APPROACHES TO SOFTWARE FAULT TOLERANCE

Brian RANDELL

The University of Newcastle
Dept. of Computing Science
Newcastle upon Tyne, NE1 7RU, UK

ABSTRACT - A personal and rather discursive account is given of the background to the start of work in the early 1970s at Newcastle on software fault tolerance, and of how work has developed to encompass forward as well as backward error recovery, and parallel and distributed software as well as sequential programs. A major theme of the paper is that of the links between this work and that carried out elsewhere in connection with the topic of object-oriented programming, in particular on concepts such as generic classes and functions, exception-handling, delegation and reflection.

1 - INTRODUCTION

During the current academic year I have had the pleasure of being involved in no less than three Silver Jubilees. One was of the Annual Newcastle International Seminar on the Teaching of Computing Science, where my role was an organizational one, but two have called for me to speak in France. Several months ago I had the honour of speaking at the INRIA 25th Anniversary Conference. Now I have the equally great honour, and even greater pleasure, of speaking at that of LAAS-CNRS.

At INRIA it seemed appropriate to concentrate in my lecture largely on the overall work of PDCS. This is the ESPRIT Basic Research Project on Predictably Dependable Computing Systems that I, or more accurately Jean-Claude Laprie and I, lead. However Jean-Claude and his group here at LAAS play such a central role in PDCS that I am happy to leave discussion of the PDCS project largely to them on this occasion. In fact, partly because I feel so at home here at LAAS, and partly because it seems appropriate for an anniversary celebration, I will instead discuss how ideas on software fault tolerance originated and how work on them has developed over a period almost as long as the lifetime of LAAS. The talk will as a result be rather discursive and, I am afraid, overly autobiographical in parts.

2 - FIRST INFLUENCES

I have chosen “Approaches to Software Fault Tolerance” as the title of this talk. The ambiguity in this title is deliberate, since I wish to mention how the topic of software fault tolerance is perceived by others as well as discuss how it originated and has developed. But first let me give you my perspective on the origins of the topic.

Twenty-five years ago my personal research interests were focussed on operating system design methodology. I was then at the IBM T.J. Watson Research Center in the USA, where a colleague and I were aiming to aid system designers by providing means of predicting the likely impact of their various design decisions on the ultimate performance of the overall final system. The approach we developed was based on expressing the system design as a simulation program. Such a simulator would evolve and grow vastly more detailed as the design process progressed and hence its careful structuring would be of paramount importance.

This requirement led us to develop a “multi-level modelling” scheme for structuring the simulator using what we termed “levels of abstraction” [Zurcher and Randell 1968]. Each such level was in effect a separate simulator which represented those design decisions that were naturally expressed at that level. Each level was linked to the one above it, but in such a way as not to compromise the structure of either level. This is in my opinion an interesting idea whose time has yet to come - a point I will return to briefly later.

It was these twin issues of system structuring and design decision-making, both at the time motivated mainly by performance considerations, that were uppermost in my mind when I received an invitation to the first NATO Software Engineering Conference, held at Garmisch Partenkirchen in Germany in 1968. Another attendee at the NATO conference was Edsger Dijkstra, whom I had known since the early 1960s when he had greatly helped me and my colleague Lawford Russell in our development of an Algol-60 compiler for the English Electric KDF9 computer.

Dijkstra had recently completed his very influential work on a multiprogramming system for the Electrologica X-8 computer at the Technische Hogeschool, Eindhoven. The elegant structuring of the THE system, as it was called, also relied heavily on the notion of level of abstraction, and had a major influence on many people’s thinking about how to structure complex software systems.

The impact of the NATO conference, especially on many of the attendees, was very great. For example, both Dijkstra and I have since gone on record as to how the discussions at this conference on the "software crisis", and possible catastrophic system failures due to software problems, strongly influenced our thinking and our subsequent research. In Dijkstra's case the discussions led him into an very successful long term study of the problems of producing high quality programs. In my own case, following my move to Newcastle soon after the NATO Conference, they led me to consider the then very novel, and still somewhat controversial, idea of software fault tolerance. Suffice it to say that our respective choices of research problem match our respective skills at program design and verification.

