

Fault Tolerance in Decentralized Systems

Brian Randell,

University of Newcastle upon Tyne, NE1 7RU, UK

Abstract: In a decentralised system the problems of fault tolerance, and in particular error recovery, vary greatly depending on the design assumptions. For example, in a distributed database system, if one disregards the possibility of undetected invalid inputs or outputs, the errors that have to be recovered from will just affect the database, and backward error recovery will be feasible and should suffice. Such a system is typically supporting a set of activities that are competing for access to a shared database, but which are otherwise essentially independent of each other - in such circumstances conventional database transaction processing and distributed protocols enable backward recovery to be provided very effectively. But in more general systems the multiple activities will often not simply be competing against each other, but rather will at times be attempting to co-operate with each other, in pursuit of some common goal. Moreover, the activities in decentralised systems typically involve not just computers, but also external entities that are not capable of backward error recovery. Such additional complications make the task of error recovery more challenging, and indeed more interesting. This paper provides a brief analysis of the consequences of various such complications, and outlines some recent work on advanced error recovery techniques that they have motivated.

Key Words — Concurrency, error recovery, co-ordinated atomic (CA) actions, exception handling, dependability.

1 Introduction

The problems of preventing faults in decentralised systems from leading to system failures vary very much in difficulty depending on the (it is hoped justified) assumptions that the designers make about the nature of the faults, and the effectiveness of the fault tolerance mechanisms that are employed. For example, one might choose to assume that operational hardware faults can be cost-effectively masked by the use of hardware replication and voting, or that any residual software design faults can be adequately masked by the use of design diversity. In such circumstances error recovery is not needed. In many realistic situations, however, if the likelihood of a failure is to be kept within acceptable bounds, error recovery facilities will have to be provided, in addition to whatever fault prevention and fault masking techniques are used.

In a decentralised system, i.e. one whose activity can be usefully modelled by a set of partly independent threads of control, the problems of error recovery will vary very greatly depending on what design assumptions can be justified. For example, if the designer concerns him/herself simply with a distributed database system and disallows (i.e. ignores) the possibility of undetected invalid inputs or outputs, the errors that have to be recovered from will essentially be ones that are wholly within the database. In this situation backward error recovery will suffice,

regarded as absolute, so that once the commitment has been made, there is no going back, i.e. no provision for the possibility that an output was invalid. The notion of nested transactions can be used to limit the amount of activity that has to be abandoned when backward recovery (of small inner transactions) is invoked. However, this notion typically still assumes that there are absolute “outermost” transactions, and that outputs to the world outside the database system, e.g. to the users, that take place after such outermost transactions end must be presumed to be valid.

The conversation scheme [Campbell and Randell 1986] provides a means of co-ordinating the recovery provisions of interacting threads so as to avoid the domino effect, without making assumptions regarding output or input validation. Figure 2 shows an example where three threads communicate within a conversation and the threads T1 and T2 communicate within a nested conversation. Communication can only take place between threads that are participating in a conversation together. The operation of a conversation is: (i) on entry to a conversation a thread establishes a checkpoint; (ii) if an error is detected by any thread then all the participating threads must restore their checkpoints; (iii) after restoration all threads then attempt to make further progress; and (iv) all threads leave the conversation together.

