A REUSE FRAMEWORK FOR SOFTWARE FAULT TOLERANCE

K. S. Tso, E. H. Shokri, A. T. Tai
SoHar Incorporated
Beverly Hills, CA 90211

R. J. Dziegiet, Jr.
USAFA Rome Laboratory
Griffiss AFB, NY 13441

Abstract

Software errors have become the major source of failures in complex systems. The current software engineering practice to achieve reliable software is to avoid errors during design and implementation and to remove committed errors through extensive testing. Although quite effective, experience has shown that it is insufficient to ensure the degree of correctness required by critical applications such as avionics control. Software fault tolerance (SWFT), which enhances the system's ability to tolerate the errors that survived during the testing phase, has been proposed as an additional measure to achieve ultra-high dependability. Although first introduced nearly two decades ago by Randell and Avižienis [2, 3], SWFT is still not widely used partly because it is quite difficult to implement in practice. And, if not properly incorporated, it can actually decrease the overall system reliability. There are two major reasons why SWFT schemes are difficult to incorporate in complex systems:

1. Although several SWFT schemes have been proposed, only a few research efforts have been reported to establish guidelines for cost-effective implementation of SWFT schemes.

2. The majority of SWFT schemes are not transparent to the application developer, meaning that the developer has to implement the elements of the chosen SWFT scheme together with the application. This in turn increases the complexity of the system to be developed.

Therefore, an effort to identify components of well-established SWFT schemes and to develop a framework of the desirable components to be reused by application developers has become a critical research need. A properly developed framework of reusable SWFT components together with a repository management system which facilitates efficient retrieval of these components will reduce the barriers that currently exist in implementing SWFT schemes.

There have been several efforts to develop application-specific frameworks of reusable components. One of the major reuse efforts to date is the Common Ada Missile Packages (CAMP) Project [4]
which has the objective of identifying common missile functions and the design of reusable Ada components. The CAMP experience has identified four major technical issues that have impeded meaningful levels of software reuse. These problems, along with our solutions, are:

1. **Inappropriateness to the domain**
   Domain analysis has been found to be the key factor in the success of reusability [5]. Components that result from domain analysis are better suited to reuse because they capture the essential functionality required in that domain; thus developers find them easier to include in new systems. During the research effort reported here, we conducted an in-depth domain analysis of software fault tolerance to identify reusable components.

2. **Difficulty in integration of reusable and custom code**
   A typical complaint about reusability is that, in spite of having software components available for reuse, programmers prefer to create their own module because available components are difficult to understand and adapt to new applications. This is the case when components designed for reuse lack a rigorous specification for their functionalities and interfaces. Components designed for systematic reuse must be rigorously specified in terms of both their behaviors and their interfaces to potential user components. Object-oriented design — using (i) data abstraction and encapsulation to hide implementation details, and (ii) inheritance and polymorphism to describe complex hierarchical representations — has been found to be the most promising technique for attaining the goals of extensibility and reusability [6, 7, 8]. In this research, an object-oriented methodology, based on the Booch method and Rational Rose CASE tool [8], has been used for analysis, design, and implementation of the reusable SWFT components.

3. **Skepticism about the quality of reusable components**
   Reusable components should be efficient, robust, and most importantly reliable. Rigorous testing of reusable software components is the only conceivable way of assessing their reliability. Therefore, it is important that these components support testing and data collection. Our approach to reliability assessment and assurance is to provide reusable components that support fault injection and data collection, and which therefore allow subsequent assessment of the fault detection and recovery capabilities of SWFT components.

4. **Lack of support tools for reuse**
   Although identification and implementation of reusable components are by their nature the most important tasks facing the developer of these components, another key item is an efficient retrieval system by which the potential user will select the most appropriate components to reuse. Tools can facilitate the identification, retrieval, evaluation, and incorporation of reusable software components. We plan to adapt a domain-specific repository management system supporting the object-oriented paradigm to enhance productivity gains by decreasing the cost of reuse.

