Abstract:

A formal definition is given for an activity to "appear atomic at some level of abstraction". This definition is intended to make precise such concepts as "uninterruptibility of actions". It is particularly stressed that the notion of atomicity relates intimately to process structure.

These definitions form a general framework for applications. We consider two important applications: (a) transaction-oriented data base management, (b) error recovery techniques for distributed systems. A miscellany of results is included throughout the text.

A Formal Model of Atomicity in Asynchronous Systems

by

E. Best and B. Randell

TECHNICAL REPORT SERIES

Series Editor: Dr. B. Shaw

Number 130
December, 1978

© 1978 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.
A formal model of atomicity in asynchronous systems.


Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1978.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 130).

---

Abstract

A formal definition is given for an activity to "appear atomic at some level of abstraction". This definition is intended to make precise such concepts as "uninterruptibility of actions". It is particularly stressed that the notion of atomicity relates intimately to process structure.

These definitions form a general framework for applications. We consider two important applications: (a) transaction-oriented data base management, (b) error recovery techniques for distributed systems. A miscellany of results is included throughout the text.

---

About the author

Mr. E. Best received the diploma in Computer Science from the University of Karlsruhe, W. Germany in 1974. Since 1976 he has been a Research Associate at the University of Newcastle upon Tyne.

Professor Randell has been Professor of Computing Science in the Computing Laboratory of the University of Newcastle upon Tyne since 1969.
## Contents

1) **Introduction**  
   1

2) **An Example**  
   5

3) **Occurrence Graphs**  
   9
   3.1 Occurrence graphs, and related definitions  
   9
   3.2 Subgraphs, and related definitions  
   10
   3.3 Collapsing  
   10

4) **Structuring and Atomicity of Activities**  
   12
   4.1 Basic occurrence graph  
   12
   4.2 Structuring the basic graph  
   12
   4.3 Structure tree and lattice of levels  
   12
   4.4 The atomicity criterion  
   15
   4.5 Functionality of activities  
   16

5) **Interference and Contractions**  
   18
   5.1 Interference amongst activities  
   18
   5.2 Contractions  
   19

6) **R/W - Graphs and Data Base Consistency**  
   23
   6.1 R/W - graphs  
   25
   6.2 Valid, consistent and serial schedules  
   26
   6.3 Existence of fully consistent schedules  
   28
   6.4 Typed links  
   29

7) **Atomicity and State Restoration**  
   34
   7.1 State restoration in an unstructured graph  
   34
   7.2 State restoration in a structured graph  
   42

8) **Discussion**  
   43
1) Introduction

The present paper is intended as a contribution towards the goal of achieving a valid formal understanding of the very important notion of "atomicity" of actions. Suppose that we write a program fragment intended to change part of a global data structure; for example, we might write a procedure \texttt{rem} for removing an element of a shared list. Although an arbitrary invocation of \texttt{rem} takes a finite time, we nevertheless wish to be able to consider this activity as "instantaneous", thus guaranteeing that no other activity has interfered with it. Interference by other activities may cause quite unintended effects to occur, as the fuller treatment of this example in section 2 shows.

We break up the problem in order to present an overview of the subsequent material.

First, there is the programmer who may wish to structure his program by atomic actions. For his benefit, we may provide a language facility such as the one put forward in [LON77]:

\begin{verbatim}
atomic action rem ... end
\end{verbatim}

This will enable him to treat calls to \texttt{rem} as "instantaneous", which will in turn facilitate the verification that his code observes any required consistency constraints. For example, it is presumably required that \texttt{rem} transforms a valid, non-empty list into another valid list. Interference due to non-atomicity of \texttt{rem} could violate this requirement.

Assumptions about the atomicity of certain actions or procedures have long been made, mostly under the name of "indivisibility", or "non-interruptibility". We might mention Dijkstra's P and V primitives which are required to be atomic. More recently, for example in [LIP75] and [OWI77], it has been fully appreciated that the task of proving programs correct can be facilitated by first ensuring or proving that certain actions are atomic, and then proving that they preserve the given consistency constraints.
To date, in papers on program proving or related issues, one is unlikely to find a formal statement of what is meant by an indivisible action. Our own starting point is the observation that the concept of atomicity must be defined not in terms of actions, but of invocations of actions. The invocation of an action we term an activity.

Our aim is to give a formal criterion for an activity – not an action – to appear atomic. To this end, we shall use occurrence graphs ([HOL68], [PET76]) to model (concurrent) computations; this model is informally introduced in section 2 and described fully in section 3. Each activity corresponds to a subgraph of an occurrence graph, the rest of the graph being its environment formed from other on-going activities.

If an action such as rem is written with the qualifier "atomic" specified, this indicates the requirement that each invocation of rem should satisfy the atomicity criterion. However there is the possibility that the environment does not attempt to interfere, so that a version of rem which lacks such a specification would on occasion give rise to an atomic activity. In other words, whether an activity appears atomic or not will normally depend on its environment. Accordingly, the atomicity criterion must be context-dependent.

We shall argue that moreover the atomicity criterion must be level-dependent. By this we mean that cases are conceivable in which activity A appears atomic provided that another activity B in its neighbourhood is not seen as atomic. On the level on which B is known in detail, A indeed appears atomic; on a higher level of abstraction on which B is perceived as just an "instantaneous" activity, A can however not be perceived as atomic.

In order to formulate this precisely and generally, we shall permit arbitrary nesting of activities, and consider the resulting structured graph. All will be the subject of section 4.
Given our understanding of atomicity, the requirements for its implementation are easily stated. It has to be ensured that each programmer-defined atomic action leads only to activities satisfying the atomicity criterion on all levels.

The implementation problem has great importance in the context of transaction-oriented database management, the first of two application areas which are discussed in the remaining sections of this paper. Transactions are, in effect, atomic actions. Again, a considerable body of work exists. There are basically two types of solution. In the first of these the user is required to follow a restricted pattern when writing code qualifiable by the prefix "atomic". This pattern has been termed "two-phase" ([ESN76]). Two-phase actions give rise to special activities, which may be called contractions. Contractions will be the subject of section 5 of the present paper.

