Abstract:

This paper concerns an important aspect of the problem of designing fault-tolerant distributed computing systems. The concepts involved in "backward error recovery", i.e. restoring a system, or some part of a system, to a previous state which it is hoped or believed preceded the occurrence of any existing errors, are formalised and generalised so as to apply to concurrent, e.g. distributed, systems. The formalisation is based on the use of what we term "Occurrence Graphs" to represent the cause-effect relationships that exist between the events that occur when a system is operational, and to indicate existing possibilities for the state restoration. A protocol is presented which could be used in each of the nodes in a distributed computing system in order to provide system recoverability in the face even of multiple faults. This presentation includes a proof of the protocol's correctness.

Consistent State Restoration in Distributed Systems

By

P.M. Merlin and B. Randell

TECHNICAL REPORT SERIES

Series Editor: Dr. B. Shaw

Number 113
October, 1977.

© 1977 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.
MERLIN, Philip M.

Consistent state restoration in distributed systems. [By] Philip M. Merlin and Brian Randell.
Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1977.
(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, No. 113.)

RANDELL, Brian
UNIVERSITY OF NEWCASTLE UPON TYNE. Computing Laboratory.

Suggested classmarks (primary classmark underlined)

Library of Congress:
Dewey (17th):
U.D.C.:

001.64404
681.322.06

Suggested keywords
CONCURRENCY
DISTRIBUTED COMPUTING SYSTEMS
ERROR RECOVERY
FAULT-TOLERANCE
STATE RESTORATION

Abstract

This paper concerns an important aspect of the problem of designing fault-tolerant distributed computing systems. The concepts involved in "backward error recovery", i.e. restoring a system, or some part of a system, to a previous state which it is hoped or believed preceded the occurrence of any existing errors, are formalised and generalised so as to apply to concurrent, e.g. distributed, systems. The formalisation is based on the use of what we term "Occurrence Graphs" to represent the cause-effect relationships that exist between the events that occur when a system is operational, and to indicate existing possibilities for state restoration. A protocol is presented which could be used in each of the nodes in a distributed computing system in order to provide system recoverability in the face even of multiple faults. This presentation includes a proof of the protocol's correctness.
About the authors

Dr. P.M. Merlin is a Lecturer in the Department of Electrical Engineering at the Technion and currently holds an S.R.C. Senior Visiting Fellowship at the Computing Laboratory.

Professor Randell has been Professor of Computing Science in the Computing Laboratory of the University of Newcastle upon Tyne since 1969.
1. **Introduction**

The principle utility of state restoration is in connection with fault tolerance. Fault-tolerant systems incorporate special additional features which attempt to detect internal states whose further processing by the normal algorithms of the system would lead to system failure, and prevent this from happening. We term such internal states **errors** and their causes **faults**. The identification of both errors and faults involves subjective judgement, and the relationship between them can be complex, in that a fault may give rise to many errors, and the determination of which fault caused a given error can be extremely difficult. Fault tolerance depends utterly on the provision of means for error detection, and also involves the problems of deciding what other damage (i.e. other errors) might also now exist in a system, how these errors can be corrected, and what should be done to locate and remove the fault(s) that are presumed to have caused the errors. In fact, removal of faults by, for example, replacement of failing (hardware or software) components may not be necessary for failure-free system operation, as long as the errors caused by such faults can be "recovered from".

One important form of error recovery involves restoring a system, or some part of a system, to a previous state which it is hoped or believed preceded the occurrence of any existing errors, before attempting to continue normal processing. Such **backward error recovery** [RAN77] is illustrated in figure 1, which shows the history of a system which has suffered from a number of state restorations. (Dashed lines represent abandoned activity, dotted lines state restoration.) However, this method of describing, and illustrating, backward error recovery disguises many of the problems that exist in distributed computing systems (or in any system involving concurrent activities) in which the notion of "system state" (and for that matter of "previous") is by no means straightforward.

The present paper gives, in section 2, a formal model of system behaviour which enables a precise definition to be given of state restoration in concurrent computing systems. A protocol is presented in section 3 which could be used by each of the nodes in a geographically distributed system in order to provide
system recoverability in the face even of multiple faults, together with a proof of the correctness of this protocol. Further sections of the paper discuss such matters as the problems of state restoration in the presence of contention for shared resources and the problem of reducing the amount of information about past system activity that has to be maintained for the use of such protocols.
2. **Description of the Model**

In this section we introduce the Occurrence Graph model of the dynamic behaviour of a concurrent system. Such graphs are similar to the Occurrence Nets (also called Causal Nets) described in [PET76,PET77]. The main difference is that Occurrence Graphs are viewed as a dynamic structure which is "generated" as the system that it is modelling executes. The Occurrence Graph also contains certain additional features related specifically to the problem of state restoration.

2.1 **The Occurrence Graph Model**

We introduce the model using an example. Suppose that there exist files F1 and F2 (possibly at different locations) and a terminal T. The terminal requests that copies of the files be sent to a location where they will be merged into a single file F3, which replaces F1. A copy of F3 is also kept at the merging location for possible further use. Figure 2(a) represents the initial state of the system. In this model a condition (indicating a state of, for example, a given data structure, communication line, register, etc.) is represented by a "place", such places being denoted graphically by circles. Place 1 represents the existence of file F1, place 2 represents the existence of file F2 and place 3 represents the fact that the terminal is "ready to send" the requests. (In figure 2 the names F1, F2 and T are given only for convenience and are not part of the formal model.) The first event which takes place is the sending of the requests by the terminal. The result of this event is that the previous condition of the terminal (e.g. "ready to send") does not hold any longer, and that two new conditions, representing the requests to A and B, are created. In the model, the occurrence of an event is denoted by a "bar". The new situation is shown in figure 2(b), where bar 1 represents the event of sending the requests, the input arcs of the bar indicate which conditions were necessary to generate the occurrence of the event (i.e. "caused" the event) and the output arcs point to the conditions resulting from this event. Bar 1 and its associated arcs thus show the cause-effect relationships between the occurrences of conditions 3, 4, and 5.

