A Formal Model of Atomicity in Asynchronous Systems

Eike Best and Brian Randell
Computing Laboratory, University of Newcastle upon Tyne, England

Summary. We propose a generalisation of occurrence graphs as a formal model of computational structure. The model is used to define the “atomic occurrence” of a program, to characterise “interference freeness” between programs, and to model error recovery in a decentralised system.

Contents

1. Introduction ................................................. 93
2. A Model of Computational Structure ............... 96
   2.1. Occurrence Graphs .............................. 96
   2.2. Collapsing of Subgraphs .................... 99
   2.3. Structured Occurrence Graphs ............ 102
3. Atomicity of Activities .............................. 106
   3.1. Cycle-Freeness and Serialisability ....... 106
   3.2. Interference-Freeness and Atomic Occurrences 107
   3.3. Context-Independence and Inherently Atomic Occurrences 110
   3.4. Two-Phase Occurrences ..................... 112
4. A Model of Recoverability in Decentralised Systems 114
   4.1. Recovery Points and Recovery Graphs .... 114
   4.2. Error Propagation and Recovery Collapsing 115
   4.3. Classification of Recovery Points ......... 119
5. Concluding Remarks and Notes on Further Work .......... 120
References .................................................... 123

1. Introduction

Atomic actions have long been recognised as an important programming concept. The ideas behind atomic actions can be traced back at least to Floyd’s seminal paper [5] in which he proposed to characterise programs by their input/output relations. Our own interest in atomic actions stems from the awareness [18] that they could help in generalising the concept of a recovery
block [17] for concurrent programs. We use the term in the same way as in [11]: by intuitive definition, an atomic action is a piece of program that enjoys the status of a simple “primitive” with regard to its environment, while it may however possess a “complicated” internal structure.

The concept of an atomic action is related to the so-called inductive assertion method for proving properties of programs (see for example [12]). In this technique an assertion can be proved invariant by showing that it holds initially and that its validity is preserved over successive portions of a program. This technique, which has been well-established for sequential programs, can be generalised to concurrent programs using atomic actions: to establish the invariance of an assertion one has to prove that it holds initially and that its truth is preserved by every action. Examples of correctness proofs for concurrent programs using this method include [1, 2] and [3].

Another area of application is that of shared database systems where atomic actions are usually called “transactions” [4, 7, 19]. The invariant property to be preserved in a database is usually called its “consistency”. Transactions typically are short sequential user programs which individually preserve database consistency; one of the tasks of a database manager is to ensure that they occur atomically so that the consistency of the database is maintained at all relevant times.

In this paper we base our discussions on the use of a simple programming language consisting of the usual constructs (assignment, etc.) augmented by the parallel operator || and the angular bracket facility to express atomic actions [3]. The semantics of the parallel operator partly depends on the semantics of the atomic actions used in connection with it, but could intuitively be expressed as “execute as much as possible in parallel”. For example, the program

$$\langle s := s + 1 \rangle \parallel \langle s := s + 1 \rangle$$

is taken to indicate the (possibly partly parallel) execution of two “primitive” incrementations of s, so that its net effect should be to increase s by 2 rather than by any other value (i.e. the result should be $s = s_0 + 2$ where $s_0$ is the initial value of s).

We consider atomic actions in the first place as a programming tool rather than “a hardware feature” or “a synchronisation method”. As such, atomic actions serve to build new “primitives” out of given “primitives”. It therefore stands to reason that atomic actions be allowed to be the sub-actions of larger atomic actions [18], although such nesting of atomic actions has sometimes been forbidden [3].

For instance, the program

$$\langle\langle s := s + 1 \rangle \parallel \langle s := s + 1 \rangle \parallel \langle s := 2 * s \rangle$$

differs from the program

$$(\langle s := s + 1 \rangle \parallel \langle s := s + 1 \rangle) \parallel \langle s := 2 * s \rangle$$

in that the set of possible results is \{s = 2 * s_0 + 2, s = 2 * s_0 + 4\} for the former and \{s = 2 * s_0 + 2, s = 2 * s_0 + 3, s = 2 * s_0 + 4\} for the latter. Thus as opposed to,
say, [4] where the atomicity of straight sequences of primitives is considered at only one level of nesting, we shall in the sequel not put any restrictions on either the depth of nesting of atomic actions or the degree of their internal concurrency. It will be shown that this generality introduces non-negligible complications.

The salient properties of atomic actions have frequently been characterised by what is known as the “serialisability” property [4], expressed in [11] as follows: “Actions are atomic if they can be considered, as far as their environment is concerned, to be indivisible and instantaneous, such that the effect on the system is as if they were interleaved as opposed to concurrent”. In [3] we find: “We require all accesses to shared variables to be part of an atomic action and postulate that the net effect of our concurrently operating processes is as if atomic actions are mutually exclusive, i.e. the execution periods of atomic actions don’t overlap.” And in [2] we read: “Atomic actions ... can be implemented by ensuring between their executions mutual exclusion in time.”

These characterisations raise a number of questions. Firstly, atomicity which on the face of it is a “local” property of a single action is being expressed by serialisability which is a “global” property of a set of actions. In this paper we shall deal with this question by defining and comparing a variety of “local” and “global” atomicity criteria. Secondly, mutual exclusion seems but one, in general unnecessarily restrictive, way of implementing atomicity; in a shared data base, for instance, the proposition that all transactions be mutually excluded is unacceptable [7]. We shall in this paper regard any method which ensures the truth of our atomicity criteria as a valid implementation of atomicity. Thirdly, the “as if” clause in two of the above characterisations tends to blur the distinction as to whether atomicity is a static or a dynamic property. In this paper, by taking executions of atomic actions as our basic formal objects, we shall give purely dynamic atomicity criteria.

The formal model of computation we are using is that of occurrence graphs [8]. The present paper is in fact a direct successor of [14] where the use of occurrence graphs for the purpose of characterising atomicity has first been suggested. An occurrence graph serves to model a single computation as a set of interdependent events. Event dependencies may arise through usage of data (as in all of our examples), but other types of dependencies such as producer-consumer relationships can be modelled as well. Atomicity is represented by “collapsing” parts of an occurrence graph, i.e. by reducing a subgraph to a single event. In order to accommodate graph collapsing we generalise the occurrence graph model to that of “structured occurrence graphs”.

The first part of the paper, comprising Sect. 2 and 3, is organised as follows. The occurrence graph model is introduced in Sect. 2.1 and Sect. 2.2 shows how atomicity can be represented by graph collapsing. This leads to “structured occurrence graphs”, defined in Sect. 2.3. Using structured occurrence graphs we derive a “global” atomicity criterion (in essence a generalisation of serialisability, Sect. 3.1), a “local” atomicity criterion (interference-freeness, Sect. 3.2) and a “context-independent” atomicity criterion (Sect. 3.3) which we consider to generalise the notion of “two-phase executions” [4] (Sect. 3.4). A series of propositions establishes the precise relationship between these atomicity criteria.
In the second part of this paper (Sect. 4) we examine the topic of error recovery in a decentralised system, such as, for example, a multi-process message passing system. Our starting point is the assumption that during the activity of such a system recovery points have been established which may function as local fall-back points, so enabling the system to revert to an earlier valid state. We deal with two questions arising in this situation: (a) in the event of the detection of an error, which are the relevant “recovery lines”, i.e. sets of recovery points to which the system can revert; and (b) following the detection of an error, which parts of the surrounding activity must be treated as suspect, in the sense of possibly being prey to the same error.