Alternatively the user may be allowed to specify anything he likes as "atomic", which leaves the task of ensuring proper execution to a run-time support system. The resulting problems have been investigated in [GRA76], [GRA78] (and shall be generalised presently in section 6), for the restricted case of a database transaction system. Our concern shall be twofold: firstly (in section 6.3), we shall show under which conditions an implementation of atomicity is not possible; secondly (in section 6.4), we shall contrast our approach with the idea of [GRA76] of implementing atomicity only to a certain degree.

In section 7 of the present report we apply our definitions and conceptual framework to the topic of error recovery. Again, as with sections 5 and 6, our main aim will be to elucidate and generalise earlier reported work, specifically that described in [RAN75], [RLT78], [LW77], [MR77] and [MR78]. This sequence of papers illustrates a growing awareness of the relevance of the concept of an atomic action. An important outcome is the idea that the structure of an application program supports error recovery to the extent that the atomic actions coincide with the units of recovery. This idea has prompted investigations relating to program structure ([RAN75], [AK76]). In section 7 we present state restoration
techniques based on the work of [MK77]; however, we are concerned not 
so much with program structure as with the problem of capturing what 
is meant by "units of recovery" in a distributed system.

Sections 2–4 contain basic definitions and concepts which are 
relevant for the remainder of the paper. Sections 5, 6 and 7 can be read 
in any order provided, at the beginning of section 6, one bears in mind 
that "contractions" are defined in section 5 in such a way that they 
contain a single conceptual "moment of occurrence".
2) **An Example**

Imagine a doubly linked, shared list in the following current state:

![Diagram of a doubly linked list](image)

The list consists of a start pointer, \( s \), an end pointer, \( e \), and two proper elements, \( a \) and \( b \); each element is subdivided into fields as shown.

Imagine a pointer, \( p \), to a list element; the algorithm for removing this element can make use of concurrency in the following way:

```plaintext
atomic action \( \text{rem} \)(p): \( x := p.\text{before} \) // \( y := p.\text{next} \);
\( x.\text{next} := y \) // \( y.\text{before} := x \) end.
```

(where \( x \) and \( y \) are local, and // separates sub-actions that can be carried out concurrently).

We assume without further justification that all assignments in \( \text{rem} \) are atomic actions, giving rise to basic activities. We call basic activities \( \text{events} \), represent each event by a square symbol, and use arrows to represent the "earlier" relationship between events. With this, each invocation of \( \text{rem} \) gives rise to the following pattern:

![Diagram of event flow](image)

**Figure 1**
The arrows are due to the semicolon in the definition of `rem`.

Assume now that two processes wish to remove "a" and "b" concurrently; in other words, we are considering the command `rem (a) // rem (b)`. Figure 2 shows three possible schedules:

**Figure 2a**

```
x1:=s    y1:=b
```

```
...next:=b  b.before:=s
```

```
x2:=s    y2:=e
```

```
s.next:=e  e.before:=s
```

**Figure 2b**

```
x1:=s    y1:=e
```

```
...next:=e  c.before:=s
```

```
x2:=a    y2:=e
```

```
s.next:=e  e.before:=a
```

**Figure 2c**

```
x1:=s    y1:=b
```

```
...next:=b  b.before:=s
```

```
x2:=a    y2:=e
```

```
s.next:=e  e.before:=a
```
Only the schedules shown in Figures 2a and 2b have the desired effect, namely the empty list.

In contrast, the schedule of Figure 2c gives the list

![Diagram showing the list](image)

that is, an undesired result.

We may now "collapse" the subgraphs of Figure 2 which correspond to invocations of `rem`, and obtain the following graphs, respectively:

![Diagram showing `rem (a)`](image)

![Diagram showing `rem (b)`](image)

![Diagram showing `rem (a)` and `rem (b)`](image)
Intuitively, this "collapsing" gives the activities an instantaneous appearance; in Figures 2/3a, "a" is removed before "b"; in Figures 2/3b, "a" is removed after "b"; in Figures 2/3c, the two activities mesh, i.e. interfere with each other in a damaging and atomicity-violating way. We shall argue in section 4 that such meshing is seen to take place whenever the "collapsing" operation leads to a cycle.
3) **Occurrence Graphs**

3.1 **Occurrence graphs, and related definitions**

We define an occurrence graph \([\text{HOL68}]\) (or causal net \([\text{PET76}]\)) to be a "causally interconnected set of events": events (denoted by small squares) are connected by arrows as follows:

![Diagram showing an occurrence graph with an arrow from e to e']

In general, we write "e \(\prec\) e'" if event e **precedes** event e', that is, if there is a causal path leading from e to e':

![Diagram showing a directed path from e to e']

If there is only an arrow from e to e', then e is said to **immediately precede** e'.

Clearly, \(\prec\) is a transitive relation. We say that there is a causal link between e and e' if e \(\prec\) e' or e' \(\prec\) e. We write e \(\prec\) e' if either e \(\prec\) e' or e = e'.

In occurrence graphs we disallow directed cycles (i.e. \(\prec\) is non-reflexive) because this would indicate an event being its own cause. Hence one can never have both e \(\prec\) e' and e' \(\prec\) e. If neither e \(\prec\) e' nor e' \(\prec\) e then we call e **concurrent** to e' ("e co e'").

Occurrence graphs are in fact simply acyclic graphs with the nodes interpreted as events. We shall however also need to consider cyclic graphs in full generality. When using the term "graph" we therefore explicitly include the possibility of \(\prec\) being reflexive.

Usually, arrows are interpreted as **conditions**, being the "entities changed by events". Conditions are sometimes also represented as circles \([\text{PET76}]\); that is, an arrow

![Diagram showing a circle to represent a condition]

is represented as
We shall, however, use the first notation.

We make the overall assumption that all graphs are finite.

3.2 Subgraphs, and related definitions

A subgraph $A$ generated by a set $E$ of events is defined as the set $E$ together with just those arrows which have their endpoints in $E$. Figure 4 presents an example:

![Figure 4](image)

A in Figure 4 is the subgraph generated by $E = \{e_1, e_2\}$. For each subgraph, we can identify three sets:

- the set $\mathcal{A}$ of events generating $A$ ( = $\{e_1, e_2\}$ in Figure 4)
- the set $\mathcal{A}'$ of immediate predecessors of $A$ ( = $\{e_3\}$ in Figure 4)
- the set of $\mathcal{A}$ of immediate successors of $A$ ( = $\{e_4\}$ in Figure 4)

3.3 Collapsing

Given a subgraph $A$ of a graph, we can construct a new graph by collapsing $A$, i.e. by replacing the whole subgraph by a single vertex $A'$ and drawing a single arrow from each individual member of $\mathcal{A}$ to $A'$ as well as from $A$ to each individual member of $\mathcal{A}'$.