Assuming that, at this level of abstraction, copies of F1 and F2 can be acquired independently, different bars as shown in figure 2(c) will represent the events of copying the files. These events can occur concurrently and, in
practice, we do not mind the order of their occurrence, or even whether this order is observable. Bar 2 is generated by places 1 and 4 and this event results in the continued existence of the original file F1 (represented by place 6) and the sending of a new copy of F1 (represented by place 7) to the location where F1 and F2 will be merged. Bar 3 takes a similar action with respect to F2. Figure 2(d) shows the entire Occurrence Graph model of the history of the cause-effect relationships between conditions and events for the dynamics of the given example. Bar 4 represents the merging of F1 and F2, and bar 5 the replacement of F1 by F3. The final result is F2 (condition 9), F3 (condition 10) and another copy of F3 (condition 11), which may reside at different, indeed possibly remote, locations.

In this model, we represent each place as influencing the occurrence of no more than a single event. Thus we explicitly represent those conditions which still hold after generating events, e.g. place 6 of figure 2(c) represents the fact that although file F1 (place 1) generates event 2, after the occurrence of this event the file is still available and able to influence further events. A similar relationship exists between event 3 and places 2 and 9. On the other hand, event 5 makes file F1 (place 6) unavailable while, and after, being replaced by F3. Hence the only conditions which may generate new events are those represented by places having no outgoing arcs. Such conditions are called active conditions, and are represented by Active Places; in the graphic representation these are, for convenience, indicated using a black triangle. (Inasmuch as it is appropriate to refer to the instantaneous "global state" of a distributed system, this is what the set of active places represents.)

Notice that the Occurrence Graph does not represent algorithms (either hardware or programs) but rather the actual occurrence of events during execution and the pertinent conditions which actually influence them. The Occurrence Graph model is generated by the progress of the algorithmic execution, and from our point of view, many algorithms may generate the same Occurrence Graph. Depending on the actual timing of events, and, presumably, on the values of input data, a given algorithm may generate a variety of Occurrence Graphs. Moreover, as we may choose to consider different aspects of concurrent execution, the same algorithms may generate different Occurrence Graph models.

For illustration, the previous example can be "seen" as generating the graph of figure 3. As it will be shown in latter sections, the way in which the execution is "seen" largely determines the states to which consistent rollback is permitted.
In the Occurrence Graph model, each event is atomic. All the conditions that are directly influenced by an event are explicitly connected to it by arcs and each condition has at most one incoming arc and at most one outgoing arc. The number of places and bars in a graph is allowed to be infinite. Similarly, there may be bars having an infinite number of incoming and/or outgoing arcs. There may also exist bars without incoming or outgoing arcs, representing, respectively, lack of causes or effects.

2.2 State Restoration

If an error is detected, a previous consistent state of the system should be restored at which it is possible to ignore those events and conditions which originally followed that state. By a "previous consistent state", we mean a state the system might have been in according to the cause-effect relationships between events and conditions, rather than one which had actually existed before. If the restored state is prior to the presumed set of events and conditions (i.e. the fault or faults) which caused the error, then the faults and their consequences can thus be effectively ignored.

State restoration is achieved by choosing the state to be restored, reactivating appropriate non-active conditions, and deactivating appropriate active conditions. The error detection, location of presumed faults, and reactivation and deactivation of conditions are performed by some "external mechanism" which is not considered as part of the system we model; we are only concerned with the effects that such mechanisms may have on the behaviour of the normal system. The main theme of this section is the representation of state restoration using Occurrence Graphs and the characteristics that such restorations should have to maintain consistency in the sense described above.

Restoration of a condition can be achieved by the "external mechanisms" in different ways, e.g. having its original value "checkpointed", recomputing its value from related information provided by other conditions, etc. At the level of abstraction of the Occurrence Graph it only matters whether or not a condition is restorable, regardless of how such restoration can be done. In the Occurrence Graph, a restorable condition is represented by a restorable place, which is graphically denoted by a double circle, as shown in figure 4. We assume that if at a certain point in time a condition is non-restorable it cannot
become restorable later. (In section 4 we generalise to the case in which a restorable condition can become temporarily non-restorable.) Therefore, we assume that a single-circle place cannot become a double-circle place. The opposite is clearly possible, and it is called a commitment [RAN77], such as occurs when a checkpoint is discarded. A commitment has no impact on the regular execution of the system; it may only influence the possible consistent states to which the system can be restored.

The reactivation of a place can be performed, provided that the place is restorable and non-active. When a place is reactivated we want to ignore previous effects due to the place. This is represented in the Occurrence Graph by replacing its outgoing arc by a dashed arc, and marking the bar to which it is connected by that arc with an "x". Such a bar is called an invalid bar - our aim is to make it appear as if the event that it represents had never occurred. We show later how a subgraph including these invalid bars, and also invalid places (to be defined below), can be ignored without causing any inconsistencies.

The "external mechanisms" should be able to deactivate those conditions associated with activities which are to be ignored as a result of a state restoration. In the Occurrence Graph, the deactivation of an active place is represented by removing the black triangle from the place. However, also in this case, since a deactivation is performed by an "external mechanism" it does not correspond to the normal operation of the system and, therefore, the situation of such a place is invalid. Hence, when a deactivation is performed, the place is marked as invalid by an "#".

In addition, any arbitrary sets of bars and nodes can be declared to be invalid by the 'external mechanism' because of errors they are presumed to have caused. Our main goal is to find ways by which invalid places and bars can be ignored without causing any inconsistent behaviour by the system.

A component of an Occurrence Graph is a subgraph having no outgoing arcs of any kind to other subgraphs, and having no ordinary incoming arcs from other subgraphs. Incoming dashed arcs are permitted. Suppose that a component includes neither restorable nor active places. Such a component will never have an active place and therefore it will never be able to generate new bars. Since
there are no outgoing arcs, the occurrence of the events and conditions of the component has no effect on the state of other parts of the system. Furthermore, since all incoming arcs are dashed, all places which are external to the component and which generated bars of the component have been reactivated afterwards. Therefore, with respect to other parts of the system, a component with neither restorable nor active places appears as if it never had occurred. Such a component is called an Ignorable Activity. Ignorable Activities can be freely deleted from the Occurrence Graph.

