COMPUTING LABORATORY

The Duality of Fault-Tolerant System Structures

S.K. Shrivastava, L.V. Mancini and B. Randell

TECHNICAL REPORT SERIES
No 305 February, 1990
The Duality of Fault-Tolerant System Structures

S.K. Shrivastava, L.V. Mancini and B. Randell

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 and conversations. Applications in the areas such as office information and database systems typically are described and built in terms of the first model while applications in the area of process control are usually described and built in terms of the second model. The paper claims that the two models are duals of each other and presents arguments and examples to substantiate this claim. An interesting conclusion to be drawn from this study is that there is no inherent reason for selecting one model over the other. 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.

Series Editor: M.J. Elphick

©1990 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne, Computing Laboratory, Claremont Tower, Claremont Road, Newcastle upon Tyne, NE1 7RU, England.

Newcastle upon Tyne: University of Newcastle upon Tyne: Computing Laboratory, 1990.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 305)

Added entries
MANCINI, Luigi Vincenzo
RANDELL, Brian
UNIVERSITY OF NEWCASTLE UPON TYNE.
Computing Laboratory. Technical Report Series. 305

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 and conversations. Applications in the areas such as office information and database systems typically are described and built in terms of the first model while applications in the area of process control are usually described and built in terms of the second model. The paper claims that the two models are duals of each other and presents arguments and examples to substantiate this claim. An interesting conclusion to be drawn from this study is that there is no inherent reason for selecting one model over the other. 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.

About the author
Professor Shrivastava joined the Computing Laboratory in August, 1975, where he is a Professor.
Dr. L.V. Mancini was at the Computing Laboratory from May 1985 to March 1989 as a Research Associate.
Professor Randell has been a Professor of Computing Science at the Computing Laboratory of the University of Newcastle upon Tyne since 1969.

Suggested keywords
ATOMIC ACTIONS
CONCURRENT PROCESSING
DISTRIBUTED SYSTEMS
FAULT TOLERANCE
OBJECT-BASED SYSTEMS
OPERATING SYSTEMS
RELIABILITY

Suggested classmarks (primary classmark underlined)
Dewey (18th): 001.64404
U.D.C. 519.687
The Duality of Fault-Tolerant System Structures

Santosh K. Shrivastava, Luigi V. Mancini and Brian Randell

1 Computing Laboratory
The University of Newcastle upon Tyne
Newcastle upon Tyne, NE1 7RU, U.K.

2 Dipartimento di Informatica
Universita di Pisa
Pisa, Italy

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 and conversations. Applications in the areas such as office information and database systems typically are described and built in terms of the first model while applications in the area of process control are usually described and built in terms of the second model. The paper claims that the two models are duals of each other and presents arguments and examples to substantiate this claim. An interesting conclusion to be drawn from this study is that there is no inherent reason for selecting one model over the other. 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.

Key words: Atomic actions, distributed systems, object-based systems, concurrent processing, reliability, fault tolerance, operating systems
1. Introduction

An investigation of backward error recovery based fault tolerance techniques employed in a variety of systems reveals two general classifications corresponding to two largely separate groups of application areas, each with its own terminology and techniques. We propose two canonical 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) controlling operations on objects. 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 model (OM) has found usage include database oriented applications in banking, office information, and airline reservation 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 (conversations) by processes. We will refer to this way of structuring an application as employing the process and conversation model (PM).

In this paper we claim that the OM 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. 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 [1] 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, a short preliminary version of which was published in [2], may be considered as an extension of the ideas put forward in the Lauer and Needham paper with regard to fault tolerance in distributed systems.

The paper is structured as follows: sections two and three describe the essential aspects of the OM and PM respectively. Section four contains the arguments intended to establish the duality mapping between OM and PM. Sections five contains two simple examples, followed by three sections illustrating the benefits obtained from the duality mapping. The concluding section summarizes the paper and discusses possible implications of the duality claim.