Additionally, we obey the following convention: if the subgraph $A$ contains one or more cycles, that is, if some of its events and arrows form a cycle, then we append a loop to the new graph in which $A$ is just
a single vertex.

This convention is somewhat arbitrary. Its significance is explained in section 4.3.
4) Structuring and Atomicity of Activities

We argue that a precise definition of the atomicity of an activity must depend on the level of abstraction on which the activity and its environment is seen. Levels of abstraction arise from structuring a basic occurrence graph; the relevant definitions are contained in sections 4.1 - 4.3. In section 4.4 we give the cycle criterion for atomicity, and in section 4.5 relate this criterion to the functional activities of [MR77].

4.1 Basic occurrence graph

We assume as a starting point that there exists a detailed record of the computation history, in the form of an occurrence graph in which all events are basic activities.

Basic activities can be thought of as individual invocations of some primitive instructions; they are assumed atomic without further justification: there is no need nor any possibility of decomposing them into smaller primitives.

4.2 Structuring the basic graph

The basic occurrence graph can be structured by choosing a subgraph A, and forming a new graph by collapsing A. Another subgraph of the new graph may be chosen and the collapsing may be repeated.

Thus, overlapping becomes inherently impossible; a structure with that characteristic, that is, non-overlapping, may be termed box-structure. We assume that the specification of (possibly nested) atomic actions imposes a box structure on the basic graph.

4.3 Structure tree and lattice of levels

A box structure can be visualised in a way which is similar to the so called "contour model"; upon opening and looking inside a box A - i.e., a subgraph -, one sees a set of sub-boxes, namely A, and this box opening process can be carried out repeatedly until one reaches the basic graph.
Consider the following example:

![Diagram](image)

**Figure 5**

Opening $Z$ reveals $Z = \{A, B\}$, and opening $A$ and $B$ reveals the graph shown in Figure 2c.

The assumption that there is always a box containing the whole graph enables us to represent the relationship between the boxes of a box structure by a tree; corresponding to Figure 5 we have:

![Diagram](image)

**Figure 6**
A "horizontal" cut through the tree identifies a level (of abstraction); in Figure 6 we have five levels L0, ..., L4:

![Tree Diagram](image)

**Figure 7**

Each level defines its own graph:

**L0:**

![Graph L0](image)

**L1:**

![Graph L1](image)

**L2:**

![Graph L2](image)

**L3:**

![Graph L3](image)

**L4:**

![Graph L4](image)

**Figure 8**
The loop at Z arises from the convention of section 3.3.
Intuitively, it indicates that "inside Z some activities mesh".

The \( \prec \) relation of a level graph is completely determined by the \( \prec \) relation of the basic LO-graph, and the collapsing rule.
Notationally, we indicate the relevant \( \prec \) by prefixing or suffixing corresponding statements with the phrase "at level L".

A level L is said to contain a box A if A belongs to L. Each level L divides the tree into a portion below L (by definition including L) and another portion above L (excluding L). L is said to cover a box A if A belongs to the portion below L.

The set of all levels forms a lattice; for Figures 5–8, we have

![Diagram](image)

**Figure 9**

Given two levels L1, L2, there exists a lowest upper bound level \( \text{lub}(L_1, L_2) \); for example, \( \text{lub}(L_1, L_2) = L_3 \) in Figure 9.

4.4 The Atomicity Criterion

In Figure 8, level L3 relates to L0 as Figure 3c does to Figure 2c (from which it was derived). We do not interpret the cycle on level L3 as distorting the cause-effect relationship because the basic occurrence graph actually describes a real situation in which no event is its own cause.
Rather, we interpret the cycle as follows: at L3, one is bound to perceive the relationship between A and B as an irretrievable meshing. This meshing can only be untangled if at least one of A and B is considered in more detail, i.e. opened. At L3, neither A nor B can be called atomic.

In general, we define an activity A to appear atomic at level L if and only if it is not contained in a cycle at L.

The following immediate observation can be made:

**Theorem:** if L contains a cycle then each level above L also contains a cycle.

**Proof:** suppose L contains a cycle. This cycle cannot be absorbed by collapsing part of itself; nor can it be absorbed by collapsing it completely because then a loop is introduced according to the rule of section 3.3.

Thus, the lattice of levels can be partitioned into an upper part (possibly = Ø) of cyclic levels, and a lower part the levels of which are not cyclic.

4.5 **Functionality of activities**

Let L be a level containing a box A; let L′ be the level obtained from L by opening A, that is, replacing A by ˘A. According to [MR77], A can be called functional at L if, at L′, no directed path leads out of ˘A and back into ˘A.

Consider, for example, Figure 8. Box A is functional at L = L1 because at L′ = L0 no path can be found leading out of ˘A and back into ˘A. However, A is not functional at L = L3 because at L′ = L2 there is a path leading from ˘A to B and back to ˘A.
Functionality at $L$ characterises boxes not contained in a cycle at $L$:

**Theorem:** $A$ is functional at $L$

$\iff$ at $L'$ (as above), $\neg \exists A1 \epsilon^* A, A2 \epsilon A$: $A2 \subseteq A1$

$\iff A$ is not in a cycle at $L$

**Proof:** $A$ is functional at $L$

$\iff$ (by def.) no path from $A$ to $A$ exists outside $A$:

\[ \text{not} \]

\[ \text{there are no } A1 \epsilon^* A, A2 \epsilon A \text{ such that } A2 \subseteq A1 \text{ at } L' \]

$\iff$ collapsing $A$ does not close a cycle.

Thus, provided a level of abstraction is fixed, the cycle criterion of section 4.4 is the same as the functionality definition contained in [MR77].
5) **Interference and Contractions**

5.1 **Interference amongst Activities**

Given a level $L$ and a cycle at $L_1$, then at $L$ it appears as though the boxes of the cycle inherently mesh. We now turn to the problem of whether boxes of the cycle can be determined which, by virtue of their internal structure, can never break the cycle when opened. Intuitively, such boxes cannot be considered subject to "interference by others". For clarity, we contrast the example treated in Figures 5–9 with the following graph:

![Figure 10](image)

which leads to a similar tree to that shown in Figure 7. Its level graphs are:

![Figure 11](image)
These differ from the ones shown in Figure 8 only at levels L0 and L2.