Suppose figure 4(a) is the Occurrence Graph of figure 2(d) including the marking of a set of restorable places. Since restorable marks cannot be added we assume that they existed initially. Suppose that an error is detected which is presumed to have been caused by the event represented by bar 5. Thus this bar is declared to be invalid. Hence, we have to find a way of producing an Ignorable Activity that includes bar 5. This can be done by deactivating places 11 and 12, reactivating 6, 7 and 8, and committing 11. The resulting Occurrence Graph is shown in figure 4(b), in which places 11 and 12 are invalid because they were deactivated, bar 5 is invalid by declaration as well as because of the reactivation of 6, and bar 4 is invalid because of the reactivation of 7 and 8. The bars 4, 5 and the places 10, 11, 12 form an Ignorable Activity that includes all the invalid elements, thus the system is restored to a state it could have been in; in fact this is the state that was shown in figure 2(c). The Ignorable Activity can now be deleted from the Occurrence Graph.

A Recoverable Activity of an Occurrence Graph is a subgraph having no outgoing arcs of any kind to other subgraphs, and which is such that each incoming arc is either dashed, or ordinary and coming directly from a restorable place. This set of restorable places is called the Recovery Line of the Restorable Activity. If all the places of a Recovery Line are restored, the arcs connecting the Recovery Line to the corresponding Recoverable Activity become dashed and the Recoverable Activity becomes a Component of the Occurrence Graph. Such a component can be turned into an Ignorable Activity by deactivating all active places and committing all restorable ones. Thus, a Restorable Activity is a viable candidate for an Ignorable Activity, and in fact, only Restorable Activities can be converted into Ignorable ones. Moreover all the bars which are invalidated by the reactivation of the Recovery Line, as well as the places which are invalidated by the deactivation of active places, will be included in the Ignorable Activity.
The construction of an Ignorable Activity is as simple as described above only when one can assume that the reactivation of the Recovery Line, the deactivation of active places and the commitment of restorable places can all be done atomically, i.e. when it can be assumed that there are no other changes in the Occurrence Graph while these operations are performed. The more complex case (and more realistic in many practical situations) where such an assumption cannot be made is discussed in section 2.3.

We conclude this subsection by showing the two additional state restorations that can be performed in the system of figure 4(a). If the places 1, 2 and 3 are chosen as a Recovery Line and the rest of the graph is transformed into an Ignorable Activity, the system will be restored to the consistent state shown in figure 2(a). If the entire Occurrence Graph of figure 4(a) is considered a Recoverable Activity which is converted into an Ignorable Activity, then the entire graph will be ignored. Notice that any system can by definition be "restored" to such a consistent (albeit vacuous) state.

2.3 Decentralised State Restoration

State restoration involves the choice of a Recovery Line, the deactivation of each active place and the commitment of each restorable place of the corresponding Recoverable Activity, and the reactivation of each of the places of the Recovery Line. In many concurrent, and in particular distributed, systems it is not efficient or, in practice, even possible to perform all these operations atomically, i.e. assuming that other parts of the graph do not change while the operations are being performed. In such cases, each reactivation, each commitment and each deactivation is performed separately, and should all be co-ordinated in such a way as to ensure that, in spite of the possible changes which may occur in the graph between operations, the state restoration will be properly completed. The constraints that decentralised co-ordination mechanisms must satisfy, in order to perform consistent state restoration are the main concern of this subsection.

We illustrate the type of problems which may arise while performing decentralised state restoration by the following example. Suppose that in the example of figure 4(a) bar 5 is declared invalid, and a state restoration such as was described in the previous subsection is initiated. Assuming that each restoration, each reactivation, and each commitment is performed
independently, a possible intermediate state of the Occurrence Graph is shown in figure 5(a). This corresponds to the situation after the reactivation of place 6 and the deactivation of place 12. In this state, places 7 and 8 form a Recovery Line. To complete the restoration we need to reactivate them, and to deactivate and commit place 11. However, if in the meantime place 8 is committed then it cannot be reactivated and that restoration cannot be completed. Nevertheless, it is still possible to restore the system, albeit to the consistent state defined by the Recovery Line of places 1, 2 and 3. The resulting Occurrence Graph after such restoration is shown in figure 5(b). In this case, place 6 had to be deactivated in spite of the fact that it was reactivated as part of the state restoration. This would not be necessary if, for example, in figure 5(a) place 5 was restorable. In such circumstances it would be possible to restore the state of the system by choosing places 2 and 5 as a Recovery Line.

A recovery Line may be lost not only by commitment of one of its members but also by the generation of new bars. For example, in figure 5(a) a new bar involving places 9 and 11 can be generated, as shown in figure 5(c), in which case places 7 and 8 no longer form a Recovery Line. (Such an occurrence is termed an "interaction commitment" in [RAN77].)

More subtle situations could appear if the reactivated places generate new bars, possibly in conjunction with places which are from a Recovery Activity in the process of restoration and are about to be deactivated. For illustration, in figure 5(a) a bar involving 6 and 11 could be generated resulting in the Occurrence Graph of figure 5(d). In such a case, place 6 again becomes part of the Recovery Line of places 6, 7 and 8, and therefore has to be reactivated again. This problem cannot be ruled out even by assuming that all places of the Restorable Activity are committed and deactivated before reactivating the places of the Recovery Line. In such a case, the Recovery Line might still be lost by commitment of one of its places. Indeed, it would also be possible for an already reactivated place to generate a bar in conjunction with other Restorable Activities which are in the process of restoration because of a completely independent, and possible remote, presumed fault. In both cases, new places would therefore have to be committed and deactivated before the original state restoration is completed.
Since in decentralised systems, there may be no central authority able to observe the entire Occurrence Graph, such an apparently simple task as that of determining a Recovery Line could be impossible, because while observing one part of the graph other parts may change. Section 3 describes a protocol which guarantees consistent state restoration in such a distributed system where arbitrarily many faults can be detected at different times in different parts of the system. The protocol also ensures that not only those elements which are declared invalid because of presumed faults, but also those elements which are invalidated by deactivation or reactivation operations, will ultimately be included in Ignorable Activities.