Throughout the paper, we will assume a distributed system composed of a number of nodes (computers) connected by some communication system. In common with the assumptions made in
the literature surveyed for this paper, a node will be assumed to have the fail-silent property: it
either works as specified or simply stops working (crashes). After a crash a node is repaired
within a finite amount of time and made active again. A node can have both stable storage, a type
of storage which can survive node crashes with high probability, and non-stable (volatile) storage
or just non-stable storage. All of the data stored on volatile storage is assumed to be lost when a
crash occurs; any data stored on stable storage, as stated earlier, remains unaffected by a crash. It
is also assumed that the faults in the communication system are responsible for transient failures
such as lost, duplicated or corrupted messages. Well-known network protocol level techniques are
available for coping with such failures, so their treatment will not be discussed further.

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, an operation on a remote object is
invoked by making a remote procedure call (RPC). Programs which operate on objects are
executed as atomic actions with the properties of (i) serializability, (ii) failure atomicity and (iii)
permanence of effect (see [3-8] for a representative sample and discussion). 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 (committed), producing
the intended results or be aborted, producing no results. Typical failures causing an action to be
aborted are node crashes and communication failures such as persistent loss of messages. It is
reasonable to require 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. 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, committed 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], who pioneered the development of the atomic action concept through his work on *spheres of control*). According to Fig. 2, action B's constituents are actions B₁, B₂, B₃ and B₄. A directed arc from an action (e.g. A) to some other

---

![Diagram](image-url)

*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 that 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 OM 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 may well require a suitable mapping on to processes (clients and servers) for carrying out the required functions, since most present day operating systems support processes as the entities to be manipulated. 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 OM, objects are long lived entities and are the main repositories for holding system states, while actions are short lived entities.

3. Process and Conversation Model

In contrast to the OM, where processes and messages play at most a secondary role, the PM has them as the primary entities for structuring programs. An application is structured out of a number of concurrent and interacting processes. The topic of backward error recovery among interacting processes has been studied extensively [13-18], beginning with the study reported in [19].

The PM 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). So, if a node crashes, then after a crash recovery its processes are initialized from their latest checkpoints. In the following, when we say that a 'process fails' we will mean that either the process detects an error which necessitates the initiation of a rollback to the latest checkpoint or that the node of the process crashes, causing the process to be initialized from its checkpoint as stated earlier.
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 (i.e., represents a 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.

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 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, 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 after taking fresh checkpoints. 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 indicated in the figure. The need for taking checkpoints at the start of a conversation can often be dispensed with. Assume that after exiting from conversion C₁ and before entering C₂ (see fig. 6), p and q do not modify their states; then there will be no need for p and q to take checkpoints upon
entering C₂. If a process fails within C₂, both the processes will be rolled back to their latest checkpoints established when exiting from C₁. From now on, brackets will not be drawn explicitly in the conversation diagrams.

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 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 systems stem from the need to synchronize with the events in the external environment, rather than from any inherent needs of processes themselves. Fig. 7 depicts a typical synchronization requirement. An informal interpretation of such a synchronization graph is as follows (see [22] for a precise formulation): process P₁ repeatedly initiates a computation at time T₁ which must finish by time T₃ (T₃ > T₁); processes P₂, P₃ and P₄ complete two iterations in the interval T₁ to T₃. Any interactions between P₂, P₃ and P₄ can be performed within the confines of two conversations: one starting at T₁ and finishing at T₂ and the other starting at T₂ and finishing at T₃. The use of conversations for introducing fault tolerance in the manner indicated here is discussed at length in [22, 23].

The most important aspects of the PM 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 for structuring applications. Conversations can be imposed on such applications by exploiting naturally occurring synchronization points among interacting
processes. In the PM, 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 differ somewhat from that used here. The duality between the OM and PM can be established by considering objects and actions to be the duals of processes and conversations respectively. Further, object invocations can be considered duals of message interactions [2]. A given conversation diagram (e.g. Fig. 8.a), can be translated into an action diagram quite simply (e.g. Fig. 8.b) by replacing each conversation Ci with a corresponding action Ai, and adding an arrow from Ai to Aj if Ci and Cj have at least one process in common and that process enters Cj after exiting from Ci. An arc from one action to the other is labelled with the 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 OM and PM.

