Chapter 14
Software Fault Tolerance: An Aspect Oriented Approach

Kashif Hameed, Rob Williams, and Jim Smith

Abstract Software fault tolerance demands additional tasks like error detection and recovery through executable assertions, exception handling, diversity and redundancy based mechanisms. These mechanisms do not come for free; rather they introduce additional complexity to the core functionality. This paper presents light weight error detection and recovery mechanisms based on the rate of change in signal or data values. Maximum instantaneous and mean rates are used as plausibility checks to detect erroneous states and recover. These plausibility checks are exercised in a novel aspect oriented software fault tolerant design framework that reduces the additional logical complexity. A Lego NXT Robot based case study has been completed to demonstrate the effectiveness of the proposed design framework.

Keywords Aspect oriented design and programming · executable assertions · exception handling · fault tolerance · plausibility checks

14.1 Introduction

Adding fault tolerance measures to safety critical and mission critical applications introduces additional complexity to the core application. By incorporating handler code, for error detection, checkpointing, exception handling, and redundancy/diversity management, the additional complexity may adversely affect the dependability of a safety critical or mission critical system.

One of the solutions to reduce this complexity is to separate and modularize the extra, cross-cutting concerns from the true functionality.

At the level of design and programming, several approaches have been utilized that aim at separating functional and non-functional aspects. Component level

K. Hameed (✉), R. Williams, and J. Smith
University of the West of England, Bristol Institute of Technology, BS16 1QY, UK
e-mail: Kashif3.Hameed@uwe.ac.uk; Rob.Williams@uwe.ac.uk; james.smith@uwe.ac.uk

approach like IFTC [1], computational reflection and meta-object protocol based MOP [2] have shown that dependability issues can be implemented independently of functional requirements.

The evolving area of Aspect-Oriented Programming & Design (AOP&D) presents the same level of independence by supporting the modularized implementation of crosscutting concerns.

Aspect-oriented language extensions, like AspectJ [3] and AspectC++ [4] provide mechanisms like *Advice* (behavioural and structural changes) that may be applied by a pre-processor at specific locations in the program called *join point.* These are designated by *pointcut* expressions. In addition to that, static and dynamic modifications to a program are incorporated by *slices* which can affect the static structure of classes and functions.

In the context of fault tolerance, an induced fault can activate an error that changes the behaviour of the program and may lead to system failure. In order to tolerate a fault, abnormal behaviour must be detected and transformed back by introducing additional behaviour changes (Exception Handler) or alternate structure adoption (Recovery Blocks, N-Version Programming) strategies.

The rate of change (ROC) of signals or data can be used to detect erroneous conditions that can help in tolerating faults and avoiding failures by triggering appropriate recovery mechanisms. ROC-based plausibility checks for error detection and recovery in the form of executable assertions have been addressed by Hiller in [5, 6] . In [7] the author utilizes dynamic signal values for modeling and predicting future sensor values. Unfortunately, these mechanisms will add to the complexity of the true functionality that could affect the overall dependability of the system. None of the previous studies propose the separation of these error handling concerns from true functionality. However Aspect Oriented Design and Programming approaches may be used to separate out these concerns from the true functionality of a computer based system.

In this paper the rate of change based executable assertions have been extended with more refined time bounded instantaneous and mean rate checks that reduce false positives and false negatives. Secondly an empirical method for determining the maximum instantaneous and mean rates of change has been devised.

The current work also proposes generalized aspect-oriented software fault tolerance design patterns. These design solutions provide an implementation framework to incorporate and validate the proposed ROC-based checks.

14.2 ROC Plausibility Based Error Detection and Recovery

Error detection is the basic step in deploying any fault tolerance strategy. Executable assertions are often utilized as an error detection mechanism. Rate of change (ROC) based plausibility checks on input and output data may be used to detect some erroneous conditions that could lead to failure. Although ROC based executable assertions have been addressed by Hiller in [5], these constraints are based on changes

