-- --

The RB Language

Jonathan M. Smith

Distributed Systems Laboratory
Department of Computer and Information Science, University of Pennsylvania
Philadelphia, PA 19104-6389

Gerald Q. Maguire, Jr.

Computer Science Department, Columbia University, New York, NY 10027

ABSTRACT

Typical algorithms for distributed or parallel computations are *cooperative,* meaning that the sequential component is broken down into cooperating pieces, which are distributed across available hardware. An approach wich has recently gained some attention is *competitive* processing, where several versions of a sequential program are distributed across available processors to gain performance from algorithmic diversity. There is also potential for fault tolerance from available hardware by executing the sequential versions, called *alternatives,* on a distributed configuration. Schemes for implementing competitive concurrent processing have been described in the literature, but there is little implementation experience. RB is a practical step towards gaining such experience.

RB is a programming language for specifying alternative methods of performing a computation, where at most one of the results of the alternatives is used. Our prototype implementation uses a combination of a language preprocessor for C and a runtime library to provide the desired semantics. Using other base programming languages, e.g., Ada, or other methods of managing alternatives is straightforward.

Keywords: Distributed Execution, Parallel Processing, Programming Languages.

1. Introduction

There are many situations where there exist several alternative methods for computing a result, where a result in the most general case is a state change. When there are differences in the execution times of the methods, ‘‘Multiple Worlds’’ [9, 10] exploits this difference by picking the first process to complete and eliminating slower processes. The theoretical basis for the

This work was supported in part by equipment grants from the Hewlett-Packard Corporation and AT&T, NSF grant CDR-84-21402, and the Industrial Affiliates of the Distributed Systems Laboratory.

technique comes from order statistics [2]. Consider independent identically-distributed random variables $\{X_1, . . . , X_n\}$ which measure execution time and whose distribution function is [1]. *257text: S12 <- F; b=0,h=60,lf=2,rf=2* $F(t) = Prob(X_i \le t)$. We can compute $X^* = \min_{i=1}^{n}\{X_i\}$ which in practice is the random variable defined as the execution time of the fastest execution. A straightforward analysis shows that [1]. *257text: S12 <- F; b=0,h=60,lf=2,rf=2* $F^*(t) = 1-(1-F(t))^n$. For an

The recovery block [7] is a language construct analogous to a block in block structured programming languages. A block has both private variables and access to global variables (those declared external to the block). The 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 have typically been rank-ordered by some metric, e.g., observed performance. Assuming that the acceptance test performs perfectly, the method fails on inputs where all methods fail the acceptance test. Note that the acceptance test is application-specific; Hecht [5] provides a detailed discussion of the forms such acceptance tests might take. Recovery blocks provide a useful model to base RB's syntax and some of its semantics upon. In reality, RB is an outgrowth of an attempt to generate more practical examples of the scheme described by Smith and Maguire for ''competitive computation''. [9]

RB allows the specified alternative methods to be executed concurrently. In the paper, we begin by discussing RB's syntax and semantics. We follow with some discussion of the implementation experiences and measurements we have gathered from components that have been constructed. Section 4 relates RB to other work, and Section 5 concludes the paper.

2. RB syntax and semantics

An example of a module designed to perform a numerical calculation is given in Figure 1, using RB notation.

```
#define TOLERANCE (1.0e-5)
#define dabs(_x)\
  (((_x)<0.0)? -(_x) : (_x))
#define EQUAL(_a,_b)\
  (((_b) == 0.0) ?\
   (dabs(_a) < TOLERANCE) :\
   (dabs(((_a)-(_b))/(_b)) < TOLERANCE))

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

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

   return( y );
}
```

Figure 1: Simple RB example

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 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 2 illustrates the control flow in a world-line diagram;