(1) **Freedom from interference.** In the OM, 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, 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 OM, 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. 9 illustrates that if "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 is required to ensure that the states of all the processes participating in a conversation are made stable.

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

<table>
<thead>
<tr>
<th><strong>Object Model</strong></th>
<th><strong>Process Model</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Objects</td>
<td>Processes</td>
</tr>
<tr>
<td>Actions</td>
<td>Conversations</td>
</tr>
<tr>
<td>Object invocations</td>
<td>Message interactions</td>
</tr>
<tr>
<td>Concurrency control for serializability</td>
<td>Conversation rules preventing no outside communication</td>
</tr>
<tr>
<td>Stable objects</td>
<td>Stable processes</td>
</tr>
<tr>
<td>Growing phase (two-phase locking)</td>
<td>Processes entering a conversation</td>
</tr>
<tr>
<td>Shrinking phase (two-phase locking)</td>
<td>Processes leaving a conversation</td>
</tr>
</tbody>
</table>

**Table 1. Duality mapping.**

5. **Simple Examples**

This section contains two examples, one taken from the database area and normally expressed in the OM and the other taken from the process control area and normally expressed in the PM. 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 kinds of abstract data types will be assumed: standing-order, and credit-debit:

```
  type standing-order
     -- 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
     -- 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 types can be created as objects:
order : standing-order;
acc1, 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.

    task type standing-order

    -- 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
task type credit-debit

- 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 tasks can be created as processes:

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.

Process control application. The second example is taken from a process control application in the coal mining industry [24]. Fig. 12 shows a simplified pump installation. It is used to pump mine-water collected in the 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 (representing the pump control station), surface (representing the control room), level (representing the water level in the sump), pump (representing the pump) and monitor (representing the environment monitor station). Some sketchy details are given here for the pump controller.

Pump-controller process. Some of its functions are to receive start/stop commands 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 pump-controller process can send start/stop commands to the pump process which controls the pump.

A study of the process structure discussed in [24] 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.
<table>
<thead>
<tr>
<th>(a) Process Model</th>
<th>(b) Object Model</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>task type</strong> pump-controller</td>
<td><strong>type</strong> pump-controller</td>
</tr>
<tr>
<td>- - variables - -</td>
<td>- - variables - -</td>
</tr>
<tr>
<td><strong>select</strong></td>
<td><strong>action</strong> on/off(...)</td>
</tr>
<tr>
<td><strong>conversation</strong> on/off(...)</td>
<td><strong>invoke</strong> start/stop operation</td>
</tr>
<tr>
<td>send start/stop command to the pump process</td>
<td>on the pump object</td>
</tr>
<tr>
<td><strong>end conversation</strong></td>
<td><strong>end action</strong></td>
</tr>
<tr>
<td>- - other selections - -</td>
<td>- - other actions - -</td>
</tr>
<tr>
<td><strong>end select</strong></td>
<td><strong>end pump-controller</strong></td>
</tr>
<tr>
<td><strong>end pump-controller</strong></td>
<td></td>
</tr>
</tbody>
</table>

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. If it fails, the pump state is unchanged. 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 for 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. For the sake of simplicity, we have not incorporated any exception handling in these examples, but would like to point out that exception handling techniques proposed for conversations [25] are broadly similar to those for atomic actions [12], thus more sophisticated versions of these examples will continue to show the similarities exhibited here.

6. Benefits of the duality mapping: a simple example

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 simple example.

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. [26]. However, to our knowledge, no such strategies have been studied for conversations, although they can be developed quite easily using the duality mapping. 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 and $x$ write locked; the commit protocol will involve only making object $x$ stable to its new state and the action needs generate no recovery data for $y$ and $z$. Fig. 14 shows a possible conversation to perform the same computation. In this particular case it

![Diagram](image)

Fig. 14. Read only requests.

is only necessary for process $x$ to establish a checkpoint. Message $m_1$ ($m_2$) is a 'read' 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 OM. It is worth noting that, just as locking can cause deadlocks among actions, similar problems can occur in conversations.

