STATE RESTORATION IN DISTRIBUTED SYSTEMS

Philip M. MERLIN
Department of Electrical Engineering
Technion - Israel institute of Technology
Haifa, Israel

Brian RANDELL
Computing Laboratory
The University of Newcast!e-upon-Tyne
Newcastle-upon-Tyne, England

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. Since in distributed systems there may exist a great deal of independence between activities, the system can be restored to a state that could have existed rather than to a state that actually existed.

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.

1. INTRODUCTION

One important form of error recovery for fault-tolerance 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 Fig. 1, which shows the history of a system which has suffered from a number of state restorations.

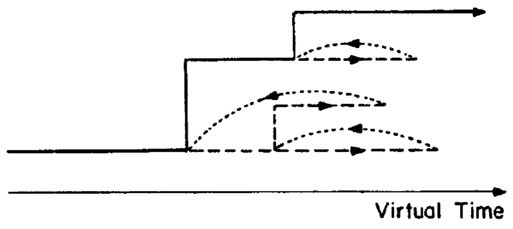


Fig. 1 - State restoration in a sequential system. (Dashed lines represent abandoned activity, dotted lines state restoration.)

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. A proof of the correctness of this protocol is presented in [MER77a], together with a study of such matters as the problem 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 useof such protocols.

2. DESCRIPTION OF THE MODEL

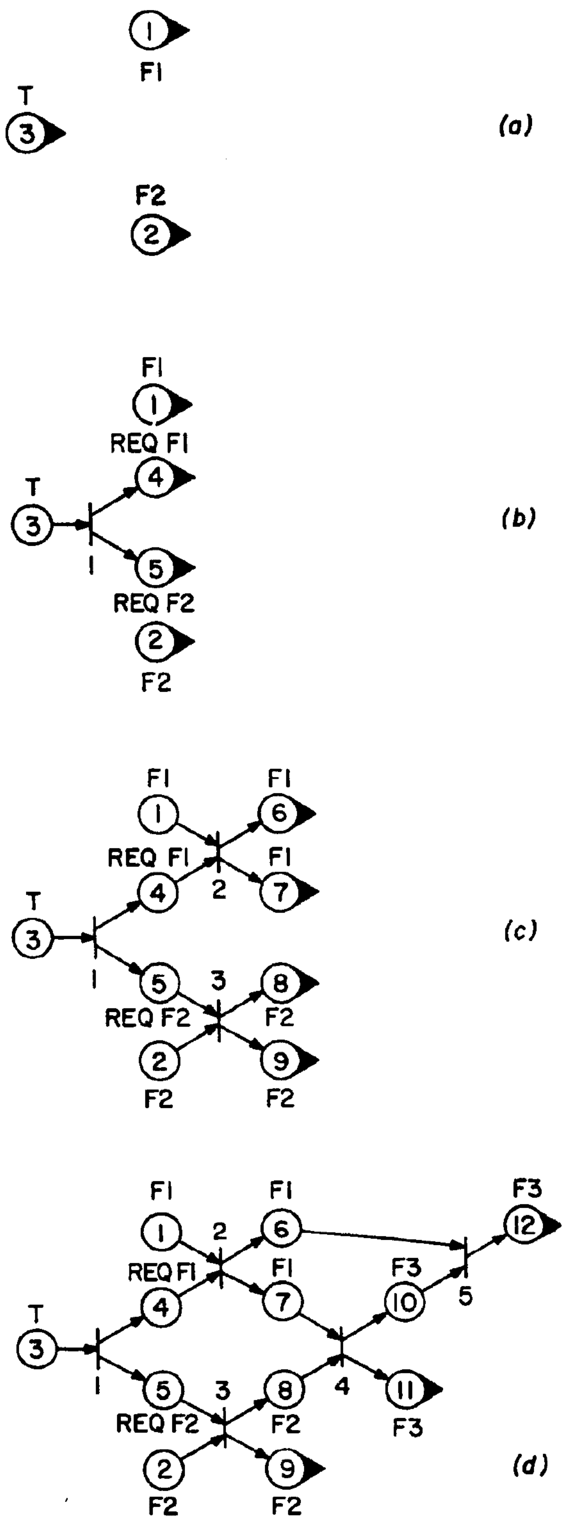
In this section we introduce the Occurrence Graph model of the dynamic behaviour of a concurrent system. Such graphs are similar to the Occurrence Nets (also called Causal Nets) described in [HOL68, PET76, PET77]. The main difference is that Occurrence Graphs are viewed aa a dynamic structure which is "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 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 Fl, place 2 represents the existence of file F2 and place 3 represents the fact that the terminal is "ready to send" the requests. (In Fig. 2 the names Fl, 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 F1 and F2, are created. In the model, the occurrence of an event is denoted by a "bar". The new situation is shown in Fig. 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 Fig. 2(c) will represent the perhaps concurrent events of copying the files. 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. Fig. 2(d) shows the entire Occurrence Graph model of the history of the cause-effect relationships between conditions and events for the <u>dynamics</u> 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 12) 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