One other event, again 25 years ago, also had a great though largely negative influence on my subsequent activities. I had been a member of the IFIP Algol Committee since 1964. In 1968 the

Committee finally split asunder, with a group of us unable to share the majority's enthusiasm for what became known as Algol-68. For my part, this incident led me, from 1968 onwards, to deliberately avoid further involvement in programming language design per se, though I remained enthusiastic for taking linguistic, as opposed to mechanistic, approaches to computer system design problems.

Although it was not made explicit in the Minority Report that Edsger Dijkstra, Tony Hoare and several others of us produced before resigning from the Algol Committee, most of us were convinced even then that Simula-67 contained more important programming language ideas than Algol-68 - Simula-67 was of course the start of object-oriented programming. I thus can claim that my enthusiasm for object-oriented programming ideas is not simply in response to the current bandwaggon. However I have no excuse for the fact that my enthusiasm for the ideas has on a number of occasions not led as quickly as they should to my recognizing their applicability in some new situation.

3 - RECOVERY BLOCKS

The Reliability Project that I and colleagues started at Newcastle in about 1971 at first involved a study of a number of large software systems, and of their typically rather ad hoc, though often quite effective, provisions for (mostly hardware) fault tolerance. We were well aware that if we were to develop techniques aimed explicitly at tolerating software faults we would have to allow for the fact that the principal cause of residual software design faults is complexity. Therefore the use of appropriate structuring techniques would be crucial - otherwise the additional software that would be needed might well increase the system's complexity to the point of being counter-productive.

The first structuring technique that we developed was what we termed a "recovery block". This was an extension and generalization of existing error recovery techniques, whose first linguistic formulation was due to Jim Horning, visiting Newcastle from Toronto during the Summer of 1973, and Mike Melliar-Smith. To us it was a virtue that this scheme was not tied to any particular programming language or even type of language.

The structure required a suitable mechanism for providing automatic backward error recovery. I produced the first such "recovery cache" scheme, a description of which was included in the first paper on recovery blocks [Horning et al. 1974] (although it was later superseded[Anderson and Kerr 1976]), together with a discussion of "recoverable procedures" - a rather complex mechanism that Hugh Lauer and I had proposed as a means of extending the recovery block scheme to deal with programmer-defined data types. This part of the paper would undoubtedly have been much clearer if we had expressed our ideas in object-oriented terms. But instead, in the paper and in our immediately subsequent work we concentrated on processes as we enlarged the domain for which we explored means of software fault tolerance.

Our initial work had simply concerned sequential programs. But by 1975 we had moved on to consider the problems of providing structuring for error recovery among sets of co-operating processes. Having identified the dangers of what we came to term the "domino effect", we had come up with the notion of a "conversation"[Randell 1975] - something which we later realized was a special case of a nested atomic action, or transaction.

Such a "process-oriented" approach had seemed natural to me - indeed some years earlier Jim Horning and I, starting from the ideas involved in multi-level modelling, had laboured for several years off and on over a paper entitled "Process Structuring". The eventual result was a lengthy survey paper in which we developed a set of formal definitions of such notions as process abstraction and combination [Horning and Randell 1973]. However, such a view of things tends to down play the problems of structuring a system state - just as it was rather easy when considering the object-oriented ideas introduced by Simula-67 to concentrate on the idea of associating data structures with the procedures manipulating those data structures, and to pay inadequate attention to the relevance of the ideas to activity structuring. And of course both are equally important.

As work continued at Newcastle, and extended into the realm of distributed systems, somewhat of a friendly division grew up between those of us who used the terminology of communicating processes and those of us who used the terminology of objects and actions. However, as my colleagues Santosh Shrivastava, Luigi Mancini and I belatedly came to realize, this terminological division was of only surface importance.

