RB: Programmer Specification of Redundancy.

Jonathan M. Smith

Gerald Q. Maguire, Jr.

Computer Science Department
Columbia University
New York, NY 10027

Technical Report No. CUCS-269-87

ABSTRACT

RB is a programming language for specifying redundancy in various dimensions. Avizienis's notation T / H / S, for Time / Hardware / Software, describes the different types of redundancy possible in a computation: repetition (nT / H / S), redundant hardware (T / nH / S), and program (software) (T / H / nS). These can each be controlled by the programmer with RB. RB derives its name from its use of the recovery block notion to specify fault-tolerant segments of software. RB also supplies the programmer with the ability to specify degrees of replication and repetition for a given recovery block alternate; the underlying support software can then take this advice to replicate in time or hardware, based on available resources. An implementation of RB based on the C programming language is described in this paper. This implementation uses a combination of a language preprocessor for C and a runtime library to provide the desired semantics. Modification of RB to support other programming languages, or programmer specification of N-Version Programming as the decision mechanism, is straightforward.

Keywords: Fault Tolerance, Redundancy, Programming Languages.

1. Introduction

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 [1-3] ; precise specifications combined with testing [4] ; and structured programming techniques. [5] Experience [6, 7] has shown that these techniques are insufficient, i.e., "bugs" remain. Gerhart and Yelowitz [8] 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*.

Redundancy (multiple copies) is used to *detect faults* and *mask failures*. Avizienis and Kelly [9] suggest the notation T / H / S, for Time / Hardware / Software, to describe the different types of redundancy possible in a computation:

- time (repetition) nT / H / S
- space (hardware) T / nH / S
- program (software) T / H / nS

The use of redundancy relies on the *statistical independence* [10] of the failures of the component subsystems.

1.1. Software Fault Tolerance

Hardware faults can be the result of physical degradation [11] 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, 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]

1.) Software reliability has an extensive literature; Musa, Iannino, and Okumoto [12] provide a detailed reference on software reliability; the survey paper by Ramamoorthy and Bastani [13] discusses a wide variety of models and issues.

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

Several techniques used to deal with faults in software systems are discussed in the following sub-sections.

1.1.1. N-Version Programming

N-version Programming [19-21] is conceptually similar to N-modular redundancy, a hardware reliability technique. The basic idea is that an odd number of independently developed versions of a software system are executed against an input. Voting[3] is used to produce a result or determine failure. The simplest case is bitwise equality as it is independent of the application. Some practical issues of constructing such systems are being addressed by the DEDIX [22] system under construction at UCLA.

1.1.2. Recovery Block

The recovery block [23-27] 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. Assuming that the acceptance test performs perfectly, the Recovery Block method fails on inputs where all methods fail the acceptance test. The acceptance test is application-specific. Hecht [28] provides a detailed discussion of the forms such acceptance tests might take. Scott, Gault, and McAllister [29] and Scott, et al. [30] have shown that acceptance test failures can be tolerated within a certain range, in particular failure rates f, $0.0 \leq f \leq 0.25$. Scott's thesis [31] proposes a synthetic software fault tolerance method, which is discussed in the next section.

2.) Knight and Leveson's [14-16] experimental work suggests the independence assumption of multiple software versions is not upheld in practice. Eckhardt and Lee [17] provide a theoretical analysis of coincident errors and their effects on some software fault tolerance schemes. Recent work by Littlewood and Miller [18] 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.) There are several ways that the vote could in fact be implemented. For example, a 2-out-of-N scheme could be used, where as soon as 2 respondents agree on a result, the result is determined to be valid. However, the rule is typically majority vote, as this intuitively provides the greatest protection against random errors.

1.1.3. Consensus Recovery Block

Scott [31] observed that the major difficulty with the Recovery Block scheme is the acceptance test[4]; his analysis shows that it is the most crucial component of the scheme if reliability is to be increased. He proposed that N program versions be independently developed, and that an acceptance test for the programs be developed as well. The programs are run concurrently; 2-of-N voting is used to select a correct result. If no two agree the results of each version are presented to the acceptance test until one passes. The scheme avoids the acceptance test in cases where there is agreement in 2-of-N of the voters, and thus a relatively unreliable acceptance test may not have much impact on the Consensus Recovery Block's output. If the versions fail independently, the Consensus Recovery Block is more reliable than Recovery Block or N-Version Programming in almost all circumstances, and never less reliable.

