The National Aerospace and Electronics Conference,
Dayton, OH, May 20-24, 1996.

DEVELOPMENT OF SOFTWARE FAULT-TOLERANT APPLICATIONS WITH ADA95 OBJECT-ORIENTED SUPPORT *

Eltefaat H. Shokri Kam S. Tso
SoHaR Incorporated
Beverly Hills, CA 90211

Roger J. Dziegiel, Jr.
Rome Laboratory/C3CB
Rome, NY 13441

Abstract

Experience has shown that the current software engineering practice is inadequate for producing error-free software. Thus, software fault tolerance (SWFT) must be employed in developing complex safety-critical applications. However, developing applications which are capable of tolerating software errors is a challenging task because the developers have to conquer not only the complexity of the application but also the complexity of fault-tolerance protocols. A middleware which provides SWFT services and establishes a well-defined interface with the application modules will allow the application developer to focus solely on the application complexity. This paper presents such a middleware consisting of reusable SWFT components. It also explores the way these components interface with the application in order to tolerate faults in the application. The paper also reports our experience on using real-time and object-oriented features of the new standard of Ada (Ada95) for implementing the middleware.

1 Introduction

Software errors and inadequacies are currently accounted for a high percentage of failures in complex computer applications [1]. The main source of software faults is the improper design and/or implementation, resulting in the violation of the system specification. The current software engineering practice to achieve reliable software includes: (i) fault avoidance through extensive testing and verification, and (ii) fault tolerance through redundancy in system components [2]. Although an essential attribute of reliable software, software fault tolerance (SWFT) is not widely used in practice mainly because it is difficult to implement. Since SWFT schemes are mostly non-transparent to the application, the application developers have to deal with the complexity of both the application as well as the chosen SWFT scheme. An effort to identify generic components of the SWFT schemes and to establish a well-defined interface between these components and the application modules is essential for complexity control in safety-critical applications. There are two main motivations for such an effort: (i) separating fault handling concerns of SWFT schemes from functional concerns of the application, and (ii) achieving component-level reuse in developing various SWFT schemes.

Our approach to accomplish these objectives is to develop a middleware which provides essential functionality of the SWFT domain. The middleware includes reusable software components which are used as building blocks for implementing a wide variety of SWFT schemes. This paper presents development of the SWFT middleware and illustrates its potential benefits in developing fault-tolerant applications. The paper also reports our experience on using real-time and object-oriented features of the new standard of Ada (Ada95) [3] for implementing the SWFT middleware. For illustration purpose, the paper focuses on the component-based implementation of one of the identified SWFT schemes — the Distributed Recovery Block scheme.

The rest of the paper is organized as follows: Section 2 briefly introduces the object-oriented and real-time features of Ada95 supporting the development of the SWFT middleware. Section 3 gives a short introduction of the basics of software fault tolerance. Reusable components of the SWFT middleware are discussed in Section 4. In Section 5, the integration of the application modules into the SWFT middleware will be discussed. Sections 6 and 7 discuss the implementation of the SWFT middleware using Ada95. Finally, Section 8 provides a conclusion for the paper.

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

2 Ada95 Support for Developing Object-Oriented Real-Time Systems

Ada95 is an extension of the original Ada language [4] and provides powerful constructs for composition of robust software components. It has increased the applicability of Ada while retaining the inherent reliability for which it has been originally designed. Ada95 consists of a core language as well as several annexes to support broad application areas and offers several new features some of which are discussed here.

2.1 Support for Large System Development

Ada is a programming language which is mainly designed for promoting the development of large safety-critical systems. Ada95 in turn increases the flexibility of Ada thus making it more suitable for emerging complex applications in the following ways:

- *Object-Oriented Programming*:

  Ada had originally provided data abstraction and encapsulation facilities. Ada95 fulfills object-oriented programming goals by supporting inheritance, dynamic binding, and polymorphism while remaining a very strongly typed language.