After examining the structure of a variety of systems, we developed two canonical models of fault-tolerant systems, one representative of the techniques and terminology used within the database and transaction processing community, the other more closely allied to the real time and process control applications area. The techniques employed in each model to provide various fault tolerance properties, in particular freedom from interference, backward error recovery, and crash resistance were found to have duals in the other model. Similarly any behaviour observed in one model has its dual in the other. Thus techniques and mechanisms which happen to have been developed in the domain of one model can be mapped and applied to the other model, and there is no inherent reason for favouring one approach over the other.

This duality should not be regarded as surprising - it is in effect an extension of the duality between procedure-based and message-based operating systems observed earlier by Lauer and Needham in the context of operating systems [Lauer and Needham 1978]. Nevertheless the completeness of the duality, and the utility of its recognition, was somewhat of a surprise. The particular benefit was that it enabled us to identify problems and techniques that each community had worked on, whose potential relevance the other community had not apparently realized [Shrivastava et al. 1987].

4 - EXCEPTION HANDLING

After the early work on recovery blocks, the Newcastle project went on to consider error recovery in more generality, and to develop an overall approach to exception handling in multi-level systems. It became clear that recovery blocks are just a convenient notation for a stylized form of exception handling in which backward error recovery is the only form of error recovery used. However, forward error recovery is often more appropriate for predictable and well-defined errors (e.g. simple hardware component faults and input errors), leaving backward error recovery to be used just for faults whose consequences have not been, or cannot be, accurately specified (such as unmasked residual design faults).

Much of our latest work related to error recovery, and on general linguistic support for software fault tolerance, in fact exploits various characteristics of C++, a language that has been used extensively at Newcastle in connection with work on distributed systems. The work takes advantage of the recent extension of C++ to include generic classes and functions ("templates"), and exception handling ("catch" and "throw"). For example, Robert Stroud and Cecilia Calsavara have shown that the existence of these facilities make it possible to implement both forwards and backwards error recovery in C++ in the form of reusable components that separate the functionality of the application from its fault tolerance[Rubira-Calsavara and Stroud 1993]. More generally, Jie Xu has shown that such facilities have the potential of providing a convenient means of achieving high levels of software reuse. This would apply both to general software components implementing various fault tolerance strategies (including generalizations and combinations of recovery blocks, and N-version programs[Avizienis 1975; Chen and Avizienis 1978], and encompassing the use of parallelism) and to application-specific software components embedded in fault-tolerant programs [Xu and Randell 1993].

5 - DISTRIBUTED SYSTEMS

During the late 1970s our interest in extending our ideas on fault tolerance to distributed systems led us to search, in fact in vain, for a suitable distributed system to use as a basis for experiments. We had for some years been using UNIX as our development platform, and in 1982 almost by accident we came up with a scheme for joining together multiple UNIX systems so as to form a transparently-distributed system [Brownbridge et al. 1982]. This involved inserting a software layer (for which we coined the name "The Newcastle Connection") between the UNIX kernel and the application programs running on this kernel. The Connection layer, whose design is mostly due to Lindsay Marshall, secretly intercepted the calls each application made to the kernel, and if necessary redirected them to the UNIX kernel on another machine. Symmetrically, it also had the task of receiving the response from this distant kernel and passing it up to the local application program as if it had come from the local kernel. The Connection in effect took advantage of certain characteristics of the UNIX system call interface, especially the recursive file naming scheme it supported [Black et al. 1987].