7. Benefits of the duality mapping: Object and Process Replication

In this next section we present a major case study illustrating further the advantages gained by applying the duality mapping. We consider the following problem. Given is a set of concurrent processes interacting via conversations. We assume that a subset of these processes (server processes) provide various services to the remaining processes (client processes) and that these services have to be made available despite a bounded number of server node crashes. Thus it is necessary to replicate servers on different nodes. Algorithms for implementing such a system
where client processes interact with replicated servers within the framework of conversations, to the best of our knowledge have not been studied extensively. Here we will show how such algorithms can be developed easily by applying the duality mapping to object replication techniques which have been studied extensively [27].

There is a large body of literature on the topic of replicated data management for increasing the availability of data, (see [28,29,30] for a representative sample). A number of algorithms have been developed and their correctness properties have been stated and proved using rigorous mathematical techniques. The correctness criterion for the algorithms is that the copies of an object should remain mutually consistent such that they appear like a single logical object (this is referred to as the one copy serializability property). However, replication techniques for communicating processes have not been studied and analyzed to the same degree of detail or rigour. To be specific, we will consider two well-known object replication techniques: the available copies and the primary copy schemes. In the following we will briefly describe these techniques.

7.1. The available copies scheme

The available copies scheme does not require an action to update all copies of each object. An action should send every operation request to all of the copies that it can, but it may ignore any copies that are down. After sending an operation request to all copies of object \( x \), an action may receive rejections from some nodes (if the operation is conflicting with some other action), positive responses from others (meaning the operation has been accepted and performed), and no responses from the remaining (those that have failed). Operation requests for which no responses are received are called missing. If any rejection is received or if all operation requests to \( x \)'s copies are missing, then the whole operation is rejected and the action must abort. Otherwise, the whole operation is successful. Since a fail-silent behaviour of the nodes is assumed, any one of the positive responses can be taken as the result of the operation invocation.

7.1.1. Recovery

So far we have assumed that a failed node does not recover. This limitation can be removed by providing a reconfiguration mechanism which can support the recovery of a copy from a failure, and in general the creation and removal of copies of an object. To achieve reconfiguration after a node crash, the set of nodes holding the available copies of an object must be established dynamically.

A solution is to employ directories to record for each object \( x \) the set of \( x \)'s copies that are available. Like any other object, a directory may be replicated, that is, it may be implemented as a set of directory copies at different nodes. In the following discussion of a scheme to support dynamic reconfiguration, we assume that there is a fixed set of copies for each directory, known to every
node. That is, new directory copies are never created - in fact the method for creating new object copies can be easily extended to create new directory copies.

Directories recording available object copies are manipulated by two special actions, Join for creating new object copies, and Delete for deleting unavailable object copies. When a node \( N \) containing a copy of \( x \), say \( x_N \), recovers from a failure, it runs an action \( \text{Join}(x_N) \). \( \text{Join}(x_N) \) brings the state of \( x_N \) up-to-date by: (1) finding a directory copy \( d \) listing the set of copies of \( x \); (2) reading \( d \) to find an available copy of \( x \), say \( x_M \); (3) copying \( x_M \)'s state into \( x_N \); (4) declaring \( x_N \) to be available by making an entry for \( x_N \) in each available copy of the directory \( d \).

When a node fails, some client that tries to invoke an object operation at that node observes the failure. Then it runs a Delete action for each copy stored at the failed node. Delete declares the relevant object copy to be unavailable by removing the entry for this copy from every available copy of the directory. Because objects are replicated, it is possible to eliminate the need for stable storage for recording their states, since a recovering node can use a Join action for acquiring up-to-date versions of objects.