1.1.4. Choice of Recovery Block Method

Scott's analysis [31] of these three methodologies concludes that the independence assumption is unwarranted[5] based on a statistical analysis of many independently-developed[6] versions of a program. His analysis of this situation is that independently-developed programs tend to fail on the same problems because the difficult cases remain difficult[7] across versions. Thus, the independence or lack of it is as much a characteristic of the problem space as it is of the solution space.

His experimental data indicated that N-Version Programming decreased the reliability and the Recovery Block scheme increased reliability for acceptance tests with reliability 0.75 or greater. The Consensus Recovery Block performed between the other two schemes, showing a reliability gain for acceptance tests with reliability 0.90 or greater.

Thus, it appears that the Recovery Block scheme currently provides the best protection against software faults. The use of the Recovery Block scheme is discussed in the next section.

4.) Cha, *et al.* [32] 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.

5.) This concurs with Knight and Leveson's results.

6.) It is possible to criticize such an experiment on the basis of the programmers being students in the same class, with the same training. Then again, academic environments place constraints on sharing of information not posed by the real world, e.g., accusations of plagiarism.

7.) This agrees with Eckhardt and Lee's [17] analysis.

2. RB: Specification of Redundancy

An example of a Recovery Block designed to perform a numerical calculation is given in Figure[8] 1, using RB notation[9]. 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. 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. 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. The execution can be modeled as a sequence of actions, where an action consists of computation of an alternate. Each alternate is associated with an instance of the acceptance test. Figure 3 illustrates the control flow in a world-line diagram; the path marked "pass" is taken if the acceptance test associated with *{Alternate #1}* is passed; external variables are then updated. Control returns to the box marked "Normal Program" in Figure 3 when the recovery block terminates, either in error or successfully.

There is opportunity for parallel execution as each recovery block alternate is assured of beginning its execution with the state of the computation as it was when the block was entered. Thus, the activities of any other alternate are irrelevant, as they are not allowed to affect the state of a given alternate. Thus, since, recovery block alternates do not communicate, they can be executed concurrently, giving rise to the model of execution illustrated in Figure 4. The END keyword marks the place in a program where the synchronization takes place. We have not decided how the synchronization method is to be selected. This could be done by using a fixed method, by selecting a library which implements a `synchronize()` primitive generated by the RB source-to-source transformation, or by creating new syntax, e.g., WITH SYNCHRONIZATION *{method}*.

The acceptance test can execute with the alternates or at the synchronization point. Results from failed tests need not be sent. We do not show the conditional control flow in examples which exhibit space redundancy; multiple alternates imply a third dimension not illustrated in the figures. Mapping concurrently executing alternates onto distinct pieces of hardware can take advantage of available space redundancy.

With support for concurrent execution, the alternates can be used to effect an N-Version Programming scheme, by setting the acceptance test to ENSURE TRUE, and implementing the synchronization as voting.

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.

2.1. Replication

One of the possibilities 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. The effect given by 2 REPLICATES OF *{Alternate #1}* is shown in Figure 5. Replication reduces the likelihood of a hardware failure destroying the results of a correct software alternate. In addition, it may more effectively serve to take advantage of differences in processor loads if we synchronize by accepting the results of the first successful execution. Note also that if we specify an ENSURE TRUE acceptance test and REPLICATES OF a single alternate, we have pure replicates of a computation; several systems of this style are mentioned in Section 3 on related work.

2.2. Iteration

Intermittent failures can be dealt with by retry, thus we would like to specify that multiple REPETITIONS OF a computation are to take place. Figure 6 illustrates the control flow if we had specified 2 REPETITIONS OF *{Alternate #1}*. In our current design, we make multiple attempts to pass the same acceptance test, exposing a user to more danger with poor quality acceptance tests.

2.3. 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()`; this is illustrated in Figure 7. 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, if we specified BY 2 REPLICATES OF 2 REPETITIONS OF *{Alternate #1}*, we would achieve the behavior illustrated in Figure 8; BY 2 REPETITIONS OF 2 REPLICATES OF would perform analogously.

