A Configurable Fault Tolerant Architecture for Component-Based Systems

Farzana Yousuf, Aamer Nadeem, and Zahid Zaman
Center for Software Dependability
Mohammed Ali Jinnah University
Islamabad, Pakistan.
safar_284@hotmail.com, anadeem@jinnah.edu.pk, zahid_zaman81@hotmail.com

Abstract
Component-based software engineering is an emerging paradigm for software development. Benefits of component-based development include significant reduction in the development cost, time and improvement in the dependability requirements. Commercial off-the-shelf (COTS) components are used without any code modification and inspection. When such components are integrated to build safety-critical systems, faults within individual COTS components or faults due to their collaboration may cause catastrophic failures. Therefore, for systems with high dependability requirements, it is essential to incorporate ways for tolerating the software faults at architectural level to deal with the faults that are not catered within the components. The existing component based fault tolerant architectures provide fault tolerance through either exception handling or design diversity.

In this paper, we propose a fault tolerant architecture, which supports design diversity and exception handling fault tolerance strategies. The proposed fault tolerant component architecture employs special-purpose connectors called design diverse-multiple version connectors (DD-MVC). These connectors allow design diverse N-variants of COTS to run in parallel. Moreover, proposed architecture also has fault tolerant architectural level connector. The proposed architecture can be configured to adjust the tradeoff between dependability and efficiency and exhibits the ability to tolerate the anticipated and unanticipated faults effectively. The applicability of proposed architecture is demonstrated with a prototype implementation and a case study.

1. Introduction
Recent advancements in Component Based Software Engineering (CBSE) have given a new trend to building large and complex systems using pre-built components with definite architectures [1]. This software technology has gained much attention of the software industry and research organizations. The reason is that it amplifies reliability, quality of the system and saves time to market and development cost.

Now-a-days, designers rely on commercial off the shelf (COTS) components for component based software development (CBSD). These components perform well in their intended environment, but problem arises when they integrate to satisfy the requirements of a new environment. COTS components are used without any modification to their code. They may have either too fine or too coarse granularity for the system being developed. Furthermore, there is a possibility of specification errors in them [2]. When integrating such untrustworthy components to build systems with high dependability requirements, faults within individual COTS components or faults due to their collaboration may cause catastrophic failures. Thus, to avoid such situations and
to achieve high dependability requirements, a fault tolerance scheme is an accepted mechanism. Fault tolerant component based systems ensure that the component-based program has a fault tolerance mechanism to deal with the faults. As components are used “as it is” so neither code inspection nor modification can be made in them. Thus, a solution is needed at the architectural level to incorporate fault tolerance into component-based software.

Components are a major part of the software architecture that concentrates on only one aspect of development. Another aspect is the interaction among the components, which is handled by connectors. Connectors are the fundamental elements of a component-based architecture that separate communication from computation. Hence, in addition to the components level, fault tolerance should be provided at architecture-level connectors to detect and handle the faults that could not be handled inside the components.

The existing fault tolerant component based architectures either provide fault tolerance through exception handling or design diversity. Exception handling is the most common and widely used methodology for detection and handling of anticipated faults. In exception handling, occurrence of faults transfers the flow of control from normal component to abnormal component. Then, appropriate handler in abnormal component handles the raised exception. If it is not capable of handling the exception then it raises interface exception but if the request to provide the service is invalid then handler signals failure exception. Design diversity [3] is a significant and main fault tolerance strategy. It makes use of different implementations of software components that give the same functionality to tolerate the software faults. These implementations are called versions, variants or modules. In contrast to exception handling, design diversity has the capability to tolerate both the expected faults and unexpected faults.

It has been observed from the literature survey that most of the existing component based architectures provide fault tolerance through exception handling while only a few support design diversity. This shows that there is little work on fault tolerance through design diversity in component-based architectures.

An analysis of existing fault tolerant component based architectures shows that the existing architectures based on design diversity are not flexible as they do not provide different coverage of faults, lack the independent assumption of failure of versions and do not allow adjusting the tradeoff between dependability and efficiency at run time. Some of the architectures are not automatable while various provide fault tolerance at component level only. The C2 architectural style [4] is a commonly used style for component-based architectures, which separates computation from communication by introducing connectors. A major limitation of the existing fault-tolerant architectures based on C2 style is that they do not provide fault tolerance at architecture-level connectors. Some of the existing architectures are highly dependable but are complex that affect their performance. Among the twelve existing architectures, nine are not implemented.

In this paper, we propose a fault tolerant component based architecture called Fault Tolerant C2 Architecture (FTC2A). The proposed architecture is based on the C2 style and provides fault tolerance at both component and architecture-level connectors. It employs both exception handling and design diversity to tolerate the anticipated and unanticipated faults effectively. The proposed architecture assumes that the diverse variants are available as COTS components, which provide similar functionality but
have different designs. In general, the cost of providing design diversity in a system may be higher as compared to exception handling but in systems with high dependability requirements, the cost of the system is usually not a major consideration. Moreover, as Popov notes, the cost of providing design diversity is, in general, lower than the cost of improving the reliability of a single component [5].

The rest of paper is organized as follows. Section 2 gives a related work. Section 3 gives proposed fault tolerant component based architecture. Section 4 and 5 shows prototype implementation of proposed idea and case study. Evaluation of proposed architecture is illustrated in section 6. Conclusion and future work is given in 7.