### 7.1.2. Read optimization

An optimization is possible with the available copies scheme if one takes advantage of the semantics of the operations. The operations that are exported by each object may be partitioned into two broad classes: the class write of operations that modify the state of the object, and the class read which incorporates operations that do not alter the state of the object. Operations of class read then need not be invoked on all available copies of an object but just on one, while the operation requests of class write need to be sent to all available copies of an object. For example, in the case of a replicated stack object, the operation top, which returns the value at the top of the stack without modifying it, can be invoked just on any available copy of the stack. The operations push and pop are of class write, and must be invoked on all available copies of the stack.

A distributed two-phase locking scheme can be employed for concurrency control. The following rule is required: whenever an operation of class read is invoked on an object, the action must first acquire a read lock (if not already acquired) on any available copy of the object; for a write operation, the action must first acquire write locks (if not already acquired) on all the available copies of the object.

When this form of read optimization is used, a validation protocol is required to prevent inconsistencies in the presence of failures (see [31] for details). To understand the need for such a protocol, consider the case of two replicated objects \( x \) and \( y \), with copies \( x_1, x_2 \) and \( y_1, y_2 \) respectively. An action, \( A_1 \), reads a value from a copy of \( x \) and writes it to all the available copies of \( y \), while another action, \( A_2 \), performs a similar update from \( y \) to \( x \). Both the actions simultaneously read from objects \( A_1 \) from \( x_1 \) and \( A_2 \) from \( y_1 \) after which two failures occur making \( x_1 \) and \( y_1 \) unavailable. The actions then update the available copies of the objects.
Clearly, this not a serializable behaviour (updates on x1 and y1 have been missed). The validation protocol is intended to detect such occurrences. An action’s validation protocol starts during commit processing, by which time the action’s operations on copies have been acknowledged or timed out. At that time the action knows all the copies it has actually accessed. The validation protocol makes sure that all copies that were available (unavailable) are still available (unavailable), else the action is aborted. In the example considered above, both A1 and A2 will be aborted.

The read optimization with validation protocol scheme suffers from the limitation that in certain situations an action has to be aborted if a failure occurs. As we have discussed previously, without read optimization, the completion of an action can be guaranteed in the presence of a specified number of node failures, by distributing operation requests to all available copies. Distributing information about read requests, and in particular about read locks, may seem unreasonably expensive. However, in [29], a scheme for lazy propagation of read locks is mentioned which guarantees that read lock information is delivered to a node before any action that requires this information is executed.

The Join and Delete actions discussed previously are still required to support recovery with read optimization. In the subsequent discussions, we will use the term pure available copy scheme to refer to the particular scheme without read optimization.

7.2. The primary copy scheme

With the primary copy scheme, executing actions use a non-replicated view of the system. That is, for each object that the actions access, the operations are carried out on the same copy of the object, called the primary copy. The distribution of the operations to other backup copies is delayed until the action is ready to commit. It is necessary therefore to maintain an intentions list of deferred operations. During the termination of an action, the appropriate portion of the intentions list has to be sent to each node that contains backup copies of the relevant objects. Alternatively, the primary copy of an object can send its new state in place of the intentions list. In any case, the information can be piggybacked on the prepare messages of the first phase of the commit protocol. If the node of the primary copy fails then the executing action is aborted; it can be resubmitted to use a different copy that will take over as the primary (see below).

In order to support recovery after a failure of a primary copy, it is necessary to elect a backup copy as the new primary. A simple scheme, that does not involve additional communication, is to fix a static ordering of the copies. An alternative is to run a consensus protocol among the backup copies - the election of the new primary copy can be based on the current load on the system.

With the primary copy scheme, it is possible to put all deferred operation requests destined for the same node in a single message. This tends to minimize the number of messages required to execute an action. By contrast, with available copies scheme, the action sends operation requests