8.) Referenced figures are found at the end of the document.

9.) The notation almost exactly follows Randell's. [23]

3. Related Work

Birman, et al. [33] discuss fault tolerant distributed objects in the ISIS system, developed at Cornell. In the discussion of *k*-resilient objects in ISIS, a specification for a 3-resilient ticket vendor is given, which implies 4 replicates of the ticket vendor object must be created. The ISIS specification is concerned only with replication. While repetition and application of software fault tolerance schemes can be implemented, no explicit notation is presented.

Cooper [34] discusses the use of *generators* to make the replication in his system 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. This sort of specification is also done in Liskov's ARGUS [35] system, where replicated components of a *guardian* are created with *iterators*. As with ISIS, although repetition and software fault tolerance can be implemented on Cooper's CIRCUS [36] 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 a Recovery Block implementation.

Welch [37] and Kim [38] 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 only two alternates, a primary and a secondary, for each recovery block.
- — Replication features were 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 [39] 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; it uses 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 [40] 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 [41] 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 [42] 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.

4. RB Implementation Notes

RB has been designed as a C pre-processor; this approach is much like that of a sophisticated macro processor. This approach has been used previously in the construction of programming languages, in particular, Stroustrup's [43] C++ programming language uses the approach of a C preprocessor coupled with a powerful support library.

RB's syntax is specified using common UNIX [44] tools 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 correct, RB 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, the support library must provide facilities which allow relevant state to be transferred to a specified remote machine. Thus, we also need a mechanism which allows synchronization of the replicates.

Research on several components of a support library is continuing at Columbia; while we can specify redundancy, we need a complete implementation of the support library in order to draw conclusions about the use of RB on running software.

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 use of repetition to deal with intermittent faults and the use of distribution and replication to deal with hardware faults seem well understood. This level of maturity is not evident in methods for "software fault" tolerance; the complexity of the medium makes metrics of reliability difficult to devise and apply. Thus, it often not clear how to model and measure the effects of software fault tolerance methods on system reliability. While the Recovery Block scheme for providing redundancy in the logic component of a program has been used for the version of RB described here, RB is easily modified to support other software fault tolerance schemes, and development of the support library will provide us with useful mechanisms for support of distributed computation and reliable software.

References

[1] R.W. Floyd, "Assigning meanings to programs," in *Proceedings, American Math. Society Symposium in Applied Mathematics* (1967), pp. 19-31.

[2] C.A.R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM* 12, pp. 576-580, 583 (October 1969).

[3] E.W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, N.J. (1976).

[4] R.E. Prather, "Theory of Program Testing - An Overview," *The Bell System Technical Journal* 62(10, part 2), pp. 3073-3105 (December 1983).

[5] O.-J. Dahl, C.A.R. Hoare, and E.W. Dijkstra, *Structured Programming*, Academic Press, New York (1972).

[6] F. P. Brooks, Jr., *The Mythical Man-Month*, Addison-Wesley, Reading, Mass. (1975).

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

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

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

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

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

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

[13] 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).

[14] 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).

[15] 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.

[16] 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.

[17] 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).

[18] 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.

[19] 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.

[20] 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.

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

[22] A. Avizienis, P. Gunningberg, J. P. J. Kelly, L. Strigini, P. J. Traverse, K. S. Tso, and U. Voges, "The UCLA DEDIX system: a Distributed Testbed for Multiple-Version Software," in *Digest of FTCS-15, the 15th International Symposium on Fault-Tolerant Computing*, Ann Arbor, Michigan (June 1985), pp. 126-134.

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

[24] 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.

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

[26] 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).

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

[28] H. Hecht, "Fault-Tolerant Software," *IEEE Transactions on Reliability*, pp. 227-232 (August 1979).

[29] 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.

[30] 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).

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

[32] 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.

[33] 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).

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

[35] 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).

[36] 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.

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

[38] 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.

[39] 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.

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

[41] 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.

[42] 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.

[43] Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley (1986).