2. Related Work

It has been observed from literature survey that most of the existing component based architectures provide fault tolerance through Exception Handling. There is little work on fault tolerance through design diversity in component-based architectures. This section discusses several other fault tolerant component based architecture. The criteria on the base of which discussion is made are architectures based on C2-style, architectures that employ exception handling at component and connector level, those use exception handling at component level and those based on design diversity.

The fault tolerant architectures based on C2 architectural style [6,7,8] presented the structure of fault tolerant component that is divided into normal and abnormal activity. These architectures detect the faults based on pre-conditions, post-conditions and invariants of the operation. Although connectors are considered as first class entities for communication but these architectures do not provide fault tolerance at architectural level connectors. Unlike our proposed architecture they belong to single version environment, but our proposed architecture belongs to multiple version environments and provides exception handling at C2 connectors. These architectures considered the concept to add redundancy through specialized exception handler but they do not properly mentioned how design faults are cater through handlers. Our work is distinct from above mentioned work as it employs different design diverse OTS as versions and apply fault tolerance technique to deal with errors. For integrating COTS in [9] author use protective wrappers that provides application-dependent fault tolerance capability. One COTS item is wrapped by connectors that are composed of eight ABCs coupled with error handler. The overhead of architecture is increased when request is passed through protected interface, which enhances its complexity also because of assertions embedded in the protector.

The other contributions in providing fault tolerance in component based systems at architectural level is done by work of authors [10,11,12,13] that considers only exception handling at local and global level. They are effective in providing fault tolerance through exception handling mechanism but due to their complexity they are either not scalable or not automatable.

Some of the contributions [14,15,16,17] considers the components as only the place to make fault tolerant.

Redundancy by mean of design diversity is used in order to obtain fault tolerance in systems. The basic aim of design diversity is to minimize the coincident failure but the work in [18,19] although considers multiple versions but they do not assumes adequate goal of design diversity. These architectures do not assume assumption of independent
failures. Furthermore, both of these architectures return the result of only authoritative version to the client and use voting only for resolving situations where no authoritative is decided.

The research work from where our work has been inspired is the concept of MVC [19] where special-purpose connectors called multi versioning connectors (MVC) are used to run old and new versions in parallel. A new version in MVC provides either same functionality, some additional functionality or has some overlap with old version functionality but with different implementation. In case of fault, engineer does the decision about the version’s replacement, which is an overhead of involvement of an engineer. Furthermore, this architecture does not describe how versions are used. Like their contributions, we also consider N (multiple) versions but these versions provide same functionality with different design or implementation. Our work is distinct from this work, as we do not consider any version as authoritative so the correct output of versions is voted upon by voting mechanism. In addition to this, connectors of DD-MVC are responsible for replacement of versions so no engineer is involved to make decision about re-configuration. Unlike our work, MVC does not provide fault tolerance at architectural level connector and it is not flexible, general and uses maximum resources to give result. Our work is flexible as it provides different fault coverage, can behave as other fault tolerance techniques depending on requirements and use least number of resources to produce result. Like MVC, new versions can also be easily inserted into the system.

3. Proposed Architecture

This section describes the proposed fault tolerant component based architecture called FTC2A (Fault Tolerant C2 Architecture) [20] that is based on Chiron-2 (C2) architectural style [4], exception handling and Self Configuring Optimal Programming (SCOP) scheme [21]. FTC2A is a generic architecture that provides fault tolerance by combining the capabilities of exception handling and design diversity fault tolerance strategies. It uses design diverse COTS components and makes a majority voting system by using SCOP. It is a flexible architecture as it provides different levels of dependability or fault coverage at run time.

C2 architectural style is a layered structure used for the development of component based and message based systems. The basic reason for choosing C2 architectural style is that it has the capability to assimilate different off the shelf components having different low-level design or implementation. SCOP is a fault tolerance technique and the rationale for using SCOP approach is that it is an efficient, cost-effective, generic and flexible scheme for tolerating software faults. SCOP can enhance run-time efficiency and provides dynamic tradeoff between parallel and sequential execution of variants by dynamically selecting the variants according to the delivery conditions given by the user. The other techniques for tolerating design faults are neither generic nor flexible. SCOP allows changing its dependability conditions at run time to avoid any catastrophic situation but the other fault tolerance techniques change their dependability conditions only after the occurrence of failure.

3.1. FTC2A
FTC2A is composed of two parts, i.e., FTC and DDMVC, which provide fault tolerance at connector level and at component level respectively. At the component level, fault tolerance is provided by Design Diverse-Multiple Version Connectors (DD-MVC) while at the connector level, Fault Tolerant Connector (FTC) provides tolerance against faults by using connector-level exception handler. According to the topological and communication rules of the C2 style [4], components can interact with each other only through connectors. The direction of the request is upward, i.e., from lower components to upper components and direction of notification is downward, i.e., from upper to the lower component as shown in the figure 1.

![Figure 1. Basic elements of C2 style (Reproduced from [4])](image)

In fact, components and connectors in existing C2 style are not fault tolerant. They neither notify exceptions nor do they understand the exceptions raised by other C2 components. To resolve this situation, C2 messages in proposed fault tolerant component based architecture are extended as described by [6], i.e., the following extensions are made:

- The request of service and response to service are simple C2 messages – request and notification respectively.
- The exception raised by FTC or DD-MVC is considered as notification’s subtype.