Opening A at L3 reveals L2—a graph in which there is still a cycle. Intuitively speaking, "A interferes with B" rather than vice versa. This lack of symmetry is concealed at L3.

We leave the exact determination of the nature of such asymmetries to future attention; suffice it to mention that strange graphs may be constructed in which parts of A appear interfered with while other parts appear unaffected.

5.2 Contractions

As we have just seen, a box which is contained in a cycle but whose opening does not break the cycle cannot be blamed for violating atomicity. We identify a class of such boxes, which we call "contractions" (for want of a better phrase).

A box $A$ is called an immediate contraction at $L$ if $L$ contains $A$ and if $A \prec A$ at $L'$ (where $L'$ is obtained from $L$ by opening $A$, and $A \prec A$ means: $\forall A \in A, A \not\in A$: $A1 \prec A2$ at $L'$).

In Figure 11, for example, $A$ is an immediate contraction at $L1$ as well as at $L3$. At $L1$, $A = \{A1\}$, $A' = \{A2\}$, and at $L1' = L0$, $A1 \prec A2$. At $L3$, $A = A' = \{B\}$, and at $L3' = L2$, $B \prec B$ (i.e. $B$ is still in a cycle). In Figure 8, however, $A$ is not an immediate contraction at $L1$. 

\[19\]
Observe that it is always the case that \( A \preceq A \) at the level \( L \) where \( A \) is just a single box. The contraction property requires that also \( A \preceq A \) at the level \( L' \) obtained from opening \( A \). Hence \( A \) can, at \( L' \), be thought of containing a single "moment" which comes after all of \( A \) but before all of \( A' \); this moment can conceptually stand for "the moment of \( A \)'s occurrence". Collapsing \( A \) can be thought of as contracting \( A \) into this "moment" — hence the term.

We are interested in the stronger property that a box \( A \) has the immediate contraction property all the way down to the basic level: \( A \) is called a contraction at \( L \) if \( A \) is an immediate contraction at \( L \) and all its sub-boxes are either basic activities or contractions at \( L' \) (where \( L' \) is defined as above).

From this definition it follows:

**Theorem:** if \( A \) is a contraction at \( L \) and \( A \) is not functional at \( L \) (i.e. \( A \) is contained in a cycle at \( L \)) then opening \( A \) (and any of its sub- or sub-sub-, etc. boxes) will not break the cycle.

For example, \( A \) at \( L3 \) in Figure 11 is a contraction but not functional, and opening \( A \) does not break the cycle. Conversely, an activity may be functional at \( L \) but not a contraction at \( L \); for example, in Figure 8, \( A \) is functional at \( L1 \) but not a contraction at \( L1 \).

The definition of contraction is to some extent level-independent:

**Theorem:** if \( A \) is a contraction at \( L \) then \( A \) is also a contraction at all \( L^* \) containing \( A \) and covering "\( A \) and \( A' \)."

**Proof:** Assume that \( A \) is a contraction at \( L \). Clearly, if \( L^* \) contains \( A, A' \) and \( 'A', \) then \( A \) is also a contraction at \( L^* \). Suppose \( L^* \) is obtained from \( L \) by collapsing only boxes of \( 'A'; \) then the contraction condition is preserved.

Suppose \( L^* \) is obtained from \( L \) by collapsing only boxes of \( A' \); then the condition is again preserved.

Suppose \( L^* \) is obtained from \( L \) by collapsing boxes of "\( A \) and \( A' \); then there is a cycle at \( L^* \) and the condition is again preserved.
The contraction property provides a sufficient condition for the absence of cycles since it follows immediately from the preceding discussion that all levels are cycle-free provided all boxes are contractions at all levels. In other words, cycles may only (but do not necessarily) arise on levels covering at least one box which is not a contraction. We conclude that provided all actions give rise to contractions, cycles cannot occur and atomicity is automatically satisfied.

Conversely, there is a sense in which the contraction property is necessary for the absence of cycles:

**Theorem:** Suppose A is not an immediate contraction at L; then it is possible to rearrange the structure of the tree portion above L in such a way that A is contained in a cycle at some higher level K; moreover, opening A breaks the cycle.

**Proof:** Generalise the following example:

![Diagram](image)

**Figure 12a**

In Figure 12, A is not an immediate contraction at L, and we have A1 co A2 at L'; we arrange for a level K in which A is contained in a cycle:
In other words: suppose we cannot rely on run-time support for implementing atomicity, and suppose we run a non-contraction; then we can conceive of a "worst case mixture" of neighbourhood actions in such a way that a cycle is introduced. Thus if we want to make sure from the outset that atomicity is not violated, we have to ensure the contraction property of all action invocations.
6) R/W - Graph and Data Base Consistency

In practice, ensuring the contraction property may be achieved at the expense of performance, for example by "low granularity of locking" in the context of data base management systems [see GRA76]. In this context, actions are considered which perform read and update functions on shared data. In order to ensure atomicity, actions are required to set so-called long locks on any data entities they access. This means that an action does not release any entities for use by other actions until it has acquired all entities it needs for itself. Thus, the contraction property is ensured; conceptually, such an action occurs at the moment when it has acquired the maximum number of data entities. Performance is impaired because an action will in general lock out entities for a longer period than necessary.

In this scheme, the user, i.e. the writer of the actions, is responsible for setting the locks. If long locks are always set, atomicity is guaranteed. If no such stringent assumption about the proper setting of locks can be made, more sophisticated scheduling of entities becomes necessary. In general, if nothing is assumed about the actions, it is not necessarily true that run-time support of atomicity is possible in principle. For example, if the user is allowed to influence scheduling by using wait or similar commands, then there is no way of ensuring the atomicity of, say,

```
call A(B) // B(A),
```

where A and B are actions of the following type:

```
atomic action P(Q); signal Q;
wait Q; end.
```
One conclusion from this simple example is that the atomic action concept and the `wait` statement do not necessarily go together naturally; this remark is slightly counter to the belief expressed in [LOW77]. Even if `wait` statements are disallowed, they can still be "modelled" by actions performing only read or write to shared data. For example, define

```
atomic action P(x,y); x:=1;
   while y ≠ 1 do skip end .
```

Then the atomicity of

```
a:=b:=0; call P(a,b) // P(b,a);
```

cannot be implemented. This, of course, corresponds to "busy waiting".

