Choices in the Design of a Language Pre-Processor for Specifying Redundancy

Jonathan M. Smith

Gerald Q. Maguire, Jr.

Computer Science Department
Columbia University
New York, NY 10027

Technical Report No. CUCS-270-87

Choices in the Design of a Language Pre-Processor for Specifying Redundancy

Jonathan M. Smith

Gerald Q. Maguire, Jr.

Computer Science Department
Columbia University
New York, NY 10027

Technical Report No. CUCS-270-87

1. Introduction

RB is a set of constructs which can be added to a programming language to specify *redundancy* in a program; a complete implementation of the macro processor and its support library is underway at Columbia University. While the main focus of our research is fault tolerance, we have made some observations which we believe are of interest to the programming language community.

We decided that implementation of RB as a macro preprocessor for some underlying programming language (we've chosen C) best suited our needs and design goals. The remainder of this report details those goals and our decisions.

1.1. Goals of RB

The goal we have for RB is to specify redundant portions of programs; the specification and its meaning are discussed in another report [1] where the fault tolerant aspects of the language constructs are emphasized. We recap three portions of that material here in order that the design issues be properly motivated and that this report be self-contained. These are: what is fault-tolerance, how does RB attempt to address it, and what does RB look like.

1.2. Fault Tolerance

There are two approaches to constructing highly reliable systems, *fault intolerance*, and *fault tolerance*. The basic notion of fault intolerance is that the systems should be as fault-free as the construction process allows, by using careful design and quality components. In software, *failures*, and *faults* causing errors, can be reduced or removed by verification, precise specifications combined with testing, and design methodologies such as structured programming techniques. Experience [2] has shown that these techniques are insufficient, i.e., "bugs" remain. Gerhart and Yelowitz [3] point out that applications of the methods may themselves have bugs.

Hence, an alternate point of view is that faults exist, and that we must provide reliable systems in spite of them. This approach is called *fault-tolerance*. The basic idea behind all of fault tolerance is the use of *redundancy*, which is used to *detect faults* and *mask failures*.

1.3. Redundancy

Redundancy is used in the construction of fault-tolerant software. There can be redundancy in space, where multiple identical copies of a program exist; redundancy in time, where multiple applications of a software system are attempted; and redundancy in logic, where multiple methods of achieving a specified result are applied. Avizienis and Kelly [4] suggest the notation T / H / S, for Time / Hardware / Software, to describe the different types of redundancy possible in a computation.

In each of these dimensions, RB allows a programmer to specify the desired level of redundancy, and in the case of logic, the means for achieving it. The use of redundancy relies on the *statistical independence* [5] of the failures of the component subsystems.

1.4. Software Fault Tolerance

Hardware faults can be the result of physical degradation [6] or they can result from poor design. Software can only have faults due to mistakes in design and implementation; the logic encoded by a program does not degrade with time. Thus, assuming a fixed underlying system (that is, a fixed set of hardware and a set of conditions that retry will not change), the only variation we can effect is in the program logic. Logic redundancy, that is, a multiplicity of methods for computing results, has been applied as a technique to achieve reliable[1] software. Statistical

1.) Software reliability has an extensive literature; Musa, Iannino,

independence of method failures is often assumed[2] in order to derive reliability estimates.

Several techniques used to provide fault-tolerance in software systems have been developed. The major techniques are N-version Programming [14-16] and the Recovery Block [17-21] scheme. The Consensus Recovery Block [22] is a synthesis of these two methods. We have chosen the Recovery Block as the basis for RB's logic redundancy specification; see Smith and Maguire [1] for justification. The Recovery Block is a language construct analogous to a block in block structured programming languages, in that a block has both private variables and access to global variables (those declared external to the block). The recovery block either reliably updates the external variables, or fails. The scheme is conceptually similar to the "standby spare" technique used in hardware. *N* alternate methods of passing an *acceptance test* are provided. The first such method is referred to as the *primary;* they are rank-ordered based on some metric such as observed performance. Each method is tested, and the first which passes the acceptance test provides the result of the recovery block. Each alternate is *guaranteed* to begin execution with the system's state as it was when the recovery block was entered. This prevents previously executed alternates from damaging state assumed by the currently executing alternate. Assuming that the acceptance test performs perfectly[3], the Recovery Block method fails on inputs where all methods fail the acceptance test.