The resultant configuration of C2 style by embedding DD-MVC and FTC is shown in the figure 2.

### 3.2. Components of the FTC2A

The purpose of this section is to describe the structure and the functionality of the components of FTC2A as shown in the figure 3.

#### 3.2.1. Fault Tolerant Connector (FTC):

Connectors are an important part in software architecture that separate communication from computations. Consequently, they must also be made fault tolerant so that they can detect and handle errors that cannot be handled inside the components.
Figure 2. C2 Configuration with FTC2A

In FTC2A, connector in C2-style is also made fault tolerant and called Fault Tolerant Connector (FTC). All communication among components is performed through FTC. FTC sends a normal response if a request is valid; otherwise, an interface or failure exception is raised. Interface exception is raised when the service request is not valid and failure exception is signaled when user’s request is not serviced due to time deadline or failure in satisfaction of delivery conditions. It has the capability to deal with:

- Input related faults.
- Internal logic errors of connectors, which may occur while filtering the messages, broadcasting and routing the message.
- Communication errors between connector and components attached to it.

3.2.2. DD-MVC: DD-MVC is a special-purpose connector in which multiple COTS components with the same functionality are wrapped as diverse variants. DD-MVC allows multiple variants of COTS components to execute in parallel. DD-MVC is composed of two parts, i.e., Lower DD-MVC and Upper DD-MVC, set of variants of the components, a set of delivery conditions that may be chosen at run time. DD-MVC interacts with the other C2 components by requesting or providing services. Each component and connector in DD-MVC has a top, a bottom interface, i.e., the top interface of Lower DD-MVC is connected to bottom interfaces of COTS components, and bottom interface of Lower DD-MVC is connected to top interface of FTC as shown in the figure 3.

Lower DD-MVC: Lower DD-MVC plays the role of the controller. It receives the request from lower component through FTC, invokes Upper DD-MVC and multiple variants of COTS components. It is involved in:

- Distribution of inputs to the variants and invoking of Upper DD-MVC.
- Sending notification to the caller component through FTC.
- Selecting variants’ subset at run time according to the delivery condition.
- Provides support to the different types of decision mechanisms depending upon delivery conditions.
**Upper DD-MVC** : Upper DD-MVC acts as an adjudicator. It compares the results of variants by voting mechanism to select the output and perform error compensation. It performs following functions:

- It takes result from appropriate subset of variants, and produces the result, if any, by verifying the delivery conditions.
- It performs majority voting on the results of variants.
- It is also responsible for fault detection and handling of individual COTS components erroneous output.

![DD-MVC and FTC Overall Structure](Image)

**3.3. Delivery Conditions**

DD-MVC delivers the result of variants based on delivery conditions. Delivery conditions are the criteria or the user’s requirements to deliver the result, which may be given by the user at run time or are default optimal values. User can provide the delivery condition and few mandatory values by placing them in the configuration file. Different delivery conditions can have diverse coverage of faults. Delivery conditions may contain following user’s requirements [21] and shown in the figure 4.

**3.3.1. Maximum Number of Phases:** Delivery condition may contain number of phases to make dynamic use of redundancy by using minimum number of variants to provide the results within specified timing constraints. The reason for running the process in phases is to reduce the wastage of resources. Each phase involves selection of a subset of variants from the total variants available in the system. The variants that were not selected in any of the preceding phase are used in the current phase. If number of phases is not provided by the user then the number of phases is determined by \((N - |V_1| + 1)\), where \(N\) = total variants available and \(V_1\) = set of the variants to be executed in the first phase as specified in SCOP [21]. In the above-mentioned formula, \(V_1\) depends
on the number of agreeing result required. Suppose in the 1st phase, $V_1 - 1$ variants produce the similar results. In order to fulfill the requirement, 1 more variant should be executed. Now, in the 2nd phase, 1 more variant is executed. Now, if selected variant does not produce the similar result then in the next phase again 1 variant will be executed. This process may continue until all $N - |V_i|$ remaining variants have executed. Thus, maximum phases will be $N - |V_i| + 1$ where 1 is added for the 1st phase. The maximum number of phases is represented by maxPhase in the configuration file as shown in the figure 4.

3.3.2. Timing Constraints: Other dependability requirement in delivery conditions is timing constraint to deliver the result within the specified time. If no timing constraints are specified then default optimal values are used which is the time of slowest variant in the current phase. At the start of each new phase, timing constraints are checked. If Time is out then degraded service can be provided to the user. The timing constraint is represented with maxTime in the figure 4.

3.3.3. Maximum Number of Faulty Variants: Third delivery condition is the maximum number of faulty variants allowed in the system. The faulty variants mean maximum number of variants that can produce faulty output and do not pass an Acceptance Test (AT). During execution of Upper DD-MVC, if number of faulty variants crosses the specified number of faulty variants then failure state is reached. It is represented by maxIncorrectVariants in figure 4. If no value is given by the user then default optimal value should be $N - k$ is selected where $N$ = total variants available and $k$ is the number of agreeing result required. If more than, $N-k$ variants produce, the faulty result then specified number of agreeing result cannot be obtained and the result cannot be provided to the user according to the requirement.

