Fault self-detection of automatic testing systems by means of Aspect Oriented programming

Pasquale Arpaia[1], Mario Luca Bernardi[2], Giuseppe Di Lucca[2],
Vitaliano Inglese[3], Giovanni Spiezia[3]

[1]Department of Engineering, University of Sannio, Corso Garibaldi 107, 82100 Benevento, Italy.
Ph : +39 0824 305804-17, Fax: +39 0824 305840, E-mail: arpaia@unisannio.it
[2]Research Centre on Software Technology (RCOST), University of Sannio, via Traiano 1, 82100 Benevento, Italy.
Ph : +39 0824 305533, Fax: +39 0824 50552, E-mail: dilucca/mlbernar@unisannio.it
[3] Department of Electrical Engineering, University of Naples, Federico II, Via Claudio, Napoli, Italy; CERN, Dept. AT (Accelerator Technology), Group MTM, CH 1211, Genève 23, Switzerland, E-mail: Giovanni.Spiezia@cern.ch.

Abstract- *An Aspect Oriented approach to implement fault detection in automatic measurement systems is proposed. Faults are handled by means of "aspects", a specific software unit to better modularize issues transversal to many modules ("crosscutting concerns"). In this way, maintainability and reusability of a measurement software are improved: indeed, once a modification of the fault detection policy occurs, only the related aspects have to be modified. As an experimental case study, this technique has been applied to the fault self-detection of a flexible framework for magnetic measurements, developed at the European Organization for Nuclear Research (CERN).*

I. Introduction

Nowadays, test automation is a *must* for assuring product quality and reliability both in industry and in research worlds. Intelligent measurement systems are used deeply in critical measurement tests involving several instruments. In particular, one of main peculiarities is the their capability of assuring a proper termination to the test process. With this aim, the automatic system has to know the state of each its own parts. Thus, a fault self-detection is a key peculiarity to provide adequate reactions to anomalous working. Devices always provide information about their status, and, if a failure or an abnormal working occurs, the fault event is notified. The controller of an automatic system must know the exceptions of each device to communicate the fault to the external world and perform adequate actions to eliminate or reduce the fault effect. Software implementation of fault detection is a well known strategy for dealing with failures caused by both hardware and software faults. Compared to hardware implementation, it has the advantage of being more flexible and cost effective. Indeed, today, it is a widely used technique, and emerging application areas for cost-effective dependable systems will further increase its importance. Thus, the software implementation of a fault detector is a crucial issue in an automatic platform, and the way it is implemented may affect the overall system quality, in particular maintainability and reusability.
Today, the development of an automatic system, with reference to its software parts, is usually made exploiting development techniques such as Object Oriented [1]-[2], component-based [3]-[4], and agent-based ones [5]. These techniques aim to organize the software system in modules each responsible of specified responsibilities/functionalities. Anyway, crosscutting concerns can negatively affect the quality of even well modularized systems implemented by these techniques [6]. Crosscutting concerns are related to issues that are transversal to many modules, such as the fault detection in an intelligent measurement system. This causes the duplication of parts of very similar code in several different modules, by negatively affecting the maintainability and reusability.
In this paper, Aspect Oriented Programming (AOP) is proposed to implement fault self-detection to overcome such drawbacks. The cross-cutting concerns related to the fault self-detection of a large measurement software system are separated and handled better by encapsulating them into *aspects*. In this way, the system maintainability and reusability is improved. As an experimental case study, a first application of AOP in designing and implementing a fault self-detection sub-system to be used in the Flexible Framework for Magnetic Measurements (FFMM), to be develop at the European Organization for Nuclear Research (CERN), is presented. The following of the paper is structured as follows: section 2 provides some basic concepts of AOP; section 3 describes the overall architecture of the AOP fault detector; section 4 give some snapshots about the implementation of the fault detector in the FFMM project; and some conclusive remarks are provided in section 4.

II. AOP background

Aspect-Oriented Programming (AOP) [7] is an extension of the object-oriented paradigm that provides new constructs to improve the separation of concerns and support their crosscutting. AOP defines a kind of program unit, the *aspect*, for specifying concerns separately, and rules for weaving them to produce the overall system to be run. Like a class of objects, an *aspect* introduces a new user-defined type into the system's type hierarchy, with its own methods, fields and static relationships to other types.

