Ada95 Object-Oriented and Real-Time Support for Development of Software Fault Tolerance Reusable Components

Eltefaat H. Shokri Kam S. Tso
SoHaR Incorporated
Beverly Hills, CA 90211
{shokri,tso}@sohar.com

Abstract

This paper reports our experience on exploiting the object-oriented and real-time features of Ada95 to support the development of a reusable software fault tolerance testbed. The testbed is a hardware and software platform for the creation of software fault tolerance systems from reusable components and provides a fault-injection environment for evaluating their effectiveness. The reusable components were identified from an in-depth analysis of the software fault tolerance domain and designed using an object-oriented approach based on the Booch Method. The result of the analysis and design is a set of objects and their communication patterns. The identified objects were classified as active, passive, and shared-data objects and implemented as reusable components by mapping them into the corresponding Ada95 object-oriented constructs. A distributed recovery block system with a simplified air traffic control application were developed from the reusable components to demonstrate effective reuse and meeting soft real-time requirements.

1. Introduction

This paper reports our experience on exploiting the object-oriented and real-time features of Ada95 [5] to support the development of a software fault tolerance testbed based on reusable components. Ada95 is the revised standard of the original Ada language (Ada83) [1]. The revised language increases the flexibility and applicability of Ada by introducing new features supporting object-oriented programming, hierarchical libraries, and development of real-time systems. Since the ISO approval of Ada95 in early 1995, only a handful of pre-release Ada95 compilers are available and limited experience has been reported on its use. This research represents an early effort of using the new features of Ada95 to validate its effectiveness in supporting software reuse and real-time applications.

The Reusable Software Fault-tolerance Testbed (ReSoFT) [13] is a hardware and software platform developed under the sponsorship of the U.S. Air Force for experimenting with software fault tolerance. Software fault tolerance (SWFT) has been proposed as an additional measure to achieve ultra-high dependability required for large-scale safety-critical applications [11, 2]. Although several SWFT techniques have been proposed, they are not widely used partly because (1) their effectiveness in tolerating design faults is yet to be validated, and (2) they are difficult to implement in practice. Therefore, an effort to develop a flexible and reusable testbed to allow the creation of SWFT systems from reusable components and the evaluation of their effectiveness through experimentation is a critical research and development task. In response to this need, the ReSoFT project was initiated. Ada95 was chosen for implementing ReSoFT mainly because it provides object-oriented and soft real-time facilities while retaining its main reliability goal.

The rest of the paper is organized as follows: Section 2 gives an overview of the main characteristics as well as the overall structure of the ReSoFT testbed. In Section 3, we describe Ada95 support for implementing the SWFT reusable components. Section 4 discusses the implementation of fault-injection facility and the support by Ada95. Finally, Section 5 provides a conclusion to the paper.

2. ReSoFT: A Testbed for Software Fault Tolerance

ReSoFT (Reusable Software Fault-tolerance Testbed) [13] is an integrated software fault tolerance testbed developed on an open architecture and uses standard off-the-shelf hardware and software components. The main goals for developing ReSoFT are satisfying (i) the need to experiment with various well-established SWFT schemes for investigating its effectiveness and (ii) the desire to reduce repetitive effort in implementing those SWFT systems. Re-

SoFT consists of a number of computer nodes (workstations) communicating over a redundant network. Software components of ReSoFT are developed and executed on top of the Sun Solaris operating system. It also makes use of available POSIX 1003.1b thread libraries for improved response times. Figure 1 depicts the system architecture for ReSoFT.

The software architecture of ReSoFT, which consists of the reusable components identified during an in-depth domain analysis [14], is an integration of the following layers:

- *Reusable SWFT Components*:

  To identify reusable SWFT components, an object-oriented analysis of well-established SWFT schemes was conducted [13]. In this process, maximum care was taken to make sure that (i) the components capture the common functionality of a wide range of SWFT schemes, and (ii) application-dependent components are identified and separated from the generic (application-independent) components. The Reusable SWFT layer include full implementations of application-independent components as well as templates for application-dependent components.

- *Network Communication Components*:

  Network communication components provide application transparent reliable inter-node communications among the ReSoFT nodes. Moreover, a diagnosis facility for communication faults is provided for monitoring.