1.5. RB Syntax

An example of a Recovery Block designed to perform a numerical calculation is given in Figure 1, using RB notation[4]:

```
#define TOLERANCE (1.0e-5)
#define EQUAL(_a,_b) \
   ((((_a)-(_b))/(_b)) < TOLERANCE)

double ft_sqrt( x )
double x;
{
   double y, newton(), bisection();

   ENSURE EQUAL( y*y, x )
   BY
      y = newton( x );
   ELSE_BY
      y = bisection( x );
   ELSE_ERROR
      fail();
   END

   return( y );
}
```

Figure 1: Recovery Block using RB

The goal of the routine is to provide an output which is the square root of the numerical argument. The ENSURE keyword indicates that what follows is to be used as the acceptance test for this recovery block; the acceptance test is an arbitrary sequence of language statements which results in a Boolean value. In this example, we have defined a macro EQUAL which defines equality in terms of a relative error measure to make the example more realistic.

The BY keyword ends the specification of the acceptance test and denotes the beginning of the primary alternate, which can consist of arbitrary program text which is intended to perform the computation which the acceptance test is verifying, in this case the square root of x. ELSE_BY is used to specify further alternates; the ELSE_ERROR keyword specifies arbitrary code to be executed upon failure of the set of alternates to produce an acceptable answer. The END keyword terminates the recovery block syntactically.

RB performs a source-to-source translation; source text between keywords is copied verbatim. Hence, any syntactically correct construction in the base language, e.g., compound statements, may be used in the alternates.

RB provides several features to specify redundancy other than the redundant logic expressed by the recovery block method. In order to expose these features, we'll expand on the previous example with the routine in Figure 2:

---

and Okumoto [7] provide a detailed reference on software reliability; the survey paper by Ramamoorthy and Bastani [8] discusses a wide variety of models and issues.

2.) Knight and Leveson's [9-11] experimental work suggests the independence assumption of multiple software versions is not upheld in practice. Eckhardt and Lee [12] provide a theoretical analysis of coincident errors and their effects on some software fault tolerance schemes. Recent work by Littlewood and Miller [13] has shown that while multiple versions using a single methodology have little hope of independence with respect to their failures, multiple methodologies offer some promise.

3.) Scott, Gault, and McAllister [23] and Scott, et al. [24] have shown that reliability of software can be improved even with acceptance test failure rates of up to 25 per cent. Cha, *et al.* [25] have shown that Self-Checks (a generalization of the Acceptance Tests used by Recovery Blocks) can be effective in finding faults. However, there is difficulty both in the writing of the Self-Checks and their placement within the program structure. They also note a great variation in the ability to write effective self-checks, and the efficacy of combining code-based checks with specification-based checks compared to specification-based alone.

4.) The notation almost exactly follows Randell's. [17]

```
#define TOLERANCE (1.0e-5)
#define EQUAL(_a,_b)\
   ((((_a)-(_b))/(_b)) < TOLERANCE )

double ft_sqrt( x )
double x;
{
   double y, newton(), bisection();

   ENSURE EQUAL( y*y, x )
   BY 2 REPETITIONS OF
      y = newton( x );
   ELSE_BY 2 REPLICATES OF
      y = bisection( x );
   ELSE_ERROR
      fail();
   END

   return( y );
}
```

Figure 2: RB: Repetition and Replication

1.5.1. Repetition

Intermittent failures can be dealt with by retry, thus we would like to specify that multiple REPETITIONS OF a computation are to take place. In our current design, we make multiple attempts to pass the same acceptance test, exposing a user to a danger of poor quality acceptance tests.

1.5.2. Replication

Another possibility for robust execution of computations is to have multiple *identical* copies of a piece of software executing, as is specified with the REPLICATES OF keyword used in Figure 2. If multiple hardware nodes are available and utilized to execute the replicates, replication reduces the likelihood of a hardware failure destroying the results of a correct software alternate. In addition, it may take advantage of differences in processor loads for better real time performance if we synchronize by accepting the results of the first successful execution. We have not included syntax to specify a synchronization mechanism, but further research may require this. In any case, it is easy to add, e.g., a WITH SYNCHRONIZATION keyword to specify either arbitrary code or a limited set of synchronization primitives. Note also that if we specify an ENSURE TRUE acceptance test and REPLICATES OF a single alternate, we have pure replicates of a computation.