Usually, an AOP system can be seen as composed by two parts: one consisting of traditional modularization units (e.g. classes, functions) and referred as the *base system* or *core concern*, the other one consisting of aspects, encapsulating the crosscutting concerns involved in the system, and usually referred as the *secondary concerns*.

The features AOP provides for implementing crosscutting concerns in aspects can be classified in:

• dynamic crosscutting features**:** implementation of crosscutting concerns by modifying the runtime behavior of a program;

• static crosscutting features: modification of the static and structural properties of the system.

Dynamic crosscutting is implemented by using *pointcuts* and *advices*. An advice is a code fragment that will be executed in specified points within the program execution. The points in the dynamic control flow at which the advice code is executed are called *join-points*. A *pointcut* defines the events (such as method call or execution, field get and set, exception handling and softening) triggering the execution of the associated advices; a pointcut is an expression pattern that, during execution, is matched for join points of interest. Every advice is associated to a pointcut that defines the join-point(s) at which it must be applied. Advice code can be executed either before, after or around the intercepted join-point. A *join point shadow* is the static counterpart, in the code, of a join point; equivalently, a join-point is a particular execution of a join-point shadow. Aspects and base program are composed statically by a *weaving* process.

The weaver is the component of an AOP programming language environment (such as AspectJ [8]) responsible to perform the weaving process. The weaver inserts instructions at join point shadows to execute the advice to apply at the corresponding join points. The weaver may need to add runtime checks to the code inserted at a join point shadow in order to perform parameters binding and other requested computations. These code fragments are known as *dynamic residues*. If a dynamic residue specifically tests the applicability of advice at a given join point, it is called an *advice guard*.

The static crosscutting features of AOP implement crosscutting concerns by modifying the static structure of the system. An aspect can introduce new members (i.e. fields, methods, and constructors) to a class, or interface; change or add parents for any class or interface; extend a class from the subtype of the original superclass or implement a new interface. These features are called *intertype declarations*.

Figure 1 reports an example of a very simple AOP program (an AO version of the 'Hello World' program), just to exemplify how an AOP program works. The programming language used is AspectJ. In Fig. 1, the code of the class `HelloWorld` and the aspect `GreetingsAspect` are reported. The aspect defines a pointcut and two advices. The `callTellMessage()`captures calls to all public static methods with names that start with `tell`. In the example, the pointcut captures the calls to `tell(...)` and `tellPerson(...)` methods in the `HelloWorld` class taking any arguments.

```
// HelloWorld.java
public class HelloWorld {

    public    static    void    tell(String
message) {
      System.out.println(message);
    }

    public  static  void  tellPerson(String
message, String name) {
      System.out.println(name  +  ",  "  +
message);
    }
}
```

```
// GreetingsAspect.java
public aspect GreetingsAspect {
    pointcut      callTellMessage()      :
call(public          static          void
HelloWorld.tell*(..));

    before() : callTellMessage() {
       System.out.println("Good
morning!");
    }

    after() : callSayMessage() {
       System.out.println("Bye Bye!");
    }
}
```

Figure 1. - A simple AO program.

The two advices, one before and one after, associated to the `callTellMessage()` pointcut will cause, respectively, the printing of the "`Good morning!`" and "`Bye Bye!`" text strings just before and after each message printed by the `tell()` and `tellPerson()` methods, respectively.

III. The architecture of the AOP fault self-detector

The proposed AOP-based architecture for fault self-detection in measurement systems is based on:

- a **fault detection subsystem,** designed for:
  - monitoring the 'health' state of the measurement system's component devices;
  - catching software faults such as stack overflow, live-lock, deadlock, and application-defined faults as they occur.
- a **fault notification subsystem,** responsible for
  - constantly receiving the sequence of faults occurring from all the system components;
  - storing the diagnostic history and providing means to other components or to external humans to access it and adequately react to faulty events.