  Ada95 provides primary benefits of object-oriented paradigm by extending Ada83 constructs. For example, the tagged type, which is the vehicle for implementing inheritance in Ada95, is an extension of the type derivation. Derived types in Ada83 provides a very simple inheritance facility by which a derived type inherits the operations of its parent type and can add new operations. However, it is not possible to add new items to the derived type. By contrast, in Ada95 a tagged type can also be extended to add new items. This allows the implementation of the inheritance mechanism in Ada95.

- *Hierarchical Libraries*:

  An important strength of Ada has been the library package where the distinct specification and body decouple the user interface specification from its implementation. This enables the body of a package and its clients to be compiled separately without interference, provided the specification remains unchanged. However, it is desirable for large systems to extend an existing package by adding more functionality without forcefully recompiling its clients. This cannot be done in Ada83. Ada95 provides a solution by introducing the hierarchical library structure. If a package is defined as a child of another package, the child package shares private types and procedures defined in its parent package. Thereby, by creating a child package, the parent package can be extended without recompilation of its existing clients.

2.2 Support for Inter-Task Communications

Normally, inter-task communication and synchronization is either for data sharing or for synchronous communication. Ada83 offers only the rendezvous mechanism for both purposes. Unfortunately, the rendezvous mechanism is not a very efficient technique and has a high overhead especially when used for data sharing.

Ada95 has introduced the protected type construct to provide protected access to shared data without using the high-overhead rendezvous construct. Multiple tasks can safely access individual components of a protected record type simultaneously and the Ada95 runtime system will guarantee mutually exclusive access to the shared data.

2.3 Support for Real-Time Development

One of the major deficiencies of Ada83 in real-time applications is the non-negligible timing inaccuracies incurred by statements such as **delay**. For example, **delay T** only guarantees that the corresponding task will be delayed for *at least* **T** seconds. Moreover, a task in Ada83 does not have any control on its execution time. Another deficiency of Ada is that its scheduling rules are unsatisfactory especially with regard to the rendezvous. First-in-first-out queuing on entries and the arbitrary selection from several open alternatives in a **select** statement conflict with the normal preemptive priority rules. For instance, priority inversion occurs when a high priority task is behind a lower priority task in an entry queue. However, Ada95 has introduced the following to allow better real-time behavior:

- Ada95 offers more flexibility in the choice of priority and scheduling rules. It has, for instance, the ability to modify the priorities of tasks at run-time. This capability is provided by the **Ada.Dynamic-Priorities** package. Moreover, The Real-Time Annex of Ada95 bounds the duration of priority-inversion by the following policy. When a task enters a protected object, it inherits the priority of the protected object. The task regains its own priority when it exits the protected object. The bounded priority-inversion facilitates the timely execution of Ada programs.

- A more accurate real-time clock is available in Ada95. The Real-Time Annex provides the **Ada.Real-Time** package containing the **Real-Time.Time** type. The type **Real-Time.Time** is intended to represent a real-time clock with potentially smaller granularity than the time-of-day clock associated with

Calendar.Time. Furthermore, the value returned by **Real-Time.Clock** is guaranteed to be monotonically non-decreasing.

- Time-bounded actions can be realized by **select delay ... then ... abort ... end select** statement. If the execution of the piece of code after **abort** is not completed in the time-duration specified in the delay statement then it will be aborted and the code located between **select** and **then** will be executed.
- **delay until** statement should be used to delay a task until a particular point of time.

As will be discussed in subsequent sections, some of these new features were employed in developing the SWFT middleware. The tagged types were used for implementing reusable generic software components. We also used the child/parent packaging for developing fault-injection components [5]. Fault-injection components, which are not discussed here due to the space limitation, are responsible for injecting various faults in reusable components. Asynchronous select was used to prevent the infinite waiting for the occurrence of events.

3 Software Fault Tolerance

Due to the inadequacy of fault avoidance and fault removal techniques in eliminating errors from large-scale software, highly reliable software for complex applications is possible only through software fault tolerance. SWFT is normally achieved by using redundancy and/or diversity in the system or some of its components. Most existing SWFT schemes [6] tolerate design errors in the application software by providing multiple versions of the application, assuming that the same design errors will not be committed by different application programmers using different tools.