1.5.3. Combinations

Combinations of redundancy specifications lead to interesting behaviors that may be exploited by the programmer. For example, Figure 2 specifies 2 REPETITIONS OF `newton()` and 2 REPLICATES OF `bisection()`. While we don't currently make provision for arbitrary nesting in RB, we do allow a fashion of nesting for combinations of specification of replication and specification of iteration. For example, we could specify 2 REPLICATES OF 2 REPETITIONS OF `newton()`; in this case we would create two copies of the computation, each of which would make up to two attempts to pass the acceptance test. 2 REPETITIONS OF 2 REPLICATES OF would perform analogously.

2. Design of RB

RB has been designed as a C pre-processor; this approach is much like that of a sophisticated macro [26] processor. This approach has been used for example, in Stroustrup's [27] C++ programming language; its implementation consists of a C preprocessor coupled with a powerful support library.

RB's syntax is specified using common UNIX [28] tools, in particular *lex*[29] and *yacc*,[30] for specifying lexical analyzers and grammars. RB processes an input file which consists of intermixed C code and RB keywords. If the input is syntactically acceptable to RB, the preprocessor generates an output file which mixes the C code from the input with calls to a support library. It is this support library which defines the mapping between the language syntax and semantics, so that the programmer can precisely specify what is to occur.

For example, the degree of replication may be specified; if this is to be supported as a distribution across multiple hardware nodes, the support library must provide a facility which allows relevant state to be transferred [31] to a specified remote machine. Consequently, we require a mechanism which allows synchronization of the replicates.

As the implementation of RB's support library is not complete, we cannot yet offer observations about the efficacy of the design; use typically points out failures in the design process most effectively. However, we feel that several choices in our design are well-supported, in particular that of RB as a language preprocessor. This is argued in the next section.

3. Design Choices

As was mentioned in the introduction, our research is in fault-tolerance, not programming languages. Consequently, we have a set of goals for RB which may be different than other programming language systems:

1.) We want to focus on fault-tolerance.

2.) We need to be able to measure reliability.

3.) The design should be language independent.

4.) We need control over I/O.

The applicability of RB to these goals is argued in the

following sections.

3.1. Focus

The focus on fault-tolerance is important to us; we have neither the time, interest, nor inclination to construct a complete programming language system. However, as our research involved the development of fault-tolerant software, we needed a vehicle to carry out some experiments in applications of redundancy. We identified a small number of constructs which expressed the types of redundancy we believe possible in software systems. Reasoning about the support necessary for the constructs convinced us that we could implement the constructs with a few primitives, for, e.g., address space copying, remote execution, and process synchronization.

As the other details did not concern us and the constructs and primitives were so few, building on top of an existing language offered programmers all the features they expect from a language in addition to the redundancy specification offered by RB, as well as limiting our efforts.

This approach has proved fruitful, as the parser and "C" code generator were operational in about two weeks of programming; our current effort is focused on the implementation of the support library.

3.2. Reliability

In order to make any claims about increases in reliability offered by RB, we need to be able to measure component reliabilities. For hardware, and hence replication, these reliability figures and their measurement are well understood; measuring the effectiveness of the retry effected by repetition is not particularly difficult either. The difficulty is the measurement of reliability [32] of software. The topic has a large literature, as mentioned in the introduction, but in many cases, e.g., the time-domain models [33, 34] the data is difficult to gather because programmers must cooperate, recording failure times, etc. In addition, the validity of many models is only established for large programs, e.g., those of greater than 5000 [8] source lines. Data-domain modeling [22] is one technique to simplify reliability data gathering.

In order to show reliability gains, we must first measure reliability and then compare the reliability figures from the "fault-tolerant" implementation combining a set of versions to the best performing single version in that set. In order that the reliability figures for the single and composite versions be comparable, we would like to perturb the single-versions as little as possible. This is aided by our terse set of constructs; they cause only minimal perturbation in software complexity metrics applied to the single versions and the composites. In addition, as there is no need for rewriting of software to use RB, there is little chance of introducing new errors into existing code. Such errors are often caused by unfamiliarity with language features and subtleties introduced by the complexity of a new syntax and semantics.

3.3. Language independence

The design of RB as a language preprocessor embodying a few simple concepts about redundancy enables it to be adapted readily to almost any block-structured programming language, since the RB processor is concerned solely with its own syntax and performs simple text-to-text transformations. The hard work is done by the support libraries and the underlying programming language system. We are quite sure that a variant form of RB, or at least access to its support library, can be implemented in other language environments, e.g., Common Lisp. [35]

One limitation on portability may be the semantics implied by the support library; when it is complete it must be examined to see which design choices affect portability. One choice we have made is that we wish to control certain aspects of the I/O behavior of program modules, particularly the writing of data. This may significantly reduce our choice of implementation environments and is discussed in the next section.