to replicated copies while it executes. Thus the available copies scheme tends to use more messages than the primary copy scheme. Another advantage of the primary copy scheme is that aborts often cost less compared with the available copies scheme. In the available copies scheme, when an action aborts, it is likely that many of the action's operations have already been distributed to replicated copies. Not only are these operations wasted, but they must also be undone. With the primary copy scheme, the distribution of these operations are delayed till commit time, making aborts cheaper. Fast aborts in the primary copy scheme are at the expense of commits, which can be more time consuming than in the available copies scheme. This is because during the first phase of the commit protocol a node may be asked to process a potentially large number of deferred operations on backups. With the primary copy scheme, read optimization is possible - the intentions lists of only write operations need be distributed to backup copies when the executing action is ready to commit.

The most important aspects of the object replication techniques relevant to this paper are summarized below. The pure available copies scheme provides \textit{k-object-resiliency}, meaning that out of \( k \) copies of an object, all the \( k \) copies have to become unavailable before the action using it is forced to abort. With read optimization, \textit{k-object-resiliency} is not always reachable; this is the price paid for obtaining higher efficiency. The primary copy scheme does not provide \textit{k-object-resiliency} in the sense mentioned above; the executing action has to be aborted if the primary fails. The action can be resubmitted once a secondary is elected to be the primary.

\section*{7.3. \textit{Process replication techniques}}

Process replication techniques take advantage of the existence of multiple nodes by replicating critical processes on two or more nodes. A terminology commonly used for classifying the redundancy employed in PM is to differentiate \textit{active replication} from \textit{passive replication}. With an active replication scheme, a given computation is executed simultaneously by a number of processes, while with a passive replication scheme, if the process running the computation fails then a designated backup process takes over. Not surprisingly, it turns out that active replication techniques correspond to the available copy schemes and passive replication techniques to the primary copy schemes.

The duality mapping between object and process replication schemes is shown in Table 2.

\subsection*{7.3.1. \textit{The available processes scheme}}

The dual of the available copy scheme results in an approach where replicated processes behave like a single process. Interactions with a replicated process implies interactions with all of its replicas. A copy of a request to a replicated process is sent to all replicas, and all replicas execute each request. In case a reply is required, all replicas generate replies; only the first reply received is considered, and the others are discarded (since the replies from all working replicas should be identical under the fail-silent assumption on processor behaviour). A reconfiguration strategy for
<table>
<thead>
<tr>
<th></th>
<th>Active replication</th>
<th></th>
<th>Passive replication</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>pure form</td>
<td>recovery</td>
<td>read optimization</td>
<td>pure form</td>
</tr>
<tr>
<td><strong>Object replication</strong></td>
<td>available copy</td>
<td>available copy + join</td>
<td>available copy + validation</td>
<td>primary copy</td>
</tr>
<tr>
<td><strong>Process replication</strong></td>
<td>available process</td>
<td>available process + join</td>
<td>available process + validation</td>
<td>primary process</td>
</tr>
</tbody>
</table>

Table 2. Object and process replication.

the available processes scheme can be designed by adopting the directory-based scheme. Note that replicated processes need not record their checkpoints on stable storage (see Section 7.1.1).

7.3.2. Read optimization

Read optimization can be achieved in PM using the approach employed in OM. Assume that processes receive message via message ports which are data structures capable of holding messages of a certain class. Message ports can be of class read, capable of receiving messages whose processing does not alter the state variables of the receiver process, and write, capable of receiving messages whose processing can alter the state variables of the process. A message intended for a read port of a process need not be sent to all available copies of that process.

Read optimization in PM will require the conversation rules to be modified as indicated in section 6. Instead of having all the copies of a replicated process participating in only one conversation at a time, a replicated process can take part in more than one conversation, if only requests of the class read are being served.

This optimization may lead to problems of consistency similar to those encountered in OM, as can be appreciated by considering the process diagram of Figure 15. Here a system with non-replicated processes p and q and replicated processes x (copies x_A, x_B) and y (copies y_C, y_D) is considered. Suppose that within the conversation C_1, p reads the state of x and updates the state of y (denoted by r_x and w_y, respectively) and that, within the conversation C_2, q reads the state of y and updates the state of x (denoted by r_y and w_x). Conversation C_1 begins by reading from x_A, and conversation C_2 begins by reading from y_D. After p and q complete their reads, processes x_A and y_D fail. Then p and q perform their writes. Since y_D is down, y_C is the only available copy of y. So C_1’s w_y is invoked only on y_C. Similarly, since x_A is down, C_2’s w_x is invoked only on x_B.