- *Fault Injection Components*:

  Fault injection components are designed to support testing and evaluation of the dependability of the SWFT components and systems. They can inject a vast variety of faults into other layers such as SWFT components and application modules.

- *Graphical User Interfaces*:

  Graphical user interfaces have been implemented:

  1. to automate integration of SWFT schemes with application modules, such that the application developer is relieved from the complexity of the SWFT schemes (Graphical Application Integrator).
  2. to accept fault injection commands from the user and direct the fault injection experiment (Graphical Fault-Injection Manager).
  3. to monitor the execution of SWFT schemes and to supervise them to a safe state when an unexpected situation arises (System Status Monitor).

- *Application Modules*

  The reusable SWFT components are designed in a way that can be integrated by various applications with a minimal change. The application developer should implement the application preferably in the form of reusable components and then complete the templates of application-dependent SWFT components according to the application characteristics using the Graphical Application Integrator. The Graphical Application Integrator will then automatically build the program(s) supporting fault-tolerant execution of the application.

In this paper, our focus is on the first three layers (i.e., the Reusable SWFT layer, Network Communication layer, and Fault Injection layer). Reusable components in these three layers are classified into three general categories (i) *active components* which have their own execution threads, (ii) *passive components* which do not possess any execution thread but rather provide services to other components, and (iii) *shared-data components* which store shared data items. The mapping of reusable components into Ada95 tasks and/or packages will be discussed in details in the following sections.

3. Object-Oriented Support of Reuse in ReSoFT

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 [8] 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 [10]. Components that result from domain analysis are better suited to reuse because they capture the essential functionality of the domain. In the ReSoFT project [13], 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 concern about reusability is that software developers prefer to create their own modules 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. Object-

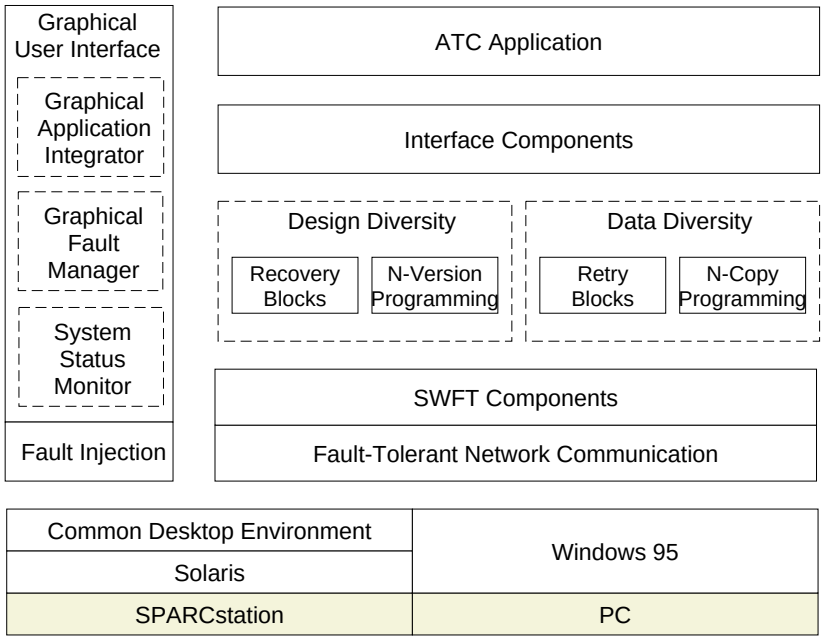

Figure 1. System Architecture of ReSoFT

oriented design has been found to be the most promising technique for attaining the goals of extensibility and reusability [9, 12]. In the ReSoFT project, an object-oriented methodology, based on the Booch method [3] and Rational Rose CASE tool [15], has been used for analysis and design 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 supports for fault injection and data collection for evaluating 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 issue is an efficient integration of system reusable software components into application modules to create a robust application. We implemented a graphical application integrator which builds the SWFT scheme chosen by the user and integrates it with the application.

3.1. Mapping Reusable Components into Ada95 Constructs

The result of object-oriented design is a set of objects, their interfaces as well as communication patterns. Identified objects are classified into three types:

- *Active objects*
  An active object is an object which possesses its own execution thread(s). Additionally, an active object may also provide services to other objects.