Figure 1 depicts the object-oriented development process for the SWFT components incorporating the solutions to the technical problems of reusability described earlier. The first step is an in-depth domain analysis of software fault tolerance. The inputs include the expert knowledge of the investigators, a survey of the existing SWFT systems, and an extensive review of past and current literature. Based upon the past experience in domain analysis [7, 9] and motivated by the object-oriented design and programming methodology chosen, object-orientation is selected as the underlying paradigm for the modeling and architecture of the domain. The output of the object-oriented domain analysis is a SWFT domain model summarized in a taxonomy. Based on the developed domain model, the design process performs an object-oriented decompo-

![Figure 1: Development Process of Reusable Components Based on Object-Oriented Paradigm](image-url)
sition which identifies key abstractions (classes and objects) and invents mechanisms whereby sets of objects interact. The products of the design process are a set of class and mechanism diagrams. The key abstractions and mechanisms are then implemented in Ada using abstract data types, inheritance, and polymorphism provided by Ada 95 [10, 11]. The resulting components will be managed by a domain-specific reuse librarian which supports the identification, retrieval, evaluation, and incorporation of object-oriented reusable software components.

The rest of the paper is organized as follows: Section 2 describes results of the domain analysis identifying the SWFT reusable components. Section 3 describes the development of the SWFT components and their integration into SWFT systems for demonstration. Section 4 provides a conclusion to the paper and discusses future work.

2 Domain Analysis and Taxonomy

Domain analysis is an examination of a specific problem area which seeks to identify common operations, objects, and structures which are candidates for software components [5]. For the following reasons, we decided to use an object-oriented approach to domain analysis which examines requirements from the perspective of the classes and objects found in the vocabulary of the SWFT domain [7]:

- Domain-specific knowledge of the SWFT application can be captured in a form that lends itself to careful, point-by-point verification by domain experts.
- Domain knowledge can be accurately transferred to software designers and programmers.
- The analysis can be communicated in a form that is easily mapped into an object-oriented design and implementation.

Examination of existing complex applications indicates that high failure rates (particularly those resulting in system crashes) occur in application software components. Based on this experimental finding, we decided to concentrate mainly on analyzing the techniques for masking (or recovery from) application software failures. Hardware faults can be tolerated by duplicated components, while design and software faults can only be tolerated by some types of diversity. Figure 2 depicts a taxonomy of existing software fault tolerance techniques based on the most viable diversity utilized.

1. **Design Diversity Class**

   Considering the level of complexity in emerging applications, one cannot exclusively rely on rigorous design and validation techniques for obtaining sufficiently reliable software. It is sensible to seek means for enhancing software systems to cope with residual design errors/inadequacies that appear at unpredictable instances during system operation. A well-established class of techniques for tolerating residual software design failures is to provide multiple designs (or so-called versions) for the given application software specification [12, 13, 14]. An executive component may then be dedicated to employ the given versions and provide a desirable system output based on the selected or voted outputs created by the versions. The underlying assumption is that the different versions are created from different designs and thereby do not contain the same set of failures. The Recovery Blocks [2] and N-Version Programming [3] are among well-established design diversity techniques.

2. **Data Diversity Class**

   Extensive testing of a software system typically eliminates the software errors that manifest themselves in most frequently occurring cases. Hence, most of the remaining software errors are of the types which occur only in very special and intuitively-rare cases overlooked in the testing and validation phases [15]. Moreover, some of the remaining errors occur for special values of the data set (i.e., system input and state spaces) [16]. The data diversity class is based on the partially validated assumption that if the data set is re-expressed (i.e., it is either expressed in some other form, or transformed into an equivalent data set), the software which has failed with the original data set, does not fail with the re-expressed data set. Retry Block, N-Copy Programming [16], and Progressive Retry [17] techniques belong to the data diversity class.

3. **Temporal Diversity Class**

   In many applications, one does not have to make pessimistic assumptions about the possible residual software failures. Therefore, approaches mentioned earlier would be too expensive for use in such applications. In fact, it is well known that most of hardware/software failures in computer systems are soft (also called
Heisenbugs) [1], meaning that if the failed scenario is repeated (in different time or different hardware), the failure disappears and the system will succeed to produce the desirable results. This class of failures typically includes transient hardware failures and unanticipated sequences of events.

The temporal diversity scheme suggests that to recover from a soft failure, the failed scenario should be repeated immediately after the failure is detected. Checkpointing and rollback techniques [17, 18, 19, 20] utilize temporal diversity for system recovery.

4. Spatial Diversity Class

The logic behind spatial diversity is based on the assumption that if the failed scenario is repeated in different hardware, most failures will not be repeated [1]. Under this scheme, the application software is concurrently executed in multiple hardware units and an executive component specifies the correct output of the system using the output of all (or a subset of) hardware units. Primary/backup protocols [21, 22] employ spatial diversity to tolerate hardware/software failures.