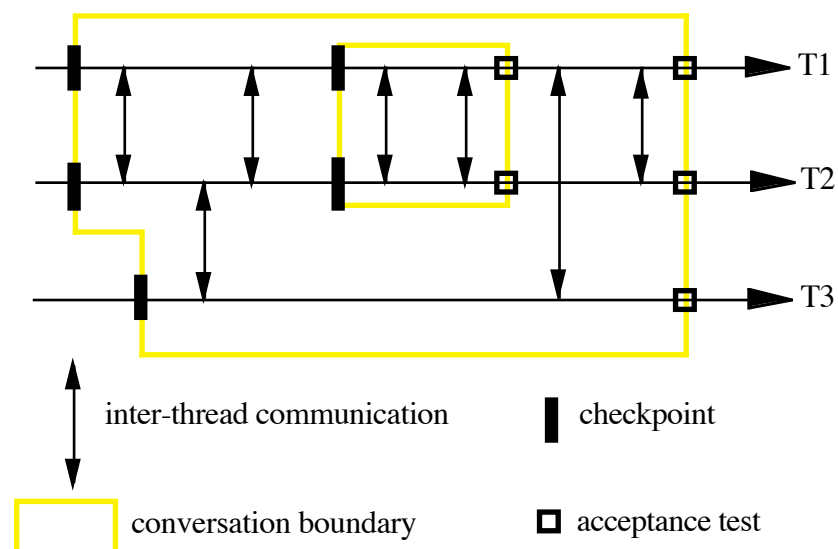


Figure 2: Nested conversations.

Both transactions and conversations are examples of atomic actions, in that viewed from the outside, they appear to perform their activity as a single indivisible action. (In practice transaction-support systems also implement other properties, such as “durability”, i.e. a guarantee that the results produced by completed transactions will not be lost as a result of a computer hardware fault.)

The basic transaction involves a single thread, whereas a conversation normally involved multiple threads. The notion of a conversation is still a very restrictive one, however. It assumes that all the threads can perform backward error recovery, and that it is possible to determine which threads need to take part in which conversations, i.e. to synchronize their taking of recovery points so as to enter a conversation, and to synchronize their exit from that

conversation. In other words, it addresses co-operative but not competitive concurrency, since it does not provide a convenient means of protecting, and providing error recovery for the activities of threads that are merely competing for access to a shared resource, e.g. a database, something that transactions do very well.

3 Co-ordinated Atomic Actions

The Co-ordinated Atomic (CA) Action scheme [Xu et al. 1995] was arrived at as a result of work on extending the conversation concept so as to allow for the use of forward error recovery, and to allow for both co-operative and competitive concurrency. A CA action is a mechanism for co-ordinating multi-threaded interactions and ensuring consistent access to objects in the presence of concurrency and potential faults. CA actions can be regarded as providing a programming discipline for nested multi-threaded transactions [Caughey et al. 1998] that in addition provides very general exception handling provisions. They augment any fault tolerance that is provided by the underlying transaction system by providing means for dealing with (i) unmasked hardware and software faults that have been reported to the application level to deal with, and/or (ii) application-level failure situations that have to be responded to.

The concurrent execution threads participating in a given CA action enter and leave the action synchronously. Within the CA action, operations on objects can be performed co-operatively by *roles* executing in parallel. To co-operate in a CA action a group of concurrent threads must come together and agree to perform each role of the action, with each thread undertaking a different role. Inside a CA action, some or of all its roles can be involved in further (nested) CA actions. If an error is detected inside a CA action, appropriate forward and/or backward recovery measures must be invoked co-operatively, by all the roles, in order to reach some mutually consistent conclusion. To support backward error recovery, a CA action must provide a recovery line that co-ordinates the recovery points of the objects and threads participating in the action so as to avoid the domino effect. To support forward error recovery, a CA action must provide an effective means of co-ordinating the use of exception handlers. An *acceptance test* can and ideally should be provided in order to determine whether the outcome of the CA action is successful. Error recovery for participating threads of a CA action generally requires the use of explicit error co-ordination mechanisms, i.e. exception handling or backward error recovery within the CA action; objects that are external to the CA action and so can be shared with other actions and threads must provide their own error co-ordination mechanisms. These external objects, which are in effect being competed for, must behave atomically with respect to other CA actions and threads so that they cannot be used as an implicit means of “smuggling” information [Kim 1982] into or out of a CA action.