For the remainder of this section we consider the problem of implementing the atomicity of actions which do \textit{not}, in one way or another, involve `wait` or "busy waiting" (by which we also include all kinds of infinite loops). Apart from this restriction, we might allow any type of basic command as a building block for actions. However, we constrain ourselves to actions consisting only of read and write commands; this we do because we shall only briefly outline the problem and will stay in the context of the much more detailed studies [GRA76, GRA73].

In section 6.1, R/W-graphs are introduced which correspond to an execution of actions consisting only of reads and writes. An R/W-graph must represent a proper schedule of the reads and writes. The schedule must be valid, i.e. not contradict user specifications (this should actually go without saying). The schedule must be consistent, i.e. two different read/write or write/write accesses to an entity must not be concurrent. One could also say, in analogy to section 4.5, that reads and writes have to be functional at the basic level. If the schedule is furthermore fully consistent, this is taken to mean that all activities are functional at all levels, i.e. that atomicity is guaranteed. These definitions are contained in section 6.2.
The term "consistency" derives from [BSW76] where it is used both generally, meaning "a database agrees with its semantics" and specifically, meaning that "this (particular) transaction schedule does not destroy the (general) consistency of the database". We use "full consistency" as a synonym for "no violation of atomicity".

Throughout this section, we consider (causal) links which are defined by the relation $<$(see chapter 2). If $e < e'$ for two events then there is a causal link from $e$ to $e'$ which may or may not be represented by a single arrow from $e$ to $e'$; if not, we have a sequence of arrows from $e$ to $e'$.

Two links $e < e'$ and $d < d'$ are, of course, different if $e \neq d$ or $e' \neq d'$; otherwise they are equal.

6.1 R/W-Graphs

We consider a graph of reads and writes to a set of shared entities $a, b, c, \ldots$. An event which reads entity "a" is denoted by

\[ \square \text{Ra} \]

an event which writes into entity "b" is denoted by

\[ \square \text{Wb} \]

For example, we assume the execution of an assignment statement "b := a" to give rise to the part

\[ \text{Ra} \quad \square \rightarrow \square \text{Wb} \]

of the graph.
Graphs in which all events are thus labelled are called R/W-graphs. Figure 2a, for example, can be redrawn as an R/W-graph as follows:

![Diagram](image)

Figure 14

6.2 Valid, consistent and serial schedules

We first consider an R/W-graph as such, for the moment neglecting any structuring, for example, the graph of Figure 14; here, all links except the cross-links \( \sigma_1 \) and \( \sigma_2 \) are user-defined, in fact by virtue of his definition of rem (section 2). We assume that it is true in general that the user defines some links. We call an R/W-graph validly scheduled if all user-defined links occur in it; in what follows, we consider only validly scheduled R/W-graphs.

The cross-links \( \sigma_1 \) and \( \sigma_2 \) are supplied so as to resolve the conflicting reading/writing to the same entity. In general, a link is called a consistency link if it relates read/write or write/write events to the same entity. Some consistency links may have been supplied by the user, others by the scheduler(s). We call an R/W-graph consistently scheduled if each write access to an entity is linked to all other accesses to that entity in such a way that the graph is acyclic.

**Theorem:** if an R/W-graph is consistently scheduled

then adding any further consistency link creates a cycle.

This follows immediately from the definition.
Redundant links may be added to a consistent schedule so that cycles are not introduced. The schedule can be called minimal if there are no redundant links; obviously, minimal schedules are maximally concurrent. Any minimal consistent schedule can be extended to a serial one in which the links represent a linear order; for example, the schedule of Figure 14 can be extended to the following one:

\[\text{Figure 15}\]

By removing all redundant links, any serial schedule can be reduced to a minimal schedule.

Now we may consider R/W-graphs on which a box structure is imposed. Such an R/W-graph is called fully consistently scheduled if it is consistently scheduled and if there are no levels at which cycles are present. A fully consistent schedule corresponds to a computation which has properly resolved all conflicting read/write
accesses and has paid attention to the atomicity specifications of the box structure.

As an example, with the box structure defined by rem (Figures 2/3), Figure 14 shows a fully consistent schedule. Figure 15 shows a valid, serial and consistent, but not fully consistent, schedule.

In general, each fully consistent schedule can be extended to a fully serial schedule which induces a linear ordering of activities on all levels; conversely, each fully serial schedule can be reduced to a fully consistent schedule.

6.3 Existence of fully consistent schedules

We prove the existence of fully consistent schedules by proving the stronger statement that there exist fully serial schedules.

By centralised scheduling, we construct a fully serial schedule for the example statement

\[ \text{rem (a) // rem (b)} \]

of Figures 2a, 3a and 14. Eventually, we shall have executed it in the way indicated by the following structured graph:

![Figure 16](image)

with the same labelling as Figure 14, and the structure tree:
In the course of the execution, we traverse the tree in the manner shown. The basic commands of an action (rem (a) or rem (b)) are executed in a valid order. In general, should one of the commands of an action not be executable, then this command is some form of wait which we have excluded. On the other hand, if we enter an infinite loop at some stage, we have encountered some form of "busy wait" which we also have excluded. In short, our restriction says that once an action is entered, it can be completed independently of other activities. Therefore, the construction will always lead to a fully serial schedule.

The interesting problem is, of course, to approach a minimal schedule as closely as possibly by dropping links and letting non-linked activities run concurrently. This problem occupies the main body of the work of [GRA76] but falls outside the scope of the present paper.

6.4 Typod links

Atomicity, or full consistency for that matter, can be implemented by enforcing the contraction property. This is likely to impair performance, as has been indicated at the beginning of section 6. On the other hand, atomicity can be implemented by centralised scheduling. Obviously this also impairs performance to an often intolerable extent. However, [GRA76] allows for the fact that in some cases full consistency is not required. It may suffice to
implement consistency only to a certain degree.

That is — returning to the language of section 4.5 —, it may suffice that only a certain number of boxes are functional at certain levels, while they or others are not functional at certain (other) levels. Thus we may distinguish various degrees of consistency, depending on what boxes at what levels one requires to be functional. For a start, we would certainly require that the primitive commands, read and write, are basically functional. Thus "consistency" as defined at the beginning of section 6.2 is the minimal degree of consistency to be distinguished; on the other hand, "full consistency" is the maximal degree since this means that all boxes are functional at all levels.