In fact two different types of recovery can be provided in concurrent systems. **Strong Recovery** guarantees that in spite of any possible stream of failures the total system will eventually arrive at a consistent state. **Weak Recovery** guarantees that each particular failure will be consistently repaired, even in the presence of an arbitrary stream of other possible failures. Assuming a finite number of failures, or bursts of failures with enough time between the bursts to ensure that previous failures are repaired before the next burst, Weak Recovery becomes identical to Strong Recovery. In some computer systems, Strong Recovery is needed in order to provide any meaningful service. However, in many distributed systems Weak Recovery is enough. For illustration consider the world-wide telephone network. It is virtually certain that at all points in time there is at least one faulty line. Thus, the system is never in a full globally consistent state. However, each line can (one hopes) be guaranteed to be repaired in finite time, even in the presence of other related failures (e.g. if a concentrator breaks down). Such a system provides meaningful (though imperfect) service in spite of being continuously in a globally inconsistent state.

Using our model, we can define the two types of recovery as follows:

**Strong Recovery**: It is guaranteed that the Occurrence Graph will eventually arrive at a situation where all invalid places and bars will belong to Ignorable Activities.

**Weak Recovery**: It is guaranteed that each invalid place and bar will eventually belong to an Ignorable Activity.
The type of recovery that is achieved will depend on the characteristics of the system, and on the particular recovery protocol employed. The recovery protocol described in section 3 provides weak recovery for a distributed system.

NOTE: If Strong Recovery is guaranteed or in cases where Strong and Weak Recovery are equivalent there is never any need for the dashed arcs, which can be deleted. The dashed arcs are needed only for some implementations of Weak Recovery, in order to prevent certain unwanted effects which may exist in infinite Occurrence Graphs.

2.4 Operations on Occurrence Graphs

We view the Occurrence Graph as being built up (from an initially empty graph) and modified by the use of a number of atomic operations. These operations are intended to model the activities that could reasonably be expected to be performed atomically in a decentralised computing system, even one involving geographically distributed computers.

Each operation has one or more parameters, identifying existing or new places and/or bars in the Occurrence Graph. In some cases pre-conditions involving these parameters are given - it is assumed that an operation can be performed only if its parameters satisfy these pre-conditions. An infinite number of the operations can be performed; any set of operations can be performed concurrently, providing that they have disjoint parameters.

The various different operations are as follows:

1. **GENERATE** \((P_1, b, P_2)\)

   where \(P_1\) is a subset of the active places, \(b\) is a new bar, and \(P_2\) is a set of new places. The sets \(P_1\) and \(P_2\) can be empty, finite, or infinite. This operation makes the places of \(P_1\) non-active, adds bar \(b\) and active places \(P_2\) to the Occurrence Graph, and adds ordinary arcs from \(P_1\) to \(b\) and from \(b\) to \(P_2\).

2. **COMMIT** \((p)\)

   where \(p\) is a restorable place. This operation makes \(p\) non-restorable.
(3) **DEACTIVATE (p)**

where \( p \) is an active place. This operation makes \( p \) **non-active and invalid.**

(4) **REACTIVATE (p)**

where \( p \) is a non-active restorable place. This operation makes \( p \) active, changes its outgoing ordinary arc to a dashed arc, and makes the bar to which this arc is connected invalid. (It is not possible for a place to have more than one ordinary outgoing arc.)

(5) **INVALIDATE (e)**

where \( e \) is either a bar or a place. This operation makes \( e \) invalid.

Any Occurrence Graph resulting from the use of the above operations will have the following properties:

(1) The graph given by the union of the ordinary and the dashed arcs has no directed loops. (No event can be, directly or indirectly, its own cause!)

(2) Each place has a single incoming arc, which must be ordinary, and though it has at most one ordinary outgoing arc, it might have a number of dashed outgoing arcs. (The bars representing the events that caused initial conditions to hold have, for simplicity, been omitted from earlier figures.)

(3) An active place cannot have ordinary outgoing arcs, but may have dashed outgoing arcs. Such arcs point to events which were caused previously by the place, and which must now be ignored.

(4) There cannot be a directed path of ordinary arcs between the places of a Recovery Line. (Otherwise the places cannot constitute a consistent set of states.)

(5) Any connected component of the graph of ordinary and dashed arcs is a Recoverable Activity having an empty Recovery Line — the "restoration" of such a component merely involves forgetting that it ever existed. (A connected component of a directed graph is a subgraph such that, taking no account of the directions of the arcs, there is a path between any of its elements and there are no connections to other subgraphs.)
3. **A Decentralised Recovery Mechanism - The "Chase Protocols"**

In this section we demonstrate the use of the Occurrence Graph model by discussing and validating a protocol which guarantees consistent state restoration in decentralised systems.

Suppose a system is composed of a finite set of nodes communicating by means of messages through a set of prescribed virtual links connecting them. Such a system could be a packet switching network, a distributed application, or any other system where only message communication is permitted. In such a case, there is no central means of performing atomic state restoration, and a decentralised recovery mechanism is required.

A node may send messages only to the nodes to which it is directly connected by a virtual link. Clearly, we abstract ourselves from the physical links, i.e. the virtual links could be provided by a lower level protocol. Each node can generate messages "spontaneously", or as a result of receiving messages. The dynamics of such a system can be modelled by an Occurrence Graph in which places represent messages and bars the generation of these messages. Copies of messages could be retained for recovery purposes, in which case the corresponding places will be marked as restorable. We assume that each node "remembers" that part of the history (i.e. the Occurrence Graph) which relates to it, and thus that between them the nodes "remember" the structure of the whole history but only the content of those messages explicitly marked as restorable.

3.1 **Description of the protocols**

The system can freely perform operations of type GENERATE and COMMIT, but the operations DEACTIVATE and REACTIVATE are completely controlled by the recovery protocols that are described below. The protocol is independently performed for each bar and for each place. As described later, each protocol can be in either of two states, called the LIVE state and the DEAD state. The protocol for each bar or place can communicate with the protocols for those places or bars having incoming or outgoing arcs to it by sending and receiving a special message called FAIL. For simplicity of presentation, we will say
that a bar or a place performs an action of its protocol (e.g. enters the DEAD state, sends a FAIL message, etc.) meaning by this that the mechanism that implements the protocol for that bar or place performs that action. We will first present the protocol informally, then give a formal definition and validation, followed by a discussion of possible improvements.