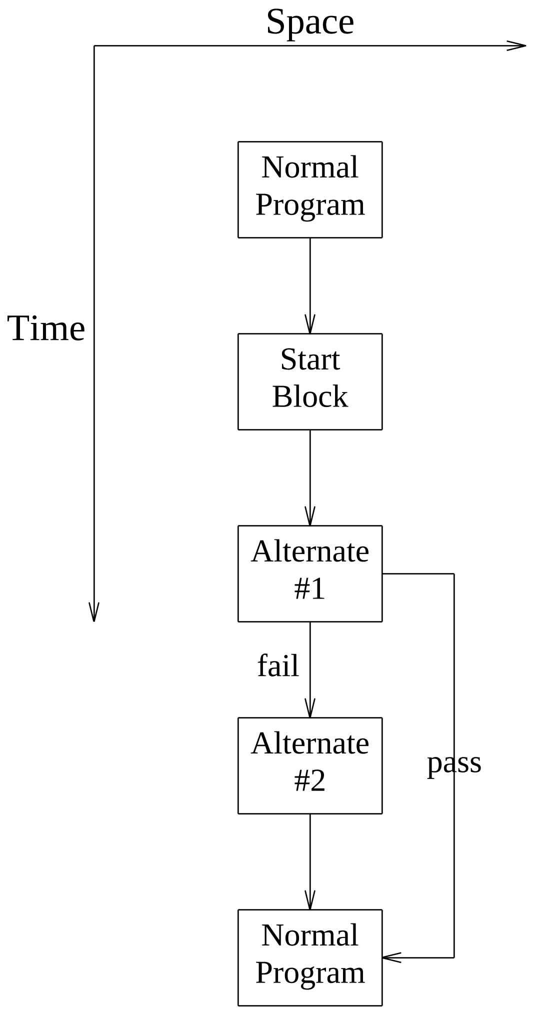

Figure 2: Sequential Execution of two RB Alternates

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 2 when the 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 with each other, they can be executed concurrently, giving rise to the model of execution illustrated in Figure 3.

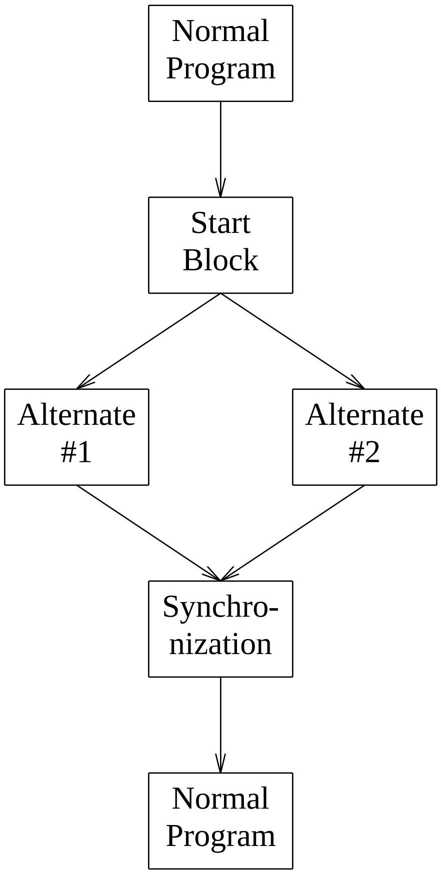

Figure 3: Concurrent Execution of two RB Alternates

The **END** keyword marks the place in a program where the synchronization takes place. Synchronization can be done by using a fixed method, by selecting a compiled library which implements a `synchronize()` primitive generated by the RB source-to-source transformation, or by creating new syntax, e.g., **WITH SYNCHRONIZATION** *{method}*. In the RB prototype, a fixed method drawn from a support library is used.

The acceptance test can execute on remote nodes with the alternates, at the synchronization point, or in both places. Results from failed tests need not be sent. We do not show the conditional control flow in examples which exhibit spatial 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 spatial 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 additional syntax to specify forms of redundancy other than the redundant logic expressed by the recovery block method. These forms are repetition and replication; the modifications to RB to support copies of the same routine are trivial; it's obvious how the modifiers we will describe could be implemented with simple text manipulation.

First, we will look at *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. The effect given by 2 **REPLICATES OF** *{Alternate #1}* is shown in Figure 4.

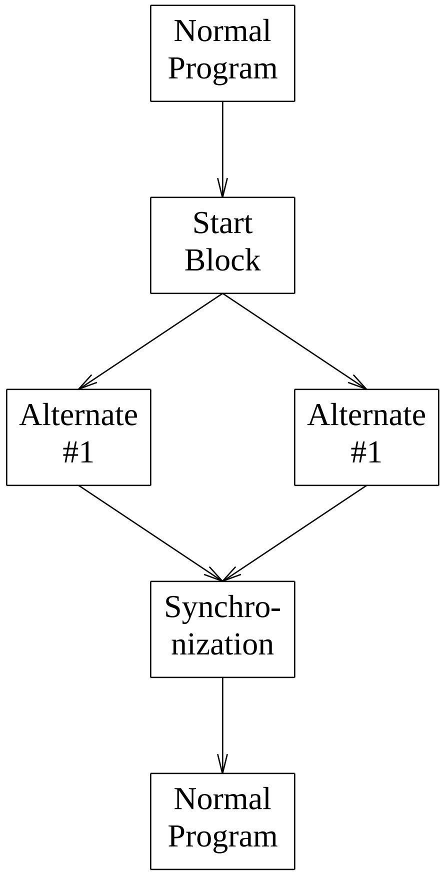

Figure 4: Concurrent Execution of Replicas

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 replication of a computation. Using replicas which have a random component gives us the induced execution time distributions described by Smith and Maguire [10].