We shall show that the structured occurrence graph model can be adapted to deal with these questions. In Sect. 4.1 we extend the occurrence graph model to that of “recovery graphs” providing for the representation of recovery points. In Sect. 4.2 we define the “units of recovery” as those events which describe the activity “find the appropriate recovery line” atomically, and we show that this defines a structured occurrence graph. In Sect. 4.3 we examine in particular the question of whether or not a recovery point can be invalidated as a consequence of an error detected in a different part of the system, which leads to a classification of different types of recovery points.

We conclude this paper by putting our atomicity criteria into perspective (Sect. 5). In particular, we discuss the connection between what we have called “interference” in Sect. 3.2 and other related properties, such as data dependency and information flow. We also outline different possible implementation strategies for atomic actions.

2. A Model of Computational Structure

2.1. Occurrence Graphs

We define an occurrence graph as a pair \( G = (E, B) \), where \( E \) is a non-empty set and \( B \subseteq E \times E \) is a (possibly empty) relation over \( E \). We use occurrence graphs to describe computations, whereby the elements \( ee \in E \) are interpreted as the events of the computation and the elements \( b \in B \) are interpreted as “conditions” holding between events and indicating an ordering of events as described below. Pictorially, events are represented as squares and conditions are represented as arrows between squares, as for example in

![Diagram of occurrence graph](image)

We define \( e = \text{tail}(b) \) and \( e' = \text{head}(b) \) iff \( b = (e, e') \). We call a sequence \((e_0, \ldots, e_n)\), \( n \geq 1 \), of events a (directed) path from \( e_0 \) to \( e_n \) iff \( e_i \mathrel{B} e_{i+1} \) for \( 0 \leq i < n \). A path is called a cycle iff \( e_0 = e_n \). A path is called “simple” iff its constituent events, except possibly the two endpoints \( e_0 \) and \( e_n \), are distinct. We write \( e < e' \) (“\( e \) before \( e' \)”) iff \( eB^+ e' \), i.e. iff there is a path from \( e \) to \( e' \). We write \( e \leq e' \) iff \( e < e' \) or \( e = e' \), and \( e \) and \( e' \) are said to be “concurrent” iff neither
\( e \prec e' \) nor \( e' \prec e \). In Fig. 1, for example, \( e_1 \prec e_1, e_2 \prec e_3 \) but \( e_3 \) and \( e_4 \) are concurrent.

\[ \text{Fig. 1} \]

We call an occurrence graph acyclic iff for no \( e \in E, e \prec e' \); i.e. iff it does not contain any cycles\(^1\). The structure \((E, \prec)\) derived from an acyclic occurrence graph is a partial ordering of events. In contradistinction to [8] we allow (for the moment) occurrence graphs to be cyclic because we wish to attach a particular significance to cycles.

We next define the notion of an “immediate predecessor”. Care must be taken in this definition because our graphs may be cyclic. Usually [15] an event \( e \) is called an “immediate predecessor” of another event \( e' \) if \( e \prec e' \) but for no \( e'' \), \( e \prec e'' \prec e' \). By this definition, in Fig. 1 \( e_1 \) is not an immediate predecessor of \( e_3 \), nor is \( e_2 \) an immediate predecessor of \( e_4 \). This is contrary to our subsequent intentions; we therefore define immediate predecessors slightly differently.

We call a path from an event \( e \) to an event \( e' \) a “(proper) extension” of another path from \( e \) to \( e' \) iff the former contains the same events in the same order as the latter, and besides also at least one other event. In Fig. 1, for example, \((e_1, e_2, e_1, e_3)\) is a proper extension of \((e_1, e_3)\). We call a path “max-

\(^1\) There is a close connection between the occurrence graph model and the occurrence net model as defined in [6]. Occurrence nets are triples \((B, E, F)\) where \( B \) and \( E \) are disjoint non-empty sets (with the same interpretation as above) and \( F \subseteq E \times B \cup E \times B \) is a relation, such that

\[
\forall b \in B: |F(b)| \leq 1 \quad \text{and} \quad |F^{-1}(b)| \leq 1, \quad \text{and}
\]

\[
F^+ \quad \text{is irreflexive.}
\]

Every acyclic occurrence graph \((E, B)\) in which \( B \neq \emptyset \) can be considered an occurrence net

\[
(B, E, \{(e, b), (b, e')\} \mid b \in B \text{ and } e = \text{head}(b) \text{ and } e' = \text{tail}(b))
\]

satisfying the somewhat stronger property

\[
\forall b \in B: \ |F(b)| = |F^{-1}(b)| = 1.
\]

Conversely, every occurrence net satisfying \((1')\) can be considered an acyclic occurrence graph

\[
(E, \{(e, e') \mid e, e' \in E \text{ and } e F^2 e'\}).
\]

We use occurrence graphs rather than occurrence nets because they permit the collapsing operation to be described more easily.
imal" iff it cannot be properly extended. In Fig. 1, for example, both $(e_1, e_2, e_1, e_3)$ and $(e_2, e_1, e_2, e_4)$ are maximal paths.

We require all occurrence graphs under consideration to satisfy the property that every path can be extended to a maximal path. This is a discreteness property which we subsequently refer to as the "maximality axiom". It is always satisfied for finite graphs.

We finally define $e \in E$ to be an "immediate predecessor" of $e' \in E$ (or $e'$ an "immediate successor" of $e$), and write $e < e'$, iff there exists a maximal path $(e_0, \ldots, e_n)$ in which $e$ and $e'$ are neighbours, i.e. $e = e_i$ and $e' = e_{i+1}$ for some $i \in \{0, \ldots, n-1\}$. Note that by this definition, $e_1$ is an immediate predecessor of $e_3$ in Fig. 1 because they are neighbours in the maximal path $(e_1, e_2, e_1, e_3)$; similarly, $e_2$ is an immediate predecessor of $e_4$.

As an example of the use of occurrence graphs to describe computations, we consider a simple program operating on a doubly linked list. Let the list consist of a "start" pointer, an "end" pointer and two proper elements "x" and "y" in the following current state:

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

Fig. 2

The algorithm for removing a list element pointed to by a pointer $p$ can make use of concurrency in the following way:

$$rem(p): p_1 := p.\text{before} \parallel p_2 := p.\text{next};$$

$$p_1.\text{next} := p_2 \parallel p_2.\text{before} := p_1,$$

where $p_1$ and $p_2$ are local pointer variables.

Every execution of $rem$ thus consists of executions of its four constituent assignments in some valid order, the validity of the order being determined by the semantics of the semicolon and the $\parallel$ operator. With the abbreviations

$$e_1 = \text{execution of } p_1 := p.\text{before},$$

$$e_2 = \text{execution of } p_2 := p.\text{next},$$

$$e_3 = \text{execution of } p_1.\text{next} := p_2,$$

$$e_4 = \text{execution of } p_2.\text{before} := p_1,$$

the following two occurrence graphs describe two possible (valid) executions of $rem$:

---

The following lemma shows that this definition agrees, for acyclic graphs, with the usual one referred to above.

**Lemma 1.** (i) $e < e'$ and $\exists e'' : e < e' < e''$ implies $e < e'$.

(ii) For acyclic graphs, the converse of (i) also holds.
The two executions differ in that in the first one, the pairs of events \((e_1, e_2)\) and \((e_3, e_4)\) both occur concurrently whereas in the second one, \(e_1, \ldots, e_4\) occur in linear order. The first execution, in fact, is maximally concurrent in the sense that any more concurrent execution would no longer be a valid execution of the program.

2.2. Collapsing of Subgraphs

We represent programmer-defined atomic actions by “collapsing” the subgraphs corresponding to their executions into single events, thus giving them an “instantaneous” appearance. Through the collapsing operation we obtain from a given occurrence graph a new one which describes the same computation on a different level of abstraction. Before defining the collapsing operation formally, we give an example of its use.