Figure 3 shows an example in which two concurrent threads enter a CA action in order to play the corresponding roles. Within the CA action the two concurrent roles communicate with each other and manipulate the external objects co-operatively in pursuit of some common goal. However, during the execution of the CA action, an exception *e* is raised by one of the roles. The other role is then informed of the exception and both roles transfer control to their respective exception handlers *H1* and *H2* for this particular exception, which then attempt to perform forward error recovery. (When multiple exceptions are raised within an action, a resolution algorithm based on an exception resolution graph [Campbell and Randell 1986; Xu et

al. 1998] is used to identify the appropriate “covering” exception, and hence the set of exception handlers to be used in this situation.) The effects of erroneous operations on external objects are repaired, if possible, by putting the objects into new correct states so that the CA action is able to exit with an acceptable outcome. The two threads leave the CA action synchronously at the end of the action. (As an alternative to performing forward error recovery, the two participating threads could undo the effects of operations on the external objects, roll back and then try again. This would in general require the use of diversely-designed software alternates if the aim was to tolerate residual design faults, though a simple “re-try” strategy can be effective when the fault is in effect transient [Gray 1990].)

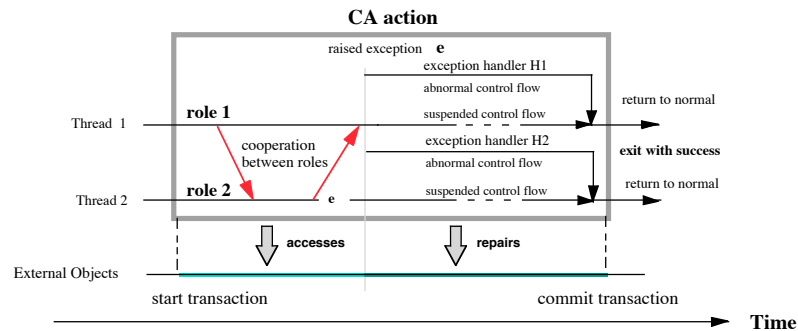


Figure 3: Example of a CA Action

In general, the desired effect of performing a CA action is specified by an acceptance test. The effect only becomes visible if the test is passed. The acceptance test allows both a normal outcome and one or more exceptional (or degraded) outcomes, with each exceptional outcome signalling a specified exception to the surrounding environment (typically a larger CA action). The CA action is considered to have failed if the action failed to pass the test, or the roles of the action failed to agree about the outcome. In such circumstances, it is necessary to try to undo the potentially visible effects of the CA action and signal an `abort` exception to the surrounding environment. If the CA action is unable to satisfy the “all-or-nothing” property (e.g. because the undo fails), then a `failure` exception must be signalled to the surrounding environment (in general an enclosing CA Action). This `failure` exception indicates that the CA action has not passed its acceptance test and its effects have not been undone, so that the system has probably been left in an erroneous state, which it is now the responsibility of the environment to deal with. Thus, ideally, execution of a CA action will only produce one of the following four forms of outputs: a normal outcome, an exceptional outcome, an `abort` exception, or a `failure` exception.

The fact that CA actions provide a discipline for using forward error recovery has the important effect that they can be regarded as defining a protocol for confining and dealing with errors that arise from (i) faults in the environment of a (possibly distributed) computing system, as well as with signalled faults in the computer system itself, and (ii) the problems that arise because inputs to or outputs from the computer system cannot be immediately validated. They thus are aimed at the problems of general decentralized systems, not just decentralized computer systems, and thus can be seen as embodying some of the advanced of C.T. Davies’ pioneering ideas of “spheres of control” [Davies 1979]. (The spheres of control concept originated from consideration of the problems of managing a nation-wide-wide semi-automated US Air Force

stores inventory project. The basic ideas influenced early work on basic database transactions, and the work still has many lessons for today's system designers.)

An Example

One of the experimental uses that I and my colleagues have made of co-ordinated atomic (CA) actions is as a system structuring tool to design and validate a sophisticated control system for a complex industrial application that has high reliability and safety requirements, the specification and simulator for which were defined and developed by FZI (Forschungszentrum Informatik, Germany). The FZI "Fault-Tolerant Production Cell" [Lewerentz and Lindner 1995] represents a manufacturing process involving six devices: two conveyor belts (a feed belt and a deposit belt), an elevating rotary table, two presses and a rotary robot that has two orthogonal extendible arms equipped with electromagnets - see Figure 4. The task of the cell is to get metal blanks from its "environment" via the feed belt, transform them into forged plates by using one of the presses, and then return them to the environment via the deposit belt.