Iteration of the creation and modification of an address space by an alternate is a potentially useful feature. Intermittent failures, for example those in subsystems such as communications media (which are often made to resemble memory devices!), can be dealt with by retry; thus we would like to specify that multiple **REPETITIONS OF** a computation are to take place. For example, we might specify 2 **REPETITIONS OF** *{Alternate #1}* as a method by which to pass the acceptance test. In our current system, we make multiple attempts to pass the same acceptance test, exposing a user to more danger with poor quality acceptance tests. That is, if the acceptance test rejects many good results, and accepts many bad ones it is deemed to be of poor quality. Repetitive application of such a test may obscure any fault-tolerating value the alternatives (repeated or not) may have had. In any case, the addition of these features took little effort and make an RB program easier to read and understand than if these methods had been employed using the sparser syntax.

3. Implementation Experience and Performance

RB has been designed as a C pre-processor; new semantics are provided with a powerful support library.

RB is built using common UNIX® tools for specifying lexical analyzers and parsers. 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 run-time system aggressively distributes alternates across available processors. Thus, the support library must provide facilities which allow relevant state to be transferred to a specified remote machine. A mechanism which synchronizes the alternates is obviously necessary.

3.1. An example

To see precisely what RB accomplishes, we will examine the output produced by running RB on an input file. We chose the program of Figure 1 as input; Figure 5 shows output generated by RB.

```
#define TOLERANCE (1.0e-5)
#define dabs(_x)\
  (((_x)<0.0)? -(_x) : (_x))
#define EQUAL(_a,_b)\
  (((_b) == 0.0) ?\
   (dabs(_a) < TOLERANCE) :\
   (dabs(((_a)-(_b))/(_b)) < TOLERANCE))

double
ft_sqrt( x )
double x;
{
   double y, newton(), bisection(), fail();
   {
 int frz_ret, frz_alternate, frz();
 char *frz_p, *frz_start_sys,
   *frz_temp_file(), *frz_get_uname();

 frz_setup_synch();
 frz_start_sys = frz_get_uname();
 frz_p = frz_temp_file();
 frz_ret = frz( frz_p );
 if( frz_ret < 0 )
 {
```

® UNIX is a registered trademark of AT&T.

```
  fprintf(stderr,
   "Fatal RB runtime error: can't checkpoint" );
  fprintf(stderr,
   "to file %s. Exiting.", frz_p );
  exit( 1 );
 }
 if( frz_ret == 1 ) /* RESTORE */
 {
 switch( frz_alternate )
 {
 case 0:

      y = newton( x );
   break;

case 1:

      y = bisection( x );
   break;

case 2:
 default:

 frz_err_synch();

      y = fail();
    }   /* close switch */
 if (  /* ENSURE condition... */
 EQUAL(y*y,x)
    )
 {
  frz_synchronize();
  frz_migrate( frz_start_sys );
 }
 else
  frz_fail(); /* probably just exit( 1 ) */
 }
 else
 {     /* in spawning process (ret == 0) */
  for( frz_alternate = 0; frz_alternate <
3 ; frz_alternate += 1 )
 {
  frz_update( frz_p, &frz_alternate,
   sizeof(frz_alternate) );
  frz_launch( frz_p );
 } /* close for */
 } /* close else */
 } /* close RB block */


   return( y );
}
```

Figure 5: RB output from previous example

The general idea is as follows: We generate a C language `switch()` statement, where the `case` instances are mapped 1-to-1 from the RB alternates. In order to facilitate correct translation, C symbols beginning with `frz` are reserved. A particular alternate is then selected by setting the `switch()` argument to the proper value. In order to instantiate several copies of the computation, we create a *checkpoint* of the process by dumping the state of the process into a file in such a way that the file is executable; a bootstrapping routine restores the registers and data segments and returns control to the caller of the checkpoint routine when this file is executed. Figure 6 illustrates the mechanics.