Between these two, we may identify intermediate degrees by, admittedly, rather arbitrarily imposing restrictions on the functionality of boxes. Roughly, the idea is that we may allow activity B only to read but not to change data previously updated by activity A, whilst A is still in progress. In this case, we might say that "W/W-links bind stronger than W/R-links".

We start by assigning types to causal links. A link can be of type R/R, R/W, W/R or W/W according to what kinds of events (read or write) it relates. For example, σ1 in Figure 14 is of type W/R, σ2 is of type W/W.

Link types are unique at the bottom level; to all links of Figure 14, say, a unique type can be attached. Consider, however, the collapsed version of Figure 14:

![Figure 12]

Figure 12
A is linked to B by two competing links: one of type W/R and another of type W/W. If link types are to be unique on all levels, a convention is necessary to select this type (link type selection rule). This can be conveniently done by defining a lattice of types; for example:

![Lattice Diagram](image)

**Figure 19**

This lattice can be thought of as indicating "how strongly" links bind; "W/R binds stronger than R/R"; "R/W binds as strongly as W/R". The shape of this lattice is arbitrary; it only depends on which results are to be achieved.

The link type selection rule reads: the resulting type is the lowest upper bound of all competing types. For example, in Figure 18, A is connected to B by a W/W-link:

![Link Diagram](image)

since W/W = lub(W/R, W/W) in the lattice of Figure 19.

Given, now, a level L and a cycle at L. We can characterise the cycle by a type: its type is R/R (R/W, W/R, W/W) if all links of the cycle are of type R/R (R/W, W/R, W/W) or higher.
This definition depends on the type lattice: with Figure 19, a W/W-cycle is also an R/R-cycle because W/W is higher than R/R in the lattice; but an R/R-cycle is not necessarily also a W/W-cycle.

It is now possible to distinguish degrees of consistency of a schedule:

A schedule is called \textit{d0-consistent} if it is consistent in the sense of section 6.2.

A schedule is called \textit{d1-consistent} if it is d0-consistent and if there are no W/W-cycles at any level.

A schedule is called \textit{d2a-consistent} if it is d1-consistent and if there are no R/W-cycles.

A schedule is called \textit{d2b-consistent} if it is d1-consistent and if there are no W/R-cycles.

A schedule is called \textit{d3-consistent} if it is d2a-consistent and d2b-consistent.

A schedule is called \textit{d4-consistent} if it is d3-consistent and if there are no R/R-cycles; this is the highest degree of consistency and has been termed full consistency in section 6.2.

For all these degrees of consistency, the implementation problem is viable to a lesser or greater extent. d0-consistency demands no restrictions from the side of the user and is easily implemented by distributed scheduling (i.e. each entity follows a local scheduling algorithm, independently of other entities). d4-consistency can be implemented by enforcing the contraction property; d3, d2 and d1-consistency can be implemented by enforcing gradually more and more relaxed contraction properties. The corresponding "lock protocols" have been identified in [GRA76].
It appears, however, that in [GRA76] a slightly different type of lattice is implicitly used, namely

\[ \begin{align*}
&\text{W/W} \\
&\text{W/R} \\
&\text{R/W} \\
&\text{R/R}
\end{align*} \]

**Figure 20**

If this second lattice is used, the above definitions of d0-, d1-, d2- and d3-consistency respectively generalise the notions of d0-, d1-, d2- and d3-consistency of [GRA76]; for clarity, we list the relevant definitions of [GRA76], pp. 382-386:

- **d0-consistency**: an entity is always locked in exclusive mode before written into
- **d1-consistency**: no W/W-cycles
- **d2-consistency**: d1-consistency and no W/R-cycles
- **d3-consistency**: d2-consistency and no R/W-cycles

If we compare the lattices of Figures 19 and 20, we find that in the latter R/W-links and W/R links are not treated on a symmetrical footing; rather, "W/R binds stronger than R/W". Perhaps this is not quite as it intuitively should be, and accounts for the surprise expressed in [GRA76] at the finding that d2- and d3-consistency give rather similar performance figures.
7) Atomicity and State Restoration

This section applies the structuring concepts of section 4 to the state restoration algorithm of [MR77]. Again (as in 6.2) we first assume an unstructured occurrence graph belonging to a computation in which checkpointing has been done at some stages; then we consider the situation where this graph is user-structured.

7.1 State restoration in an unstructured graph

We assume that, as part of a strategy for providing a measure of fault tolerance, the user ensures that certain conditions are preserved, in case backward error recovery has to be undertaken. In [MR77], such conditions are called restorable conditions and are represented by double circles. In accordance with what has been said in 3.4, we represent conditions by arrows, and therefore use double arrows for indicating restorable conditions.

For example, consider Figure 21:

![Figure 21](image-url)
Some computation has started with e1. Conditions b1-b5 are restorable; b6 and b7 are "active": the graph keeps growing from them when normal progress is being made. Condition b6 is marked as "invalid" by an asterisk; this indicates that an error has been detected at b6.

Whenever an error is detected, state restoration activity is initiated. This works as follows for the example graph shown in Figure 21: all events e6, e4, e3 have to be invalidated in turn, because none of them have had their outcomes preserved. But e3 has been a cause of the restorable conditions b3, b4 and b5; hence b3-b5 are also invalidated. Furthermore, during the back-up the graph may grow at b7. But this growth has to be "chased" and invalidated since b7 is also an effect of e3 and hence is invalid.

In summary: the error detected at b6 causes a back-up to the nearest recovery line [b1, b2], and a chase of the growth of the graph at b7. For the moment let a recovery line be defined as a set of restorable conditions which can be restored after backing up. In general, whenever an error is detected, an attempt is made to find the nearest recovery line and to chase the growth of the rest of the graph affected by the error.

This state restoration algorithm is described in [MR77] by special "chase protocols". Such protocols could be implemented in the various computers comprising a distributed computing system by mechanisms which use the records kept of recent system activity, in the form of an occurrence graph. The activity of the protocol can be reformulated as follows:

(i) Each part of the graph which is of the form

\[ e \xrightarrow{b} e' \]

should be collapsed into a single box; this is because of two desiderata:

- an invalidation of e should entail the "chasing" of its effect, e', and
an invalidation of e' should entail a back-up to e
(since b is not restorable).