The execution above does not violate the conversation rule for read optimization (just as, in OM, the two-phase locking rule will not be violated). However, this execution is not equivalent to any
serial execution. A serial execution of $C_1$ and $C_2$ on a non-replicated system would have either $C_1$ reading the value of $x$ written by $C_2$, or $C_2$ reading the value of $y$ written by $C_1$ - in the example neither conversation reads the data written by the other.

The dual of the validation protocol is required at the end of conversations to ensure correctness. The aim of the validation protocol in the PM is to make sure that all processes found available (unavailable) during the execution of a conversation are still available (unavailable).

### 7.4. The primary process scheme

With the primary process scheme, the same copy of a replicated process, called the primary process, takes part in conversations. A replicated process is provided with backup processes on different nodes (in particular, just one backup process might be employed). Request messages are sent to the primary process, which handles the requests. The distribution of requests to other backup processes is delayed until the end of the conversations. At that time, the primary copy sends the list of requests served during the conversation to the other backup processes. Alternatively, the primary process can send a checkpoint of its new state to the backup processes. In the event of a primary process failure, the executing conversation is rolled back to the beginning and restarted with a backup process which takes over to become the primary. The dual of the election scheme mentioned earlier will be required to select the new primary process.

### 7.5 Summary

The main objective of this section was to illustrate that the duality concepts can be utilized for developing algorithms of practical interest. Given that a large class of data replication techniques
has been proposed in the database literature, it seems natural to enquire whether these techniques can be adapted to process control systems. Assuming that a set of processes interact via message passing and use checkpoints for recovery and that the servers are to be replicated for increased availability, can we use existing data replication techniques for deriving process replication techniques? That this is indeed possible was demonstrated. Moreover, some of the existing process replication techniques can also be shown to be special cases of the techniques developed here [27].

8. Benefits of the duality mapping: further examples

To round off the discussion on the benefits gained by exploiting the duality mapping, we present three additional interesting examples.

8.1. Fault tolerance in distributed Ada

The concepts presented here can be exploited in deciding on the right set of fault tolerance mechanisms. Consider the following practical problem. There is a growing need for developing suitable techniques for the construction of fault-tolerant distributed systems written in Ada. Two broadly differing approaches to partitioning Ada programs for distribution have emerged: (i) tasks (corresponding to processes in our terminology) as the main units of distribution, communicating via remote rendezvous (synchronized messages) [32]; and (ii) packages (corresponding to objects in our terminology) as the main units for distribution, communicating via RPCs [33]. Neither of the approaches is entirely satisfactory (see [33] for a discussion) since it turns out that Ada was not particularly designed with distributed processing in mind. Indeed, much of the ongoing discussions on Ada distribution is intended towards influencing the future versions of the language. The problem is then further compounded by the need for fault tolerance in such distributed programs. (Should tasks be allowed to take checkpoints? Should package bodies be executed as atomic actions? Should rendezvous be structured as conversations or atomic actions? Should tasks or packages or both be replicated? etc.). When faced with such a bewildering choice of possibilities, we would suggest that any investigation of fault tolerance issues should consider the PM-based approach for any task-oriented solution and the OM-based approach for any package-oriented solution. A PM based approach has been examined in [32,34]. The duality mapping of this paper provides not only the right framework for developing an alternative, OM-based approach, but also for evaluating the two approaches. The duality mapping further suggests that it may well be possible to develop certain generic mechanisms capable of supporting both approaches.

8.2. Choosing the functionality of layers in hierarchically structured systems

One of the criticisms levelled at the duality argument is based on the claim that the PM describes the system level activities at a lower level of abstraction than the OM (the interactions between objects are typically implemented using the client-server model of processes), so the OM and PM