3.3.4. Subset of Variants: The minimum number of variants executed in the phases depends on the number of resources available in the system. In single processor, threads are used to run the variants in parallel. The selection of the subset of variants is depends on the minCorrect Variants value specified in the configuration file as shown in the figure 4. If no value is given, then it configures the set of variants “$V_i$” depending on the default minimum correct variants to satisfy conditions in the absence of faults. Minimum correct variants mean minimum number of variants that produce the correct result. In design diverse system, to make a voting atleast two variants are required so its default value will be two. During the voting, if, atleast, the specified number of agreeing results of variants is obtained, then success state is reached.

```
totalVariants=7
maxIncorrectVariants=2
minCorrectVariants=4
interfaceName=ddmvc.components.cots.Sort
maxTime=1
maxPhase=5
```

Figure 4. Configuration File

Besides delivery conditions, the configuration file also contains some mandatory parameters such as:
**totalVariants:** Total variants available in the system that can be used to compute the results.

**interfaceName:** Complete name of the interface implemented by the variants, it also includes the package like `dmvc.components.cots.sort`.

Lower DD-MVC, upon receiving the request for service, validates the delivery conditions. If the user gives no delivery conditions, the default optimal values are used. The Upper DD-MVC then uses these conditions as read-only values at run time to deliver the results according to user’s requirements.

### 3.4. Working of FTC2A

According to the communication rule of C2 style, requests flow upward to the layers of system and notifications flow downward. Therefore, a service request is made to the DD-MVC from lower component. As direct communication cannot occur between components, so FTC receives request. Working of FTC2A is explained in the following steps:

- **Step 1:** Upon receiving the request, FTC checks for any kind of incompatibilities between COTS and rest of the system that might include wrong method name, incorrect number of parameters and method’s return type. If any input related fault or communication fault occurs, the user is asked to retry, otherwise the request is sent to the Lower DD-MVC.

- **Step 2:** FTC starts its timer to check whether notification is received from Lower DD-MVC within the specified time or not.

- **Step 3:** Upon receiving the request and user requirement, Lower DD-MVC validates the delivery conditions that are configured in the configuration file.

- **Step 4:** Lower DD-MVC then loads all the “N” variants, with different execution time, available in the system. Each variant implements the same interface so for identification of each variant, an instance of variant is created and then checks that whether that instance belongs to an interface or not. If that instance of variant belong to an interface then that variant is executed.

- **Step 5:** After loading, it decides on the number of phases during which results is obtained.

- **Step 6:** After deciding on the number of phases, Lower DD-MVC selects set of minimum number of variants that can be executed to verify the user’s requirement.

- **Step 7:** Lower DD-MVC then invokes the set of variants, with minimum response time, and invokes Upper DD-MVC as well to get the results by majority voting.

- **Step 8:** The variants perform their computation and the result of execution is stored in syndrome whose size increases as more phases are executed. A syndrome is a data structure that is created at run time and contains the result of variants along with their name, which is used by an adjudicator at the end of each phase to find out the result.

- **Step 9:** Upper DD-MVC then performs adjudication on the results to find out the results, if any, that satisfied the delivery condition. Acceptance test, part of Upper DD-MVC, is used to check the erroneous output of an individual COT(s). Upper DD-MVC
also checks the states that have been reached after the completion of each phase. These states can be:

- Terminal state (T) where delivery condition is satisfied and correct result is obtained within specified timing constraints.
- Intermediate state (I) where obtained result does not satisfy delivery condition but remaining set of variants i.e., $N - V_1$ can be used to get required result that meet the requirement of the user.
- Failure state (F) occurs when there are no remaining variants to be executed or time specified for giving result has been exceeded.

The delivery conditions can also be changed dynamically before the start of a new phase when degraded services can be considered. In this situation, if before starting next phase, timing constraint has been accomplished and set of variants is empty, then signal is raised to the user that the result cannot be produced within specified time. This process continues until either success or failure condition occurs. Upper DD-MVC then signals either result in case of success or signals failure.

For the reconfiguration purpose, Upper-DDMVC saves the results of the variants with their names in two files called correctFile and incorrectFile. The correctFile contains results and variants’ names that have passed the Acceptance Test. File incorrectFile contains results and variants’ name of those variants that produce incorrect result. At start, both files are empty.

4. Implementation of FTC2A

In this section, we describe the prototype implementation of the proposed FTC2A architecture. We have implemented FTC2A by using Java, IDE Eclipse and JDK1.6.

The implementation of FTC2A fulfills the topological and communication rule of C2 architectural style. For implementation, we have used C2 framework (Medvidovic et al., 1997) of C2 style that has been implemented in Java. C2 framework is composed of interfaces of Components, Connectors and Messages. Existing C2 framework is not fault tolerant. We have used that existing framework and implemented our main classes of FTC2A by using its interfaces of components and connectors to make it fault tolerant. The algorithm of FTC2A is given below that is split in FTC algorithm and DD-MVC algorithm respectively. In figure 5, an algorithm for FTC is given.