This idea was very successful, and led to a number of other developments which similarly operated secretly at the UNIX system call level. For example, by this means we explored and implemented schemes for load balancing, for hardware fault masking via triple modular redundancy, and for designing multi-level secure systems [Randell 1983]. As a result for several years the main Newcastle reliability group did little language-oriented work, though during 1981-84 an allied project directed by my colleague Tom Anderson undertook a major exercise to provide a quantitative assessment of the cost-effectiveness of software fault tolerance based on recovery blocks and recoverable "dialogues" between multiple processes. (A dialogue provides a restricted form of the concept of a conversation.) This project involved the implementation of a model Naval Command and Control System. The model, though only partial, was as realistic as possible, and experiments conducted with it led to the conclusion that "by means of software fault tolerance a significant and worthwhile improvement in reliability can be achieved at acceptable cost"[Anderson et al. 1985].

The UNIX-based Newcastle Connection work was followed by a return to a more language-oriented approach, led by my colleague Santosh Shrivastava, and to the explicit use of object-oriented programming techniques. His Arjuna project has produced a distributed programming system based on C++ [Shrivastava et al. 1991]. This system provides a number of fault tolerance provisions, each in the form of a class definition to be used by application programmers, though it does not provide any specific facilities for software faults per se.

There were a number of similar projects at other places, but what distinguished the Arjuna project was the extent to which it made use of the existing object-oriented language facilities provided in C++. For example in Arjuna objects can inherit persistence or atomicity characteristics, and no changes were made to the C++ compiler or run time system. This characteristic of Arjuna is if course very convenient, not least because of the portability that it implies, and the system has now been made widely available. Unlike the Newcastle Connection's provisions, its various fault tolerance provisions can easily be applied selectively, just to such parts of an application, i.e. to such object classes, as the application programmer chooses. Moreover the predefined strategies that the classes employ can be over-ridden if desired.

6 - REFLECTION

The exception handling structures, and the structures developed in the Arjuna project that were outlined above in Sections 4 and 5 greatly facilitate the use of certain types of fault tolerance, and enable many implementation details to be hidden from application programmers. However, there remain certain strategies and types of structuring that cannot be implemented entirely in a language like C++ even given such mechanisms as generic functions and inheritance. Instead, the programmer who wishes to employ these strategies has to obey certain conventions. For example, the application programmer who wishes to make use of Arjuna's provisions for atomic objects has to state explicitly when an operation modifies the state of an object.

Adherence to such conventions can be automated by embodying them into a somewhat enhanced version of C++ and using a pre-processor to generate conventional C++ programs automatically. For example, in Arjuna such a preprocessor (the "stub generator") is used to provide distribution transparency. Although the preprocessor approach can be quite practical it does have disadvantages. In particular the language provided to application programmers becomes non-standard, and programmers have in some circumstances during program development to work in terms of the C++ program generated by the pre-processor, rather than of the extended C++ program that they had written. The alternative, that of leaving it to the programmer to adhere to the conventions is of course a fruitful source of residual program faults.

Basically, the problem is that typical object-oriented languages such as C++, for all their merits, do not provide adequate expressibility to enable the implementation of certain types of fault tolerance. Specifically they do not provide access, from within the object-oriented language, to some of the basic mechanisms involved in supporting the language. For example, one could imagine a scheme for transparently-invoked triple modular redundancy based on intercepting and repeating selected method invocations, and a scheme for transparent persistence that involved temporary modifications to the way in which objects were created and later destroyed. The contrast to be drawn, of course, is with the relative ease with which equivalent sorts of

interception were utilized, albeit in very restricted form, at the UNIX system call level by the Newcastle Connection.

Recent developments in the object-oriented language world, under the term "reflection" [Maes 1987], show considerable promise in this regard [Stroud 1992]. The notion of reflection comes from logic, and concerns the ability of a program to observe and manipulate the computational behaviour of the system. The idea is to define the basic abstractions of the programming language, such as objects, classes and methods, as "meta-objects", and to provide means in the language by which the definitions of these meta-objects can be changed dynamically at run time. In practice, what is being done is to introduce an additional level of indirection in the mechanisms used to invoke certain basic facilities, and to provide linguistic means of intercepting and temporarily diverting this indirection [Agha et al. 1992].