N-Version Programming (NVP) [7] and *Recovery Block (RB)* [8] are two most widely used SWFT schemes. The NVP scheme uses multiple functionally equivalent versions of critical software with a voting mechanism for fault detection and result selection. To achieve a better response time, each version should run on a separate computing node. This parallel execution also facilitates tolerating hardware failures.

Software structuring in the RB scheme is based on the recovery block abstraction [8] which has the following syntax:

ensure T by P
else by A_1
...
else by A_n
else error

Here, T denotes the acceptance test, P the *primary* try block, and $B_k, 1 \leq k \leq n$, the *alternate* try blocks. All the try blocks are designed to produce the same or similar results. The acceptance test is a logical expression representing the criterion for determining the acceptability of the results of the try blocks. Prior to the execution of a try block, a recovery checkpoint is established by saving the current system state. The execution of a try block is followed by an acceptance test. If an error is detected during the execution of a try block by the acceptance test, then the system will rollback (i.e., restoring the last checkpoint) and retry the application by using the next try block.

The Distributed Recovery Block (DRB) scheme [9] is an extension of the RB scheme to achieve a better response time. Under the DRB scheme, try blocks and the acceptance test are replicated and executed concurrently on different computing nodes. Figure 1, in which the control flow in the fault-free scenario is distinguished by heavy lines, depicts the structure of the DRB scheme with two nodes.

The *active* node normally executes the primary block and produces the output, while the other node, the *shadow* node, executes the alternate block and acts as a standby backup. The shadow node takes over the control when the active node fails the acceptance test or crashed.

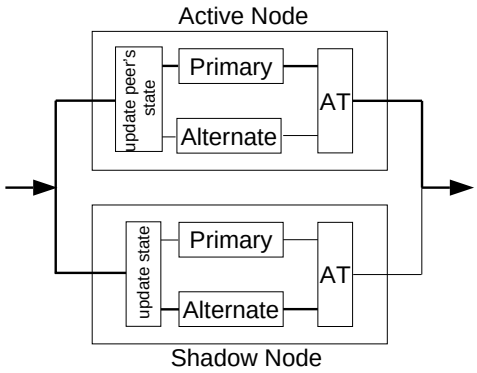

Figure 1: Structure of the Distributed Recovery Block

Several other SWFT schemes have been proposed in last two decades [10, 11, 12, 13, 14]. To decide on which SWFT schemes should be included in the SWFT middleware, we performed an in-depth analysis of the SWFT domain. Maximum care was taken to make sure that the analysis covers majority of existing schemes with applicable approaches for tolerating software faults. The analysis resulted in a taxonomy of existing SWFT schemes based on the type of redundancy employed. A detailed discussion on the selection criteria and the chosen SWFT schemes can be found in [15].

4 SWFT Middleware

There are two main objectives for building the SWFT middleware:

- Separating fault-handling concerns of fault-tolerance services from functional concerns of the application.
- Achieving component-level reuse in implementing various SWFT schemes.

To fulfill the first objective, the SWFT middleware includes a variety of SWFT schemes to meet different fault-tolerant requirements of the applications. The middleware encapsulates the implementation of SWFT services and thereby facilitates isolation of fault-tolerance functionality from the application. It also provides a set of templates (denoted as the Interface components) for the application to interface with the SWFT services.

To achieve the second objective, the middleware is constructed from a set of reusable software components from which various SWFT schemes are composed. Besides a high degree of reusability, component-based implementation of SWFT schemes facilitates platform-independent testing of the SWFT services by injecting various faults into individual components.

Figure 2 depicts the overall structure of the SWFT middleware and its sub-layers. The SWFT-Component layer is discussed in details here. The next section presents the Interface layer.