In our algorithm, upon receiving the request, FTC starts its functionality by calling startComputation() method. This method takes name of method given by the user and list of parameters as arguments. Inside the method startComputation(), validateInput(), checkAck(), checkValue and checkCommError() are called. The method validateInput() checks for any kind of input related faults. In this method, Reflection mechanism available in Java and is used to find out the classes and methods of classes at run time. Through this mechanism, it is checked that whether the name of the method(s) and types of argument(s) match with the name and parameters type given in the interface. If the type of arguments or method name does not exist then error occurs and ask user to retry. Otherwise, method checkAck() is called to set the time to get the acknowledgment from Lower DD-MVC check whether the time is out or not. If time is not out then LowerDDMVC.startPhase() method is called. Methods checkValue() and
checkCommError() are used to checks for any kind of internal logic errors of FTC itself, and the communication faults between connector and components and between connectors respectively.

![Algorithm for FTC](image)

Next, request goes to Lower DDMVC by calling startPhase(). This method takes method name and parameter list as arguments.

We modified SCOP algorithm provided in [23] by providing reconfiguration process and degradation services as shown in figure 6. Furthermore, the decision to run the variants in a phase is not only based on timing constraints but also on the minimum correct results required by the user. In algorithm for DDMVC, when startPhase() is called then Lower DD-MVC validates the delivery conditions and based on delivery conditions decides on the number of phases, minimum correct variants and maximum time to produce the result according to user requirement or using default optimal values. Lower DDMVC calls loadClasses() method to load the available variants in the system. loadClasses() method further calls LowerDDMVC.getClasses() method to get
the package name that contains all the available variants. Every variant implement the
same interface, so for identification of a variant, an instance of loaded variant is
created. This instance is checked using instanceofLoadedClasses instanceofInterface. If
an instance belongs to that interface then that variant is executed. LowerDDMVC.startVariant() method is used to execute one of the variant from the
variant list that saves its result in the resultList. LowerDDMVC.getResult() method invokes upperDDMVC.getResults() method to perform adjudication on the result of
variants stored in the resultList. Inside the upperDDMVC.getResults(), method
compareResult() is used to verify the variants’ result. In this method Acceptance Test is
used to test the output of variants. The results of variants that have passed the
Acceptance Test remain in the resultList while results of variants that have failed the
Acceptance Test are removed from resultList. The method upperDDMVC.getExecutionState() returns the current state of the program which can
be Terminal state = T, Intermediate state = I or Failure state = F. If Intermediate state is
obtained then upperDDMVC.startNextPhase() is called. In this method, remaining
variants that were not executed in the previous phase are selected and executed but
selection of variants depends upon the number of agreeing result required. In this
method, again startVariant() is called. The method createFile() creates two files called
correctFile and incorrectFile for reconfigur ation purpose. The correctFile contains
results and variants’ name that produce correct result. File incorrectFile contains results
and variants’ name of those variants that produce incorrect result. At start, both files are
empty.

5. Computer Assisted Dispatch System (CAD): A Case Study

The case study of Computer Assisted Dispatch System (CAD), taken from [22], is
used to provide the ambulance services in an emergency. This CAD system uses
Geographic Information System (GIS) that contains design diverse COTS components
to perform sorting functions in order to get the optimal distance between incident
location and an ambulance. The chosen example is safety-critical because an erroneous
as well as a late rescue of an emergency service may have disastrous results.

Fan used client server style as system architecture. The GIS COTS component is not
fault tolerant in that style. Moreover, author has identified failure mode of GIS
component, perform fault tree analysis and determine critical level for each failure
mode. However, he used manual process for searching the map. Besides this, we used
C2 style as system architecture that can incorporate heterogeneous off the shelf
components. We use different COTS component, for performing sorting, as variants and
make majority voted system where COTS can give result even when at least one variant
is remain there to produce the correct result. The overall C2 configuration of FTC2A is
represented in the figure 7.

Geographical Information System (GIS): The main functionality performed by GIS
in DD-MVC is as follows:

- Map display: Map display is used for displaying digital map on the base of
  address received during receipt of the calls.
- Address searching: It is used to acquire address of incident at level of street
  according to name of street name / postal code / main sites such as buildings,
parks or mosques or it can automatically identifies the location of payphone.
Figure 6. Algorithm for DD-MVC

Inputs: methodName:String, param:Object[]
Output: finalResult.

Initialize: totalVariant, resultList, State = N, maxPhase, minCorrect, maxTime, maxFail.

Procedure:
1. call validateDeliveryConditions()
2. call validatePhase()
3. call validateMinCorrect()
4. call validateMaxTime()
5. call validateTotalVariants()
6. call loadClasses() // load all variants in the system.
7. call getClasses() // get the package name containing variants.
8. If valid package name
    then load classes
9. Else
    print “package name is not valid”
End If
10. ArrayList<Class> variantList;
11. LowerDDMVC.methodName = methodName
12. LowerDDMVC.parameters = param
13. call LowerDDMVC.getTimeBound()
14. While (maxPhase <= totalVariant - minCorrect +1 and State = N) perform Steps 19-62.
15. Repeat the steps 21-58 until minCorrect variants are executed.
16. If maxTime
    String decision = print “for degraded service press Y”
    If (decision = ‘Y’)
        Then Continue
    Else
        Exit
    End If
End If
17. If maxfail ≠ 0
    tempResult = startVariant();
    Class loadedClasses;
    Object obj = loadedClasses.newInstance();
    If obj instanceof InterfaceName
        Method[] methods = loadedClasses.getMethods();
        Initialize i = 0;
18. Repeat step 35-37 until i < methods.length
19. If methods[i].getName() = methodName
    return variant result
    minCorrect ++;
End If
20. UpperDDMVC upperDDMVC;
21. call LowerDDMVC.getResult()
22. result := upperDDMVC.compareResult(TempResult);
23. If (result = true)
    minCorrect --;
Else
46. maxFail --;
47. End If
48. resultList.add(tempResult);
49. call upperDDMVC.getExecutionState()
50. If (result = null && maxPhase = totalVaraint - minCorrect + 1
   && maxTime = true && maxFail > totalVaraint - minCorrect)
   State = F
51. Else If (result != null && maxPhase <= totalVaraint -
   minCorrect +1 && maxTime = false && maxFail <= totalVaraint -
   minCorrect)
   State=N
52. Else
   State = I
53. End If
54. call LowerDDMVC.startNextPhase()
55. End While
56. finalresult := upperDDMVC.getResults(resultList);
57. upperDDMVC.createFile(resultList);
58. maxPhase --;
59. End While
60. If State = N
   Then deliver finalresult
61. Else Failure
62. End If
63. End If