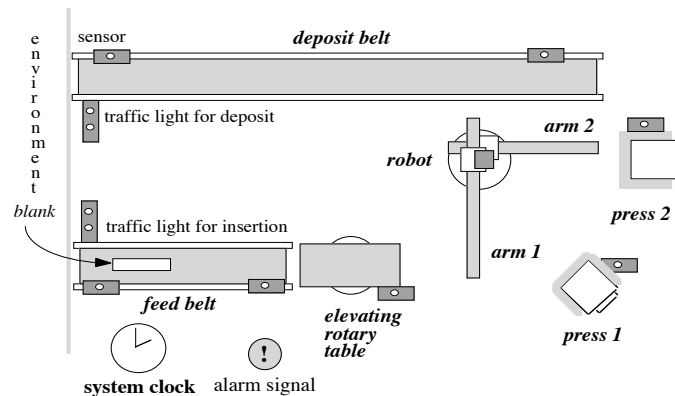


Figure 4: The Fault-Tolerant Production Cell (Top View)

The challenge posed by the model specification is to design a control system that maintains specified safety and liveness properties even in the presence of a large number and variety of device and sensor failures, and which will continue to operate even if one of the presses is non-operational. Based on an analysis of the possible failures, we provided a design for a control program that uses CA actions to deal with both safety-related and fault tolerance concerns, undertook the formal verification of this design based on the use of model-checking, and implemented a system that controls the FZI-provided graphical simulator.

The main characteristics of our design are the way it separates safety, functionality, and efficiency concerns among a set of CA actions, which thus could be designed, and validated, independently of each other, and of the set of device/sensor-controllers that dynamically determine the order in which the CA actions are executed at run-time. In particular, the safety requirements are satisfied at the level of CA actions, while the other requirements are met by the device/sensor-controllers. Our design for the Fault-Tolerant Production Cell uses 12 main CA actions; each action controls one step of the blank processing and typically involves passing a blank between two devices.

We have implemented this design for a control program using a Java implementation of a distributed CA action support scheme [Zorzo et al.]. (This scheme makes use of the nested

multi-threaded transaction facilities provided by the Arjuna transaction support system [Parrington et al. 1995].)

Figure 5 is a screen-shot of the FZI simulator of the production cell. Overlaid on the screen are rectangular outlines we use to portray the scopes of these CA actions. When the simulator is operated by our control program each of these outlines is gradually shaded in each time the corresponding CA action operates - the colour of this shading is changed when the CA action is involved in exception handling, in response to simulated faults. (Multiple concurrent faults, of many different types, can be injected via an on-screen control panel, not shown in Figure 5.)

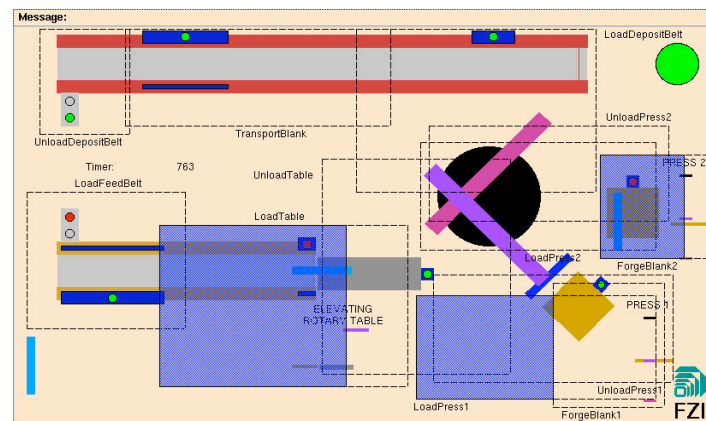


Figure 5: The Fault-Tolerant Production Cell Simulator

We confirmed that CA action structuring facilitated both the design and verification of what was a really quite sophisticated and essentially decentralized control system by enabling the various safety problems (involving possible clashes of moving machinery) to be treated independently of each other. Even complex situations involving the concurrent occurrence of any pairs of the many possible mechanical and sensor failures were handled simply yet appropriately.