In the following sections, some well-established schemes using design and data diversity classes will be discussed in details.

2.1 Recovery Blocks

A recovery block consists of a primary module which is executed to perform the required task under normal situations, an acceptance test, an expression of the criterion used for accepting the result, which is used to detect failures occurring in the execution, and a number of alternate modules which are executed in sequential order if errors are detected. Before the execution of a recovery block, a recovery checkpoint is established by saving the current system state using a checkpointing mechanism for possible rollback. An executive is used to coordinate the execution, checkpointing, and recovery processes, as shown in Figure 3, in which the control flow in the fault-free scenario is distinguished by heavy lines. This scheme is sometimes called Sequential RB (SRB) because the try blocks are executed sequentially.

Figure 3: The Sequential Recovery Block Scheme

The acceptance test is the key to the success of the RB scheme. It should be efficient and have high coverage. The following checks can be utilized by the acceptance test depending on the application [23]:

- Replication Check
- Timing Check
- Reversal Check
- Coding Check
- Reasonableness Check
- Structural Check
- Diagnostic Check
- Customized Check
Checkpointing and acceptance test are overheads that the RB scheme must incur. Different checkpointing mechanisms have been devised to optimize the process depending on the system characteristics [23].

- **Checkpointing**
- **Recovery cache**
- **Audit trail**

A significant improvement in system response time can be achieved by the distributed execution of the recovery blocks. In the Distributed Recovery Block (DRB) scheme [24], try blocks (primary and alternate modules) and acceptance tests are replicated and executed concurrently on different computing nodes as shown in Figure 4. The *active* node normally executes the primary module and provides active control and processing. The other node, referred to as the *shadow*, operates as a standby. The shadow node takes over control if the active node fails the acceptance test or a timeout occurs. Since the shadow is executing another try block concurrently, it can output its result without delay should it pass the acceptance test.

![Figure 4: The Distributed Recovery Block Scheme](image)

Applying recovery blocks without coordination in concurrent processes can incur unbounded rollback, since errors can be propagated to other processes via communication among processes. For example, a fault requiring a rollback in one process can provoke a rollback in another process because it might have used the results of the faulty process in its computation. This, in turn, could affect the whole set of processes, causing the entire computation to restart from its initial state (which is called the domino effect). Domino effects occur due to the uncoordinated nature of the recovery actions undertaken by cooperating processes. Thus the recovery block scheme cannot be applied to concurrent processing environment without incorporating a cooperation scheme among recovery blocks in interacting processes.

In general, three possible ways of avoiding domino effects are conceivable: design-time methods, runtime methods, and a combination of both. Design-time methods force the application developer to obey certain restrictions. The developer must design the interacting processes, in particular their recovery points, with the assurance that domino effects cannot occur. The goal is to create simple and easy-to-follow design rules that support the design procedure. The *conversation* scheme [2] belongs to this class.

On the other hand, runtime methods rely on the operating system support to dynamically coordinate processes to establish recovery lines so that domino effects may be prevented. The *proliferation transparent coordination* (PTC) scheme [25] represents a run-time method.

The class diagram of the recovery block components using the Booch notation [8] is shown in Figure 5. As shown in the figure, the “Recovery Block Scheme” class uses “Acceptance Test”, “Try Block”, and “Checkpointing Mechanism” classes. The use relationship is denoted in the Booch notation by a line ended with an empty circle at the user side. “Recovery Block Scheme” also has an aggregation relationship (denoted by a line ended with a solid circle at the owner side) with the “Executive” class. On the other hand, the “Executive” class itself is an abstract class (denoted by a triangle enclosing an A) with “Single-Process Recovery” and “Concurrent-Processes Recovery” classes as its concrete subclasses (the subclass relationship is denoted by an arrowed line). “Single-Process Recovery”, in its turn, has two subclasses: “SRB Executive” and “DRB Executive” classes. Moreover, “PTC Executive” and “Conversation Executive” classes are concrete subclasses of the “Concurrent-Processes Recovery” class.

As illustrated in the class diagram, the “Try Block” class has two subclasses: “Primary” and “Alternate” and “Acceptance Test” is the superclass of “Timing Checks” and “Reasonable Checks” classes. Finally, “Checkpointing Mechanism” has created three subclasses: “Recovery Cache”, “Checkpoint”, and “Audit Trail” classes.