Figure 6. Algorithm for DD-MVC (Continued)

Figure 7. Resulting Configuration of FTC2A in C2 Style
• Distance searching between two locations: It is used to search the optimal distance between two locations for most favorable allocation of resources.

• Search vehicle Status: It is used to determine the mobilization of vehicles.

The basic reason for making GIS component as a fault tolerant component is that if a map is not displayed within time or fails to display correct map, fails to search the address or calculates wrong address. Moreover, if it fails to revise the status of vehicle or takes more time then specified to revise the status then it would be critical to provide the services within time at desired incident’s location with desired resources. Hence, results could be catastrophic.

To illustrate the effectiveness of FTC2A in handling expected and unexpected faults, different faults are seeded at fault tolerant connector level and at design diverse component level.

There is an interface for sorting the list of values given by user as shown in the figure 8.

```java
package ddmvc.components.Sorting;
public interface Sort {
    public Integer [] Sort (Integer [] list);
}
```

Figure 8. Sort Interface

5.1.1. Faults Tolerated by FTC

FTC can tolerate input related faults, internal logic errors of connectors, which may occur while filtering the messages, broadcasting and routing the message. It can also tolerate the communication errors between connector and components attached to it. Following are the different scenarios to show fault tolerance by FTC2A at FTC level.

**Input related faults**

FTC can tolerate input related faults and give signal to the user to retry, if user enters wrong signature of the sorting method or gives incorrect return type or gives incorrect method name. For example, in figure 4.3, the name of the function to sort the given list of values is sort. Suppose user gives “orts” instead of sort or some other function name that is not available with defined method name then in this situation, FTC detects the fault and signals that input is invalid. Similarly, if user gives float[] as return type of list and list input by the user is integer[] then again FTC signal the error of type mismatch and ask the user to try again with valid return type.

**Communication errors**

In order to find out communication errors, upon forwarding the request to the Lower DD-MVC, FTC starts its timer to check whether notification is received from Lower DD-MVC within specified time or not. In this case, FTC signals time out if acknowledgment is not received within the specified time as shown in the figure 4.4.

5.1.2. Faults Tolerated by DD-MVC
Upper DD-MVC is responsible for tolerating erroneous output by using Acceptance Test (AT). AT is used to validate the behavior of the system in order to tolerate the faulty results. An AT should be simple, effective to detect the expected faults. It should be reliable to not only reject the faulty behavior but also to minimize the introduction of more design faults.

AT is composed of application dependent assertions whose complexity can be reached to the complexity of implementing the actual system and can enhances run time outlays. It is intricate to build up AT. However, if AT is strong and effective, then there is better likelihood of tolerating errors but if it is weak, then it cannot be ensured that faulty behavior is correctly rejected.

Following are the scenarios where AT can pass or reject the correct or incorrect results of variants.

**In case of Strong AT (tolerates similar incorrect results)**

To tolerate the similar incorrect results or faulty behavior of variants, comparison algorithm is implemented in AT that checks the values from start of the array list to the end of the array list as shown below.

\[ a[i] \leq a[j] \quad i,j=1,2,..n \]

For all \( i < j \)

If any of the value in the array list is not at its appropriate position, AT will not pass the result of variant and keeps that result in incorrectFile while AT places correct result of variants in correctFile.

**In case of Weak AT (fails to cater similar incorrect results)**

In case of weak AT, it fails to reject the similar incorrect results. To show this, an algorithm is implemented in AT that will checks whether sum of array list given as input is equal to the sum of array list produce as output as shown below.

\[ \sum a[i] = \sum b[i] \quad i=1,2,..n. \]

Where \( a[i] = \) input array

\( b[i] = \) output array

However, it does not determine whether list is sorted properly or not. Each variant produces the result and submits to the Upper DD-MVC, where AT will check the results of variants. As AT is weak, so it can pass the incorrect results of variants as well and place their result in correctFile. Now, correctFile has both similar correct and incorrect results. Upper DD-MVC then performs the majority voting on the results and selects the outputs which are in majority. In this way, Upper DDMVC can be failed to find out the correct result which shows the inherent disadvantage of voting mechanism.

6. Evaluation of FTC2A

This section illustrates the evaluation of proposed fault tolerant component based architecture based on the definite parameters.