In the architecture, several kinds of classes of faults relevant in automatic measurement systems are identified: **faults in virtual device**, **faults in the measurement environment**, **faults in software components**.
The analysis of several state-of-the-art measurement systems highlighted that fault detection is usually scattered all over different hierarchies, mainly with reference to devices hierarchy. This means that concrete virtual devices classes contains code for fault detection resulting in code duplication that will be difficult to comprehend and maintain.
The proposed architecture is based on the `FaultDetector` aspect hierarchy: the fault detection logic is just encapsulated in the aspects belonging to this hierarchy. In this way we avoid the duplication of the code related to the fault detection in the several modules implementing the virtual devices. Moreover, this provides a very flexible mean for dynamically and obliviously associating the system components that act as fault sources to the components that need to be aware of and to handle the fault conditions, all that by means of a fault notification aspect mapping layer.

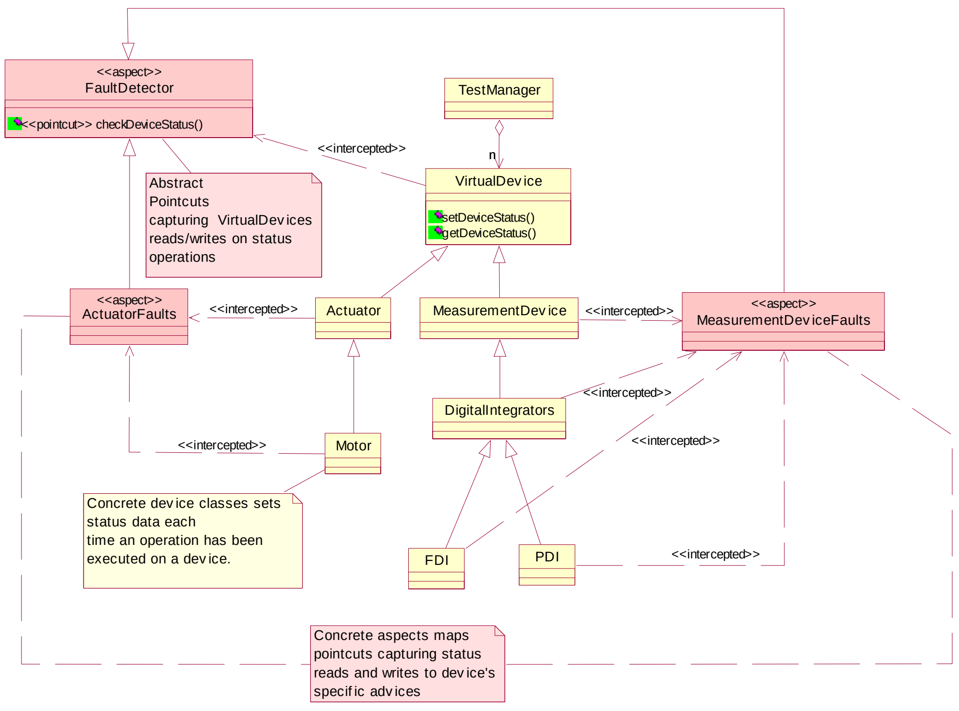

Figure 2. The hierarchy of the proposed FaultDetector.

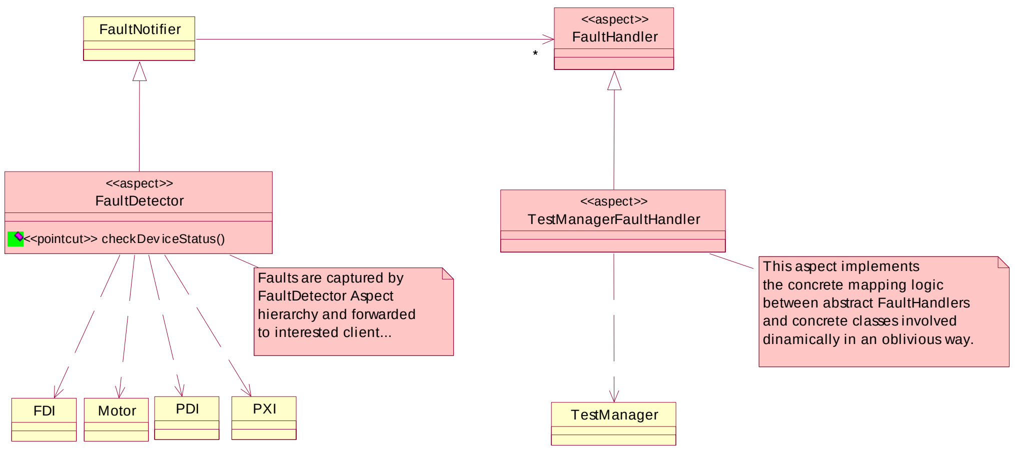
FaultNotifier
<<aspect>>
FaultHandler
*
<<aspect>>
FaultDetector
<<pointcut>> checkDeviceStatus()
<<aspect>>
TestManagerFaultHandler
Faults are captured by FaultDetector Aspect hierarchy and forwarded to interested client...
This aspect implements the concrete mapping logic between abstract FaultHandlers and concrete classes involved dinamically in an oblivious way.
FDI
Motor
PDI
PXI
TestManager