Recovery is initiated when a bar or place is declared invalid as a result of a presumed fault. Obviously, such recovery can also be started when other activities are already in progress in other elements of the Occurrence Graph. Initially, when a bar or place is created it is placed in the LIVE state. As described below in further detail, a bar or place declared invalid will become DEAD, and the DEAD state will propagate in all directions through the arcs of the Occurrence Graph by means of FAIL messages. This propagation stops when all the elements of the minimal Recoverable Activity that includes the invalid element are DEAD. The protocols guarantee that each DEAD place is neither active nor restorable, and that all the places which, though not included in the Recoverable Activity, have an ordinary outgoing arc to a bar of that Activity will be reactivated and the arc will be dashed. Thus each invalid element, together with all the DEAD elements caused by it, will become an Ignorable Activity. This guarantees Weak Recovery.

None of the operations performed on the graph (i.e. GENERATE, COMMIT, REACTIVATE, DEACTIVATE, INVALIDATE) can reduce the size of the minimal Recoverable Activity that includes a given element. In fact, the GENERATE, COMMIT and REACTIVATE operations can cause the size of the Activity to increase. Illustrations of this were given in section 2.3, where the size of the minimal Recoverable Activity was shown to increase because a place of its Recovery Line is committed, and because the generation of a new bar nullifies the Recovery Line. There is also the simple possibility of growth by adding bars and places to a Recoverable Activity without changing the Recovery Line.

As mentioned above, the DEAD state is propagated by sending FAIL messages. Whilst this propagation is in progress, the minimal Recoverable Activity can grow because of operations performed on the graph. In such a case the DEAD state will propagate to the new (i.e. larger) minimal Recoverable Activity. Therefore, the propagation of the DEAD state could be "chasing" the growth of the minimal Recoverable Activity. In order to guarantee completion, it has to be assumed that the propagation will catch up the growth. There are
many ways by which this can be guaranteed, such as giving higher priorities to FAIL messages than to ordinary messages, bounding the number of ordinary messages the system can produce, or limiting their rate of production. We consider these mechanisms to be outside the scope of this paper, and we simply assume that the "chasing" will be successfully completed.

The propagation of the DEAD state from invalid elements having disjoint minimal Recoverable Activities is performed independently, and the state restoration of one does not affect the others. If several invalid elements have the same minimal Recoverable Activity, there is no effect on the propagation, except that now the propagation is started concurrently at several elements. If two elements have overlapping minimal Recoverable Activities, at least one of the elements will eventually have a minimal Recoverable Activity which encompasses the union of their minimal Recoverable Activities; ultimately state restoration will be consistently completed for this union. Similar comments apply to the situations which can arise if a minimal Recoverable Activity having invalid elements is enlarged (e.g. by a COMMIT or a GENERATE operation) with a subgraph which already includes invalid elements.

The protocol that each bar and each place executes is the following:

If a LIVE bar is invalidated or if it receives a FAIL message, the bar will be placed in the DEAD state and FAIL messages will be sent to all places having incoming or outgoing arcs to this bar. If a LIVE place is invalidated, if such a place receives a FAIL message from its incoming arc (i.e. the event that caused it), or if it receives a FAIL message from an ordinary outgoing arc (i.e. one of the events that it caused) while being non-restartable, then FAIL messages will be sent by the place through all of its arcs independently of their direction, and the place will be left non-active, non-restartable and set to DEAD. If a LIVE restartable place receives a FAIL message from an ordinary outgoing arc, the place will remain LIVE and will be reactivated. Invalidations of bars or places in the DEAD state, as well as FAIL messages received by such bars or places, are ignored.

The protocols for an arbitrary bar b and place p are summarised in figure 6 using a notation similar to those used in [BOC76] and [MER77]. In
this notation there is a finite state machine for each bar b and for each
place p. Transition T1 is executed atomically by a 'b' machine whenever the
predicate CONDITION1 is satisfied by b, and during the transition ACTION1 is
performed. The transitions of a 'p' machine are executed in a similar way.

We assume that every FAIL message arrives at its destination within
a finite, though arbitrarily long, time after it is sent. We assume also that
FAIL messages, as well as notifications of invalidation, are received sequentially
by the protocols (e.g. by queuing). This ensures that no more than one
transition can be executed at each place at any given time.

3.2 Validation of the Protocols

Given a situation OG1 of an Occurrence Graph, let l be one of its
elements (i.e. a place or a bar). Define M(l) to be the union of the
following sets:

\[ M(l) = \bigcup_{i=0}^{\infty} M_i(l) \]

where \( M_0 = 1 \) and \( M_i(l) \) is the union of the following sets:

1. the elements having an incoming arc in OG1 from some element of \( M_{i-1}(l) \);
2. The bars having an outgoing arc in OG1 to some place of \( M_{i-1}(l) \);
3. the non-recoverable places having ordinary outgoing arcs in OG1 to
   some bar of \( M_{i-1}(l) \).

From (1) above, the subgraph of OG1 given by M(l) has no outgoing arcs. From
(2), all arcs entering into elements of M(l) from elements not included in
M(l) are arcs directed from places (external to M(l)) to bars (of M(l)).

From (3), each of those arcs is a dashed arc or is coming from a recoverable
place. Thus the subgraph given by M(l) is a Recoverable Activity. In fact,
this is the Recoverable Activity with the smallest number of elements which
includes element 1 (i.e. what has previously been termed the minimal Recoverable
Activity for element 1).

The following three theorems validate the protocol. Theorem 1 proves
that if an element is declared invalid, each element of its (possibly growing)
minimal Recoverable Activity will eventually become DEAD. Theorem 2 proves
that when the propagation of the DEAD state reaches an entire minimal Recoverable

16.
Activity this activity is Ignorable, and Theorem 3 shows that if an element becomes DEAD, it can cause neither an element external to the minimal Recoverable Activity to become DEAD, nor the growth of that Activity. These theorems hold regardless of whether restoration activity is in progress which has arisen from the invalidation of just a single element, or from the invalidation of several different elements.