Table 14.1 ROC assertions and recovery

ROC assertion (PC)	Recovery mechanism
Case: $\mathbf{y_i} > \mathbf{y_{i-1}}$ (increasing) PC1: $\frac{\mathbf{y_i - y_i - 1}}{\mathbf{t_i - t_{i-1}}} \leq \mathbf{r}_{\max-\mathbf{incr}}$	$\mathbf{y_r} = \mathbf{y_{i-1}} + \mathbf{r}_{\max-\mathbf{incr}}\Delta\mathbf{T}_{\mathbf{i}-1\to\mathbf{i}}$
Case: (PC1 & $\mathbf{y_i} < \mathbf{y}_{\max}$) PC2: $\frac{\mathbf{y_i - y_{i-1}}}{\mathbf{t_i - t_{i-1}}} \geq \mathbf{r}_{\min-\mathbf{incr}}$	If $\mathbf{y}_{\max} - \mathbf{y_{i-1}} \geq \mathbf{r}_{\min-\mathbf{incr}}\Delta\mathbf{T}_{\mathbf{i}-1\to\mathbf{i}}$ then $\mathbf{y_r} = \mathbf{y_{i-1}} + \mathbf{r}_{\min-\mathbf{incr}}\Delta\mathbf{T}_{\mathbf{i}-1\to\mathbf{i}}$ else $\mathbf{y_r} = \mathbf{y}_{\max}$
Case: $\mathbf{y_i} < \mathbf{y_{i-1}}$ (decreasing) PC3: $\frac{\mathbf{y_{i-1}} - \mathbf{y}_i}{\mathbf{t_i - t_{i-1}}} \leq \mathbf{r}_{\max-\mathbf{decr}}$	$\mathbf{y_r} = \mathbf{y_{i-1}} - \mathbf{r}_{\max-\mathbf{decr}}\Delta\mathbf{T}_{\mathbf{i}-1\to\mathbf{i}}$
Case: **PC**3 & $\mathbf{y_i} > \mathbf{y}_{\min}$ PC4: $\frac{\mathbf{y_{i-1} - y_i}}{\mathbf{t_i - t_{i-1}}} \geq \mathbf{r}_{\min-\mathbf{decr}}$	If $\mathbf{y}_{\min} - \mathbf{y_{i-1}} \geq \mathbf{r}_{\min-\mathbf{decr}}\Delta\mathbf{T}_{\mathbf{i}-1\to\mathbf{i}}$ then $\mathbf{y_r} = \mathbf{y_{i-1}} - \mathbf{r}_{\min-\mathbf{decr}}\Delta\mathbf{T}_{\mathbf{i}-1\to\mathbf{i}}$ else $\mathbf{y}_{\mathrm{r}} = \mathbf{y}_{\min}$

in variable values but without any bound on time. However true rate of change should employ the change in variable values in a specified time interval as asserted by Clegg [7]. Without considering a time boundary, there are more chances to have false positives and false negatives.

In order to apply various plausibility checks, it is first necessary to determine the characteristic range of values for key variables/signals. The characteristic parameters of variables assigned here are $\mathrm{y}_{\max}$(maximum value), $\mathrm{y}_{\min}$(minimum value), $\mathrm{r}_{\max\text{-incr}}$(maximum increase/sample time), $\mathrm{r}_{\min\text{-incr}}$(minimum increase/sample time), $\mathrm{r}_{\max\text{-decr}}$(maximum decrease/sample time), $\mathrm{r}_{\min\text{-decr}}$(minimum decrease/sample time).

When an error is detected a recovery mechanism is brought into service to avoid a failure and so tolerate the fault. The recovery mechanisms employed here are managed on the basis of running trends. The faulty data is replaced by computed values derived from past values and some increment based on the maximum and minimum rates of change. However, the forcefully assigned values are kept within the maximum and minimum data ranges as tabulated in Table 14.1.

14.3 Aspect Oriented Exception Handling Patterns

Exception handling has been deployed as a key mechanism in implementing software fault tolerance through forward and backward error recovery mechanisms. It provides a convenient means of structuring software that has to deal with erroneous conditions [8].

In [9], the authors addresses the weaknesses of exception handling mechanisms provided by mainstream programming languages like Java, Ada,C + +, C#. In their experience exception handling code is inter-twined with the normal code. This hinders maintenance and reuse of both normal and exception handling code.

Moreover as argued by Romanovsky in [10], exception handling is difficult to develop and has not been well understood. This is due to the fact that it introduces additional complexity and has been misused when applied to a novel application domain. This has further increased the ratio of system failures due to poorly designed fault tolerance strategies.

Thus fault tolerance measures using exception handling should make it possible to produce software where (a) error handling code and normal code are separated logically and physically; (b) the impact of complexity on the overall system is minimized; and (c) the fault tolerance strategy may be maintainable and evolvable with increasing demands of dependability.