3.4. Control of I/O

One of the design choices of the C programming language [36] is the deliberate omission of I/O primitives as part of the language definition. I/O is typically accomplished by procedure-like requests made of the underlying operating system called system calls; typically a "standard I/O" library is implemented using these system calls. References to these calls are resolved when the program image is created, by reference to object libraries accessible to the linkage editor. This particular design is amenable to customization, in the following sense; references to system routines such as `write()` can be replaced by routines which buffer I/O until a synchronization primitive is successfully executed; this can be accomplished *transparently*, that is with no change in the source code. This characteristic of C allows us to control a certain class of programs whose output can be delayed; highly interactive programs would not be in this class.

3.5. Conclusions on Design

The design of RB as a C pre-processor coupled with a support library allows us to achieve our research goals. The simple set of constructs allow for significant expressive power in the specification of redundant portions of software. The RB language features are portable between base languages, as a run-time support library allows the semantics to be implemented for many run-time environments. More important, the design of the system is such that we can readily modify it, and the base language compiler

reliabilities can be readily compared with the reliability of the composite systems specified with RB, therefore allowing confirmation of our method.

4. Related Work

The ISIS system [37] provides distributed *k*-resilient objects. An example specification for a 3-resilient ticket vendor is given, which implies 4 replicates of the ticket vendor object must be created. While repetition and application of software fault tolerance schemes can be implemented, no explicit notation other than for replication is presented.

Cooper [38] discusses the use of *generators* to make replication explicit. A generator is a function which rather than returning a single value, returns a sequence of values. The degree of replication of a given module can be determined using generators, but the specification of that redundancy is implicit, by repeated use of the `add_troupe_member` primitive. Such specification is also done in Liskov's ARGUS [39] system, where replicated components of a *guardian* are created with *iterators*. Although repetition and software fault tolerance can be implemented on Cooper's CIRCUS [40] system, there is not a compact notation for expressing these. While ISIS provides checkpoints as a necessary feature, it is not clear that the design of CIRCUS is amenable to the sort of state-saving behavior necessary for backward error recovery. [19]

Welch [41] and Kim [42] have discussed distributed execution of recovery blocks as a uniform approach to fault tolerance across hardware and software. While this work shows that distribution is viable, its utility to a programmer is limited:

- The recovery blocks were handcrafted to the application, rather than being provided as a general feature to the programmer.
- There were two alternates, a primary and a secondary, for each recovery block.
- Replication was not available.
- The implementation was on a shared-memory multiprocessor, and thus communications link failures were not addressed.

RB's implementation is similar in spirit to that of Herlihy and Wing's Avalon [43] language. Like RB, Avalon is used to specify reliable distributed programs; it also uses the approach of a small number of constructs added to a base language such as C, coupled with a support library. Avalon is similar to Liskov's ARGUS system in its use of transactions and other features to provide robust execution. RB is orthogonal to Avalon in that it accomplishes its goals in a different manner; if we view an RB "block" as effecting a transaction on the external variables, the specification is a description of how to accomplish the transaction reliably. It is clear that Avalon's mechanisms could be combined with RB's, and vice-versa.

Strom and Yemini's NIL [44] programming language is designed to address the problem of reliable software for distributed systems via a combination of a higher-level programming language and a programming environment providing mechanisms [45, 46] for reliable execution. The mechanisms are transparent to the programmer, unlike RB's. Process checkpoints, messages, and inter-process message dependencies form a *redundant* description of the system's state. Replaying messages to a process restored from a checkpoint results in a consistent state. Johnson and Zwaenepoel's Sender-Based Message Logging [47] provides similar mechanisms.

RB attempts to be much more explicit about the specification of redundancy and so offers a different point in the design space. Most importantly from an implementer's point of view, RB can use modules generated in a base programming language for which complexity measures and reliability estimates have been developed. Thus, we can isolate the effect of RB in measurements of reliability.

5. Conclusions

RB provides a shorthand notation to the programmer seeking fault-tolerance through the use of redundancy. It allows the specification of three types of redundancy: time, space, and logic.

The design of RB as a language preprocessor allows us to make rapid adaptations for new experiments and environments. We have argued also that it does not interfere with the reliability measurement necessary for our research. We feel that this preservation of the base language characteristics is an important feature of RB, as it allows us to separate the effects of a better programming language from the effects of the reliability enhancements RB provides.