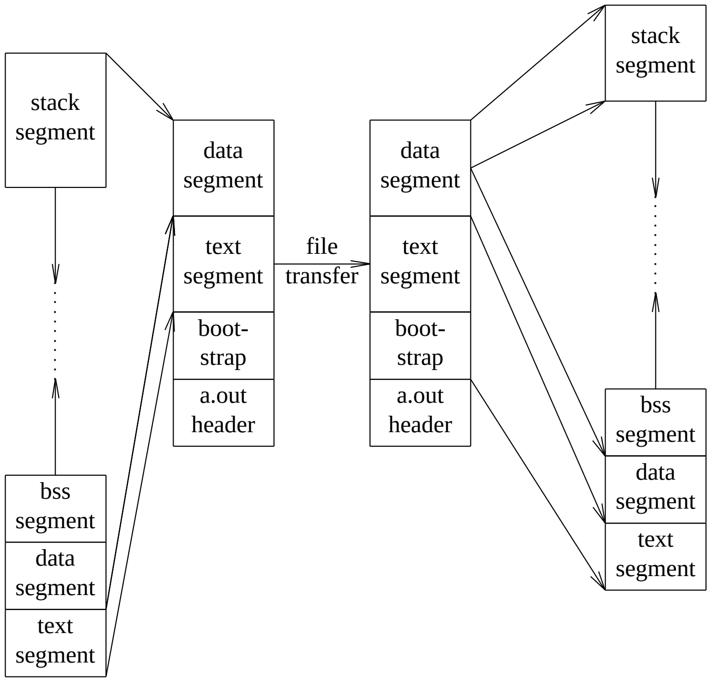

Figure 6: UNIX process migration as used for *frz()*

A return value is used to distinguish between return of control in the checkpoint and in the calling process. Our checkpointing and process migration implementation is described in detail in Smith and Ioannidis [11].

In RB, the checkpoint is created with a call to `frz()` and a control variable is iterated through the values it can take on, updating the checkpoint and transferring it to a remote system on each iteration. The advantage of this method comes from the cost of creating a checkpoint of this process; creating a checkpoint takes 0.10 second (0.18 second writing to a NFS disk), and updating an existing checkpoint can be done in a few milliseconds[1]. Each alternate performs its computation, leaves the body of the `switch()`, and is tested against the acceptance test specified by **ENSURE.** If the acceptance test succeeds, the alternate attempts to continue executing; first, it must synchronize itself with other alternates, so that *at most once* semantics are preserved. If the acceptance test fails the process exits immediately. If the process can continue, it may have to return to the system from which it was started, e.g., to deliver data to a terminal or other attached device; to do this, it calls `frz_migrate()` with an argument indicating the desired destination, in this case the saved name of the system which spawned the multiple alternates.

[1] These measurements were made on Sun-2, HP9000 Series 300, and AT&T 3B2/310 workstations. See [11] for details.

3.2. Performance

The preprocessor itself is quite fast; when applied to a large (3200 line) C program with no RB constructs (GNU Emacs' [12] *sysdep.c*) the RB prototype requires 11 seconds; this compares with the 4 seconds required by *cb,* a "pretty-printer" for the C programming language executing on the same workstation. For the example program above, the RB prototype and *cb* require 0.24 seconds and 0.14 seconds, respectively. Since most of the time consumed by the prototype is due to parsing and lexical analysis, the preprocessor can be sped up dramatically by use of optimized versions of the language tools.

The programs generated by the preprocessor have four contributing sources of execution time:

1. The execution time required by the alternative module which is being executed.
2. The time required to make copies of the state on stable storage (checkpoint). This varies somewhat with the amount of relevant state, e.g., the addressable storage.
3. The time required to transfer the checkpoints to remote systems.
4. The time required to select *at most one* of the alternative results (the error case is executed if no results are successful).

Item 1 is a function of the computation being performed; 2-4 are overhead contributed by the RB implementation. The fact that a checkpoint to NFS [8] file storage for a 30K process takes 0.2 second gives us a measure of the cost of 2 and 3; performance of our mechanism is discussed in Smith and Ioannidis [11]. The cost of 4 is quite variable, depending on the execution times of the modules and the number of messages necessary to achieve synchronization. Experiments with some special-purpose protocols indicate that this can be accomplished in about 0.10 second; synchronization usually results in another migration, so that the total execution overhead contributed by RB is around 0.5 seconds of execution time. Thus, if we have three alternates, each requiring 0.5 seconds of execution time, RB can provide better performance where a failure occurs, as the execution time will be the overhead plus the time required by the fastest alternative. In fact, if there is some variance between the execution times, the ‘‘fastest first’’ behavior will take advantage of the variance to offer improved performance. This performance improvement has been demonstrated on problems such as polynomial zero-finding [10] and searching for ‘‘semi-perfect’’ numbers.