- *Passive objects*
  Passive objects do not own any execution thread and merely act as service providers to other objects. Passive objects are stateless, meaning that they do not maintain any long-life data used by successive services.

- *Shared-data objects*
  Shared-data objects maintain data stores to which multiple objects may have mutually-exclusive access.

To illustrate this classification, Figure 2 depicts implementation of the Distributed Recovery Block (DRB) scheme [7] using reusable components. The DRB scheme is essentially an approach for structuring a duplex redundant computer nodes dedicated to execute different versions of the applications. Each version of the application is structured in the form of the recovery block abstraction [11]. The acceptability of the result of the application execution is

validated by an acceptance test. Moreover, to facilitate the detection of node-crash failures, each node sends periodical status messages called heartbeats [4]. In our implementation, the DRB Executive, Heartbeat-Generator, and Try Block are active components, Checkpoint and Acceptance Test are passive components, and RB Data-Store is the only shared-data object. As shown in the figure, five components share the data stored in the RB Data-Store component.

The role of the Application Execution Interface component needs some clarification since it interrelates the execution of the application and the DRB protocol. The Application Execution Interface component is the only interaction component used in the implementation of DRB and acts as a communication interface between the Try Block component and the application in the following way. It first makes a rendezvous with the Try Block component to receive the command for initiation of a new cycle of the application. Upon completion of the rendezvous with Try Block, Application Execution Interface makes another rendezvous with the application to start a new cycle of the application. Using a similar approach, Application Execution Interface passes the result of the completed cycle to Try Block.

To implement reusable components in Ada95, the following mapping was adopted:

- A passive object is implemented as an Ada package which does not contain any task. Care was taken to make sure the package for a passive object does not possess any persistent global data item. In other words, all data communication between a server package and its client are realized by passing parameters.
- An active object is implemented as an Ada package with a task for each of its execution threads.
- A shared-data object is implemented using the protected record type provided in Ada95, thereby the Ada runtime system will guarantee mutually exclusive accesses to the shared-data.

3.2. Tightly-Coupled Objects vs. Loosely-Coupled Objects

Two approaches to the implementation of object interactions in Ada95 may be used. The first, called *tightly-coupled* interactions, employs the rendezvous concept for the interaction a server and its clients. In this approach, all objects residing in a processor share a single address space and should be compiled into a single process consisting of concurrent tasks interacting with synchronous rendezvous. This method of packaging all objects as a single program is easy to implement. However, it has a major drawback that any change in an object leads to the rebuilding of the entire program. Moreover, one of the interacting objects should be aware of the name of the other interacting object (because of the rendezvous semantics). This reduces the autonomy of reusable components to some extent.

The other approach, which is denoted as *loosely-coupled* interaction, implements interactions between objects using lower-level communication facilities such as socket-based communication mechanisms. This approach provides a higher degree of object autonomy. However, we learned during the ReSoFT implementation that loosely-coupled interactions introduce non-negligible overhead if not used carefully.

It is hard to say whether one approach has to be generally preferred to the other. Our experiments suggested that one practical hybrid solution is to partition the system into several subsystems based on criteria such as communication pattern, and use tightly-coupled interactions among objects located in the same subsystem while employ loosely-coupled interactions among objects residing in different subsystems. In the ReSoFT implementation, reusable components and the application were considered as two separate subsystems and their interactions were implemented using loosely-coupled interactions.

3.3. Use of Ada95 in the DRB Implementation

In Section 3.1, a component-based implementation of the DRB scheme was shown to illustrate the use of reusable components. In this section, we describe how features of Ada95 were employed in this implementation.

3.3.1. Tagged Types for Inheritance

The ultimate goal in designing SWFT components is their maximum reusability. These components should be designed in the way that they can be reused by any application with no or minimal changes. However, there are functionalities which may vary from one application to another. As an example, the Checkpoint component, which is responsible for establishing and restoring the application checkpoints, is highly dependent on the application. We used the inheritance mechanism to separate application-specific functionality from application-independent features.