However quite what reflective capabilities are needed for what forms of fault tolerance, and to what extent these capabilities can be provided in more-or-less conventional programming languages, and allied to the other structuring techniques outlined in this paper, remain to be determined. In particular, the problems of the combined provision of significant software fault tolerance and hardware fault tolerance, and of evaluating their cost-effectiveness, are likely to require much further effort.

7 - DELEGATION

In a conventional object-oriented language, all the objects of a given class must follow the same behaviour. But many entities in the real world that a computer has typically been designed to interact with, e.g. for monitoring and control purposes, exhibit quite different types of behaviour in their lifetime, in particular depending on whether or not they are functioning correctly.

It is desirable to find means whereby objects representing such entities can make such behaviour changes very explicit, rather than simply through assignment to variables that are interpreted as "flags". (Such a technique means that behaviour changes have to be implicitly represented by flag-checking code within all relevant methods.) One could define a series of different (sub)classes related to a single external entity, each corresponding to a different type of behaviour by the entity. But class-based languages such as C++ do not allow a program to change the class of an object.

In so-called delegation-based programming languages, such as Self [Ungar and Smith 1987] and Actor [Agha 1986; Hewitt 1977], objects are viewed as prototypes (or exemplars) that delegate their behaviour to related objects, called delegates. Such languages, in contrast to class-based languages therefore, provide dynamic, explicit and per-object behaviour sharing. Hence one can directly reflect behaviour changes by the external entities, by delegating operations to different objects representing different types of behaviour. But the fact that these different objects are intended as representing the same entity at different stages in its lifetime are obscured.

My student Cecilia Calsavara has therefore recently been exploring a means of implementing delegation in C++, in order to improve the structuring of software whose task is to control a complex and failure-prone environment, her intended experiments actually being aimed at the control of a large model railway! And what we have realized is that by such means one can, in

effect, link together separate class hierarchies, each providing a different hierarchical view of what is intended to be the same reality [Davis 1992; Johnson and Zweig 1991]. By such means one could specify different aspects of a system's behaviour, and the intended constraints on such behaviour, each in the most appropriate terms. This is essentially the aim of the early work on iterative multi-level modelling described in Section 2, except that it is now clear to me that the original notion of levels of abstraction implied an inappropriate linearization and sequencing of a set of what should be regarded simply as co-existing abstractions. Thus I feel confident that this technique of combining class abstraction and delegation is of significant potential - though as with reflection, much work remains to be done.

8 - REACTIONS TO SOFTWARE FAULT TOLERANCE

I mentioned at the outset that the ambiguity of the title of this paper was intended, since I wished to include a brief comment on attitudes to the subject of this paper. There is, it must be admitted, something rather unsettling about the subject of software fault tolerance. This it because it involves such an explicit recognition that residual design faults might exist in systems whose construction one might claim "merely" requires careful logical thinking and attention to detail.

There has therefore been a strong demand for evidence from carefully-controlled experiments concerning the cost/effectiveness of the various forms of software fault tolerance. And indeed much effort has gone into a number of such experiments, especially ones related to N-version programming. I will not attempt to summarize here, other than to say that the experimental results in my view have been quite encouraging - though they have not surprisingly confirmed that some controversial claims about the effectiveness of software fault tolerance, based on naive assumptions about statistical independence, were highly over-optimistic.

It is my distinct impression that rather less effort has gone into providing solidly-based quantitative assessments of the cost/effectiveness of the various other techniques that are of importance to achieving high levels of software reliability. This is despite the equally controversial nature of some of the claims concerning, for example, the use of formal verification [Fenton 1993].