<u>Fig. 2</u> - The generation of an Occurrence Graph

generating events, e.g. place 6 of of Fig. 2(c) represents the fact that although file F1 (place 1) generates event 2, after the occurrence of F2 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. (In as much as it is appropriate to refer to the instantaneous "global state" of a distributed system, this is what the set of active places represents.)

In the Occurrence Graph, there is a directed path between two places or bars if and only if they are causally connected. (Notice that events or conditions which are not causally connected could have occurred simultaneously.) By definition, when a new bar is created, it may have outgoing arcs only to new places representing conditions which are generated by this bar. This implies that Occurrence Graphs are acyclic (i.e. they contain no directed loops) meaning that no event or condition can be, directly or indirectly, its own cause. The progress made by a system or algorithm is represented by the growth of the graph (in our figures, towards the right-hand side of the page).

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.

In the Occurrence Graph model, each event is atomic. All 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.

Restoration of a condition can be achieved by the 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 Fig. 3. We assume that if at a certain point in time a condition is non-restorable it cannot become restorable later. (In [MER77a] 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 placing a black triangle in the place, replacing its outgoing arc by a dashed arc, and marking the bar to which it is connected by that arc with an "*". 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 de-activate 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 places 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 sub-graphs. 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 has occurred. Such a component is called an Ignorable Activity. Ignorable Activities can be freely deleted from the Occurrence Graph.

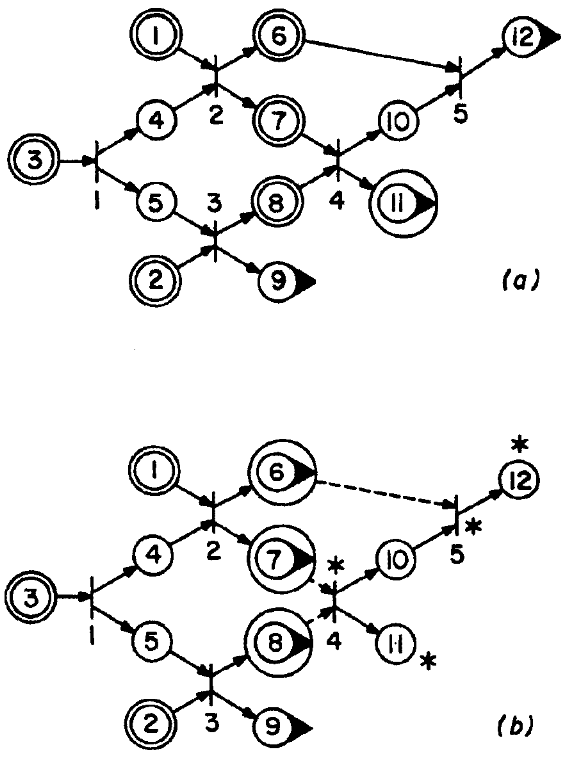


Fig. 3 - Atomic state restoration.