Ada95 introduced tagged types for implementing inheritance. A tagged type can be extended by adding new items as well new operations to the derived types. The `CHECKPOINT` record type is designed as a tagged record type on which the `GET_CURRENT_CHECKPOINT_NUMBER` procedure can be applied. The application designer can then introduce a new type `APPLICATION_CHECKPOINT` type which is defined based on the `CHECKPOINT` type. As shown in the following piece of code, two new procedures, namely

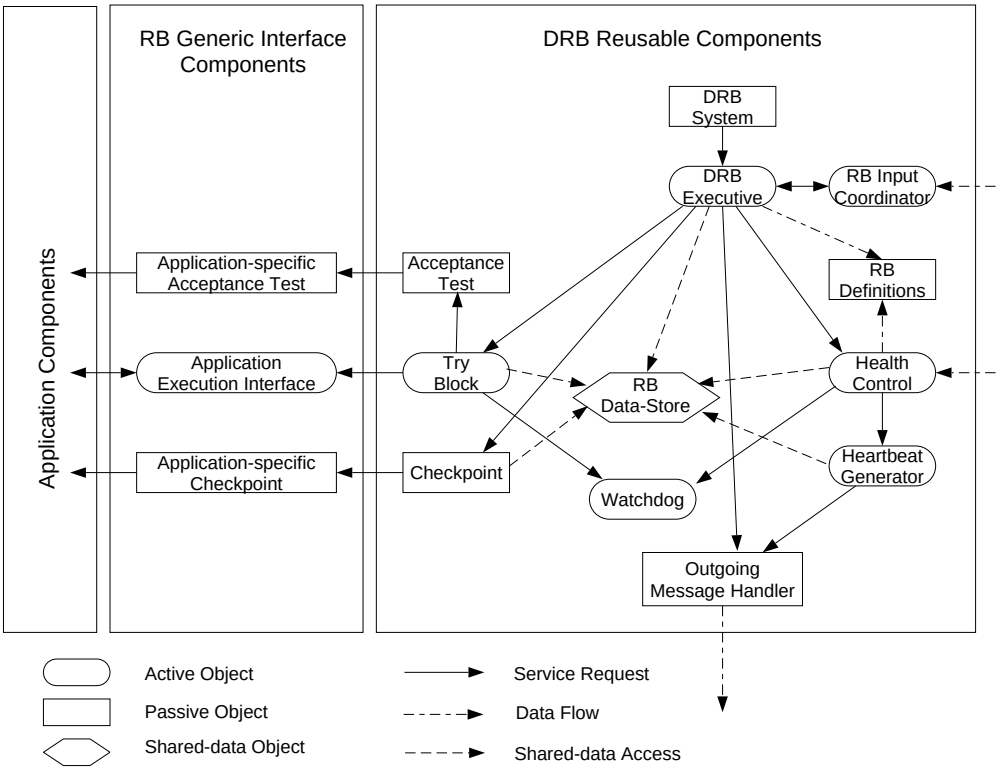

Figure 2. Component-based Implementation of the Distributed Recovery Block

ESTABLISH and RESTORE, can be additionally applied to APPLICATION_CHECKPOINT type.

```
-- Specification for general checkpoint package.
package CHECKPOINT_PACKAGE is
 type CHECKPOINT is tagged
    record
       CHECKPOINT_NUMBER : integer;
    end record;
 function GET_CURRENT_CHECKPOINT_NUMBER
         return integer;
 -- ...
 -- other application-independent procedures
 -- ...
end CHECKPOINT-PACKAGE;

-- Specification for application-specific
-- checkpoint package.
with CHECKPOINT_PACKAGE;
use  CHECKPOINT_PACKAGE;
package APPLICATION_CHECKPOINT_PACKAGE is
 type APPLICATION_CHECKPOINT is
   new CHECKPOINT with
     record
       -- Application-specific checkpoint fields
    end record;
  procedure ESTABLISH
  (CURRENT_CHECK : out APPLICATION_CHECKPOINT);
  procedure RESTORE
  (CHECK_TOBE_RESTORED:
           in APPLICATION_CHECKPOINT);
  -- ...
  -- Other application specific procedures
  -- ...
end APPLICATION_CHECKPOINT_PACKAGE;
```

We adopted the same technique for other application dependent features such as the Acceptance Test component.