### 2.2 N-Version Programming

This scheme uses multiple (i.e., N) versions with voting for fault detection and result selection. The N-Version Programming scheme executes all N versions in parallel, taking advantage of the redundant processors, which are likely to be available in any system that must tolerate hardware and software
faults. Of course, in the absence of sufficient hardware resources, one can execute the versions sequentially. A voter compares the results of all $N$ versions to determine a consensus result as shown in Figure 6. An executive component is needed to schedule version execution, broadcast and process results and to reconfigure the system based on error reports. The generic voter makes a decision by applying one of three agreement tests: (i) bit-by-bit for identical match, (ii) cosmetic for detecting character string differences caused by spacing and misspelling, and (iii) numeric for inexact integer and real number decisions.

Figure 7: The NVP Scheme (Distributed-Voter)

Since the voter could be a single point of failure, some systems [26] replicate the voter at each node as shown in Figure 7. This is particularly suitable for multi-channel systems which have multiple actuators to control.

Another variation of the NVP scheme can be realized by incorporating an acceptance test to each version in order to identify clearly erroneous results. It has been verified in practice that such a scheme will increase the performance and coverage of the voter [27].

Several industrial systems have employed variations of the N-Version Programming scheme and demonstrated potentials of the scheme as a cost-effective SWFT technology [12, 13, 14, 28, 29].

The class diagram for the N-Version Programming scheme using the Booch notation is shown in Figure 8.

2.3 Retry Block

The retry block scheme is a modification of the recovery block scheme which uses data diversity for error recovery purpose. It consists of (i) a single version to implement the application specification, (ii) an acceptance test, and (iii) re-expression module to re-express the data set if an error is detected. The try block executes the single version first, and then the acceptance test is executed. If it passes the acceptance test, the execution of the try block is completed and the result is released. However, if the acceptance test fails, the same version is executed again after the data set is re-expressed using the application-dependent data re-expression module. The class diagram for the retry block scheme is given in Figure 9.
2.4 N-Copy Programming

The N-Copy Programming scheme is a modification of the N-Version Programming scheme which uses data diversity for fault-tolerance purpose. Under N-Copy Programming scheme, N copies of the application module are executed in parallel in N hardware units each of which operates on slightly modified versions of the data set. These versions of the data set are produced by an application-dependent data re-expression module. Then the voter module votes on the results similar to the voting mechanism used in the N-Version Programming scheme. It should be noted that the data re-expression algorithm has a very strong influence on the success of the N-Copy Programming scheme and if it is not carefully chosen, results produced by different copies might diverge. However, past experience on the data diversity schemes \cite{16} indicates that for many applications dealing with continuous-value data, simple data re-expression mechanisms can be used successfully.

The N-Copy Programming can be extended to include the acceptance test in each copy in order to identify clearly erroneous results. Figure 10 shows the class diagram for the N-Copy Programming scheme.

2.5 Generic Software Components Employed by Various SWFT Schemes

During the domain analysis, the components employed by various software fault tolerant schemes were identified. These generic components can be categorized into two classes: (i) fault-tolerance specific support services and (ii) general-purpose support services.

Fault tolerant specific support services that have been identified include:

- Try Block Component
- Acceptance Test
- Voter
- Heartbeat
- Watch-dog Timer
- Checkpointing Component
- Data Re-expression Component

These components are specially important in construction of the reuse framework since they not only can be used in developing a specific SWFT scheme but also viewed as building blocks for constructing any other variation of SWFT schemes discussed earlier.

General-purpose support services that have been identified include:

- Interprocess Communication
- Network Protocols
- File Management
- Memory Management
- Message Management
- Input and Output

3 Ada Software Fault Tolerant Reusable Components

In an attempt to validate potential benefits of the identified reusable components, we have conducted an experimental development of two variations of the RB scheme, namely the sequential RB and the
distributed RB, incorporating reusable components identified in the domain analysis. Each component is encapsulated as an Ada package implementing the data structures and operations of the respective class. The implemented components include:

1. Recovery Block Executive Package
   The RB Executive schedules executions of the primary and alternate application modules and the acceptance tests. The RB Executive establishes the checkpoint before entering a block and restores it for recovery.

2. Distributed Recovery Block Executive Package
   The DRB Executive, which run in both DRB nodes, establishes communication channels to its peer and updates system state (such as local databases). It also determines which version of the application software to run (i.e., primary or alternate), and the role of the local node (i.e. active or shadow).

3. Acceptance Test Package
   This component provides acceptance tests to judge the correctness of the application results.