Figure 3 - Aspect mapping layer of the fault notification.

In Fig. 2, an excerpt of the proposed `FaultDetector`, hierarchy defined in the FFMM project, is depicted, by highlighting only the static relationships among VirtualDevice classes and FaultDetector aspects, for the sake of the clarity and brevity.

The `FaultDetector` is responsible for defining high-level pointcuts capturing relevant operations affecting the state of devices. In the measurement system, the VirtualDevice hierarchy models and organizes all the physical devices involved in the measurement process. The `VirtualDevice` class declare an abstract interface to handle a generic device status. Calls to such methods are captured by the `FaultDetector` aspect and associated to advice, by means of concrete sub-aspects, specifying the logic needed to detect if and where a device notified an internal fault. Each `FaultDetector` sub-aspect, associated to main devices categories, define inside itself the mapping logic towards concrete devices classes belonging to the same family.

The coarseness of the mapping among aspects and concrete devices allows a very flexible reuse of the fault detection logic for devices that are similar each other, still allowing to be encapsulated it in few modules (instead of being spread all over the device classes, as in the case of traditional coding).

In Fig. 3, the aspect mapping layer of the fault notification is shown. The services to associate dynamically the handlers to the fault sources in the measurement system are provided.

The sub-aspect of the `FaultHandler` aspect takes care of involving concrete classes (like the `TestManger` responsible of performing the test session) in order to make them “aware” of faults that happens in the system.

This solution allows fault handling logic to be reused in the super-aspects and does not force concrete classes in the system to implement fault handling code. Any component in the system can react to specific faults that occur anywhere in the system and perform the needed actions to handle the fault. Moreover, since concrete classes (`TestManager` or any other components interested in monitoring faults) are oblivious of being faults handlers, the monitoring relationships can be changed by simply acting on aspect mapping. Commonalities among different fault handling logics can be factored out in the aspects while multiple observations of different kinds of faults can be easily accomplished defining several mapping aspects for a single concrete class.

IV. Implementing the AOP fault self-detector in the FFMM at CERN

The Flexible Framework for Magnetic Measurement (FFMM) is a platform under development at CERN in cooperation with the University of Sannio. FFMM is aimed at collecting in a systematic way different measurement applications for testing superconducting magnets by specifying the following items: (i) device under test, (ii) quantity to be measured, (iii) measurement instruments, (iv) measurement circuit configuration, (v) measurement algorithm, and (vi) data analysis.

An AOP fault detector implementation is under development to handle the exception events in the rotating coil-based measurement, managed by FFMM, used to measure the magnet field parameters. The rotating coil working principle is simple: a coil turns into the magnet under test and its output signal is proportional to the flux derivative, according to Faraday’s law. The coil signal is integrated in the angular domain by means of the output pulses of an encoder mounted on the rotating coil shaft. A Fourier analysis of the flux finally yields the multipoles of the magnetic field generated by the magnet