Let us consider two simultaneous executions of \(rem\),

\[ rem(x) \parallel rem(y) \]

of the two elements \(x\) and \(y\) of the list as shown in Fig. 2, where at first, \(rem\) is not specified as atomic. As in the previous section, let \(e_1^x, \ldots, e_4^x\) and \(e_1^y, \ldots, e_4^y\) denote the executions, respectively, of the four assignments in \(rem(x)\) and \(rem(y)\). The following Figure shows three possible executions of the simultaneous removal, all of which can be shown to be maximally concurrent:

\[ \text{Fig. 3} \]
As in these three executions the $e^x_i$ and $e^y_i$ occur in different order, they may denote different actual assignments (for instance, $e^x_4$ denotes the setting of "y.before" to "start" in the first execution, while it denotes the setting of "end.before" to start" in the second execution). Nevertheless, as can easily be verified, the first two executions both have the same overall effect of producing the empty list whereas, by contrast, the third execution leaves the list in the following state:

Let us now consider the parallel program

\[
\text{at-rem}(x)\parallel\text{at-rem}(y).
\]

with the "remove" operation specified as an atomic action:

\[
\text{at-rem}(p): \quad \langle p_1 := p.\text{before} \parallel p_2 := p.\text{next}; \\
p_1.\text{next} := p_2 \parallel p_2.\text{before} := p_1 \rangle.
\]

This means that the simultaneous removal is to have the effect of two proper individual removals, so that of the possible orderings of events shown in Fig. 4, only the first two are to be allowed.

We represent the programmer's atomicity specifications in the following manner:
When this collapsing is applied to the occurrence graphs shown in Fig. 4, the following can respectively be obtained:

\[ \text{a} \quad \text{at-rem}(x) \rightarrow \text{at-rem}(y) \]

\[ \text{b} \quad \text{at-rem}(x) \rightarrow \text{at-rem}(y) \]

\[ \text{c} \quad \text{at-rem}(x) \rightarrow \text{at-rem}(y) \]

Fig. 7

Figure 7 shows how the invocations of \textit{at-rem}(x) and \textit{at-rem}(y) are related to each other when they are seen as atomic events. In the first two executions, these events occur in strict order, although some of their constituent events occur concurrently (cf. Fig. 4). By contrast, the invocations "interfere" with each other in the third execution, this interference being indicated by the cycle in Fig. 7c. The difference between valid and invalid executions thus manifests itself by the absence or presence, respectively, of a cycle in the collapsed graphs.

We now define the collapsing operation formally. Let an occurrence graph \( G=(E,B) \) and a non-empty subset \( E' \subseteq E \) be given. We define the subgraph \( A \) generated by \( E' \) as the set \( E' \) together with all arrows that have both endpoints in \( E' \). Formally,

\[ A=(E',B') \quad \text{where} \quad B' = \{ b \in B | \text{tail}(b) \in E' \land \text{head}(b) \in E' \} . \]

We also denote the set of events \( E' \) generating the subgraph \( A \) by \( \hat{A} \).

As \( A \) is again an occurrence graph, all the definitions relating to occurrence graphs can be transferred to subgraphs; in particular, a "before" relationship \( <_A = B'^+ \) can be defined for \( A \) which may not coincide with \( = B^+ \) on \( A \). Note also that \( A \) may be disconnected and/or that \( B' \) may be empty. We usually enclose the set \( E' \) of events in question in a rectangle. For example, in

\[ \text{Fig. 8} \]

\( A \) is disconnected, \( B' \) is empty and \( e_1 < e_2 \) in \( G \) but not \( e_1 <_A e_2 \) in \( A \).
We define the “collapsing” of \( A \) as the construction of a new graph \( G[A] \) from \( G \) such that \( A \) is replaced by a single new event and all arrows leading into and out of \( A \) are replaced by arrows ending and starting, respectively, with the new event. We assume the new event to be uniquely named and call it “\( A \)” for the purpose of this definition. Formally, \( G[A] = (E[A], B[A]) \) where

\[
E[A] = (E \setminus \hat{A}) \cup \{A\}
\]

\[
B[A] = \{(e, e') \in B \mid e \notin \hat{A} \land e' \notin \hat{A}\} \cup
\{(e, A) \mid e \notin \hat{A} \land \exists a \in \hat{A}: (e, a) \in B\} \cup
\{(A, e) \mid e \notin \hat{A} \land \exists a \in \hat{A}: (a, e) \in B\}.
\]

In the remainder of this section we present two simple facts about the collapsing operation. The first one indicates that collapsing does not tear the graph apart, in the sense that paths leading into and out of a subgraph \( A \) in \( G \) change into paths ending and starting with \( A \), respectively.

**Lemma 2.**

\[\exists a \in \hat{A}: e < a \text{ in } G \iff e < A \text{ in } G[A]\]

\[\exists a \in \hat{A}: a < e \text{ in } G \iff A < e \text{ in } G[A].\]

The proof of Lemma 2 follows immediately from the definition of the collapsing operation.

Our next lemma shows that the order of collapsing two disjoint subgraphs is immaterial. We call two subgraphs \( A \) and \( A' \) of \( G \) disjoint iff \( \hat{A} \cap \hat{A'} = \emptyset \).

**Lemma 3.** If \( A \) and \( A' \) are disjoint subgraphs of \( G \) then \( A' \) is a subgraph of \( G[A] \), \( A \) is a subgraph of \( G[A'] \), and \( G[A][A'] = G[A'][A] \).

An example is furnished by Figs. 4 and 7 above.

### 2.3. Structured Occurrence Graphs

In the previous section, we have shown how the dynamic structure arising from programmer-defined atomic actions can be represented by collapsing the subgraphs corresponding to their executions. Generalising this in the present section, we use “structured occurrence graphs” to model the (dynamic) nesting of atomic actions to arbitrary depth. We define structured occurrence graphs to consist of a “basic occurrence graph” on which a “nested structure” is imposed.

The basic occurrence graph describes the computation to such a degree of detail that its events can be decreed basic without further justification; it may be helpful to think of them as being “system-defined”. As we shall take cycle-freeness as our “basicness” criterion, we postulate that the basic graph be acyclic; any cycle in the basic graph would indicate an event being its own cause.

The nested structure imposed on the basic graph captures the dynamic structure arising from the programmer’s atomicity specifications. At the most
basic level, the computation is seen to consist of "small" basic events, interconnected as described by the basic graph. At the most abstract level, it can be viewed as a single event comprising all of its constituent activities (as if the entire user program was enclosed by outermost atomicity brackets). Depending on the depth of nesting, there may be a variety of intermediate levels of abstraction. In keeping with [18] we consider the (dynamic) overlapping of atomic actions as contrary to their nature, so that we are considering "tree structures" only.

Formally, let an acyclic occurrence graph \( G = (E, B) \) be given which we refer to as the "basic graph", \( E \) being the set of "basic events". We define a "tree structure over \( G \)" to be a finite collection \( T \) of sets of events such that

\[
\begin{align*}
(T1) &\quad E \in T \text{ and } \{e\} \in T \text{ for all } e \in E \\
(T2) &\quad \forall E_1, E_2 \in T: E_1 \cap E_2 = \emptyset \lor E_1 \subseteq E_2 \lor E_2 \subseteq E_1.
\end{align*}
\]

The sets in \( T \), which we call "(atomic) activities", model the executions of atomic actions. (T1) is motivated by the above remarks concerning the most abstract and the most basic level, while (T2) ensures the absence of overlapping.