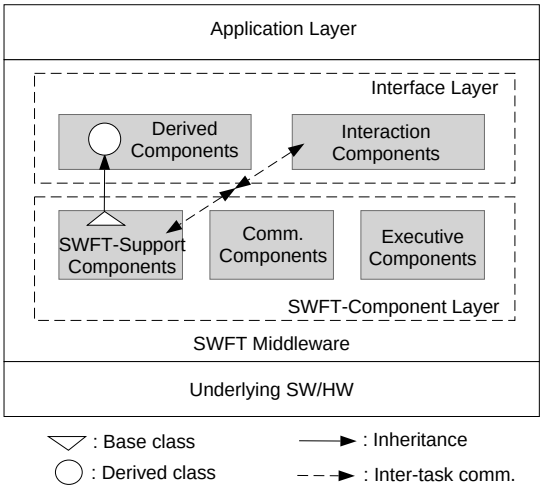

Figure 2: The Overall Structure of the SWFT middleware

To identify reusable components, an object-oriented analysis of the well-established SWFT schemes was conducted [16]. In this process, maximum care was taken to make sure that (i) the components capture the common functionality of a wide variety of SWFT schemes, and (ii) application-specific functions are identified and isolated from the the generic SWFT components. The identified reusable components were then classified as *Generic SWFT-Support* components, *SWFT Executive* components, and *Communication* components, as shown in Figure 2.

Analyzing various SWFT schemes shows that some fault-tolerance services are of generic type and are normally employed by several schemes. As an example, several SWFT schemes such as *Retry Block* [10] and *Recovery Block* schemes employ the checkpointing mechanism for a possible roll back and retry. Hence, a component with procedures to establishing and restoration of checkpoints can be shared by several SWFT schemes. Such components are classified as generic SWFT-Support components. In Section 7, some of these components will be utilized for the implementation of the DRB scheme.

Functionality of some of these generic components are application-dependent. For example, the data to be saved and restored by the checkpointing component is highly dependent on the application. In order to be applicable to various applications, generic SWFT-support components such as the checkpointing component should be developed in the form of base (very likely abstract) classes. However, subclasses of these generic classes will be derived and included as the Interface components. This will allow the reuse of the SWFT components for different applications without any change.

For each individual SWFT scheme, there should be a component which is responsible for managing orderly execution of the scheme: (1) initialization of the execution of the scheme including the activation of various tasks, (2) managing redundancy employed, and (3) producing a correct output. Such components are classified as *SWFT Executive* components.

Finally, generic *Communication* components are provided for reliable inter-node communications.

5 Integration of the SWFT Middleware with the Application Modules

The components in the SWFT middleware are designed in a way that they can be employed by various applications with no or minimal changes. The SWFT-Component layer is completely isolated from the application by the *Interface layer*. The components in the Interface layer encapsulate all the application-dependent features facilitating a clean interaction between the SWFT components and application modules, as shown in Figure 2.

Interface components are further classified into the following two types:

- *Derived Components*:
  As discussed earlier, some of the generic SWFT-support components are developed as abstract classes in order to be application-independent. The inheritance mechanism can be used to augment application specific functionalities to these abstract classes. The Interface layer provides templates for such subclasses. For example the checkpointing component can have a subclass with procedures for establishing and restoration of the application states. Templates

for such subclasses are called the Derived components and are provided in the Interface layer.

- *Interaction Components*:
  Obviously, SWFT components have to communicate with the application. This, however, may contradict with the genericity of the SWFT components. This is especially true, when the implementation platform does not provide efficient message passing communication facilities. For example, a natural way for the inter-component communication in Ada is rendezvous. For making a rendezvous between two tasks, one of the task must know the name of the other. To realize a loose interdependency between SWFT components and the application, the Interaction components (as part of the Interface layer) are introduced to act as communication interfaces between the SWFT components and the application. The templates of the Interaction components are also provided in the Interface layer.

To integrate an application with the SWFT middleware, the application developer should complete the templates provided in the Interface layer. Completion of the templates can be done automatically using a user-friendly graphical use interface (GUI). Such a GUI is currently being developed.

6 Implementing SWFT Middleware in Ada95

We now discuss how the reusable components and the SWFT middleware were implemented in Ada95. In the course of this discussion, the components will be classified according to their invocation behavior. Each class will then be mapped into a suitable Ada95 construct. Possible approaches for implementing inter-component communication in Ada95 will also be discussed.

6.1 Mapping Reusable Components into Ada95 Constructs

The reusable components are classified into three categories:

- *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.

Each individual category is mapped into a suitable Ada95 construct in the following way:

- A passive object is implemented as an Ada package which does not own 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 clients 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 guarantees mutually exclusive access to the shared data.