Concluding Remarks

My aim in this paper has been to indicate how differing assumptions regarding the faults that a decentralized system lead to very different approaches to the provision of means of tolerating such faults. I have concentrated on just one aspect of fault tolerance, namely that of how to structure a system in order to provide simple yet powerful means of error containment and recovery, capable of dealing with very complex situations. Practical experience has shown that error recovery mechanisms can often be the most difficult and failure-prone parts of a system - hence the stress that I have put on issues of system structuring. Though the experimental work that I have described has been carried out just at the Java programming language level, these structuring ideas are I claim also of potential relevance at other system levels, something that I and my colleagues hope to demonstrate in future research.

Acknowledgements

This work on CA actions described above was supported by the ESPRIT Long Term Research Project 20072 on "Design for Validation" (DeVa). My thanks for the help received from a number of colleagues in this project in preparing this paper.

References

- [Campbell and Randell 1986] R.H. Campbell and B. Randell, "Error Recovery in Asynchronous Systems," *IEEE Trans. Software Engineering*, vol. SE-12, no. 8, pp.811-826, 1986.
- [Caughey et al. 1998] S.J. Caughey, M.C. Little and S.K. Shrivastava. "Checked Transactions in an Asynchronous Message Passing Environment," in *1st IEEE International Symposium on Object-Oriented Real-time Distributed Computing*, pp. 222-229, Kyoto, 1998.
- [Davies 1979] C.T. Davies. "Data Processing Integrity," in *Computing System Reliability*, ed. T. Anderson and B. Randell, pp. 288-354, Cambridge University Press, 1979.
- [Gray 1990] J. Gray, "A Census of Tandem System Availability between 1985 and 1990," *IEEE Trans. Reliability*, vol. 39, no. 4, pp.409-418, 1990.
- [Gray and Reuter 1993] J. Gray and A. Reuter. *Transaction Processing: Concepts and techniques*, Morgan Kaufmann, 1993.
- [Kim 1982] K.H. Kim, "Approaches to Mechanization of the Conversation Scheme Based on Monitors," *IEEE Trans. Soft. Eng.*, vol. SE-8, no. 3, pp.189-197, 1982.
- [Lewerentz and Lindner 1995] C. Lewerentz and T. Lindner. *Formal Development of Reactive Systems: Case Study "Production Cell" (LNCS-891)*, Springer-Verlag, 1995.
- [Parrington et al. 1995] G.D. Parrington, S.K. Shrivastava, S.M. Wheeler and M.C. Little, "The Design and Implementation of Arjuna," *USENIX Computing Systems Journal*, vol. 8, no. 3, 1995.
- [Randell 1975] B. Randell, "System Structure for Software Fault Tolerance," *IEEE Trans. on Software Engineering*, vol. SE-1, no. 2, pp.220-232, 1975.
- [Xu et al. 1995] J. Xu, B. Randell, A. Romanovsky, R.J. Stroud and Z. Wu. "Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery," in *Proc. 25th Int. Symp. Fault-Tolerant Computing (FTCS-25)*, Los Angeles, IEEE Computer Society Press, 1995.
- [Xu et al. 1998] J. Xu, A. Romanovsky and B. Randell. "Co-ordinated Exception Handling in Distributed Object Systems: from Model to System Implementation," in *Proc. 18th IEEE International Conference on Distributed Computing Systems*, pp. 12-21, Amsterdam, Netherlands, 1998.
- [Zorzo et al.] A.F. Zorzo, A. Romanovsky, B.R. J. Xu, R.J. Stroud and I.S. Welch, "Using Co-ordinated Atomic Actions to Design Complex Safety-Critical Systems: The Production Cell Case Study," *Software – Practice & Experience*, vol. 29, no. 8, pp. 677-697, 1999.