In this respect, Garcia et al. [2] have proposed an architectural pattern for exception handling. They address the issues like specification and signaling of exceptions, specification and invocation of handlers and searching of handlers. These architectural and design patterns have been influenced by computational reflection and meta-object protocol.

However, most meta-programming languages suffer performance penalties due to the increase in meta-level computation at run-time. This is because most of the decisions about semantics are made at run-time by the meta-objects, and the overhead to invoke the meta-objects reduces the system performance [11].

Therefore we propose generalized aspect based patterns for monitoring, error detection, exception raising and exception handling using a static aspect weaver. These patterns would lead to integration towards a robust and dependable aspect based software fault tolerance. The following design notations have been used to express aspect-oriented design patterns (Fig. 14.1).

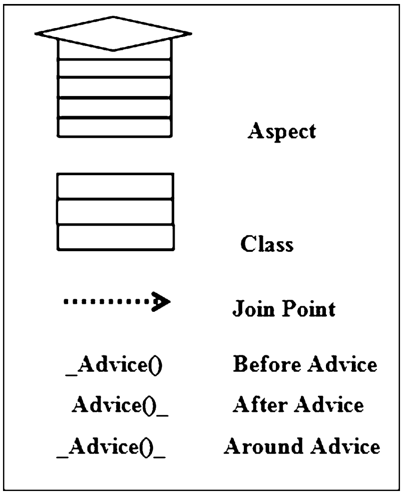
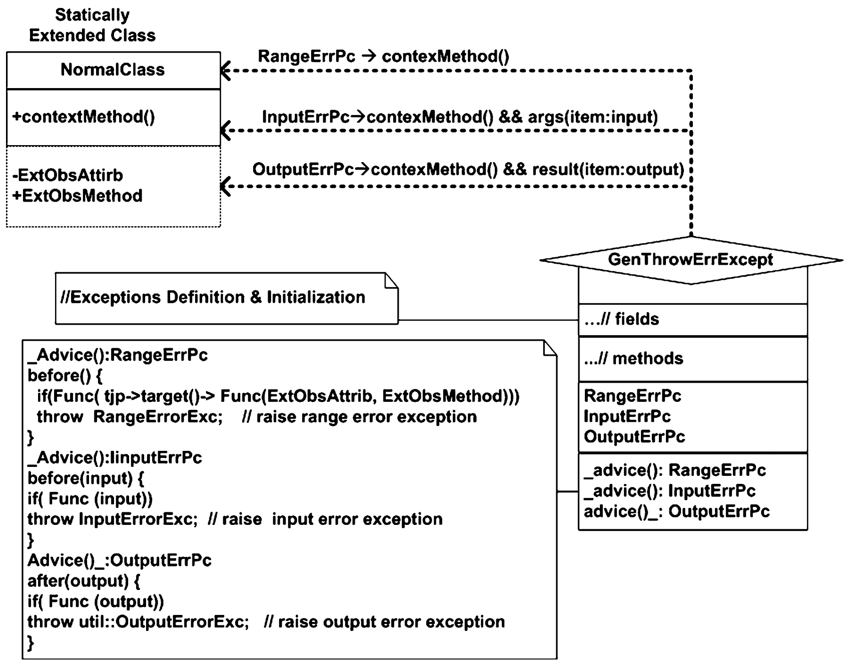


Fig. 14.1 Design notations



```
_Advice():RangeErrPc
before() {
  if(Func( tjp->target()-> Func(ExtObsAttrib, ExtObsMethod)))
  throw  RangeErrorExc;    // raise range error exception
}
_Advice():IinputErrPc
before(input) {
if( Func (input))
throw InputErrorExc; // raise  input error exception
}
Advice()_:OutputErrPc
after(output) {
if( Func (output))
throw util::OutputErrorExc;   // raise output error exception
}
```

Fig. 14.2 Error detection, exception throwing aspect pattern

14.3.1 Error Detection and Exception Throwing Aspect

Error detection and throwing exceptions has been an anchor in implementing any fault tolerance strategy. This aspect detects faults and throws range, input and output type of exceptions. The overall structure of this aspect is shown in Fig. 14.2.

The *GenThrowErrExcept* join points the *NormalClass* via three pointcut expressions for each type of fault tolerance case.

RangeErrPc: this join points the *contexMethod*() only. It initiates a before advice to check the range type errors before executing the *contextMethod*(). Incase the assertions don't remain valid or acceptable behavior constraints are not met, *RaneErrExc* exception is raised.