For example, in Figure 21 it is desirable to collapse e1, e2, and e3, e4 and e6, to produce the graph shown in Figure 22a:

(ii) Each directed cycle (as in Figure 22a), should be absorbed into a single box, because invalidation of one box of the cycle invalidates all boxes it meshes with, no matter whether or not the conditions are restorable.

We may apply the two collapsing procedures (i) and (ii) exhaustively; then this particular way of imposing a box structure on a net which is endowed with restorable conditions can be called recovery collapsing. In fact is equivalent to the technique described in [MR77], section 5. Figure 22b shows the recovery-collapsed graph for Figure 21:

A B
\begin{figure}[h]
\centering
\begin{tikzpicture}
\node [draw,rectangle,minimum size=1cm] (A) at (0,0) {A};
\node [draw,rectangle,minimum size=1cm] (B) at (2,0) {B};
\node [draw,rectangle,minimum size=1cm] (C) at (2,-2) {C};
\draw [->] (A) -- node [above] {$b_1/b_2$} (B);
\draw [->] (B) -- node [above] {$b_6$} (C);
\draw [->] (B) -- node [below] {$b_5$} (C);
\end{tikzpicture}
\caption{Figure 22b}
\end{figure}
The shape of the recovery-collapsed graph is independent of the order in which steps (i) and (ii) are carried out; it only depends on the shape of the basic graph and on the way in which restorable and active conditions are distributed in the basic graph. Thus we may say that recovery collapsing defines a unique recovery level. For instance, no matter how (i) and (ii) are applied to the basic graph shown in Figure 21, the graph shown in Figure 22b results as its recovery-collapsed version. One of the possible corresponding trees is shown in Figure 22c:
This tree results from recovery—collapsing the basic graph by means of the "chase protocols"; they work as follows: upon detection of an erroneous condition, the failure is propagated through the boxes (starting with the input event of the invalid condition) in all forward directions, and, stopping only where a restorable input condition of an invalid box is detected, in backward directions as well. The main appeal of this scheme lies in its "distributed" nature: boxes may propagate the failure independently of one another.

Thus, the chase protocol algorithm has the property of performing recovery collapsing at the latest possible time, namely when an error occurs. It is conceivable that recovery collapsing takes place at earlier times, the earliest being as soon as the graph grows. In this case the tree of our example would look slightly differently below the recovery level:

As already mentioned, however, the shape of the recovery level itself is not affected. The fact that there is some freedom to choose the time at which recovery collapsing is actually performed, leaves some scope for optimisation. We now turn to the question whether "chasing" can be avoided.
On the recovery level, each box has only restorable conditions leading into it and active or restorable conditions leading out of it (not counting loops). The recovery level contains the "units of recovery" in the following sense: if an active condition is found in error, then not only its input event but even its input box at the recovery level becomes completely invalidated, and the input conditions of this box are restored. In Figure 22b, for example, the nearest recovery line for b6 is the (proper) input condition of B, that is, \{b1, b2\}. In general, the input conditions of a box form the "present" recovery line for the box and its output conditions.

Thus, if an active condition is found to be erroneous, at least its input box at the recovery level has to be invalidated. In general, however, a larger portion of the graph than that covered by this box has to be invalidated. For example, consider Figure 22b; not only B but also b5, c7 and b7 have to be invalidated. In other words, chasing is necessary at the recovery level. In this sense, the recovery level does not contain the "units of chasing". We examine the situation more closely.

In our basic graph (see Figure 21), three types of restorable conditions can be distinguished:

- conditions which can never belong to a recovery line, such as b3 and b4; such conditions are called irrelevant (a term used in [MK77]);
- conditions which can never be invalidated, such as b1 and b2 (such conditions may be called certain); here, as throughout this section, we assume that a restorable conditions cannot be invalidated other than by "chasing";
- all others, that is, b5 in the example (these conditions may be be called uncertain).
Uncertain conditions may or may not belong to a recovery line; for example, if b\(^7\) is found to be erroneous, b\(^5\) forms its recovery line. On the other hand, b\(^5\) is invalidated if an error is detected at b\(^6\).

We have:

**Theorem** - A restorable condition is irrelevant if and only if it is absorbed at the recovery level.
- A restorable condition is uncertain if and only if it is contained in the recovery level but would be absorbed by recovery collapsing if all active conditions were joined to a common output event.

Thus in Figure 21, b\(^3\) and b\(^4\) are absorbed at the recovery level; b\(^5\) becomes absorbed at the (imaginary) recovery level if b\(^6\) and b\(^7\) were joined to an (imaginary) event.

It is easy to see that no chasing is necessary at the recovery level provided it does not contain any uncertain conditions. This can be guaranteed by a simple requirement:

**Theorem** - If, at the recovery level, the output conditions of a box are either all active or all restorable, then all restorable conditions are certain or irrelevant.

The graph shown in Figure 22b does not meet this requirement. For an illustration, consider the following recovery level:

![Diagram](image)

*Figure 23*
This graph meets the requirement stated in the last theorem; b1 and b2 are active, indicating that B covers two concurrent "strands" of activity which are currently active. Suppose that one of these strands, say b2, wishes to establish a checkpoint; if it succeeds, we have a situation similar to the one depicted in Figure 22b, hence a violation of the requirement of the last theorem, leading to an uncertain condition. Thus b2 must be synchronised with b1 upon establishing a checkpoint; b2 has to wait until both are ready to do so.

Such synchronisation requirements are typical for fault tolerance schemes not providing for "chasing". The scheme described in [HAN75] requires a "conversation" (that is, in effect, a box on the recovery level) to have a single well-defined point of exit. In [GRA78], "two-phase commit protocols" are described which serve essentially the same synchronisation purpose.

If, for two certain conditions, b1 < b2, then recovery will always stop at b2. Thus, b1 might as well be discarded (under the assumption that no external influence destroys b2). This makes "chase-free" recovery conceptually quite simple, albeit at a perhaps considerable performance cost due to synchronisation as above.