REFERENCES

[1] Jonathan M. Smith and Gerald Q. Maguire, Jr., "RB: Programmer Specification of Redundancy," Technical Report CUCS-269-87, Columbia University Computer Science Department (1988).

[2] Nancy G. Leveson, "Software Safety: What, Why, and How," *ACM Computing Surveys* 18(2), pp. 125-163 (June, 1986).

[3] Susan L. Gerhart and Lawrence Yelowitz, "Observations of Fallibility in Applications of Modern Programming Methodologies," *IEEE Transactions on Software Engineering* (September 1976).

[4] A. Avizienis and John P.J. Kelly, "Fault Tolerance by Design Diversity: Concepts and Experiments," *IEEE Computer*, pp. 67-80 (August 1984).

[5] Herbert Robbins and John Van Ryzin, *Introduction to Statistics*, SRA (1975).

[6] Daniel P. Siewiorek and Robert S. Swarz, *The Theory and Practice of Reliable System Design*, Digital Press (1982).

[7] John D. Musa, Anthony Iannino, and Kazuhira Okumoto, *Software Reliability: Measurement, Prediction, Application*, McGraw-Hill (1987).

[8] C. V. Ramamoorthy and Farokh B. Bastani, "Software Reliability - Status and Perspectives," *IEEE Transactions on Software Engineering* SE-8(4), pp. 354-371 (July 1982).

[9] J.C. Knight and N.G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multiversion Programming," *IEEE Transactions on Software Engineering* SE-12(1), pp. 96-109 (January 1986).

[10] John C. Knight, Nancy G. Leveson, and Lois D. St. Jean, "A Large Scale Experiment in N-Version Programming," in *Proceedings of the 15th Annual International Symposium on Fault-Tolerant Computing (FTCS-15)*, IEEE (1985), pp. 135-139.

[11] J.C. Knight and N.G. Leveson, "An Empirical study of failure probabilities in multi-version software.," in *Proceedings of the 16th Annual International Symposium on Fault-Tolerant Computing (FTCS-16)*, Vienna, Austria (July 1986), pp. 165-170.

[12] Dave E. Eckhardt, Jr. and Larry D. Lee, "A Theoretical Basis for the Analysis of Multiversion Software Subject to Coincident Errors," *IEEE Transactions on Software Engineering* SE-11(12), pp. 1511-1517 (December 1985).

[13] B. Littlewood and D. R. Miller, "A Conceptual Model of Multi-Version Software," in *Digest of Papers, The Seventeenth International Symposium on Fault-Tolerant Computing*, Pittsburgh, Pennsylvania (July 6-8, 1987), pp. 150-155.

[14] L. Chen and A. Avizienis, "N-version programming: A fault tolerance approach to reliability of software operation," in *Digest, 8th Annual International Conference on Fault-Tolerant Computing*, Toulouse, France (June 21-23 1978), pp. 3-9.

[15] A. Avizienis and L. Chen, "On the implementation of N-version programming for software fault tolerance during execution," in *Proceedings, COMPSAC 77, 1st IEEE-CS International Computer Software and Applications Conference*, Chicago, IL (November 8-11 1977), pp. 149-155.

[16] A. Avizienis, "The N-Version Approach to Fault-Tolerant Software," *IEEE Transactions on Software Engineering*, pp. 1491-1501 (December 1985).

[17] B. Randell, "System structure for software fault tolerance," *IEEE Tranactions on Software Engineering* SE-1, pp. 220-232 (June 1975).

[18] J.J. Horning, H.C. Lauer, P.M. Melliar-Smith, and B. Randell, "A program structure for error detection and recovery.," in *Proceedings, Conference on Operating Systems: Theoretical and Practical Aspects* (April 1974), pp. 177-193.

[19] T. Anderson and P.A. Lee, *Fault Tolerance: Principles and Practice*, Prentice-Hall International (1981).

[20] P.A. Lee, N. Ghani, and K. Heron, "A Recovery Cache for the PDP-11," *IEEE Transactions on Computers* C-29(6), pp. 546-549 (June 1980).

[21] Santosh K. Shrivastava, *Reliable Computing Systems*, Springer-Verlag (1985).

[22] Roderick Keith Scott, "Data Domain Modeling of Fault-Tolerant Software Reliability," Ph.D. Thesis, North Carolina State University at Raleigh (1983).

[23] R. Keith Scott, James W. Gault, and David F. McAllister, "Modeling Fault-Tolerant Software Reliability," in *Proceedings, IEEE 1983 Symposium on Reliability in Distributed Software and Database Systems* (1983), pp. 15-27.