InputErrPc: this join points the *contextMethod*() further scoped down with input arguments of the *contextMethod*(). It initiates a before advice to check the valid input before the execution of the context method. Incase the input is not valid it raises *InputErrExc*.

OutputErrPc: this join points the *contextMethod*() further scoped down with results as output of the *contextMethod*(). It initiates an after advice to check the valid output after the execution of the context method. Incase the output is not valid it raises **OutputErrExc**.

14.3.2 ROC Plausibility Check Aspect

This aspect is responsible for checking the erroneous state of the system based on the rate of change in critical signal/data values. Once an erroneous state is detected, the respective exception is raised. Various exceptions are also defined and initialized in this aspect. The *pointcut GetSensorData* defines the location where error checking plausibility checks are weaved whenever a critical data/sensor reading function is called. The light weight ROC-based plausibility assertions are executed in the *advice* part of this aspect.

14.3.3 Catcher Handler Aspect

The *CatcherHandler* aspect as shown below is responsible for identifying and invoking the appropriate handler. This pattern addresses two run-time handling strategies.

The first strategy is designated by an *exit_main* pointcut expression. It checks the run-time *main*() function for various fatal error exceptions and finally aborts or exits the main program upon error detection. This aspect may be used to implement safe shut-down or restart mechanisms in safety critical systems to ensure safety, if a fatal error occurs or safety is breached.

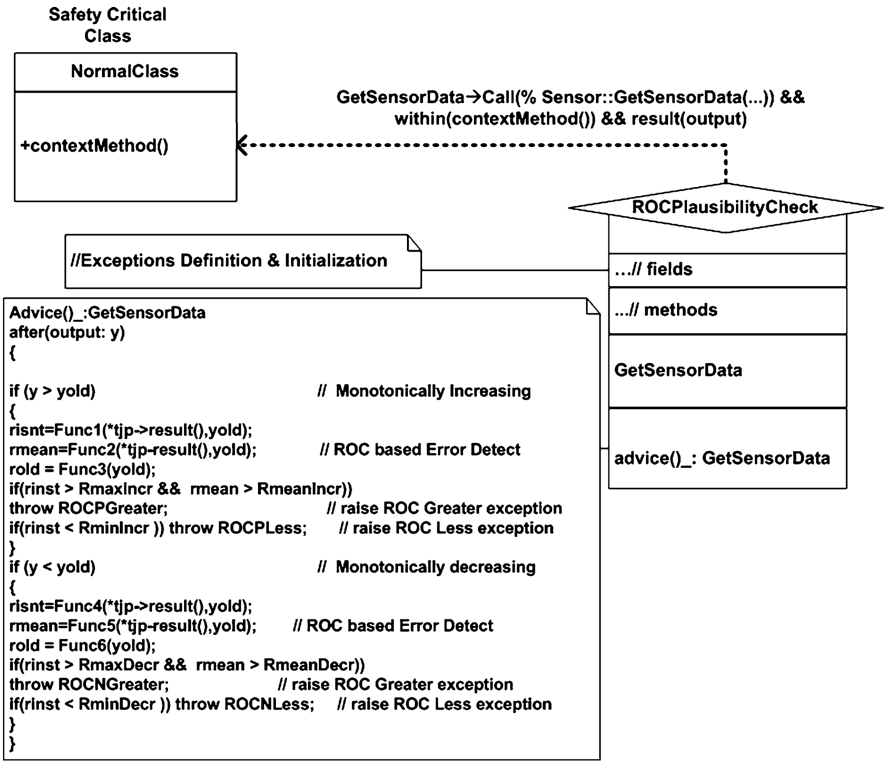
The second strategy returns from the called function as soon as the error is detected. The raised exception is caught after giving warning or doing some effective action in the catch block. This can help in preventing error propagation. Using this aspect, every call to critical functions is secured under a try/catch block to ensure effective fault tolerance against an erroneous state.

It can be seen in the diagram below that *exit_main* pointcut expression join points the main() run-time function. Whereas *caller_return* pointcut expression join points every call to the *contextMethod*(). Moreover *exit_main* and *caller_return* pointcut expressions are associated with an around advice to implement error handling. The tjp→proceed() allows the execution run-time main() and called functions in the try block.