For instance, with \( E_1 = \{e_1^x, \ldots, e_4^x\} \), \( E_2 = \{e_1^y, \ldots, e_4^y\} \) and \( E = E_1 \cup E_2 \),

\[
T = \{\{e_1^x\}, \ldots, \{e_4^y\}, E_1, E_2, E\}
\]

is a tree structure over the occurrence graph shown in Fig.4c. We represent \( T \) pictorially by enclosing its constituent sets in rectangles:

![Fig. 9](image)

This structuring represents the programmer's specification of \( \text{rem} \) as an atomic action and an implicit outermost atomic action.

We call a pair \((G, T)\) where \( G \) is an acyclic occurrence graph and \( T \) is a tree structure over \( G \), a "structured occurrence graph", and we define its structure tree as follows. The nodes of the tree are the activities in \( T \), and a node \( E' \) is called a "parent" of another node \( E'' \) iff \( E' \) is the smallest superset of \( E'' \) in \( T \).
As a consequence of (T1) and (T2), there is a unique smallest superset for all sets in \( T \) except \( E \), and the “parent” relationship therefore defines a tree with root \( E \) and leaves \( \{ e \}, e \in E \). For \( E' \in R \) we define the set of “sub-activities” of \( E' \):

\[
\hat{E}' = \{ E'' \in T \mid E' \text{ is parent of } E'' \}.
\]

Our next aim is to capture the notion of a structured occurrence graph describing a computation at different levels of abstraction. To this end we define levels of abstraction formally and then associate an occurrence graph with each level. Such a graph describes how the events of this level are related to each other, generalising the remarks made following Fig. 7 in the previous section.

For a given structured occurrence graph \((G, T)\), we call a subset \( L \subseteq T \) a “level (of abstraction)” iff

\[
(L1) \quad E = \bigcup \{ E' \mid E' \in L \}
\]

\[
(L2) \quad \forall E_1, E_2 \in L: E_1 \cap E_2 = \emptyset \lor E_1 = E_2.
\]

\(L1\) requires that all basic events are considered and \(L2\) requires that none of them is considered more than once. Levels can be visualised as “cuts” through the structure tree. For our example shown in Fig. 9 we derive the following structure tree and five levels of abstraction \( L_0, \ldots, L_4 \) (where for simplicity the leaves of the tree are labelled with the names of the basic events they represent):

![Fig. 10](image)

We further define:

- the “basic level” \( L_0 = \{ \{ e \} \mid e \in E \} \),
- the “most abstract level” \( L_{\text{top}} = \{ E \} \), and,
- for any \( E' \in T \) the level \( L_{E'} = \{ E' \} \cup \{ \{ e \} \mid e \in E \setminus E' \} \)

containing \( E' \) and all basic events outside \( E' \).

We define \( L \subset L \) for two levels \( L \) and \( L \) iff

\[
L = (L \setminus \{ E' \}) \cup \hat{E}' \quad \text{for some } E' \in T,
\]
i.e. iff \( L \) arises from \( L \) by substituting the sub-activities of \( E' \) for \( E \). We also write \( L = [E']L \) in this case; for instance, \( L_1 = [E_2]L_3 \) in Figure 10. We call \( L \) "more abstract" than \( L \) iff \( L \subseteq L \), where \( \subseteq \) is the transitive closure of \( \preceq \). The \( \subseteq \) relationship turns the set of levels into a lattice with \( L_0 \) as the minimal element and \( L_{\text{top}} \) as the maximal element. For our example we have the following lattice:

![Lattice Diagram](image)

Fig. 11  \( L_0 \)

Finally we define the occurrence graph associated with a level \( L \) by induction over the lattice of levels as follows.

(O1) The graph associated with \( L_0 \) is the basic graph.

(O2) Whenever \( L = [E']L \), \( G' \) is the graph associated with \( L \) and \( A \) is the sub-graph of \( G' \) generated by \( E' \), then the graph associated with \( L \) is \( G'[A] \). We give the new event of \( G[A'] \) the name "\( E' \)", so as to make step (O2) repeatedly applicable; the events of the graph associated with \( L \) are thus just the activities in \( L \).

As a consequence of Lemma 3 which shows that the order of collapsing disjoint subgraphs is immaterial and the requirement that all activities be non-overlapping, (O1) and (O2) properly define an occurrence graph for each level. For our example we obtain the following five level graphs:

![Occurrence Graphs](image)

Fig. 12
Due to the association of an occurrence graph with every level, all concepts defined for occurrence graphs (the \( < \) relationship, for example) now become level-dependent. In the sequel we use the phrase “at level \( L \)” in order to avoid confusion about which level is meant.

If the convention of regarding basic events as trivial subgraphs is introduced, a one-to-one relationship between activities \( E' \in T \) and subgraphs (generated by \( \hat{E} \) if \( E' \) is non-basic) can be established. We therefore use the term “activity” for subgraphs \( A_1, A_2, \ldots \) as well and extend all definitions accordingly. In particular, \( L = [A]L \) means that \( A \) is a subgraph at \( L \) which is collapsed at \( L' \); \( L_A \) denotes the level containing \( A \) and all basic events outside \( A \); and an activity \( A \) is said to “contain” another activity \( A' \) iff \( A' \) is a descendant of \( A \) in the structure tree.

We use structured occurrence graphs in Sect. 3 for the purpose of characterising atomic occurrences and in Sect. 4 for the purpose of describing the units of recovery in a decentralised system.

3. Atomicity of Activities

3.1. Cycle-Freeness and Serialisability

As exemplified in the previous section, we characterise atomicity dynamically by the absence of interference. Naturally, interference pertains not to activities in isolation but to the way in which they are related to each other. Consequently in our characterisation, which can be found in Sects. 3.1 and 3.2, we take into account the computation as a whole.

We take the characteristic (dynamic) property of atomicity to be that events are partially ordered on all levels of abstraction induced by atomicity specifications (not just the basic level). Thus we define a structured occurrence graph and the computation it describes to “satisfy atomicity” iff all of its level graphs are acyclic.

This definition generalises the “serialisability” criterion [4]. Under some very weak conditions [10] which are assumed to hold, every partial order can be “serialised” (i.e. extended to a linear order). Therefore for each acyclic occurrence graph \( G = (E, B) \) a graph \( G_{\text{lin}} = (E, B_{\text{lin}}) \) can be found such that \( B^+ \subseteq B_{\text{lin}}^+ \) and \( E \) is linearly ordered under \( B_{\text{lin}} \). More generally we have:

**Proposition 1.** A structured occurrence graph \( (G, T) \) satisfies atomicity if and only if the basic graph \( G \) can be serialised such that the resulting structured occurrence graph \( (G_{\text{lin}}, T) \) describes a linear order on all levels.

**Proof.** Assuming that \( (G, T) \) satisfies atomicity, we may serialise the basic events by processing the structure tree in the following way. Starting with the root of the tree we arrange all subactivities of non-basic activities in linear order, which is possible by assumption. This process stops when all basic events have been reached. Eventually all level graphs describe a linear order.

Conversely, assume that \( (G, T) \) does not satisfy atomicity. Then there exists a cycle at some level, the events of which cannot be serialised.
The term "serialisation" is perhaps misleading in that it may suggest that atomicity can only be implemented by actual strict sequencing (i.e. mutual exclusion in time) of the atomic actions of a program. This is not true according to our criterion which allows for the parallel execution of independent atomic actions. Even if atomic actions fail to be independent a partly concurrent execution does not necessarily violate atomicity, as demonstrated previously.

On the other hand, it may be suggested that strict sequencing always implements atomicity. However, programs such as the following cannot be serialised, i.e. are not implementable:

\[(x, y) = (0, 0);\]
\[\langle x := 1; \text{do } y = 0 \rightarrow \text{skip od} \rangle \parallel \langle y := 1; \text{do } x = 0 \rightarrow \text{skip od} \rangle.\]

Typically, in such programs the successful termination of one atomic action depends on the progress of others in a cyclic manner. To prohibit this, it seems reasonable to postulate that atomic actions always terminate (this is indeed one of the key axioms in [16]).