6.1. Dynamic use of redundancy

FTC2A decides at run time to execute the minimum number of variants depending on user’s requirement. Hence, it makes dynamic use of redundancy by utilizing minimum
number of resources and adjusts efficiency and dependability dynamically. It has been found from literature, any approach that adjusts the redundancy at run time is finest from technical and economical point of view [23]. Besides this, those fault tolerant component based architectures that are based on the concept of N-Version Programming use the maximum number of resources to produce the result.

6.2. Level of fault tolerance

Most of the existing architectures provide fault tolerance at the local level only. Only few architecture, based on exception handling only, provide fault tolerance at both local and global level. Moreover, the existing work, in literature, on C2-style provides fault tolerance only at local level, but our proposed idea provides fault tolerance at architectural level connector by tolerating faults associated with internal logic, input related, communication faults that exists between components and connectors or among connectors.

6.3. Generality

The existing architectures are either providing fault tolerance through exception handling or through design diversity. None of the existing architecture provides fault tolerance by combining the capabilities of both exception handling and design diversity. FTC2A provides the exception handling at architectural level connector and design diversity at component level to deal with expected and unexpected faults effectively.

6.4. Flexibility

FTC2A is also flexible as it provides different level of dependability or fault coverage by multiple users’ requirements selecting at the start of each phase. Thus, it provides different fault coverage and is distinct from existing work as they either provides partial or no flexibility and changes their reliability levels of services when failures appear in the system.

6.5. Automation

Automatable shows whether proposed architecture can be automated or not. It is obligatory requirement to find out the efficiency and applicability of architectures. Our proposed architecture is automatable because we have provided the algorithms for both FTC and DD-MVC.

6.6. Recovery Mechanism Used

Our proposed architecture uses both forward and backward error recovery mechanism. Through these mechanisms, proposed architecture can handle both permanent and transient faults and can efficiently use time and memory. Most of the existing architectures use only forward error recovery mechanism.

6.7. Effectiveness of Fault tolerance

This parameter is used for analyzing architectures based on exception handling only. Most of the existing architectures are highly effective in detecting and tolerating faults. FTC2A combines the capabilities of exception handling and design diversity to make a proposed architecture fault tolerant effective. It tolerates expected and unexpected faults by providing fault detection and recovery mechanism in both DD-MVC and in FT connectors.
6.8. Independent Assumption of Failure

The existing architectures based on the concept of design diversity are not assuming the independent assumption of failure. Hence, they are not able to cater failure on identical set of inputs. FTC2A meets the goal of design diversity by providing fault tolerance to common mode failures through design diversity by using diverse OTS.

6.9. Execution Overhead

Execution overhead is the time required to execute the fault tolerant component. Existing architectures has either high or medium execution overhead as they execute “N” variants in their best as well as worst case. While in FTC2A, Execution overhead is Low because in best case it executes minimum number of variants depending on the number of agreeing results required. In worst case it though execute “N” variants but still worst case of FTC2A is equal to best case of existing architectures based on design diversity. It is very rare that worst case of FTC2A occurs. Moreover, in FTC2A, voter does not wait for the execution of slowest OTS, as it starts comparing or voting on the result as soon as variant starts producing the result. Execution overhead is indicated by:

**Best case:**

\[ \sum T_i + T_l + T_u + T_{FTc} \]

- \( T_i \) = Time required to execute “m” variants and \( i = 1, 2, \ldots, m \) and \( m < n \). \( m \) is the minimum number of variants run at first phase and that fulfill the delivery condition.
- \( T_l \) = Time required to execute Lower DDMVC.
- \( T_u \) = Time required to execute Upper DDMVC.
- \( T_{FTc} \) = Time required to execute the fault tolerant connector.

**Worst case:**

\[ \sum T_i + T_l + T_u + T_{FTc} \]

- \( T_i \) = Time required to execute “n” variants and \( i = 1, 2, \ldots, n \).
- \( T_l \) = Time required to execute Lower DDMVC.
- \( T_u \) = Time required to execute Upper DDMVC.
- \( T_{FTc} \) = Time required to execute the fault tolerant connector.

6.10. Communication Overhead

Communication overhead shows the interaction among controller, an adjudicator and variants of components. The communication overhead of existing architectures based on design diversity are either have high or medium overhead because communication is decision makers and “n” variants. While in FTC2A, communication overhead is Low as Lower DDMVC invokes “m” variants and Upper DDMVC. An Upper DD-MVC takes result from “m” variants directly and gives result, if any, to the other C2 components in best case while in worst case Lower DDMVC invokes “n” variants and Upper DDMVC. An Upper DD-MVC takes result from “n” variants directly and gives result, if any.