6.2 Tightly-Coupled Objects vs. Loosely-Coupled Objects

Two approaches to the implementation of object interactions in Ada95 may be used. The first approach, called *tightly-coupled* interactions, employs the rendezvous concept for the interaction between 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 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 our implementation, the SWFT middleware and the application were considered

as two separate subsystems and their interactions were implemented using loosely-coupled interactions.

7 Middleware Implementation of DRB with Ada95 support

This section describes component-based implementation of the DRB scheme in Ada95.

Figure 3 depicts components used in implementing the DRB scheme and their control and data flow. Due to the space limitation, individual components are not discussed here. However, the role of the Application Execution Interface component needs some clarification. 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.

7.1 Tagged Types for Inheritance

The ultimate goal in designing the SWFT components is their maximum reusability. These components should be designed in the way that they can be reused by various applications 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 checkpoints, is highly dependent on the application. The inheritance mechanism provided by Ada95 are used to separate application-specific concepts from application-independent features. 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 `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.

7.2 Asynchronous Transfer of Control for Time-Bounded Computations

There are occasions during the execution of the SWFT schemes where an ongoing action should be abandoned when specific conditions (such as when the execution time of an activity exceeds a predetermined limit) arise 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 should receive the result of the execution by a predefined time-limit, say T. If T time-units elapsed and DRB Executive has not received 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 course of action can be done using "select-then-abort" [3] statement introduced in Ada95 as shown below.

```
SEND_ORDER_TO_TRYBLOCK;
select
   delay MAXIMUM_APPLICATION_EXECUTION_TIME;
   TAKE_RECOVERY_ACTION;
then abort
   WAITING_FOR_APPLICATION_RESULT(APPLICATION_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 suggests the execution of the `WAITING_FOR_APPLICATION_RESULT` procedure until `MAXIMUM_APPLICATION_EXECUTION_TIME` seconds elapse. If the control does not return from the procedure by this time-period, then the execution of `WAITING_FOR_APPLICATION_RESULT` will be abandoned

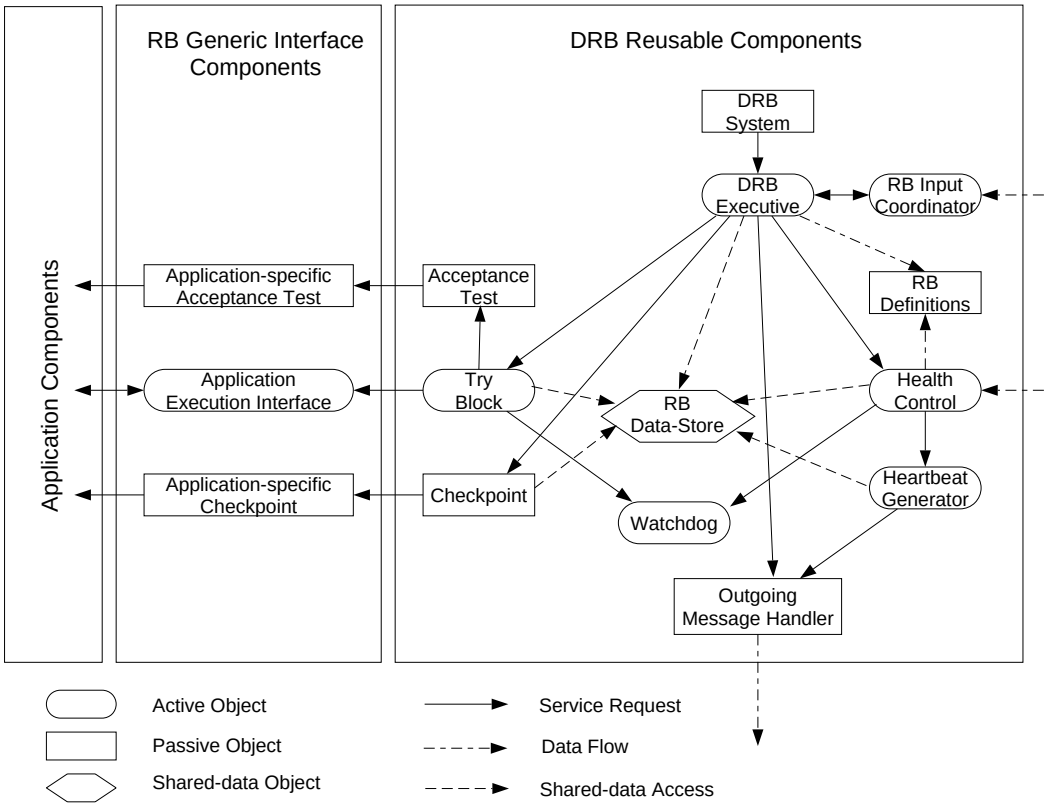

Figure 3: Component-based Implementation of the Distributed Recovery Block

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_APPLICATION_RESULT so that the task can be aborted as the time limit expires. Of course, dynamic creation and abort of tasks is not recommended in critical applications.

7.3 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 especially for data-sharing. It requires additional tasks to manage the shared data which often leads to a poor performance. Ada95 introduced the concept of protected types which provides orderly and synchronous access to shared data with no additional tasks.