The **advice** block of the catcher handler identifies the exception raised as a result of in-appropriate changes in the rate of signal or data. Once the exception is identified, the recovery mechanism is initiated that assign new values to signal or data variables based on previous trends or history of the variable (Figs. 14.3 and 14.4).

14.3.4 Dynamics of Cather Handler Aspect

This scenario shows an error handling aspect. It simulates two error handling strategies. In the first case, control is returned from the caller to stop the propagation of errors along with a system warning. In the second case the program exits due to a


Safety Critical Class
NormalClass
+contextMethod()
GetSensorData→Call(% Sensor::GetSensorData(...)) && within(contextMethod()) && result(output)
ROCPlausibilityCheck
...// fields
...// methods
GetSensorData
advice()_: GetSensorData
//Exceptions Definition & Initialization

```
Advice()_:GetSensorData
after(output: y)
{

if (y > yold)                              // Monotonically Increasing
{
risnt=Func1(*tjp->result(),yold);
rmean=Func2(*tjp-result(),yold);           // ROC based Error Detect
rold = Func3(yold);
if(rinst > RmaxIncr &&  rmean > RmeanIncr))
throw ROCPGreater;                         // raise ROC Greater exception
if(rinst < RminIncr )) throw ROCPLess;     // raise ROC Less exception
}
if (y < yold)                              // Monotonically decreasing
{
risnt=Func4(*tjp->result(),yold);
rmean=Func5(*tjp-result(),yold);      // ROC based Error Detect
rold = Func6(yold);
if(rinst > RmaxDecr &&  rmean > RmeanDecr))
throw ROCNGreater;                    // raise ROC Greater exception
if(rinst < RminDecr )) throw ROCNLess;    // raise ROC Less exception
}
}
```

Fig. 14.3 ROC plausibility aspect pattern structure

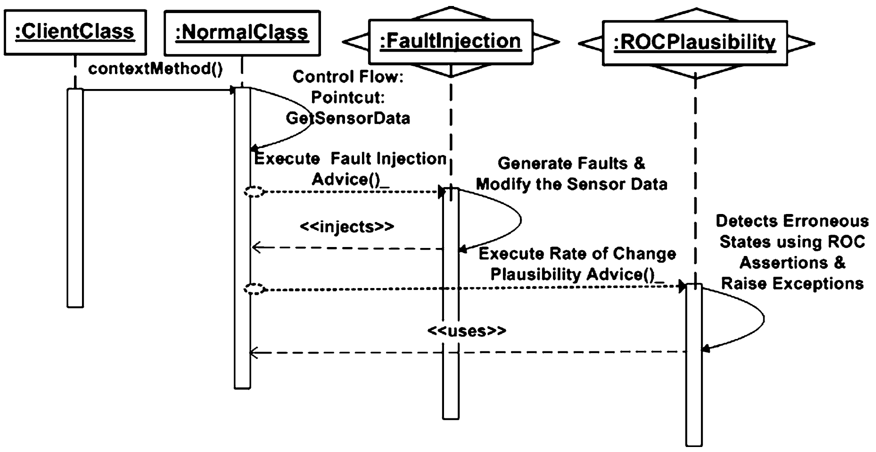

:ClientClass
:NormalClass
:FaultInjection
:ROCPlausibility
contextMethod()
Control Flow: Pointcut: GetSensorData
Execute Fault Injection Advice()_
Generate Faults & Modify the Sensor Data
<<injects>>
Execute Rate of Change Plausibility Advice()_
Detects Erroneous States using ROC Assertions & Raise Exceptions
<<uses>>

Fig. 14.4 ROC aspect pattern dynamics

fatal error. This may be used to implement shutdown or restart scenarios. Moreover the extension of a class member function with a *try* block is also explained.

1. A client object invokes the *contextMethod*() on an instance of *NormalClass*.
2. The control is transferred to *CatcherHandler* aspect that extends the *contextMethod()* by wrapping it in a *try* block and executes the normal code.
3. In case an exception is raised by previous aspect, the exception is caught by the *CatcherHandler* aspect. This is shown by the catch message. The condition shows the type of exception *e* to be handled by the handler aspect.
4. *CatcherHandler* aspect handles the exception e. the caller_return strategy warns or signals the client about the exception and returns from the caller. The client may invoke the *contextMethod2*() as appropriate. In exit_main strategy, the control is retuned to client that exits the current instances as shown by the life line end status (Fig. 14.5).