3.2. Interference-Freeness and Atomic Occurrences

In this section we take a closer look at "interference". We define an event \(e\) to interfere with an activity \(A\) if it occurs strictly after part of \(A\) and strictly before another part of \(A\). We define \(A\) to "occur atomically" if it is not interfered with in this fashion.

For example, in Fig. 13 we have \(e_2^x < A_2 < e_4^x\) at \(L_2\), which means that the event \(A_2\) interferes with \(A_1\) (though none of the constituent basic events of \(A_2\) does!):

This can also be characterised by the fact that at \(L_3\), \(A_1\) and \(A_2\) stand in a cyclic relationship which disappears when \(A_1\) is opened, i.e. at \(L_2\); we take this as the basis for our formal definition below. We first remark that an activity
may be in a cycle even though (intuitively) it is an atomic occurrence, and illustrate this point with an example.

We consider the program

$$\langle s := s + 1 \rangle \parallel \langle s := s + 1 \rangle,$$

assuming each assignment to consist of an event "r" of reading the value of s followed by an event "w" of overwriting s with the value of s + 1. Consider the following atomicity-violating (and hence invalid) execution:

![Diagram of program execution](image)

Fig. 14

giving rise to the following four level graphs:

![Graphs illustrating execution](image)

Fig. 15

Intuition suggests that $A_1$, though contained in a cycle at $L_3$, occurs atomically because it is not interfered with by other activity.

Generalising these examples, we define inductively that for a given structured occurrence graph $(G, T)$,

(A1) Basic events occur atomically.

(A2) An activity $A$ occurs atomically iff

(a) $\forall a \notin \hat{A}$: a occurs atomically, and

(b) for all levels $L$, whenever $e < A < e$ at $L$ then $\exists a \in \hat{A}$: $e < a < e$ at $[A]L$.

This definition signifies that if (A2b) is violated for some level $L$ and event $e$ then $e$ is one of the outside activities that interfere with $A$, making it non-atomic.
(A1)-(A2) define the atomic occurrence of a single activity and can therefore be called a "local" atomicity criterion, in contrast to the "global" cycle-freeness criterion defined in the preceding section. These two criteria are inter-related as follows:

**Proposition 2.** (i) If for no level either \( A \) or one of the activities it contains is involved in a cycle then \( A \) occurs atomically.

(ii) Let \( e < e \) at \( L \); then \( \exists A: e \leq A \leq e \) at \( L \) and \( A \) does not occur atomically.

Proposition 2(ii) is a weak converse of (i); as the example shown in Figs. 14/15 demonstrates, the immediate converse of (i) does not necessarily hold true. We also have the following immediate consequence of proposition 2:

**Corollary.** A structured occurrence graph satisfies atomicity if and only if all of its activities occur atomically.

*Proof. (i) If neither \( A \) nor any of the activities contained in it is involved in a cycle then (A2b) cannot be violated for \( A \).

(ii) Let \( e < e \) at \( L \).

Because of the maximality axiom, there exists a maximal simple cycle (\( e = A_0, ..., A_n = e \)) at \( L \).

Suppose that all of the \( A_i \) occur atomically.

This means that there exist \( a_i \in A_i \) such that \( a_0 < ... < a_n \) and \( a_0 = a_n \) at \([A_0] ... [A_{n-1}]L\).

Again we choose a maximal simple cycle \( (a_0, ..., a_n) \) which must consist of sub-activities of the \( A_i \) only (otherwise \( (A_0, ..., A_n) \) would not itself be maximal).

This argument is thus repeatable and leads to a contradiction because of the cycle-freeness of the basic graph.

Hence for some \( i, A_i \) does not occur atomically, q.e.d.

We finally show that our definition remains intuitively valid also in a more complicated example than considered so far.

![Diagram](image-url)
In this example, $A_1$ occurs atomically according to our definition (A1)–(A2) even though it is contained in four cycles at $L_3$, one of which (namely the one not including $e_1$ or $e_2$) disappears when $A_1$ is opened, i.e. at $L_2$. It is however perfectly in line with intuition that $A_1$ should be defined to occur atomically, while $A_2$ is clearly a non-atomic occurrence, being interfered with by $A_1$ and even by all basic events in $A_1$.

3.3. Context-Independence and Inherently Atomic Occurrences

As characterised in the preceding sections, the atomic occurrence or otherwise of an activity depends not only on its internal structure but also on its environment at large. We now show that there is a sense in which an activity can be called “atomic” by virtue of its internal structure only (i.e. without regard to the entire computation). We call activities with this property “inherently atomic occurrences” or, for short, “contractions”.

We consider the contraction property to generalise the notion of a “two-phase” execution [4]. A two-phase execution consists of a “growing phase” followed by a “shrinking phase”, whereby the conceptual moment of occurrence of the activity lies between the two phases [11]. Our claim will be substantiated in the next section where it will be shown that an activity is a con-
traction iff it contains an "internal state" which can be thought of as the moment of its occurrence. In this section we define contractions and exhibit their relation to atomic occurrences.

Our definition can be motivated as follows. Every (maximal) cycle through $A$ must also pass through an immediate predecessor of $A$ and an immediate successor of $A$. If $A$ is so structured that from every immediate predecessor of $A$ a path leads through $A$ to every immediate successor of $A$ then the opening of $A$ can never break that cycle. Accordingly, we define that in a structured occurrence graph $(G, T)$,

(C1) Basic events are contractions.

(C2) An activity $A$ is a contraction iff
(a) $\forall a \in A: a$ is a contraction, and
(b) whenever $e_1 < A < e_2$ at $L_A$ then $\exists a \in A: e_1 < a < e_2$ at $[A] L_A$.

In (C2b) we consider only the level $L_A$ as defined in Sect. 2.3. However in the proof of Proposition 3 below it will become apparent that if (C2b) holds for $L_A$ then it holds for all other levels as well.

As an example, consider $A_1$ in Fig. 18:

At $L_1 = L_{A_1}$ we have $r_2 < A_1 < w_2$ and at $L_0 = [A_1] L_1$, $r_2 < w_1 < w_2$. Hence $A_1$ is a contraction. Its collapsing can be thought of as "contracting" it into $w_1$ – hence the name. By contrast, $w_1 < A_2 < w_1$ at $L_2$ but $w_1 < w_1$ at $L_0$; hence $A_2$ is not a contraction.

With the definition (C1)-(C2) we have the following:

**Proposition 3.** (i) Contractions occur atomically.

(ii) If $A$ is not a contraction then a structure $T'$ can be defined containing the same subtree rooted at $A$ as is contained in $T$, such that $A$ does not occur atomically in $(G, T')$.

Proposition 3(ii) is again a weak converse of (i), signifying not that non-contractions occur non-atomically, but that based just on the internal structure of a non-contraction, nothing can be inferred about its atomic occurrence.
Proof. (i) Let \( e < A < e \) at \( L \).

Because of the maximality axiom, there exists a simple cycle \((e, \ldots, A_1, A, A_2, \ldots, e)\) such that \( A_1 < A \) and \( A < A_2 \) at \( L \).

Because \( A_1 < A \) at \( L \), by repeated applications of Lemma 2 one sees that \( A_1 \) must contain a basic event \( d_1 \) such that \( d_1 < A \) at \( L_A \).

Again because of the maximality axiom, there exists a basic event \( e_1 \) with \( d_1 \leq e_1 < A \), which is also contained in \( A_1 \) (otherwise \( A_1 \) would not immediately precede \( A \) at \( L \)).