Theorem 1: Given an Occurrence Graph where GENERATE, COMMIT and INVALIDATE can be executed freely and REACTIVATE, DEACTIVATE are executed by the "chase" protocols; let l be an element which is declared invalid, and let OG be any given situation which the Occurrence Graph can be in after l was created. Then, each element of M(1) is either DEAD or will eventually become DEAD.

Proof: Each element that becomes DEAD stays in this state forever. If when l is declared invalid it is LIVE, then it will become DEAD.

Given that l = M o (l), and assuming that M i-1 (l) are or will become DEAD, we prove by induction that M i (l) are or will become DEAD.

When the elements of M i-1 (l) have become DEAD (T1 or T2 of the protocols ) they will have sent FAIL messages to all the elements having incoming or outgoing arcs to them. When a LIVE element receives a message through its incoming arc, then it becomes DEAD.

(Transition T1 is executed for bars and T2 for places.) This satisfies part (1) of the definition of M 1 (l). Similarly, T1 is executed by LIVE bars receiving a FAIL message through an outgoing arc from a place of M i-1 (l). Hence such bars become DEAD, so satisfying part (2) of the definition of M 1 (l). Suppose that in OG1, a non-restoreable place p has an ordinary outgoing arc to a bar b of M i-1 (l). We check the following potential situations involving p and the arc connecting p and b at the moment p receives the FAIL message from b:

(1) non-restoreable and ordinary arc. Then T2 is performed and p becomes DEAD.

(2) restorable and ordinary arc. Since a non-restoreable place cannot become restorable, the message was received before the set M(i) was defined (i.e. an earlier situation than OG1).
However, when the message was received T3 was performed and the arc made dashed by the REACTIVATE operation. Therefore, in situation OG1, the arc connecting p and b could not be ordinary. Thus, this case could not arise.

(3) non-restorable and dashed arc. The arc was ordinary when M(1) was defined, but became dashed later, i.e. while p was non-restorable. This can be done only by T3, which requires p to be restorable. Thus, this case could not arise.

(4) restorable and dashed arc. Similar to the second case above, with the only difference that T3 is not performed. Thus, this case also could not arise.

This analysis applies to all the non-restorable places having an outgoing ordinary arc to some bar of M_1, thus (3) of the definition of M is satisfied.

Q.E.D.

Note: Each element that becomes DEAD sends messages to all elements having arcs to it, independently of the direction of those arcs. Furthermore, an element sends messages only when it becomes DEAD.

Theorem 2: Given an arbitrary element l, consider any situation OG1 of the Occurrence Graph in which the elements of M(1) are already DEAD and all the messages sent by these elements have already been received. Then the subgraph M(1) is an Ignoorable Activity.

Proof: Suppose it is possible that OG1 has a place p not included in M(1), having an ordinary outgoing arc to a bar b of M(1). Since places are generated together with their incoming arcs, and since dashed arcs do not become ordinary, b will have sent a FAIL message on that arc and p will have received this message through an ordinary arc. By definition of M(1), p is restorable in OG1, and therefore it was restorable when the FAIL message was received. Thus, when that message arrived T3 was executed by p, and the arc connecting p and b became dashed. Therefore, there cannot be any ordinary outgoing arcs from places not included in M(1) to bars of M(1).

Moreover, by definition of M(1), there are no arcs of any type.
from bars not included in M(1) to places of M(1), and from elements of M(1) to elements not included in M(1). In addition, all the places of M(1) are DEAD, and DEAD places are neither active nor restorable (by T2). Thus, M(1) satisfies the definition of Ignorable Activity.

Q.E.D.

Theorem 3: The protocols performed by the elements of a minimal Recoverable Activity do not cause elements which are not included in this Recoverable Activity to become DEAD, or the Recoverable Activity to grow.

Proof: An element can cause another element to become DEAD only by means of FAIL messages. These messages propagate through the arcs of the Occurrence Graph. Consider the elements which are not included in a minimal Recoverable Activity, but which have arcs to elements of that Activity. These elements are places having outgoing arcs to bars of the minimal Recoverable Activity. By definition, these places can only have either

(1) outgoing dashed arcs to bars of the minimal Recoverable Activity, in which case FAIL messages received through these arcs are ignored, or

(2) an outgoing ordinary arc and be restorable, in which case when a FAIL message is received through this arc T3 is performed.

Thus, when a FAIL message is received by an element not included in a minimal Recoverable Activity from an element of the Recoverable Activity, it does not cause transition to the DEAD state, and the FAIL message is not propagated any further.

Q.E.D.

3.3 Improvements of the Protocol

Several improvements of the Protocol are possible. First, it is not necessary to send a FAIL message to a bar or place which is already DEAD. Thus it is unnecessary to send a FAIL message to a place or bar from which the sending bar or place has already received such a message. In particular, if T1 is triggered by the arrival of a FAIL message, ACTION1 can exclude the arc
through which that message arrived. It can also be proved that every place has received \textit{FAIL} messages from all of its dashed outgoing arcs. Therefore, if T2 is initialised by invalidation, at most two \textit{FAIL} messages are sufficient, i.e. one to its incoming arcs, and possibly one to the ordinary outgoing arc from it. Similarly, if T2 is triggered by the arrival of a \textit{FAIL} message from an incoming arc at most one \textit{FAIL} message will have to be sent, i.e. to its ordinary outgoing arc. If T2 is triggered by a \textit{FAIL} message arriving via an outgoing arc, it can be shown that this message must arise through an ordinary outgoing arc. Thus, in this case only a single message is necessary, which is sent through the incoming arc.

\textbf{DEAD} places are neither active nor restorable and therefore they cannot generate any new bars. \textbf{DEAD} places and bars ignore any \textit{FAIL} messages received by them, and they will eventually become an Ignorable Activity. Thus, any bar or place that becomes \textbf{DEAD} can be deleted from the records kept by nodes of the underlying system. This reduces storage requirements by discarding records of history which cannot influence either the present or the future. In a similar way, dashed arcs can also be deleted.
4. Constrained State Restoration

Conditions, or even entire processes, which are perceived as independent at one level of abstraction, may in fact be dependent at a lower level. The most common example is a set of independent processes, which at a lower level of abstraction are dependent because of contention for shared resources. State restoration of contending processes is studied in [SHR77]. Here, we generalise our model to represent implicit constraints that should be maintained during normal execution, as well as during state restoration.