4. Try Block Package
   Try Block Package provides the application modules with an interface to the RB or DRB Executive. The executives utilize this interface to pass the checkpoint data and invoke the primary or alternate modules.

5. Heartbeat Package
   Heartbeats are the periodic status messages exchanged by the DRB nodes. The DRB Executive determines the failure of its peer node by recognizing the absence of the peer’s heartbeat.

6. Watchdog Package
   Watchdog Package provides the functions for setting a timer with a timeout period, starting and stopping the timer, and raising an alarm signal should a timeout occur.

7. Message Management Package
   Message Management Package encapsulates the structure and operations of the messages to be sent by the DRB nodes.

8. Network Communication Package for Read-Write Channel
   This package provides a read-write channel facility that can be, for example, used for network communications between the DRB nodes and the Monitor node, which is responsible for monitoring the activities undertaken in the DRB nodes, using the User Datagram Protocol.

9. Network Communication Package for Write-Only Channel
   This component provides the write-only channel as a subclass of the read-write channel using the Ada derived type feature.

The following subsections describe how these reusable components have been used to implement a DRB system.

3.1 Distributed Recovery Block Demonstration System

3.1.1 Overview of the DRB System
   The underlying fault tolerance algorithms and mechanisms of the DRB Demonstration System are based on the distributed recovery block [24] with real time extensions. The system runs on two Sun workstations interconnected over an Ethernet. These two computers are collectively referred to as an operational node pair. One member of the node pair, called the active node, provides active control and processing. The other node, referred to as the shadow, operates as a standby. The active and shadow nodes exchange frequent periodic status messages, called heartbeats, over a communication channel as an indication of their health status. If the shadow node senses the absence of its companion active node’s heartbeat, it will promote itself to the active status.

   Within both the active and shadow nodes are two versions of the task execution software, referred to as the primary and alternate modules. Under normal circumstances, the primary module is run on the active node while the alternate module is concurrently run on the shadow node. The primary module is coded to provide the greatest functionality, accuracy, and performance. The alternate module provides less functionality and performance, but is coded to optimize reliability. After each processing iteration, an online acceptance test checks the validity of the output of both the primary and alternate modules. If the acceptance test shows that an error has occurred in the primary module, the output will be taken from the alternate module and control is passed to the shadow node. This minimizes the recovery time in the event of an application failure. Since the primary module is more desirable in the absence of failures, the DRB Executive of the newly
promoted active node will switch the version back to the primary module after a short period of time (five successful iterations in our implementation).

3.1.2 Components of the Distributed RB System

The Distributed Recovery Block Demonstration System is implemented from instances of the following components:

- DRB Executive Package
- Acceptance Test Package
- Try Block Package
- Heartbeat Package
- Watchdog Package
- Message Management Package
- Network Communication Package for Read-Write Channel
- Network Communication Package for Write-Only Channel
- Primary Module
- Alternate Module

Figure 11 illustrates a graphical view of the inter-relationship of the components. Some of the units shown in the figure are provided either by the Ada compiler (such as text_io and unchecked_conversion) or by the TeleAda-LAN package (such as adalan_socket).

4 Conclusions

This paper described the research results in developing reusable Ada software fault tolerant components. The study was performed by SoHaR Incorporated under an SBIR contract with the Rome Laboratory of the US Air Force.

The research has demonstrated the feasibility of reusable Ada software fault tolerant components through the creation and use of the recovery block components. An in-depth domain analysis has been performed on the software fault tolerance domain and resulted in the identification of a library of reusable SWFT components. A total of nine reusable components have been identified and implemented. The object-oriented approach has been successfully applied to develop reusable Ada components.

As an attempt to validate potential benefits of the identified components, we have conducted an experimental development of two variations of the recovery block schemes, namely SRB and DRB, incorporating reusable components identified in earlier part of the research. This experiment produced positive evidences of the benefits of the developed reuse framework.

We plan to implement other SWFT schemes such as the N-Version Programming scheme using SWFT components and to conduct an extensive testing of the components using the testbed.

Acknowledgments

This work was partially supported by Small Business Innovative Research (SBIR) Contract F30602-94-C-0013 from Rome Laboratory, Griffiss AFB, U.S. Air Force.

References


Figure 9: Class Diagram for the Retry Block Scheme

Figure 10: Class Diagram for the N-Copy Programming Scheme

Figure 11: Components of the DRB Demonstration System