However, quoting an earlier paper of mine: "Perhaps the best argument that can be deployed in favour of the use of software fault tolerance is one due to Michael Melliar-Smith. Formal verification, testing and fault tolerance each depend on a set of assumptions. For example, verification depends, inter alia, on the adequacy of the theorem proving tools and techniques, testing on the choice of tests and the care with which their results are inspected, and fault tolerance on the completeness of the error detection scheme and the degree of actual design diversity achieved. In addition, the three schemes all depend on having an adequate system specification. By using formal verification and testing and fault tolerance the number of assumptions on which belief in the reliability of a system rests can be reduced to the absolute minimum, namely that the system specification is satisfactory, for this is the single assumption that is shared by all three approaches."

9 - CONCLUDING REMARKS

This paper in both its text and its references has taken an overly Newcastle-centred view of the subject of design fault tolerance, and has not attempted to cover the very considerable contributions that other groups, most notably Al Avizienis' group at UCLA, have made to the subject. And of course our work has not been pursued in isolation, and we have benefited greatly from work done elsewhere. But the intent was not to give a complete survey. Rather, my aim has been to attempt to explain how our research has evolved, and in particular to describe how it has been influenced at various points, albeit sometimes rather belatedly, by the now very fashionable topic of object-oriented systems - a topic which I should point out has also recently celebrated its Silver Jubilee, just ahead of LAAS.

One final remark: my current research, which largely stems from a very pleasant sabbatical spent at LAAS in the Winter and early Spring of 1991 and which continues to be carried out in close collaboration with Jean-Charles Fabre, very much fits the implied theme of this talk. This research concerns how the fragmentation-redundancy-scattering technique for achieving combined reliability and security developed by LAAS can best take advantage of object-oriented techniques. However this research is the subject of other, more technical, papers.

10 - ACKNOWLEDGEMENTS

Self-evidently, the work I have discussed in this paper has been carried out in collaboration with a considerable number of colleagues over the years - it is a pleasure to acknowledge the debt I owe to them all. The work has been sponsored by the UK Science and Engineering Research Council and the UK Ministry of Defence, and for the last few years has been pursued largely in connection with the ESPRIT Basic Research Action on Predictably Dependable Computing Systems.

11 - REFERENCES

[Agha 1986] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems,* Cambridge, Massachusetts, MIT Press, 1986.

[Agha et al. 1992] G. Agha, S. Frolund, R. Panwar and D. Sturman. "A Linguistic Framework for Dynamic Composition of Fault Tolerance Protocols," in *Proc. 3rd IFIP Working Conf. on Dependable Computing for Critical Applications,* pp. 345-364, Mondello, Sicily, 1992.

[Anderson et al. 1985] T. Anderson, P.A. Barrett, D.N. Halliwell and M.R. Moulding, "Software Fault Tolerance: An evaluation," *IEEE Trans. Software Engineering*, vol. SE-11, no. 12, pp.128-134, 1985.

[Anderson and Kerr 1976] T. Anderson and R. Kerr. "Recovery Blocks in Action: A system supporting high reliability," in *Proc. 2nd Int. Conf. On Software Engineering,* pp. 447-457, San Francisco, IEEE, 1976.

[Avizienis 1975] A. Avizienis. "Fault-Tolerance and Fault-Intolerance: complementary approaches to reliable computing," in *Proc. Int. Conf. on Reliable Software,* pp. 458-464, Los Angeles, California, 1975.

[Black et al. 1987] J.P. Black, L.F. Marshall and B. Randell, "The Architecture of UNIX United," *Proc. IEEE: Special Issue on Distributed Database Systems*, vol. 75, no. 5, pp.709-718, 1987.

[Brownbridge et al. 1982] D.R. Brownbridge, L.F. Marshall and B. Randell, "The Newcastle Connection, or - UNIXes of the World Unite!," *Software Practice and Experience*, vol. 12, no. 12, pp.1147-1162, 1982.

[Chen and Avizienis 1978] L. Chen and A. Avizienis. "N-Version Programming: A fault-tolerance approach to reliability of software operation," in *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8),* pp. 3-9, Toulouse, France, 1978.