The Occurrence Graph is complemented by a set of constraints that active places should satisfy. These constraints can be expressed by sets of unallowed combinations of active places, or alternatively, by assigning attributes to each place and stating relations that the attributes of active places should satisfy.

For illustration, figure 7 shows an Occurrence Graph of two sequential processes, PR1 and PR2, contending for three resources of the same type. At each place i, the attribute $N_i^1$ denotes the number of these resources used by it when active. Obviously, the required constraint is that

$$\sum_{i \text{ active}} N_i^1 \leq 3.$$

Suppose that in figure 7 place 6 is declared invalid. In this case, although place 5 is a Recovery Line for PR1, place 5 cannot be reactivated because this would result in the system using $N_5^1 + N_i^5 = 4$ resources. There are several possible alternative restoration strategies available. We could, for example, restore an older Recovery Line of PR1 (e.g. place 3), force a restoration which involves PR2 (e.g. places 5 and 10), or wait to restore PR1 until PR2 is in an active place which makes the reactivation of place 5 possible. The last policy may lead to deadlock, because there could be a set of processes waiting for each other to release their resources in order to complete state restoration. When performing constrained state restoration, the election of a feasible Recovery Line will involve policy decisions, and possibly priorities concerning which jobs and how much of each chosen job's activity to undo.

The constraints given with the Occurrence Graph of figure 7 can also be represented without using the attribute $N_i^1$, for example by a set of the pairs of places which cannot be concurrently active:
\{(1,8)(1,9)(1,10)(1,11)(1,12)(5,9)(5,11)(5,12)\}.

There are many ways by which the constraints can be expressed, and different ways may be more concise or understandable for different types of systems. In general, we could even replace the whole Occurrence Graph model with an enumeration (or other expression) of all legal states to which the system could be restored, as discussed in [MER75]. However, such a description will lack the notions of "progress" and "amount of work undone" which is represented clearly by a directed graph such as the Occurrence Graph.
5. **Simplification of the Occurrence Graph**

If the Occurrence Graph is only used as a conceptual tool for studying or validating the implementation of a state restoration mechanism, the problem of keeping the size of the Occurrence Graph small has little or no practical significance. However, if the Occurrence Graph is directly used by the implementation, as illustrated by the protocol of Section 3, it is important to reduce its size in order to maintain storage requirements within practical limits.

A trivial way of reducing the size of an Occurrence Graph is to discard all Ignorable Activities. However, further reductions are usually possible. We describe below a method which can be used to reduce any Occurrence Graph to an equivalent one having, in most practical cases, a minimal number of places.

Given a subgraph of an Occurrence Graph, we define the set of relevant places as the set of all the places of the subgraph which are active or which belong to one or more of the Recovery Lines of the Occurrence Graph. Two Occurrence Graphs are said to be recovery equivalent if and only if:

1. There is a one-to-one correspondence between their relevant places.
2. There is a one-to-one correspondence between their Recovery Lines. That is, for any Recovery Line of one Occurrence Graph, the corresponding places in the other Occurrence Graph are also a Recovery Line.
3. Each such pair of corresponding Recovery Lines will define a pair of corresponding Recoverable Activities; for each relevant place in one such Recoverable Activity, the relevant place to which it corresponds is included in the other Recoverable Activity.

As stated in the note of Section 2, and as demonstrated by the protocol of Section 3, in many practical cases there is no need to retain dashed arcs in Occurrence Graphs. Therefore, we first concentrate on a reduction method which deals only with areas of Occurrence Graphs containing no dashed arcs. If applied to an Occurrence Graph from which all dashed arcs have been deleted, it will result in an Occurrence Graph containing a minimal number of places.
We introduce the method by the example shown in figure 8. If in this Occurrence Graph, place 13 is deleted, the result will be a recovery-equivalent Occurrence Graph. This is because such a deletion will not affect any Recovery Line, or the relevant places of the Recoverable Activity of such lines.

RULE 1: Any non-active non-restorable place without outgoing arcs can be deleted.

Since the bar which has an outgoing arc to such a place will be included in any Recoverable Activity which includes that place, the invalidity of the place is equivalent to the invalidity of the bar.

In addition to those described above, a more drastic reduction rule is demonstrated in figure 8(b), where all the elements included inside the dotted line of figure 8(a) are collapsed into a single bar 12. In this process the restorable place 7 is lost, but this place does not belong to any Recovery Line. This is because if place 7 were in a Recovery Line then bar 5 would be in the corresponding Recoverable Activity. But this implies that place 9, bar 7, place 5, and bar 4 are also in the same Recoverable Activity, and if bar 4 is included then so also must place 7 be. Similarly, if one of the elements inside the dotted line of figure 8(a) is included in any Recoverable Activity, all of them must be included and therefore they can be replaced by a single bar 12. Such a group of elements consistute what we term a Collapsible Activity.

The incoming arcs of bar 12 are the incoming arcs from elements (which must be places) outside the group to elements of the group; the outgoing arcs of bar 12 are the outgoing arcs from elements of the group to elements outside the group (which again must be places).

RULE 2: Given bars b1 and b2 such that there exists a non-restorable non-active place p having its incoming arc from b1 and its outgoing arc to b1, consider the set of elements b1, b2, p, and all the elements belonging to any of the directed paths which start at b1 and end at b2. If each of the places in this set has just a single outgoing arc the set of elements, together with their interconnecting arcs, constitute a Collapsible Activity. Such an Activity can be collapsed into a single bar, and the invalidity of any element in the set is equivalent to the invalidity of that bar.
In the previous example, b1 and b2 are respectively bars 4 and 7, and p is place 5. In the example, RULE 2 can also be applied for bars 12 and 8, and place 10, resulting in the Occurrence Graph of figure 8(c).