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 the shared data and can only be accessed through procedures GET_NODE_ROLE and SET_NODE_ROLE. For each shared data in item RB_DATA_STORE, possible ways to access the data were considered and implemented as procedures.

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

8 Conclusions

This paper described a research effort in developing a software fault tolerance middleware by which the complexity of the SWFT schemes is hidden from the application developers. The SWFT middleware thus provides an easy-to-use and clean interface between the SWFT schemes and the applications. It consists of a set of reusable components from which a variety of SWFT schemes can be implemented. Our experience demonstrated the effectiveness of the SWFT middleware in isolating fault-tolerance functionality from the application. The SWFT middleware was implemented using the real-time and object-oriented features of Ada95. The implementation illustrated that new features of Ada95 are very useful for developing complex real-time applications.

References

[1] J. Gray, "A census of Tandem system availability between 1985 and 1990," *IEEE Trans. Reliability*, vol. 39, pp. 409–418, Oct. 1990.

[2] A. Avižienis and J.-C. Laprie, "Dependable computing: From concepts to design diversity," *Proceedings of the IEEE*, vol. 74, pp. 629–638, May 1986.

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

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

[5] E. H. Shokri and K. S. Tso, "Ada95 object-oriented and real-time support for development of software fault tolerance reusable components," in *Proceedings of Second International Workshop on Object-oriented Real-time Dependable Systems (WORDS'96)*, (Laguna Beach, CA), Feb. 1996.

[6] P. A. Lee and T. Anderson, *Fault Tolerance: Principles and Practice*, vol. 3 of *Dependable Computing and Fault-Tolerant Systems*. Wien, New York: Springer-Verlag, 2nd ed., 1990.

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

[8] B. Randell, "System structure for software fault tolerance," *IEEE Trans. Software Engineering*, vol. SE-1, pp. 220–232, June 1975.

[9] 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. Computers*, vol. 38, pp. 626–636, May 1989.

[10] P. E. Ammann and J. C. Knight, "Data diversity: An approach to software fault tolerance," *IEEE Trans. Computers*, pp. 418–425, Apr. 1988.

[11] J. P. J. Kelly, T. I. Mcvittie, and W. I. Yamamoto, "Implementing design diversity to achieve fault tolerance," *IEEE Software*, pp. 61–71, July 1991.

[12] K. H. Kim and T. F. Lawrence, "Adaptive fault-tolerance in complex real-time distributed computer system applications," *Journal of Computer Communications*, vol. 15, pp. 243–251, May 1992.

[13] Y. M. Wang *et al.*, "Progressive retry for software error recovery in distributed systems," in *Digest of the 23rd Annual International Symposium on Fault-Tolerant Computing*, (Toulouse, France), pp. 138–144, June 1993.

[14] A. T. Tai, J. F. Meyer, and A. Avižienis, "Performability enhancement of fault-tolerant software," *IEEE Trans. Reliability*, vol. 42, pp. 227–237, June 1993.

[15] 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*, (San Antonio, TX), pp. 490–500, Mar. 1995.

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