[Davis 1992] S.R. Davis, "C++ Objects that Change their Types," *J. of Object-Oriented Programming*, vol. 5, no. 4, pp.27-32, 1992.

[Fenton 1993] N. Fenton, "How Effective Are Software Engineering Methods," *Journal of Systems and Software (Elsevier, US)*, vol. (To appear),1993.

[Hewitt 1977] C. Hewitt, "Viewing Control Structures as Patterns of Passing Messages," *J. of Artificial Intelligence*, vol. 8, no. 3, pp.323-364, 1977.

[Horning et al. 1974] J.J. Horning, H.C. Lauer, P.M. Melliar-Smith and B. Randell. " A Program Structure for Error Detection and Recovery," in *Proc. Conf. on Operating Systems, Theoretical and Practical Aspects (Lecture Notes in Computer Science, vol. 16),* pp. 171-187, IRIA, Springer, 1974.

[Horning and Randell 1973] J.J. Horning and B. Randell, "Process Structuring," *ACM Computing Surveys*, vol. 5, no. 1, pp.5-30, 1973.

[Johnson and Zweig 1991] R.E. Johnson and J.M. Zweig, "Delegation in C++," *J. of Object-Oriented Programming*, vol. 4, no. 7, pp.31-34, 1991.

[Lauer and Needham 1978] H.C. Lauer and R.M. Needham. "On the Duality of Operating System Structures," in *Proc. 2nd Int. Symp. on Operating Systems,* IRIA, 1978.

[Maes 1987] P. Maes. "Concepts and Experiments in Computational Reflection," in *Proc. Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87) (ACM SIGPLAN Notices, 22,10),* pp. 147-155, 1987.

[Randell 1975] B. Randell, "System Structure for Software Fault Tolerance," *IEEE Trans. on Software Engineering*, vol. SE-1, no. 2, pp.220-232, 1975.

[Randell 1983] B. Randell. "Recursively Structured Distributed Computer Systems," in *Proc. 3rd Symp. on Reliability on Distributed Software and Database Systems,* pp. 3-11, Clearwater Beach, Florida, IEEE, 1983.

[Rubira-Calsavara and Stroud 1993] C. Rubira-Calsavara and R. Stroud. *Forward and Backward Error Recovery in C++,* Tech. Report 417 (in press), Dept. of Computing Science, University of Newcastle upon Tyne, 1993.

[Shrivastava et al. 1991] S.K. Shrivastava, G.N. Dixon and G.D. Parrington, "An Overview of the Arjuna Distributed Programming System," *IEEE Software*, vol. 8, no. 1, pp.66-73, 1991.

[Shrivastava et al. 1987] S.K. Shrivastava, L.V. Mancini and B. Randell. "On the Duality of Fault-Tolerant System Structures," in *Workshop on Experience with Distributed Systems (Lecture Notes in Computer Science, Vol. 309),* pp. 19-37, Kaiserslautern, Springer-Verlag, 1987.

[Stroud 1992] R.J. Stroud. "Reflection and Transparency in Distributed Systems," in *Proc. Fifth ACM SIGOPS European Workshop on Models and Paradigms for Distributed System Structuring,* Mont St Michel, France, 1992.

[Ungar and Smith 1987] D. Ungar and R.B. Smith. "Self: The Power of Simplicity," in *Proc. Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87) (ACM SIGPLAN Notices, 22, 10),* pp. 227-242, 1987.

[Xu and Randell 1993] J. Xu and B. Randell. *Object-Oriented Software Fault Tolerance: Reusability versus Design Diversity,* Tech. Report (in preparation), Dept. of Computing Science, University of Newcastle upon Tyne, 1993.

[Zurcher and Randell 1968] F.W. Zurcher and B. Randell. "Iterative Multi-Level modelling: A methodology for computer system design," in *Proc. IFIP Congress 68,* pp. D138-D142, Edinburgh, 1968.