cannot be considered as duals. We acknowledge the strength of this argument and accept that in object-oriented systems, it is quite possible to regard OM as a higher level model than PM (this is particularly so, since present day object-oriented systems are implemented over operating systems supporting the process view). However, this viewpoint does not invalidate our arguments; on the contrary, the duality mapping, together with the 'end to end argument' [35] can be usefully exploited for understanding the issues and incorporating the necessary mechanisms for fault tolerance at the right level of abstractions in layered systems. We give a practical example illustrating this observation.

What reliability mechanisms should a 'traditional' PM based operating system provide? Support for checkpointing and active replication for processes seems an attractive choice. Within the context of reliability, the end to end argument would suggest that application level reliability requirements may well make some of the lower level fault tolerance measures unnecessary. Suppose now the application level software has been designed to support atomic action based transaction processing, with the available copies scheme for replication. Will it be necessary to implement any action specific fault tolerance mechanisms, given the functionality of the underlying layer (i.e., can the underlying layer directly support the the application level requirements)? We would claim that maximum benefit from the underlying layer can only be obtained if the action implementation reflects the duality mapping; if this is not the case then some of the functionality of the layer may well turn out to be either superfluous or ill matched. An examination of an existing system supports this observation, in that some of the fault tolerance functionality (checkpointing) of the PM turns out to be unsatisfactory in certain settings [36].

8.3. Fault-tolerant garbage collection

For the last example, consider the problem of fault-tolerant garbage collection in object-oriented distributed systems, implemented on top of a PM based operating system. When a node crashes, it can leave garbage (unreferenced objects) on other nodes, which is required to be collected. Rather than designing a brand new sub-system for collecting this type of garbage, we can first examine the functionality of the lower levels to see if any process-based mechanisms can be exploited. The dual of objects are processes, so unwanted objects correspond to unwanted processes (called orphans [37]). If the underlying level supports orphan detection and elimination, then can these facilities be exploited for garbage collection? This indeed turns out to be the case, as a recent design and implementation has demonstrated [38].

9. 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 transaction processing 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, we are not aware of any earlier literature (other than our own earlier efforts \([2,27]\)) 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 began by examining the 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. Further, 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. The second detailed example indicated how object replication techniques can be used for deriving process replication techniques. Some additional examples illustrated the relevance of the duality argument in fault-tolerant system design and structuring.

It must be noted that real systems often do not possess all of the features of a given model in the clean fashion assumed here. For example, many present-day transaction processing systems do not support fully fledged concurrency control for serializability or support nested concurrent transactions. Similarly, many process control systems, typically simple embedded applications in real-time control, do not require stable storage support as in transaction processing systems, but utilize ROMs for initializing failed processes. Nevertheless, by indicating the relationships between the idealized versions of such systems (the canonical models), we believe that a deeper understanding of fault-tolerant system structures has been obtained. We further observe that many large scale systems are required to be integrated, combining both database and process control functionalities of possibly heterogeneous subsystems in a uniform manner; in this case, an understanding of the duality concepts would undoubtedly help the integration process.

The establishment of the equivalence between the two approaches to fault tolerance would mean that there is 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. It can be further stated that a single system architecture based on either model can in principle, support
both classes of applications. This observation is particularly relevant, given the current popularity of the object-oriented approach to (distributed) computing. Certainly, several industry-led research projects are trying to develop and standardize on OM-based architectures [eg., [39, 40]]. The concepts presented in this paper provide a suitable framework for recasting PM-based applications into their counterparts in the OM.

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. This work was supported in part by research grants from the UK Science and Engineering Research Council, the UK Ministry of Defence and ESPRIT projects DELTA-4 (project number 2252), ISA (project number 2267) and PDCS (project number 3092).

References