3.3.2. Asynchronous Transfer of Control for Time-Bounded Computations

There are occasions during the execution of the SWFT schemes in which an activity should be abandoned if certain condition (such as when the execution time of an activity exceeds a predetermined limit) arises and an alternative action should be taken. As an example, when DRB Executive orders the Try Block to executive a new application cycle, the DRB Executive component should receive the result of the execution by a predefined time-limit, say T. If T time-units elapses and DRB Executive does not receive the result, it should abandon normal execution of the scheme and take a recovery action. Thereby, waiting for the result of application execution should be terminated before T time-units elapsed, otherwise the action (i.e., waiting for the application result) should be abandoned and it should then be concluded that the application has either experienced a timing fault or died. This set of actions can be taken using "select-

then-abort" [5] statement introduced in Ada95 as shown below.

```
SEND_ORDER_TO_TRYBLOCK;
select
   delay MAXIMUM_APPL_EXEC_TIME;
   TAKE_RECOVERY_ACTION;
then abort
   WAITING_FOR_APPL_RESULT(APPL_RESULT);
end select;
```

SEND_ORDER_TO_TRYBLOCK orders the Try Block component to manage execution of a new application cycle. Then the DRB Executive component executes the select statement which executes the WAITING_FOR_APPL_RESULT procedure until it is completed or the MAXIMUM_APPL_EXEC-TIME timeout expires. If the control does not return from the procedure by this time-period, then the execution of WAITING_FOR_APPL_RESULT will be abandoned and the TAKE_RECOVERY_ACTION procedure will be activated. This cannot be easily implemented in Ada83. The only solution in Ada83 is dedicating a task for WAITING_FOR_APPL_RESULT so that the task can be aborted as the time limit expires. Of course, dynamic creation and abort of tasks are not recommended in critical applications.

3.3.3. Protected Types for Shared Data

In Ada83, the rendezvous model is used for both task synchronization and data-sharing purposes. Unfortunately, use of rendezvous has not been entirely satisfactory specially for data-sharing. It requires additional tasks to manage shared data and this often leads to poor performance. Ada95 introduced the concept of protected types which provides orderly and synchronous access to shared data without creating additional task.

The following piece of code illustrates the use of protected type in DRB implementation. RB_DATA_STORE maintains DRB configuration status such as the role of each node, the versions of application software being executed, and the health status of participant nodes. MY_ROLE, for example, is an item of shared data and can only be accessed through procedures GET_NODE_ROLE and SET_NODE_ROLE. For each shared data item RB_DATA_STORE, possible types of access were studies and implemented as procedures.

```
protected RB_DATA_STORE is
    function GET_NODE_ROLE return ROLES;
    procedure SET_NODE_ROLE(NEW_ROLE : ROLES);
    -- ...
    -- Other procedures to manipulate other
    -- protected data items
    -- ...
private
   MY_ROLE : ROLES;
   PEER_ROLE : ROLES;
   -- ....
   -- Declaration of other protected data items
   -- ...
end RB_DATA_STORE;

protected body RB_DATA_STORE is
  function GET_NODE_ROLE return ROLE is
  begin
     return MY_ROLE;
  end GET_NODE_ROLE;
  procedure SET_NODE_ROLE(NEW_ROLE : ROLES) is
  begin
    MY_ROLE := NEW_ROLE;
  end SET_NODE_ROLE;
  -- ...
  -- bodies of other procedures
  -- ...
end  RB_DATA_STORE;
```

4. Object-Oriented Support of Fault-Injection Testing in ReSoFT

To ensure that ReSoFT meets its reliability as well as timing requirements, a fault-injection subsystem was implemented. In order to preserve timely as well as functional characteristics of the target system, existing fault-injection approaches [6] inject faults by altering object codes or data items using underlying hardware and/or operating system facilities, making the fault-injection subsystem platform-dependent. To reduce such an undesired dependency, we decided to implement the fault-injection subsystem as a collection of reusable components which are compiled along with operational SWFT components. To be more specific, each SWFT component is paired with a fault-injection component responsible for injecting faults in the paired SWFT component. Using this component-based approach, robustness of each individual SWFT component as well as correctness of SWFT schemes can be evaluated in a platform-independent manner.