14.4 Case Study

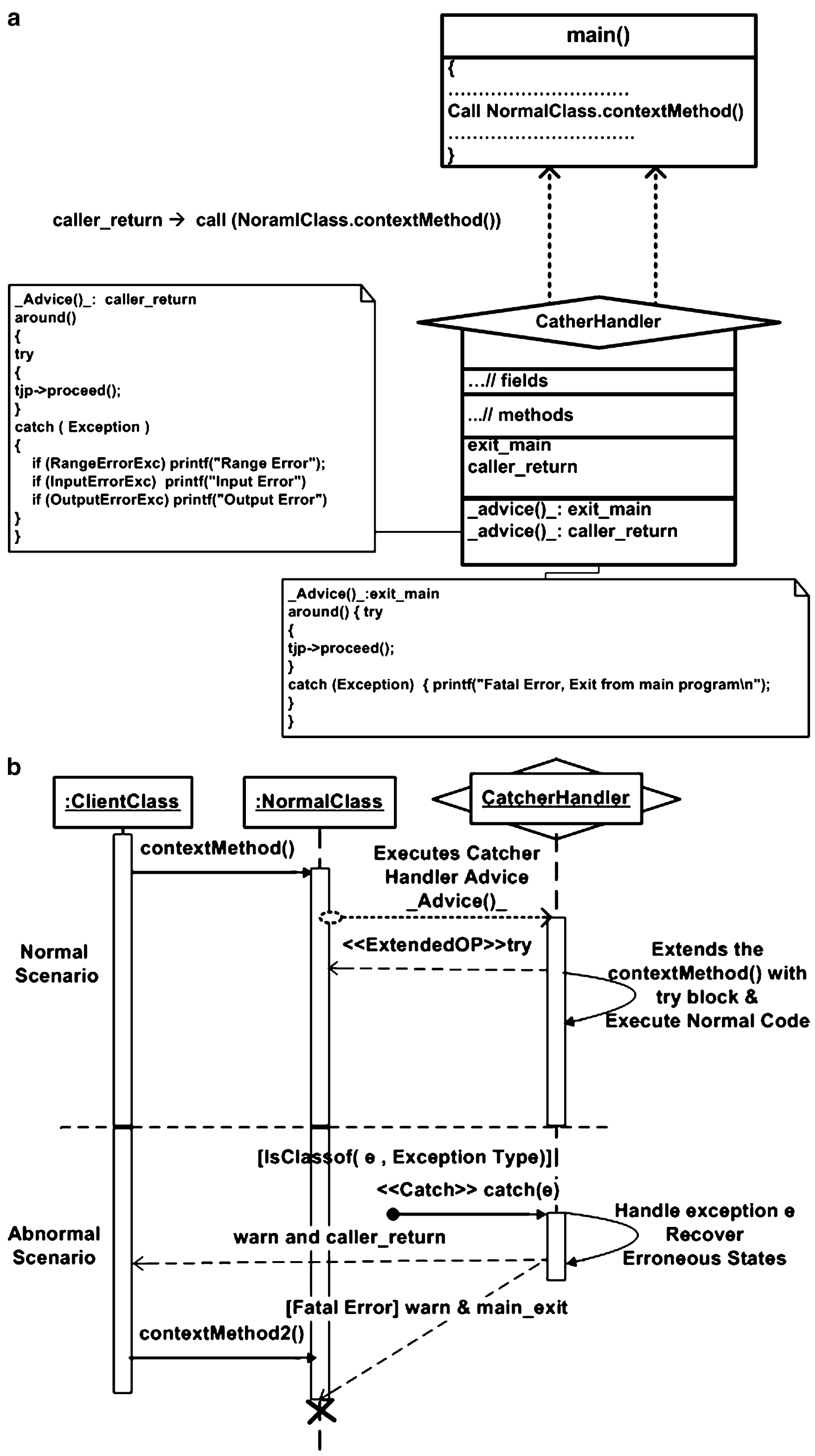
In order to validate aspect oriented fault tolerance patterns for exception handling and executable assertions as proposed earlier, a case study has been carried out using a LEGO NXT Robot (Tribot). This uses an Atmel 32-bit ARM processor running at 48 MHz. Our development environment utilizes AspectC++ 1.0pre3 as aspect weaver [4].

The Tribot has been built consisting of two front wheels driven by servo motors, a small rear wheel and an arm holding a hockey stick with the help of some standard Lego parts. Ultrasonic and light sensors are also available for navigation and guidance purposes.