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

Figures

```
#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: Simple Recovery Block example

```
#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: Using RB redundancy specification

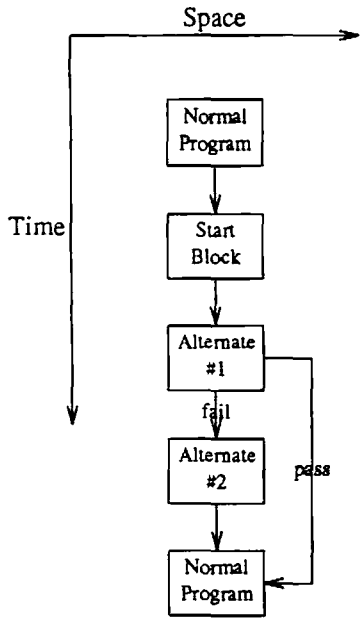

Figure 3: Sequential Execution of Alternates (Redundancy in Logic)

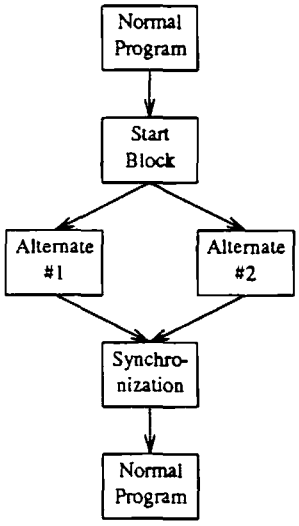

Figure 4: Concurrent Execution of Alternates (Redundancy in Logic and Space)

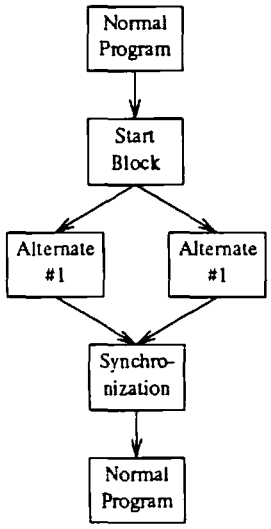

Figure 5: Replication (Redundancy in Space)

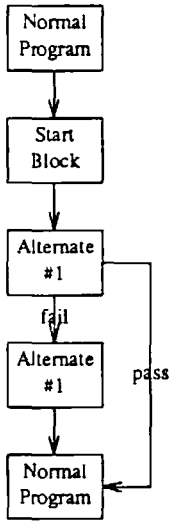

Figure 6: Repetition (Redundancy in Time)

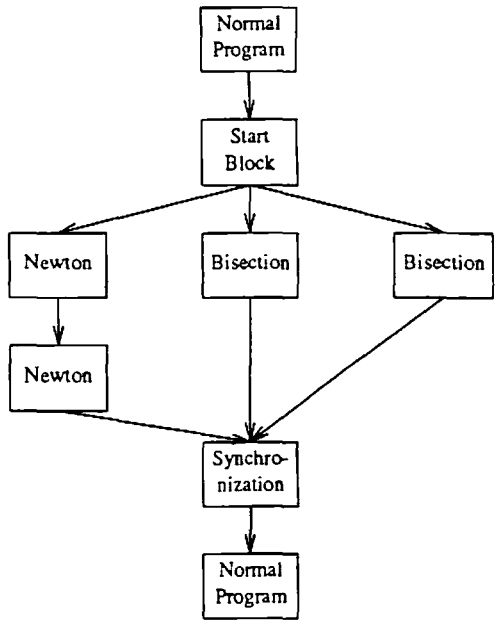

Figure 7: Example of Figure 2

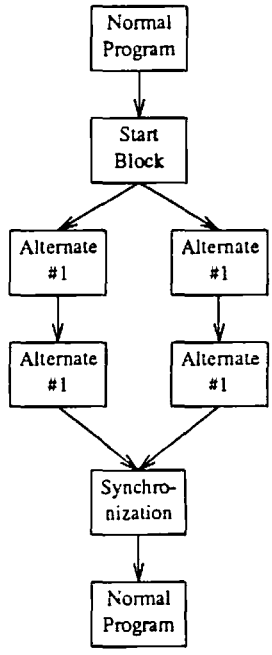

Figure 8: Replication of Repetition