The fault-injection subsystem, as illustrated in Figure 3, has the following characteristics:

- *Centralized fault-injection manager*:

  The global fault-injection manager (FI Manager) accepts attributes of the faults to be injected from the user via a graphical user interface. The FI Manager then directs the fault-injection experiment and generates a fault-injection schedule. It also sends timely orders to the corresponding nodes to emulate faults. Moreover, the FI Manager monitors any changes occurring in the system in response to the injection of a fault. Finally, it analyzes system behavior for each fault injected and logs the necessary data for off-line reliability assessment. Whenever an undesired change occurs (e.g., a nodes crashes) in response to a fault, the FI Manager takes appropriate actions (e.g., restarting the crashed node) to reset the fault tolerant system to a normal state so that the FI experiment can be continued.

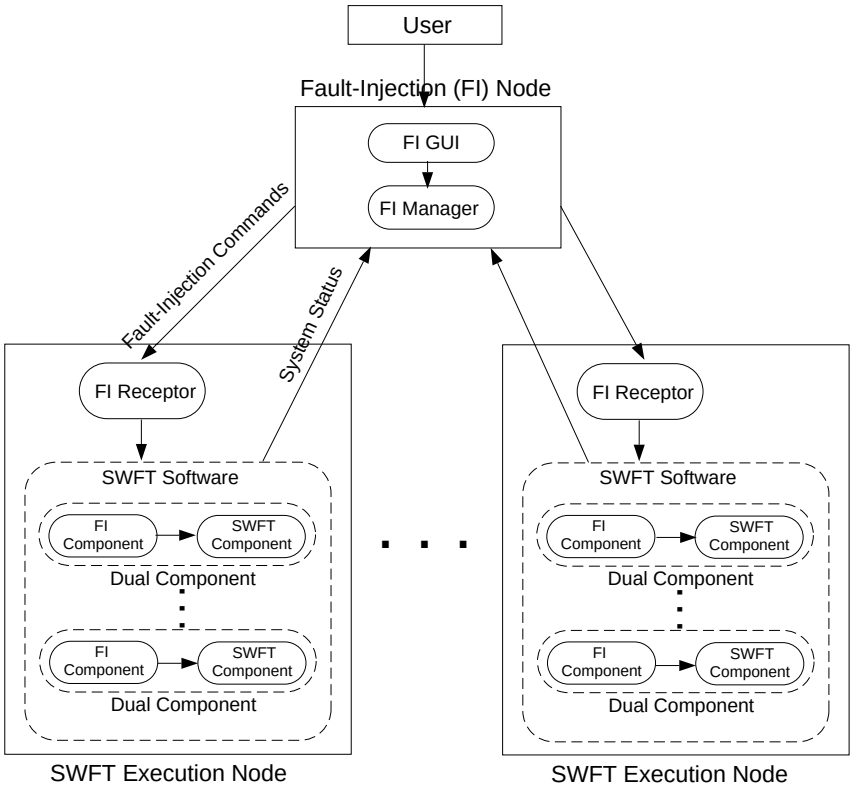

Figure 3. Overall Architecture of the Fault-injection in ReSoFT

- *Fault-injection receptors residing in each SWFT execution node*:

  Each of the nodes executing the SWFT scheme is designed to receive fault-injection commands issued by the FI Manager. After the fault-injection receptor (FI Receptor) receives a fault-injection command, it invokes the necessary procedures of the corresponding fault-injection component (FI component) to initiate the emulation of the given fault scenario.

- *Child-parent packaging to avoid recompilation*:

  As shown in Figure 3, each SWFT component is paired with a FI component creating a virtual component which is called here a *dual component*. Each FI component is designed for injecting faults in its peer SWFT component (i.e., the SWFT component located in the same dual component). Isolating fault-injection functionality from the SWFT components and locating them in separate components guarantees the minimal code alteration in SWFT components. In our Ada95 implementation, each FI component is implemented as a child package of its peer SWFT component. By doing so, SWFT components (specifications as well as bodies) do not need to be recompiled for fault-injection.