**Best case:**

\[ L \cdot m(Variants) + L \cdot U + L \cdot FTC + m(Variants)\cdot U \]
where

\[ L \ast m(\text{Variants}) \text{ shows } m \text{ message from LowerDD-MVC to “m” Variants.} \]
\[ L \ast U \text{ shows } 1 \text{ message from LowerDD-MVC to UpperDD-MVC.} \]
\[ L \ast FTC \text{ shows } 1 \text{ message from LowerDD-MVC to FTC.} \]
\[ m(\text{Variants}) \ast U \text{ shows } m \text{ message from “m” Variants to UpperDD-MVC.} \]
\[ = m + 1 + 1 + m = 2m + 2 = 2(m+1) \]

**Worst case:**

\[ (L \ast n(\text{Variants})) + L \ast U + L \ast FTC + n(\text{Variants}) \ast U + U \ast L + \text{ph} \]

where

\[ (L \ast n(\text{Variants})) \text{ shows } n \text{ message from LowerDD-MVC to “n” Variants.} \]
\[ L \ast U \text{ shows } 1 \text{ message from LowerDD-MVC to UpperDD-MVC.} \]
\[ L \ast FTC \text{ shows } 1 \text{ message from LowerDD-MVC to FTC.} \]
\[ n(\text{Variants}) \ast U \text{ shows } n \text{ message from “n” Variants to UpperDD-MVC.} \]
\[ U \ast L \text{ shows } 1 \text{ message from UpperDD-MVC to LowerDD-MVC.} \]
\[ \text{Ph is a constant and shows the maximum number of phases that can occur.} \]
\[ = n+1+1+n+1+\text{ph} = 2n+3 +\text{ph} \]

### 6.11. Implementation

We have provided implementation of our proposed architecture to show its applicability by using Java, IDE Eclipse and JDK 1.6. An implementation of FTC2A fulfills the topological and communication rule of C2 style. For implementation, we have used C2 Framework that is composed of interfaces of Components, Connectors and Messages.

### 6.12. Decision Mechanism Used

Voting is used to decide on the majority correct results and Acceptance test is used to check the erroneous output of an individual OTS. While existing architectures, use voting mechanism for resolution of domain problem where more then one variant has an ability to fix the problem of same domain instead of performing majority voting in order to produce the correct result.

### 6.13. Tolerance for Similar Incorrect Results

FTC2A has an ability to cater similar incorrect results by using AT whereas none of the existing architecture caters this problem.

### 7. Conclusion and Future Work

In this paper, we have proposed a fault tolerant component based architecture that combines the capabilities of exception handling and design diversity to make a proposed architecture fault tolerant by tolerating expected and unexpected faults effectively. We have applied our work on case study of Sorting and on example of Computer assisted dispatch system that provide ambulance services in an emergency...
within time. To show the applicability of our proposed idea we have also provided its prototype implementation.

The existing component based fault tolerant architectures have been classified based on fault tolerance strategies and evaluated with definite parameters. It has been found from literature survey that fault tolerance in component-based software is available in lesser degree and the use of fault tolerance for dealing with design faults in them is still uncommon. Most of the work has been done on exception handling while little attention has been given to design diversity.

Our analysis showed that existing component based fault tolerant architectures based on the concept of design diversity lack adequate goal of design diversity. The goal of design diversity is to build independent and diverse versions that can perform their functionality even if one variant fails to give the output so that the system can tolerate the faults. Moreover, there is no mechanism for handling exception at connector level in many component based fault tolerant architectures. In exception handling, most of existing architectures are not implemented. However, some authors give formal verification of proposed architectures by using B method or timed automata while one architecture is validated by testing strategy to determine the fault tolerance capability and reliability of component based fault tolerant architectures. In design diversity, most of the component based fault tolerant architectures are not shown with case study. Furthermore, these architectures are neither generic nor flexible and use maximum number of resources to produce the results.

We have addressed most of the above-mentioned issues to proposed fault tolerant component based architecture called FTC2A. It can be effective in providing tolerance to common mode failures and provides fault tolerance at component and connector level.

FTC2A is flexible as it provides different coverage of fault depending on the requirement of the user. It is also generic architecture that combines both exception handling and design diversity fault tolerance strategies. The proposed architecture adjusts the tradeoff between dependability and efficiency at run time and exhibits the ability to tolerate the anticipated and unanticipated faults effectively. The communication overhead and execution overhead of FTC2A is low as compared to other existing architectures based on the concept of design diversity. The reason for this is that FTC2A always execute minimum number of variants to fulfill the requirement of user to produce the result within specified time. If maximum time to produce the result is expired then degraded service can be provided to the user. In FTC2A, Acceptance test is used to check the erroneous output of an individual OTS and then Upper DD-MVC performs adjudication on the result of OTS to find out the result, if any that satisfied the delivery condition. Hence, it can tolerate the problem of similar incorrect result.

As a future work, we plan to propose a FTC2A Framework that provides the interfaces for Fault Tolerant Components, Connectors and Messages.

References


Authors

Farzana Yousuf is a lecturer at the Riphah International University, Islamabad, Pakistan, and a member of the Center for Software Dependability at Mohammad Ali Jinnah University, Islamabad, Pakistan. She received her MS in computer science from Mohammad Ali Jinnah University, Islamabad. The work described in this paper is based on her Masters Thesis. Her research interests include software fault tolerance techniques and component-based systems.

Aamer Nadeem is currently an Associate Professor in the Department of Computer Science at Mohammad Ali Jinnah University, Islamabad, Pakistan. He is also Director of the Center for Software Dependability at Mohammad Ali Jinnah University. He received his MSc in computer science from Quaid-i-Azam University, Islamabad, MS in software engineering from National University of Sciences and Technology, and PhD in software engineering from Mohammad Ali Jinnah University. He is a professional member of the ACM.

Zahid Zaman is a student of MS (computer science) at Mohammad Ali Jinnah University, and a member of the Center for Software Dependability at Mohammad Ali Jinnah University, Islamabad, Pakistan. His research interests include software fault tolerance, and testing of aspect-oriented programs.