4. Related Work

RB's implementation is similar in spirit to that of Herlihy and Wing's Avalon [6] language. RB is used to specify alternatives in order to gain performance from their diversity, and Avalon is used to specify reliable distributed programs. The approach to constructing the system is remarkably similar; Avalon also uses the approach of a small number of constructs added to a base language such as C, coupled with a support library. The notion of multiple alternatives is orthogonal to the transaction concept; if we view an RB ‘‘block’’ as effecting a transaction on the system state, the specification is a description of how to accomplish the transaction reliably. It could also be viewed as a set of ‘‘competing’’ transactions, at most one of which will take effect. It is clear that Avalon's mechanisms could be combined with RB's, and vice-versa.

Distribution of computation across several nodes offers attractive possibilities for both reliability and performance. Cooper [1] discusses the use of replicated distributed programs in order to take advantage of this potential. Cooper's CIRCUS system transparently replicates computations across several nodes in order to increase reliability. Goldberg [4] has also discussed process replication for performance improvements. Transparent replication can easily be combined with the use of parallel execution, as shown by our "REPLICATES OF" notation.

5. Conclusions

RB provides a means for programmers to specify mutually exclusive alternatives for computing a result, and to execute these alternatives across multiple processors. The features can be used both for improvements in reliability and improvements in performance. Continued development of the support library will provide us with useful mechanisms for support of distributed computations; we are currently applying the scheme to a computer vision application and the initial results have been promising. Our performance measurements on the prototype indicate that a reasonable class of problems can be addressed with this approach.

The model and implementation for the support system we presented deals successfully with the issue of some potentially conflicting side effects in the concurrently executing alternatives. Demand-based memory copying strategies may reduce the costly state copying activity which is the major cost in overhead incurred by RB.

6. Notes and Acknowledgments

Rob Strom has been extremely helpful in our gaining an understanding of the issues and approaches to building reliable systems. Discussions with Calton Pu, Yechiam Yemini, Steve Feiner and David Farber have all had a positive effect either on the ideas, their translation to text or both. Conversations with Nancy Leveson were a helpful aid to understanding the assumptions behind the various software fault tolerance methods, and how they hold up in practice. Referee's comments on related work helped us, through a critical reassessment, to make this a more solid piece of research.

7. References

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

[2] W. Feller, *An Introduction to Probability Theory and Its Applications,* Wiley, New York (1971).

[3] J. Galambos, *The Asymptotic Theory of Extreme Order Statistics, 2nd Edition*, Krieger (1987).

[4] Arthur P. Goldberg and David R. Jefferson, ''Transparent Process Cloning: A Tool for Load Management of Distributed Programs,'' in *Proceedings, International Conference on Parallel Processing* (1987), pp. 728-734.

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

[6] 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. Also Technical Report

CMU-CS-86-147

[7] 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.

[8] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and R. Lyon, ‘‘The Design and Implementation of the Sun Network File System,’’ in *USENIX Proceedings* (June 1985), pp. 119-130.

[9] Jonathan M. Smith and Gerald Q. Maguire,Jr., ‘‘Transparent Concurrent Execution of Mutually Exclusive Alternatives,’’ in *Proceedings, Ninth International Conference on Distributed Computing Systems*, Newport Beach, CA (June, 1989), pp. 44-52.

[10] Jonathan M. Smith and Gerald Q. Maguire,Jr., ‘‘Exploring ‘‘Multiple Worlds’’ in Parallel,’’ in *Proceedings, International Conference on Parallel Processing*, The Pennsylvania State University Press, St. Charles, Illinois (August 8-12, 1989), pp. 239-245.

[11] Jonathan M. Smith and John Ioannidis, ‘‘Implementing remote *fork()* with checkpoint/restart,’’ *IEEE Technical Committee on Operating Systems Newsletter*, pp. 12-16 (February, 1989).

[12] Richard Stallman, *GNU Emacs Manual, Fourth Edition, Version 17,* Free Software Foundation, Inc., 100 Mass Ave., Cambridge, MA 02138 (February 1986).