- *Minimal change in timing behaviors of the target SWFT scheme*:

  An important concern in using software fault-injection for evaluation and validation of a software system is that the fault-injection components might alter the timing as well as logical behavior of the target system. In ReSoFT, we attempt to minimize such undesired behavioral changes by the following techniques:

  1. Only one additional task (the fault-injection task) was introduced for fault-injection. Since this task will be in the blocked state most of the time waiting for a fault-injection command, the timing overhead is expected to be very small. Early fault-injection experiments on ReSoFT indicated that the timing overhead introduced by fault-injection components remains tolerable.

  2. Since injecting faults is mostly done by FI components (i.e., child packages), the code changes on SWFT components are kept minimal, thereby reducing the possibility of any behavioral changes in the SWFT target components.

5. Conclusions

The paper discussed some of the new features of Ada95 which were used in the development of the ReSoFT testbed. The development effort reported in the paper demonstrated that the new features (especially the ones supporting object-oriented development) are well-suited for developing reusable software. We used features such as class-based construction, inheritance, and child-parent hierarchical library structuring in developing reusable SWFT components and schemes. We are convinced from our experience that although Ada95 compilers and tools are not mature enough yet, but if used conservatively (i.e., using new features with extensive care and tests), they will provide significant cost and reliability benefits. We did not use any annexes mainly because none was available. We plan to experiment with features of the Real-Time System Annex to evaluate its effectiveness, when it becomes available.

Acknowledgments

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

References

[1] ANSI/MIL-STD-1815A. *Reference Manual for the Ada Programming Language*. American National Standards Institute, Feb. 1983.

[2] A. Avižienis and L. Chen. On the implementation of N-Version Programming for software fault-tolerance during program execution. In *Proceedings of COMPSAC-77*, pages 149–155, 1977.

[3] G. Booch. *Object Oriented Analysis and Design with Applications*. Benjamin/Cummings, Redwood City, CA, 2nd edition, 1994.

[4] M. Hecht, J. Agron, H. Hecht, and K. H. Kim. A distributed fault tolerant architecture for nuclear reactor and other critical process control applications. In *Digest of the 21st Annual International Symposium on Fault-Tolerant Computing*, pages 3–9, Montreal, Canada, June 1991.

[5] ISO/IEC-8652:1995. *Ada Reference Manual: Language and Standard Libraries*. International Organization for Standardization and International Electrotechnical Commission, Jan. 1995.

[6] R. K. Iyer and D. Tang. Experimental analysis of computer system dependability. In D. Pradhan, editor, *Fault-Tolerant Computing: Theory and Techniques*. Prentice Hall, 2nd edition, 1995.

[7] K. H. Kim and H. O. Welch. Distributed execution of recovery blocks: An approach for uniform treatment of hardware and software faults in real-time applications. *IEEE Trans. Comput.*, 38(5):626–636, May 1989.

[8] McDonnell Douglas Missile Systems Company. Developing and using Ada parts in real-time embedded applications. Common Ada Missile Packages — Phase 3 (CAMP-3), McDonnell Douglas Missile Systems Company, St. Louis, MO, Apr. 1990.

[9] B. Meyer. Reusability: The case for object-oriented design. *IEEE Software*, pages 201–215, Mar. 1987.

[10] R. Prieto-Diaz. Domain analysis for reusability. In *Proceedings of COMPSAC'87*, pages 23–29, Tokyo, Japan, Oct. 1987.

[11] B. Randell. System structure for software fault tolerance. *IEEE Trans. Softw. Eng.*, SE-1(2):220–232, June 1975.

[12] S. Shlaer and S. J. Mellor. An object-oriented approach to domain analysis. *ACM SIGSOFT Software Engineering Notes*, 14(5), July 1989.

[13] K. S. Tso and E. H. Shokri. ReSoFT: A reusable software fault tolerance testbed. In *Proceedings of Pacific Rim International Symposium of Fault-Tolerant Systems*, pages 98–103, Newport Beach, CA, Dec. 1995.

[14] K. S. Tso, E. H. Shokri, A. T. Tai, and R. J. Dziegiel, Jr. A reuse framework for software fault tolerance. In *Proceedings of AIAA Computing in Aerospace 10 Conference*, pages 490–500, San Antonio, TX, Mar. 1995.

[15] I. White. *Rational Rose Essentials — Using the Booch Method*. Benjamin/Cummings, Redwood City, CA, 1994.