Suppose Fig. 3(a) is the Occurrence Graph of Fig. 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 Fig. 3(b), in which places 11 and 12 are invalid because they were deactivated, bar 5 is invalid by declaration and also 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 Fig. 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 Recoverable 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 Recoverable Activity is a viable candidate for an Ignorable Activity, and in fact, only Recoverable 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 Sec. 2.3.

We conclude this subsection by showing the two additional state restorations that can be performed in the system of Fig. 3(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 Fig. 2(a). If the entire Occurrence Graph of Fig. 3(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.

We illustrate the type of problems which may arise while performing decentralised state restoration by the following example. Suppose that in the example of Fig. 3(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 Fig. 4(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 Fig. 4(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 Fig. 4(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 Fig. 4(a) a new bar involving places 9 and 11 can be generated, as shown in Fig. 4(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 Recoverable Activity in the process of restoration and are about to be deactivated.

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. The reader interested in a more formal discussion of Occurrence Graphs, of properties of such graphs, and of Recovery using Occurrence Graphs is referred to [MER77a].

3. A DECENTRALISED RECOVERY MECHANISM THE "CHASE PROTOCOLS"

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

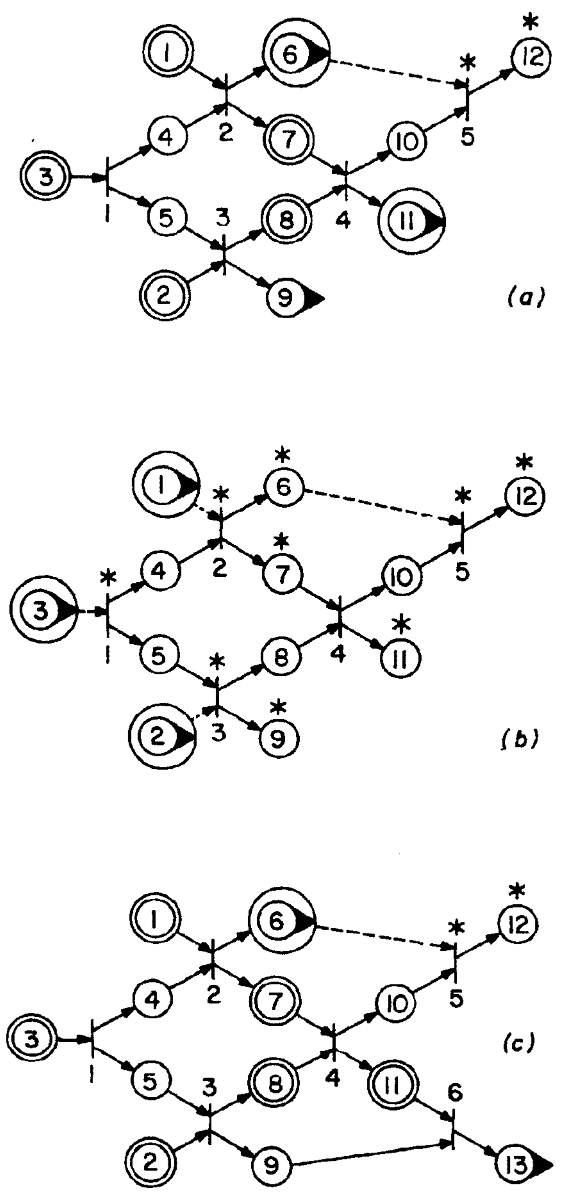


Fig. 4 - Decentralised State Restoration

Suppose a system ie 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 apacket switching network, a distributed application, or any other system where only message communications 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 generate new bars and places and commit restorable places, but the deactivation of active places and the reactivation of restorable non-active places 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.

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 recovery 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 recovery.