In Occurrence Graphs which do not contain dashed arcs, after RULE 1 has been applied wherever possible, RULE 2 is applicable for any non-restorable non-active place. Thus, by exhaustively applying RULE 2, the resulting graph will include only restorable or active places. Furthermore, any place which has an outgoing arc belongs to some Recovery Line. Clearly, this method of collapsing minimises the number of places, and also reduces the number of arcs and bars.

It is in fact possible to minimise even Occurrence Graphs in which dashed arcs are still retained. One technique for doing this involves first transforming places which have multiple outgoing arcs into sets of places where each place has at most one outgoing arc, and then applying a rule similar to Rule 2 above. Further simple transformations reduce the resulting graph to one in which all the places are active and/or restorable.

These and other graphs collapsing techniques can be used in various different ways. In particular, depending on circumstances, it might be thought appropriate always to keep the graph in minimal form; alternatively graph reductions might be performed only occasionally, or even only when storage limitations so demand.
6. Structured Occurrence Graphs - Functional Activities

Collapsible Activities and, for that matter, Recoverable Activities and Ignoorable Activities are all special cases of what we will term Functional Activities.

A **Functional Activity** is a subgraph of the Occurrence Graph in which:

1. given any two elements of the subgraph, all elements which are on a directed path between these two elements are also members of the subgraph;

2. all elements outside the subgraph which have outgoing or incoming arcs to the subgraph are places;

The set of places outside the Functional Activity which have an incoming arc from this Activity are called output places, and the set of places outside the Functional Activity which have an outgoing arc to this Activity are called input places. By definition, there is no directed path from an output place to an input place. A Recovery Line is a subset of the input places of a Functional Activity which has no output places. Similarly, an Ignoorable Activity corresponds to a Functional Activity having no output places, and such that all arcs from input places are dashed.

If we regard a Functional Activity as being replaced by a single bar (which is itself, by definition, a Functional Activity), the resulting graph will still express correctly all the cause-effect relationships that currently hold amongst the elements that remain. The new graph, in fact, provides us with a less detailed representation of the history of the system whose dynamics are being modelled. It is thus an abstraction of the original graph; an example of such an abstraction was given in figure 3, for the Occurrence Graph of figure 2(d). However, unless due care is taken such an abstraction might lose information concerning Recovery Lines and active places - the graph collapsing techniques described in Section 5 guarantee that no such loss of information occurs.

26.
Such Functional Activities provide an important method for structuring an Occurrence Graph, i.e. for dividing it into disjoint subgraphs (each subgraph being a Functional Activity) which are connected by input/output places, such that the properties of the subgraphs can be studied independently of each other, and, to a certain degree, independently of the complete graph. This independence results from the fact that the effects of a Functional Activity depend solely on its input places and are reflected solely in its output places. Clearly, such a partition of the Occurrence Graph can be done in different ways, presumably corresponding to different points of view. By definition, each structuring forms a partial ordering of the Functional Activities (i.e. the resulting Occurrence Graph is loop free) which means that Functional Activities do not effect those Functional Activities which cause them.

The degree of independence that Functional Activities have from each other is such that one can readily imagine, for example, the use of different recovery protocols and collapsing strategies inside each Functional Activity, as well as between Functional Activities. One can also imagine that each Functional Activity can be further divided into smaller Functional Activities and so forming an hierarchical structure. In fact we regard the concept of a Functional Activity as generalising and formalising the essence of the notion of an Atomic Activity [LOM 77, RAN 77], i.e. an activity which appears "logically instantaneous" to its environment, and from within which the environment seems "logically unchanging".
7. Concluding Remarks

The ideas and techniques presented in this paper provide a basic model which can be either directly implemented or used as a reference for validation for other backward error recovery mechanisms for concurrent systems. However, much further work remains to be done.

In practical systems one could, for example, expect the design of recovery protocols to take into account the planned constraints on information flow between entities of the system (e.g. using planned Functional Activities such as "conversations" [RAN75]), rather than depend totally on such records as can be provided of the history of actual information flow. Such constraints result in a priori knowledge of properties that the Occurrence Graphs of a particular system will possess, and which can be used to design more efficient protocols. Further study of practical constraints that will result in improved recovery protocols without unduly compromising system performance under normal conditions is clearly needed.

In many cases backward error recovery will be infeasible or insufficient, and some form of forward error recovery will be needed - this would involve the notion of "compensation" [BJO72, DAV77], i.e. the sending of additional corrective information to an entity which has previously received erroneous information, instead of requiring that the entity perform state restoration. Such strategies will involve considerations of the semantics associated with Occurrence Graphs, as well as their syntactic structure.
8. Acknowledgements

The development of the ideas contained in this paper has been greatly aided by the discussions we have had with numerous colleagues in the Computing Laboratory of the University of Newcastle upon Tyne, and with Eike Best in particular. The paper itself has been typed at prodigious speed and with unfailing cheerfulness by Julie Lennox. Acknowledgements are also due to the U.K. Science Research Council for their financial support for this research.
9. References


Figure 1 - State Restoration in a Sequential System
Figure 2 - The generation of an Occurrence Graph
Figure 2 (cont'd) – The generation of an Occurrence Graph
Figure 3 - A less detailed history
Figure 4 - Atomic State Restoration
Figure 5 - Decentralised State Restoration
Figure 5 (cont'd) - Decentralised State Restoration
T1:  
CONDITION1: (b is declared INVALID) OR (b RECEIVES FAIL message)
ACTION1: SEND FAIL through all arcs of b, independently of their direction.

T2:  
CONDITION2: (p is declared INVALID) OR (p RECEIVES FAIL from incoming arc) OR (p RECEIVES FAIL from ordinary outgoing arc AND p is not restorable)
ACTION2: IF p is active THEN DEACTIVATE(p);
IF p is restorable THEN COMMIT(p);
SEND FAIL message through all arcs of p, independently of their direction;

T3:  
CONDITION3: (p RECEIVES FAIL message from ordinary outgoing arc) AND (p is restorable)
ACTION3: REACTIVATE(p)

Figure 6: The "Chase" Protocol
Figure 7 - Occurrence Graphs with Constraints
Figure 8: Occurrence Graph Minimization