Similarly, \( A_2 \) contains a basic event \( e_2 \) such that \( A < e_2 \) at \( L_A \).

Property (C2b) for \( A \) requires the existence of an \( a \in A \) such that \( e_1 < a < e_2 \) at \( [A]L_A \).

For this \( a \) we also have, again by Lemma 2: \( e \leq A_1 < a < A_2 \leq e \) at \( [A]L_A \).

(ii) Since \( A \) is not a contraction there exist basic events \( e_1, e_2 \) outside \( A \) such that for no \( a \in \bar{A}, e_1 < a < e_2 \) at \( [A]L_A \).

We define \( T' \) as containing \( \{e\} \) for all \( e \in E \), \( E \), the entire subtree rooted at \( A \) and the set \( \{e_1, e_2\} \).

\( T' \) is a tree structure and \( A \) does not occur atomically in \((G, T')\).

In practical terms, Proposition 3 signifies the following. Suppose that a programmer wishes to use a set of system synchronisation primitives to ensure the atomic occurrence of his program, but cannot rely on any system-provided implementation of atomicity. Then he must ensure the contraction property (for example by employing the simple two-phase protocol [4], or by using more knowledge about the system to derive more complicated and efficient protocols [20]), in order to prevent unwanted interference which may arise due to the lack of system-provided safeguards. In this way, Proposition 3 can be seen as a generalisation of the result contained in [4].

3.4. Two-Phase Occurrences

We have seen that in the structured occurrence graph shown in Fig. 18, \( A_1 \) is a contraction and the event \( w_1 \) can be thought of as the conceptual moment of the occurrence of \( A_1 \). We now show that it is characteristic for a contraction to contain a "state" which can be thought of as the moment of its occurrence. The following example serves to illustrate this point:

![Figure 19](image-url)
In this example, \( A_1 \) is a contraction while \( A_2 \) is not. The broken line through \( A_1 \) represents a "cut" with the property that from every immediate predecessor of \( A_1 \) (\( e_1 \) or \( e_2 \)) to every immediate successor of \( A_1 \) (\( e_3 \) or \( e_4 \)) there is a path which crosses this cut. No cut with this property can be found for \( A_2 \). Following [6] we interpret cuts as "states" of an activity and we go on to show that an activity \( A \) is a contraction iff it contains a cut with the property just mentioned.

To define states formally, let an occurrence graph \( G = (E, B) \) and a subgraph \( A = (E', B') \) of \( G \) be given. We first define

\[
B^> := \{ b \in B \mid \text{tail}(b) \not\in \hat{A} \land \text{head}(b) \in \hat{A} \}
\]

(the set of arrows leading into \( A \)), and

\[
< B := \{ b \in B \mid \text{tail}(b) \in \hat{A} \land \text{head}(b) \not\in \hat{A} \}
\]

(the set of arrows leading out of \( A \)).

\( B^> \) and \( < B \) can be considered the interface between \( A \) and its environment. The relation \( _A \leq B^+ \) (see Sect. 2.2) can be extended in a natural way to elements of the set

\[
X = \hat{A} \cup B^' \cup B^> \cup < B
\]

if \( x, x' \in X \), define \( x \leq_A x' \) iff a directed path inside \( A \) leads from \( x \) to \( x' \). Two elements \( x, x' \in X \) are said to be "\( A \)-concurrent" iff neither \( x <_A x' \) nor \( x' <_A x \).

We call a subset \( C \subseteq X \) an "\( A \)-state" iff its elements are pairwise \( A \)-concurrent and it is a maximal set with this property (for instance, the \( A_1 \)-state shown in Fig. 19 is \( C = \{b_3, b_4, b_5, a_2\} \).

We are now ready to state:

**Proposition 4.** Condition (C2b) in the definition of a contraction (Sect. 3.3) can be equivalently replaced by:

(C2b') There exists an \( A \)-state \( C \) at \([A]L_A\) such that whenever \( e_1 < A < e_2 \) at \( L_A \) then \( \exists c \in C: e_1 < c < e_2 \) at \([A]L_A\).

**Proof.** (C2b') implies (C2b):

Let an \( A \)-state \( C \) be given and let \( e_1 < A < e_2 \) at \( L_A \) and \( e_1 < c < e_2 \) at \([A]L_A\) with \( c \in C \).

Because \( C \) is a subset of \( X \) and hence contains only elements in \( A \) or bordering on \( A \), one of the following must hold:

- either \( ce \in \hat{A} \), in which case (C2b) is satisfied with \( a = c \);
- or \( ce \in B \) and \( \text{head}(c) \in \hat{A} \), in which case \( e_1 < \text{head}(c) < e_2 \);
- or \( ce \in B \) and \( \text{tail}(c) \in \hat{A} \), in which case \( e_1 < \text{tail}(c) < e_2 \).

Conversely, (C2b) implies (C2b'):

Because every path from \( b_1 \in B^> \) to \( b_2 \in B^> \) must include \( \text{tail}(b_2) \not\in \hat{A} \), the elements of \( B^> \) are pairwise \( A \)-concurrent.

We define \( C_0 \) as the first \( A \)-state including \( B^> \); formally,

\[
C_0 = \{ x \in X \mid \forall b \in B^> : x \text{ is } A \text{-concurrent to } b \}
\]

and

\[
\neg \exists x' \in X : x' <_A x
\]

In the example shown in Fig. 19, \( C_0 = B^> = \{b_1, b_2, b_3\} \).

In the example shown in Fig. 19, \( C_0 = B^> = \{b_1, b_2, b_3\} \).
The elements of \( C_0 \) are pairwise \( A \)-concurrent by definition, and \( C_0 \) is maximal because no \( x'' \in X \) concurrent to all elements of \( C_0 \) can have an \( A \)-predecessor \( x' <_A x'' \) in \( X \).

We show that \( C_0 \) satisfies the requirements of (C2b').

Let \( e_1 < A < e_2 \) at \( L_A \).

Because \( A \) is a contraction,

\[ \exists a \in \hat{A}: e_1 < a < e_2 \quad \text{at} \quad [A]L_A. \]

Every path from \( e_1 \) to \( a \) must contain a pair of neighbours \((e_i, e_{i+1})\) with \( e_i \notin \hat{A} \)

and \( e_{i+1} \in \hat{A} \).

Hence \( c = (e_i, e_{i+1}) \in C_0 \) and \( e_1 < c < e_2 \) at \([A]L_A\), q.e.d.

The \( A \)-state \( C \) which exists by (C2b') can be thought of as a "moment of occurrence" of \( A \). \( C \) is by no means unique; in the proof of Proposition 4, the set \( C_1 \) defined as the last \( A \)-state including \( \triangleleft B \) would have done a similar service as \( C_0 \). \( C_0 \) and \( C_1 \) are in fact the "first" and "last" \( A \)-states, respectively, which satisfy the property required in (C2b').

Thus, in general, the occurrence of a contraction \( A \) can be viewed as consisting of the occurrences of its immediate predecessors, \( C_0 \), all intermediate \( A \)-states, \( C_1 \), and its immediate successors, in that order. In other words, \( A \) occurs quasi-sequentially, again illustrating the context-independence of its atomic occurrence.

4. A Model of Recoverability in Decentralised Systems

4.1. Recovery Points and Recovery Graphs

In this section we outline how the framework set up in section 2 can be adapted to model error recovery in a decentralised system. We assume that in a given system, as part of a strategy for providing a degree of fault tolerance, certain states have been checked (for correctness or a similar property) and saved, so that the system can fall back on these states if need be; such states are called "restorable states" [14] or "recovery points" [18]. As before, we represent the activity of the system by an occurrence graph, indicating restorable states by a special type of condition.