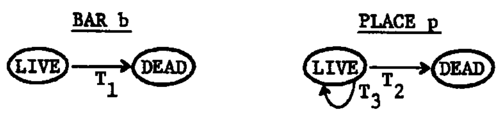
None of the operations performed on the graph (i.e. GENERATE, COMMIT, REACTIVATE, DEACTIVATE, INVALIDATE) can reduce the size of the minimal Recoverable Activity that includes a given element, In fact, the GENERATE, COMMIT and REACTIVATE operations can cause the size of the Activity to increase. Illustrations 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-restorable, 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-restorable and set to DEAD. If a LIVE restorable 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 Fig. 5 using a notation similar to those used in [BOC76] and [MER77b]. 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 medicate CONDITION1 is satisfied by b, and during the transition ACTION1 is performed. The transitions of a 'p' machine are executed in a similar way.



T1: CONDITION: (b is declared INVALID) OR (b RECEIVES FAIL message).
ACTIONI: 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 outging 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: CONDITION: (p RECEIVES FAIL message from ordinary outgoing arc) AND (p is restorable).
ACTION3: REACTIVATE(p)

Fig. 5 - The "Chase" Protocol.

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 queueing). This ensures that no more than one transition can be executed at each place at any given time. In [MRR77a] a formal validation of the protocol is given together with a discussion of several improvements which can be made in the protocol.

4. 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 of 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 "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" [BJ072, 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.

5. REFERENCES

[BJ072] L.A. Bjork, C.T. Davies, "The Semantics of the Preservation and Recovery of Integrity in a Data System", TR 02.540, IBM, San Jose, Cal., 1972.

[BOC76] G.V. Bochman, J. Gecsei, "A Unified Method for the Specification and Verification of Protocols" Pub. #247, Dept. d'Informatique, Univ. of Montreal, 1976.

[DAV77] C.T. Davis, "Data Base Spheres of Control", TR 02.782, IBM, San Jose, Cal. 1977.

[HOL68] A.W. Holt, R.M. Shapiro, H. Saint, S. Marshall, "Information System Theory Project", Appl. Data Research ADR 6606 (US Air Force, Rome Air Development Center RADC-TR-68-305), 1968.

[LOM77] D.B. Lomet, "Process Structuring, Synchronisation and Recovery using Atomic Actions", Proc. ACM Conf. on Language Design for Reliable Software. Sigplan Notices 12, 3, 128-137, 1977.

[MER77a]P.M.Merlin, B. Randell, "Consistent State Restoration in Distributed Systems", TR 113, Computing Lab., Univ. of Newcastle-on-Tyne, UK, 1977.

[MRR77b]P.M.Merlin, A. Segal, "A Failsafe Loop-Free Algorithm for Distributed Routing in Data Communication Networks", Pub. 313, Dept. of Electr. Eng., Technion, Haifa, Israel, 1977.

[PET76] C.A. Petri, "Nichtsequentielle Prozesse", Rt. 76-6, GMD-ISF, Bonn, W. Germany, 1976.

[PET77] C.A. Petri, "General Net Theory", Proc. of the Joint IBM/Univ. of Newcastle-upon-Tyne Seminar on Computing System Design (B. Shaw, Ed.); Comp. Lab., Univ. of Newcastle-upon-Tyne, U.K., 1977, pp. 131-169.

[RAN75] B. Randell, "System Structure for Software Fault Tolerance", IEEE Trans. on Software Eng SE-1, 2, pp. 220-232, 1975.

[RAN77] B. Randell, P.A. Lee, P.C. Treleaven, "Reliable Computing Systems", TR 102, Comp. Lab., Univ. of Newcastle-upon-Tyne, U.K., 1977.

6. Acknowledgement

Development of the ideas contained in this paper has been greatly aided by discussions with numerous colleagues in the Computing Lab. of the Univ. of Newcastle-upon-Tyne, particularly with Eike Best. Acknowledgements are also due to the UK Science Research Council for their financial support of this research.