[24] R. Keith Scott, James W. Gault, David F. McAllister, and Jeffrey Wiggs, "Experimental Validation of Six Fault-Tolerant Software Reliability Models," in *Proceedings of the 14th Annual International Symposium on Fault-Tolerant Computing* (1984).

[25] S. D. Cha, N. G. Leveson, T. J. Shimeall, and J. C. Knight, "An Empirical Study of Software Error Detection Using Self-Checks," in *Digest of Papers. The Seventeenth International Symposium on Fault-Tolerant Computing*, Pittsburgh, Pennsylvania (July 6-8, 1987), pp. 156-161.

[26] P. J. Brown, *Macro Processors and Techniques for Portable Software*, Wiley (1974).

[27] Bjarne Stroustrup, *The C++ Programming Language*.

Addison-Wesley (1986).

[28] D.M. Ritchie and K.L. Thompson, "The UNIX Operating System," *Communications of the ACM* 17, pp. 365-375 (July 1974).

[29] M. E. Lesk, "Lex — A Lexical Analyzer Generator," Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey (October 1975).

[30] S. C. Johnson, "Yacc — Yet Another Compiler-Compiler," Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, Murray Hill, New Jersey (July 1975).

[31] John Ioannidis and Jonathan M. Smith, "Notes on the Implementation of a Remote Fork Mechanism," Technical Report #CUCS-275-86, Columbia University Computer Science Department (1987).

[32] B. Littlewood, "How to Measure Software Reliability and How Not To," *IEEE Transactions on Reliability*, pp. 103-110 (June 1979).

[33] John D. Musa, "A theory of software reliability and its application," *IEEE Transactions on Software Engineering* SE-1, pp. 312-327 (September 1975).

[34] John D. Musa, "Validity of Execution-Time Theory of Software Reliability," *IEEE Transactions on Reliability* (August 1979).

[35] Guy L. Steele, Jr., *Common Lisp: The Language*, Digital Press, 1984.

[36] B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, Prentice-Hall (1978).

[37] Kenneth P. Birman, Thomas A. Joseph, Thomas Raeuchle, and Amr El Abbadi, "Implementing Fault Tolerant Distributed Objects," *IEEE Transactions on Software Engineering* SE-11(6), pp. 502-508 (June 1985).

[38] Eric Charles Cooper, "Replicated Distributed Programs," Ph.D. Thesis, University of California, Berkeley (1985).

[39] Barbara H. Liskov, "The Argus Language and System," in *Distributed Systems: Methods and Tools for Specification, An Advanced Course*, ed. H. J. Siegert, Springer-Verlag (1985).

[40] Eric Charles Cooper, "Circus: A replicated procedure call facility," in *Proceedings of the 4th Symposium on Reliability in Distributed Software and Database Systems* (October 1984), pp. 11-24.

[41] H.O. Welch, "Distributed Recovery Block Performance in a Real-Time Control Loop," in *Proceedings, IEEE Real-Time Systems Symposium* (1983), pp. 268-276.

[42] K.H. Kim, "Distributed Execution of Recovery Blocks: An Approach to Uniform Treatment of Hardware and Software Faults," in *IEEE Fourth International Conference on Distributed Computing Systems* (1984), pp. 526-532.

[43] M. P. Herlihy and J. M. Wing, "Avalon: Language Support for Reliable Distributed Systems," in *Digest of Papers, The Seventeenth International Symposium on Fault-Tolerant Computing*, Pittsburgh, Pennsylvania (July 6-8, 1987), pp. 89-95.

[44] R. E. Strom and S. Yemini, "NIL: An Integrated Language and System for Distributed Programming," *ACM SIGPLAN Notices*, pp. 73-82 (June 1983).

[45] R. E. Strom and S. Yemini, "Optimistic Recovery: An Asynchronous Approach to Fault-Tolerance in Distributed Systems," in *Proceedings of the Fourteenth International Conference on Fault-Tolerant Computing Systems* (1984), pp. 374-379.

[46] R. E. Strom and S. Yemini, "Optimistic Recovery in Distributed Systems," *ACM Transactions on Computer Systems* 3(3), pp. 204-226 (August 1985).

[47] D. B. Johnson and W. Zwaenepoel, "Sender-Based Message Logging," in *Digest of Papers, The Seventeenth International Symposium on Fault-Tolerant Computing*, Pittsburgh, Pennsylvania (July 6-8, 1987), pp. 14-21.