A decentralised system may at any time contain several independent active components, any one or more of which may independently discover an error and invoke recovery, so that in each case the problem arises as to which recovery points should be chosen to fall back on. This is one of the questions to be dealt with in the remainder of this paper. We also extend the occurrence graph model slightly to indicate those active states, again by introducing special conditions.

We use the term "recovery graph" to denote an occurrence graph which may contain (besides normal conditions) also restorable conditions and active conditions. Pictorially, we represent restorable conditions by double arrows and active conditions by arrows "dangling" from an event (waiting to be connected to another event):
This example represents a computation having started at $e_1$, having established the recovery points $b_1 - b_5$ and at present consisting of two strands of activity at $b_6$ and $b_7$.

By the asterisk we indicate the fact that an error has just been detected in the active component represented by $b_6$. It stands to reason that in this situation all of the activity having sprung from $e_3$ must be abandoned, because no proper restorable state lies between $e_3$ and $b_6$. Hence the nearest fall-back line (or "recovery line") in this example is the set of recovery points $\{b_1, b_3\}$, all later activity being suspect; we may represent actual recovery in the model by erasing the subgraph generated by $e_3 - e_7$ and making $b_1$ and $b_2$ the new active conditions from which further activity can spring [14].

In the remainder of this section we will discuss the following two questions. In case an error is detected in an active component of a system described by a ("basic") recovery graph, (a) which is the nearest recovery line, and (b) which portion of the graph should be treated as suspect as a consequence of the error detection? Question (a) is dealt with in Sect. 4.2 while question (b) is addressed in Sect. 4.3.

### 4.2. Error Propagation and Recovery Collapsing

Our aim is to determine the nearest recovery line in the event of an error being detected (which we represent by marking the corresponding active condition as "invalid"). We first define the propagation of error information by considering the three types of conditions separately:

**R1**

\[
\begin{array}{c}
\text{e} \\
\text{b}
\end{array}
\]

If the active condition $b$ is invalidated (representing, for instance, the failing of an acceptance test [17]) then its input event $e$ should also be invalidated as a consequence.

**R2**

\[
\begin{array}{c}
e \\
\text{e}'
\end{array}
\]

Suppose $e$ and $e'$ are connected via a non-restorable condition. The invalidation of $e'$ should entail the invalidation of $e$ ("backward error propagation")
and the invalidation of $e$ should entail the invalidation of $e'$ ("forward error propagation" or "chasing" [14]).

(R3)

Here only forward error propagation is possible: the invalidation of $e$ should be propagated to $e'$ but not vice versa, because the recovery point between $e$ and $e'$ functions as a local fall-back.

Rules (R1)–(R3) can be thought of as giving the meaning of the different types of conditions introduced in Sect. 4.1 in the context of error recovery. We have:

**Lemma 4.** In a recovery graph, an event $e'$ becomes invalidated as a consequence of another event $e$ being invalidated if and only if there is an undirected path between $e$ and $e'$ in which all restorable conditions point towards $e'$.

Example:

![Diagram of a recovery graph](image)

Repeated applications of (R2) and (R3) show that, indeed, the invalidation of $e$ eventually spreads to $e'$.

In particular, if a recovery graph contains a cycle then an invalidation of any one event of the cycle entails the invalidation of all events of the cycle. Generalising, we define a "unit of recovery" to be a maximal set of events with the property that the invalidation of any one of the events in this set is propagated to all of the events in the set. In our example (Fig. 20) there are three units of recovery, $E_1$, $E_2$ and $E_3$ as indicated in the next Figure:

![Diagram of units of recovery](image)

As we can see, the units of recovery afford a disjoint covering of the basic graph. This is true in general, as the next lemma shows:

**Lemma 5.** For any two units of recovery $E_1$ and $E_2$, either $E_1 \cap E_2 = \emptyset$ or $E_1 = E_2$. 
Proof. Suppose $e \in E_1 \cap E_2$.

Because both $E_1$ and $E_2$ are units of recovery, the invalidation of $e$ entails the invalidation of both $E_1$ and $E_2$ and the maximality property implies $E_1 = E_2$.

Lemma 5 implies that, in the terms of Sect. 2.3, the set containing all basic activities $\{e\}$, all units of recovery, and the set $E$, forms a tree structure over the basic recovery graph, in which the set of all units of recovery determines a level of abstraction. We call this the "recovery level". For our example we obtain:

![Diagram](image)

Fig. 23

The graph associated with the recovery level (which we call the "recovery collapsed graph") can be defined as in (O1)–(O2) of Sect. 2.3, except that provisions have to be made for restorable and active conditions. Because as a consequence of our definition, all conditions leading into or coming out of a unit of recovery must be either restorable or active, it is reasonable to introduce the rule that these properties are retained in the recovery collapsed graph. For our example (compare Fig. 22) we therefore obtain:

![Diagram](image)

Fig. 24

Recovery collapsing can alternatively be described by the following two rules:

(R4) Collapse all parts

of a recovery graph into a single event, whereby ordinary conditions have precedence over restorable conditions (this means that, say,
(R5) Collapse all cycles into single events.

We then have:

**Proposition 5.** A set of events is a unit of recovery if and only if it can be reduced to a single event by repeated and exhaustive applications of (R4) and (R5),

which can be proved by applying Lemma 4. Proposition 5 indicates that exhaustive application of (R4) and (R5) to the basic recovery graph will be an automatic way of producing the recovery collapsed graph.

From our definitions it follows that the invalidation of an active condition is propagated at least throughout the unit of recovery it is attached to, i.e. its input event at the recovery level, but is not propagated to the input conditions of that unit of recovery. Thus we can define for every active condition its “(nearest) recovery line” to consist of the input conditions of its input event at the recovery level.

In our example, the recovery line of \( b_6 \) is \( \{b_1, b_2\} \) and the recovery line of \( b_7 \) is \( \{b_5\} \):

The recovery collapsed graph thus helps to answer the question posed at the beginning of this section. The events of the recovery collapsed graph can also be interpreted as describing the activity “find the nearest recovery line” as single atomic events. Note however that upon invalidation of an active condition, possibly more than just the unit of recovery it is attached to have to be invalidated, as exemplified above (Fig. 25) where the invalidation of \( b_6 \) is propagated not only to \( e_3 - e_6 \) but also to \( e_7 \). We examine this situation more closely in the next section.
4.3. Classification of Recovery Points

Precisely which portion of a recovery graph becomes invalidated as a result of an active condition being invalidated is of course entirely determined by rules (R1)–(R3) of the previous section. The “chase protocols” described in [14] give a practical means of computing this portion of the graph. As has been demonstrated, this portion may in general comprise more than a single unit of recovery. In our example this was due to the existence of the restorable condition $b_5$ which, depending on the location of the detection of an error, in one case (if the error is detected at $b_7$) functions as a recovery line and in another case (if the error is detected at $b_6$) as a useless recovery point (compare Fig. 26). The existence of such restorable conditions is a consequence of the asymmetry in rule (R3). In this section we characterise such conditions and derive a criterion for their absence.

![Fig. 26](image)

In a recovery graph we distinguish three types of restorable conditions (compare Fig. 26):

- conditions which can never belong to a recovery line, such as $b_3$ and $b_4$ in our example; such conditions are called “irrelevant” [14];
- conditions which can never be invalidated as a consequence of rules (R1)–(R3), such as $b_1$ and $b_2$ in our example; such conditions may be called “certain”;
- all others, such as $b_5$ in our example; these conditions may be called “uncertain”.

We can characterise these types of conditions as follows:

**Proposition 6.** 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 Fig. 26, $b_3$ and $b_4$ are absorbed at the recovery level and $b_5$ becomes absorbed at the (imaginary) recovery level if $b_6$ and $b_7$ were joined to an (imaginary) event.