An interesting task has been chosen to validate our design. In this example Tribot hits a red ball with its hockey stick avoiding the blue ball placed on the same ball stand. It makes use of the ultrasonic and light sensors to complete this task. Any deviation in full-filling the OR goals and corresponding AND sub-goals in fulfilling the overall task are considered as a mission failure.

14.5 Result and Discussion

The dependability assessment of the proposed scheme has been done via fault injection. All the faults are injected into the most critical functionality of the system, which is reading the ultrasonic sensor, light sensor, motor speed sensor and writing motor servo commands. The faults are injected by supplementary code in an aspect oriented way using AspectC++ [4]. The faults injected are permanent stuck, noise bursts and random spikes at pre-defined or random locations. These faulty data


a
main()
caller_return → call (NoramlClass.contextMethod())
CatherHandler
...// fields
...// methods
exit_main
caller_return
advice(): exit_main
advice(): caller_return
b
:ClientClass
:NormalClass
CatcherHandler
contextMethod()
Executes Catcher Handler Advice _Advice()_
<<ExtendedOP>>try
Extends the contextMethod() with try block & Execute Normal Code
Normal Scenario
[IsClassof(e , Exception Type)]
<<Catch>> catch(e)
Handle exception e Recover Erroneous States
Abnormal Scenario
warn and caller_return
[Fatal Error] warn & main_exit
contextMethod2()

```
{
..............................
Call NormalClass.contextMethod()
..............................
}
```

```
_Advice()_: caller_return
around()
{
try
{
tjp->proceed();
}
catch ( Exception )
{
   if (RangeErrorExc) printf("Range Error");
   if (InputErrorExc)  printf("Input Error")
   if (OutputErrorExc) printf("Output Error")
}
}
```

```
_Advice()_:exit_main
around() { try
{
tjp->proceed();
}
catch (Exception)  { printf("Fatal Error, Exit from main program\n");
}
}
```

Fig. 14.5 Catcher handler aspect. (**a**) Structure. (**b**) Dynamics

scenarios may simulate both permanent and transient faults originating in a faulty hardware, software or corrupted environment within or outside a computer-based system.

Although ROC-based plausibility checks are very effective in detecting faulty data values, yet a number of false positives and false negatives were generated. The proposed recovery mechanism deviates if faults persist for a longer duration. Thus maximum instantaneous ROC assertions are augmented with mean rate base check to reduce these fault positives and negatives. Mean rate is measured in a moving average window of size m. The choice of window size m is a trade off between avoiding faulty data and reducing too much estimation bias if fault bandwidth is large. For the ultra sonic sensor of Lego NXT case study, a moving window of size $(m = 4, 5)$ provides optimal results.

In order to provide better test coverage, the ultrasonic sensor data has been injected with periodic noisy bursts and random spikes. The frequency of these noisy spikes is controlled by modulo-n of a random number. It has been observed that mission critical failures are avoided using the proposed strategy with much higher confidence level (Figs. 14.6 and 14.7).