To sum up: applying our structuring concepts to a simple model, namely occurrence graphs endowed with restorable conditions, we have been able to pinpoint and solve a number of problems relating to state restoration. This underlines, if anything, the suitability of the concepts as well as the model.
7.2 State restoration in a structured graph

Complications arise when the user is also allowed to impose his boxes on the same net. The two ways of structuring may well be incompatible; in a simple case of a user-specified box A

![Diagram of a box with connections](image)

Figure 24

recovery collapsing, if backing up through A, would like to stop right in the middle of A (this kind of situation can be described as the belated recognition of an unplanned atomic action within the current planned atomic action).

On the other hand, user-imposed boxes may support recovery purposes if certain criteria of non-overlapping are satisfied. For example, the recovery block language feature ([RAN75] and [AK76]) serves to preclude situations such as depicted in Figure 24: checkpoints are established only before a recovery block is entered. A detailed discussion of these concepts falls outside the scope of the present paper.

Ideally, if the user structures his program in such a way that his atomic actions correspond precisely to the units of recovery, then the shape of the recovery level can be derived from user specifications. Consequently, no elaborate recovery collapsing needs to be carried out; rather the recovery algorithm will be able to utilise the mechanism which actually observes user specifications, that is, the mechanism implementing atomicity.
8) **Discussion**

First, we wish to discuss whether our understanding of atomicity is adequate. Three issues need consideration:

(a) We have stated that the atomicity criterion ought to be context-dependent. In effect, we have defined an activity \( A \) to appear atomic at some level if, at that level, no other atomic activity occurs after the start of \( A \) but before the end of \( A \). This definition is evidently reasonable, and is also in accordance with more informal definitions known to us (which can be found in a number of the references below).

(b) We have argued that the atomicity criterion ought to be level-dependent. In a way, this is most obvious: if we consider \( A \) in detail, we are not considering \( A \) to be atomic; only if we forget about \( A \)'s details, can we consider \( A \) atomic. There we have two levels of abstraction. We have extended the argument by exhibiting cases in which whether or not \( A \) can be considered atomic depends on how detailed a view is taken of \( A \)'s neighbourhood.

(c) We have defined the notion of atomicity purely in terms of the "earlier" - relation between (basic) activities, not in terms of, for instance, data dependencies. As a basis for discussion, consider the following example, taken from [ESW76]: if the command

\[
S_1 \begin{cases} 
    a := a + 100; \\
    b := b + 100
\end{cases} \quad S_2 \begin{cases} 
    a := 2a; \\
    b := 2b
\end{cases}
\]

is invoked, and both \( S_1 \) and \( S_2 \) are specified to be atomic, then the meshing
may yield an undesired result. S1 and S2 are mutually data-dependent, which is why corresponding reads/writes are linked by cross-arrows.

Not all arrows, however, need to be due to data-dependency. At any rate, the concept of variables depending upon one another can be "considered harmful" (compare [COH77]: we may contrast S1, S2 with the following example:

\[ S1': a = 2 \times b \mod 2 + a \mod 2 \quad // \quad S2': b = 2 \times a \mod 2 + b \mod 2 \]

S1' and S2' could also be called mutually data-dependent because they operate on the same variable set. However, it can easily be verified that no meshing whatsoever of S1' and S2' produces the wrong result, because, in effect, S1' and S2' are not data-dependent; we realise that S1' and S2' read the last bit of a and b, respectively, but change only the rest of a and b.

In our definition of atomicity, we are not, nor need to be, worried about how the "earlier" - relations arise, be it through data dependencies or otherwise; and so we claim to have gained a more precise, general and adequate understanding. However we do not deny the importance of clarifying the concepts of data dependency and, for that matter, of information flow, if only to be able to call the cross-arrows of our first example (S1, S2) "relevant" compared with the "irrelevant" arrows of our second example (S1', S2').
Turning from the atomicity criterion, to implementation strategies, we do not wish to elaborate on the contents of sections 5 and 6. These strategies are on the whole concerned with avoiding from the outset all or some of the atomicity violating cycles of section 4.4. We might venture the idea of another strategy, namely one of coping with cycles.

In section 2, we had an example of a shared list to which damage was done by the atomicity violating schedule of Figure 2c (see Figure 2d). But this damage is not completely irreparable. Take a state of the list shown in Figure 2d, un-do rem (a) of Figure 2c/3c (i.e. reset a.next to its previous value, namely "a", and reset b.before to "a" as well), and afterwards re-do rem (a) (i.e. x1:=b // y1:=e; b.next:=c // e.before:=b). Then the correct result (the empty list) is obtained. In fact, the execution is just equivalent to the one of Figures 2b/3b. Similarly, if run (b) is un-done and re-done, we get an execution equivalent to Figures 2a/3a.

Hence, if un-doing and re-doing of activities is provided for, we might imagine a mechanism which is inactive unless and until a violation of atomicity is detected. Then, it selects a box of the cycle which breaks the cycle (and therefore has to be a non-contraction), un-does and re-does it and thus corrects the damage due to the cycle, as well as gets rid of the cycle. "Chasing" may also be necessary. The mechanism might be implemented as a centralised algorithm which is asked by on-going activities which have completed their original task whether they can finish without closing a cycle.

This idea is only mentioned because it arises naturally from our discussion; whether it can be refined so as to be of any use, we leave open. Evidently, violations of atomicity will occur more frequently than, say, deadlocks, and so, prevention is probably much better than cure. Prevention, i.e. enforcing "long locking" (see the beginning of section 6) should be especially beneficial if activities are likely to be short. On the other hand, the latter scheme does make use of the fact that if, say, N activities are involved in an atomicity violation, that is, our mechanism detects a cycle of length N, only one or at best a few of them need to be un-done and re-done.

45
Therefore, the advantages of this scheme seem to grow with the average activity running time.

More generally, the idea emphasises anew that the concepts of "atomicity" and "error recovery" enjoy a close relationship. While here we have argued that the provision of a backing-up facility of activities may ease the implementation of atomicity, in section 7.2 we have argued that, conversely, activity backup may be facilitated by the presence of an atomicity implementation. We feel that this relationship deserves a great deal further exploration.

Acknowledgements

The authors would like to express thanks to their colleagues in the Newcastle reliability project for their support. The debt owed to Phil Merlin who spent the summer 1977 with the project is evident; he also carefully read a draft of the paper and suggested many improvements. We would also like to thank Judy Hindson and Julie Lennox for the skill with which they produced neatly typed versions of marginally legible original drafts. Finally, we express our gratitude to the Science Research Council of Great Britain for sponsoring the reported work.
References