If the recovery graph does not contain any uncertain conditions then the units of recovery are precisely those portions of the graph that have to be
invalidated as a result of an error detection; hence, in this case the recovery collapsed graph describes as a single atomic event not only the activity "detect the nearest recovery line" but also the activity "determine the suspect environment of the error".

The absence of uncertain conditions can be guaranteed by the simple requirement that at the recovery level, the output conditions of an event are either all active or all restorable. In order to meet this requirement, two (or more) components which are actively engaged in a single unit of recovery would have to be synchronised upon establishing a recovery point: either all components establish a new recovery point, or none does. Such synchronisation requirements are typical for fault tolerance schemes providing constraints on the forward propagation of errors. The scheme described in [17], for instance, requires a "conversation" (that is, in effect, a unit of recovery) to have a single well-defined point of exit. [9] and [7] describe "two-phase commit protocols" (not to be confused with the two-phase protocols of [4]) which serve a similar synchronisation purpose.

Finally, we consider "certain" conditions. If, for any two certain conditions, \( b_1 < b_2 \), then recovery will always stop at \( b_2 \). Thus, under the assumption that no influence other than rule (R3) destroys \( b_2 \), \( b_1 \) might as well be discarded. This assumption, therefore, makes "chasefree" recovery conceptually quite simple, albeit at a perhaps considerable performance cost due to synchronisation as above.

Things become considerably more complicated when the programmer is allowed the nested use of recovery regions. Amongst other things, the assumption mentioned in the last paragraph may not hold in this case. These complications are sorted out in [21] which contains the design of a protocol to determine recovery point "safety", which is a property much akin to what has been defined as recovery point "irrelevance" above.

5. Concluding Remarks and Notes on Further Work

In this paper we have introduced a formal model of computational structure, that of structured occurrence graphs (Sect. 2). We have also investigated the use of structured occurrence graphs; firstly, in Sect. 3, as a conceptual tool to characterise interference-freeness and atomic occurrences. Secondly, in Sect. 4 we have used structured occurrence graphs as a means of modelling error recovery in a decentralised system. Here we have concentrated on precise characterisations of the notions of a "unit of recovery" and a "recovery line" when the activity of the system is described by a recovery graph.

Hopefully, our model of recovery could also be used as a practical means for achieving a degree of fault tolerance in a decentralised system. The idea would be to keep an occurrence graph in store (in some form) as a record of the history of the system, to be processed in the way described in Sect. 4 either as a consequence of an error being detected, or prior to that as a precautionary measure.
The “chase protocols” described in [14] can be seen as a first step towards making this scheme a practical one. In [13] a design is described which refines the “chase protocol” strategy, and a number of protocols are also given for discovering and deleting recovery points which are irrelevant in the sense described in Sect. 4.3. In [21] a variety of optimisation techniques are applied to obtain a practical implementation of recovery collapsing for a system of communicating processes, under the complicating assumption that processes may be involved in nested recovery regions and may declare their commitment to their respective recovery points unilaterally.

As defined in this paper, the structured occurrence graph model has been based on the unexplained notion of dependency between events. Some tacit assumptions have been made about event dependency, for example that it is a transitive relationship. In the remainder of this section we discuss some issues related to event dependency and the semantics of atomic actions which in our opinion need further exploration.

We first observe that the notion of event dependency we are interested in does not necessarily coincide with what might be called an intuitive and simple notion of “information flow”. This can be seen by considering a pair of actions which overwrite (but do not read) a common variable. In that case, due to the write-dependencies between the two actions, an example of an atomicity-violating execution can well be constructed, even though there is no exchange of information between the two actions, at least not in the simple and intuitive sense that one action generates the value of a variable which is then read by the other action.

This remark may be related to the following characterisation of atomicity which has been claimed in [11] to be equivalent to the serialisability property: “An action is atomic if the process performing it does not communicate with other processes while executing the action”. Our arguments indicate that the equivalence between this characterisation and serialisability must be questioned, unless “communication” is understood as referring to the same dependency relation as we are interested in.

How then can “event dependency” be positively determined? We think that the concept of maximal concurrency can be employed. In a maximally concurrent execution (such as the first one shown in Fig. 3 and the ones shown in Fig. 4, Sect. 2), all event dependencies are significant; otherwise a more concurrent execution could be found. Thus we would claim that the dependency relation could be characterised by the two statements that (a) in a maximally concurrent execution all event dependencies are significant, and (b) in a given execution, a dependency between two events is significant if it does not disappear if the given execution is transformed into a maximally concurrent one.

Thus we have reduced the notion of dependency to the equally unexplained notion of maximal concurrency. We believe that in order to determine the latter an analysis of the detail of the interaction between the program and its variables is essential. Let us, for instance, consider two 2-bit variables \( x, y \in \{0, 1, 2, 3\} \) and the program

\[
x := 2^x (y \mod 2) + x \mod 2.
\]
A detailed analysis shows that this program, in effect, reads only bit 0 of \( y \) and writes into bit 1 of \( x \) (see Fig. 27).

Let us then consider the parallel program

\[
\langle x := 2^*(y \mod 2) + x \mod 2 \rangle \parallel \langle y := 2^*(x \mod 2) + y \mod 2 \rangle
\]

and let us assume that each program consists of two reads (of \( x \) and \( y \), respectively) followed by a single write. In this case no significant cross-dependencies exist because of the way in which the two programs access the two variables \( x \) and \( y \):

![Fig. 27](image)

Hence in a maximally concurrent execution all dependencies between the two programs would disappear. This example shows that the semantic interaction between the program and its variables may vary considerably from what it appears to be syntactically, entailing difficulties in the determination of maximally concurrent executions.

We believe that the exact relationship between event dependency, maximal concurrency and information flow needs much further exploration. In the occurrence graphs which we use to formulate our atomicity criterion, if we do not assume that all dependencies are significant, then the atomicity criterion may be "too strong" in the sense that an execution which on the face of it does violate atomicity, nevertheless cannot lead to incorrect results, because some of the dependencies of a cycle may be insignificant. However, the other and more important direction of this statement always holds: if an execution does not violate atomicity then it can never lead to an incorrect result (unless, of course, one of the actions has itself been programmed incorrectly).

Finally, we would like to reconsider our example of the doubly linked list and draw attention to the fact that the damage done by the atomicity-violating execution shown in Fig. 4c (leading to the corrupt final state of the list shown in Fig. 5) is not completely irreparable. If, starting with the corrupt final state, either \( \text{rem}(x) \) or \( \text{rem}(y) \) is first "undone" by executing it in reverse order and then "redone", a correct final state can be obtained.

This suggests an alternative implementation of atomicity. Rather than avoiding cycles from the outset one could try to cope with cycles. An “exception mechanism” would be waiting for a violation of atomicity to occur and then take action along the lines described in the last paragraph. This would require the capability to undo an action, as well as (possibly) the “chasing” of corrupt information as described in Sect. 4.3, again highlighting a connection between atomicity and error recovery.
Unfortunately there seem to be difficulties in generalising the undoing and redoing method suggested above. The example of the parallel addition (see the discussion relating to Fig. 14) cannot as easily be analysed as the linked list example. It might be another interesting future task to determine those properties which make the doubly linked list behave so "nicely" in this situation.

Acknowledgements. The authors wish to express their great indebtedness to their colleagues in the Project on Reliability and Integrity of Distributed Systems in Newcastle. In particular, the comments by Pete Lee, Santosh Shrivastava and Graham Wood have led to numerous improvements in this paper. This work was done with the financial support of the Science Research Council of Great Britain.

References


Received December 18, 1978/March 27, 1981