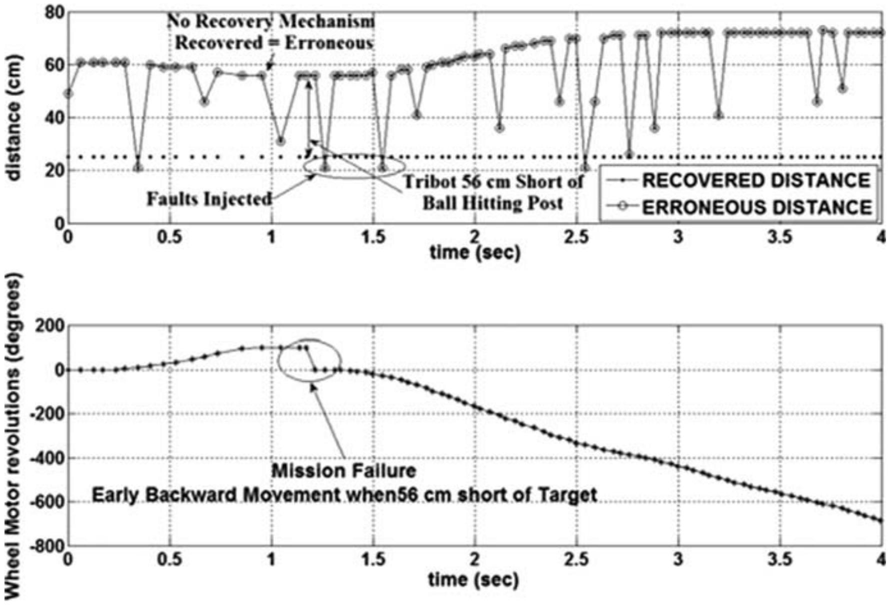


Fig. 14.6 Mission failure without recovery aspect in place

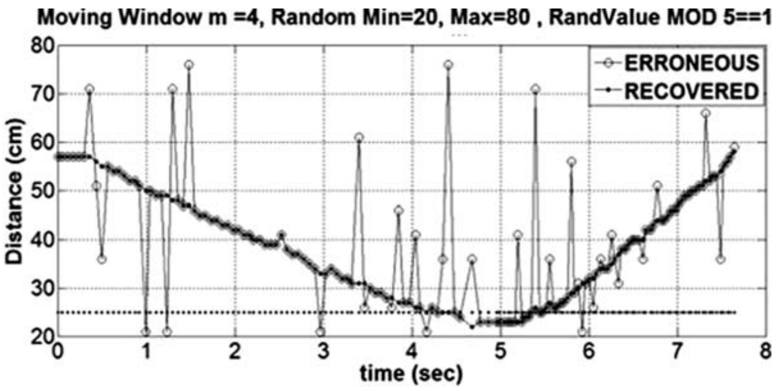


Fig. 14.7 Random spikes with error recovery

14.6 Conclusions and Future Work

The current work proposes an aspect oriented error detection and exception handling design framework. The aspect oriented design patterns under this framework bring additional benefits like the localization of error handling code in terms of definitions, initializations and implementation. Thus error handling code is not duplicated since the same error detection and handling aspect is responsible for all the calling contexts of a safety critical function. Reusability has also been improved because different error handling strategies can be plugged in separately. In this way, aspect and functional code may both be ported more easily to new systems.

The current work also investigated the use of maximum instantaneous and mean rate plausibility checks to detect and recover from erroneous states. It has been observed that mission critical variables which have monotonically increasing or decreasing trends can be augmented with carefully designed maximum instantaneous and mean rate plausibility checks to detect and recover from erroneous states.

The feedback from this initial case-study has led us to apply the same strategy to more complex applications involving the university's Merlin 521 Flight simulator. The intention is now to design and implement an aspect oriented protective wrapper that will allow students to experience physical motion within the flight simulator, under the control of their own designed autopilot, with much reduced physical risk.

This further probes the need for incorporating an error masking strategy like Recovery Blocks and N-Version Programming. An aspect oriented design version of these strategies is also under consideration.

References

1. Asterio, P., et al.: Structuring exception handling for dependable component-based software systems. Proceedings of the 30th EUROMICRO Conference (EUROMICRO'04), 2004
2. Garcia, A.F., Beder, D.M., Rubira, C.M.F.: An exception handling software architecture for developing fault-tolerant software. Proceedings of the 5th IEEE HASE USA, pp. 311–32, November 2000
3. AspectJ project homepage. http://eclipse.org/aspectj/
4. AspectC++ project homepage. http://www.aspectc.org
5. Hiller, M., et al.: Executable assertions for detecting data errors in embedded control systems. Proceedings of the International Conference on Dependable Systems & Networks, 2000
6. Hiller, M.: Error recovery using forced validity assisted by executable assertions for error detection: an experimental evaluation. 25th EUROMICRO, Milan, Italy, 1999
7. Clegg, M., Marzullo, K.: Predicting physical processes in the presence of faulty sensor readings. Proceedings of 27th International Symposium on Fault Tolerant Computing, pp. 373–378, 1996
8. Pullum, L.L: Software fault tolerance techniques and implementation. Artech House Inc., Boston, MA (2001)
9. Filho, F.C., et al.: Error handling as an aspect. Workshop BPAOSD 2007, Vancouver, BC, Canada, 12–13 March 2007
10. Romanovsky, A.: A looming fault tolerance software crisis. ACM SIGSOFT Software Engineering Notes **32**(2), 1 (March 2007)
11. Murata, K., Nigel Horspool, R., Manning, E.G., Yokote, Y., Tokoro, M.: Unification of compile-time and run-time metaobject protocol. ECOOP Workshop in Advances in Meta Object Protocols and Reflection (Meta'95), August 1995