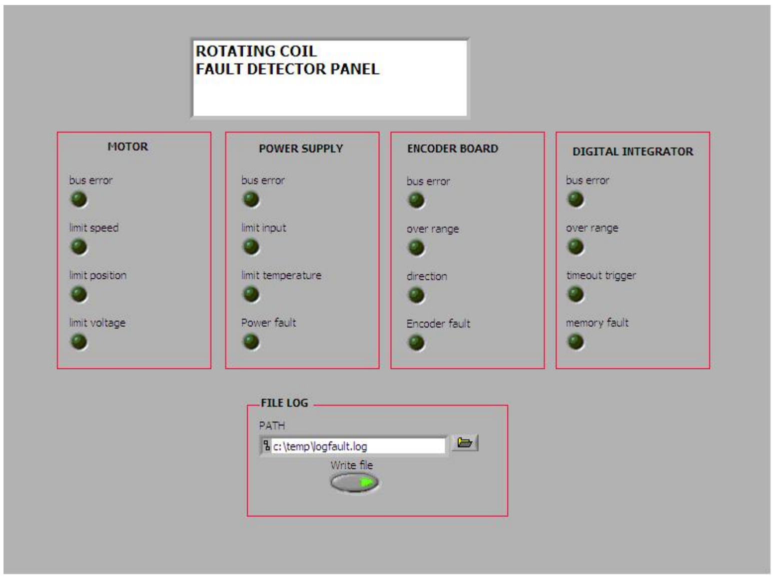

Figure 4. - Graphical interface of fault detector for rotating coil demo.

under test. A workstation is used to remotely control the motor movement, the magnet power supply, the coil signal acquisition and integration, and the current measurement. This is a typical application which involves many different devices. Each instrument has its own state and its error messages.
Pointcuts in `FaultDetector` has been defined in order to capture faulty conditions for both devices involved in the SSW measurement process and communications on system bus. Using this approach it is possible to handle uniformly different kinds of faults produced from different components. The proposed architecture allows a high level of flexibility letting to do very complex and flexible run-time binding among fault sources and fault handlers while keeping the fault detection code well modularized in fault detection hierarchy. In Fig. 4, a graphical interface of the fault detection is represented. In the FaultDetector panel, the status of each considered device can be checked.
Another main advantage of such technique is the maintainability and the reusability of the code. Indeed for each new device added to the framework, the related fault detection code is added to the fault detection hierarchy. Since all fault detection code is well modularized in few sub-aspects, commonalities among different fault detection logic is well structured and factored out.
As a consequence the FaultDetector design, with respect to ‘traditional’ OOP version, exhibits a much more centralized design reducing code duplication and greatly increasing the possibility of code reuse.

V. Conclusions

A fault self-detector based on Aspect-Oriented Programming for automatic measurement systems was presented. Such a technique allows the software quality to be improved with respect to the Object-Oriented approach, by adding specific encapsulation of crosscutting concerns. In this way, AOP allows the reusability and the maintainability of a measurement system to be enhanced. This turns out to be vital especially in software measurement frameworks, conceived specifically for developing programs for automatic measurement systems with low effort, time, and cost.
The proposed approach is experimented to develop a fault self-detector in the FFMM project under development at CERN. In particular, the improvements allowed by AOP are being verified on a measurement application based on rotating coil technique aimed at measuring parameters of superconducting magnets.

Acknowledgments

This work is supported by CERN trough the agreement K 1322 with the Department of Engineering, University of Sannio, whose supports authors gratefully acknowledge. Authors thank F. Cennamo, L. Walckiers, L. Bottura, M. L. Bernardi, L. Deniau, and G. Golluccio for their useful suggestions.

References

[1] J. Bosch, “Design of an Object-Oriented Framework for Measurement Systems” *Domain-Specific Application Frameworks*, M. Fayad, D. Schmidt, R. Johnson (eds.), John Wiley, ISBN 0-471-33280-1, 1999, pp. 177-205.

[2] S. Wang, K. G. Shin, “Constructing reconfigurable software for machine control systems”, *IEEE Trans. on Robotics and Automation,* Vol. 18, N. 4, pp. 474-486, Aug. 2002.

[3] J. M. Nogiec, J. DiMarco, S. Kotelnikov, K. Trombly-Freytag, D. Walbridge, M. Tartaglia, “A configurable component-based software system for magnetic field measurements”, *IEEE Trans. on Applied Superconductivity,* Vol. 16, N. 2, pp. 1382-1385, Jun 2006.

[4] J. E. Beck, J. M. Reagin, T. E. Sweeney, R. L. Anderson, T. D. Garner, “Applying a component-based software architecture to robotic workcell applications”, *IEEE Trans. on Robotics and Automation, Vol. 16, N. 3 pp. 207-217, Jun. 2000.*

[5] N.R. Jennings, “Agent-based computing: promises and perils”, *Proc. of the fifth International Joint Conference on Artificial Intelligence (IJCAI),* pp. 1429-1436, 1999.

[6] C. Pfister, C. Szyperski, “Why objects are not enough”, *Proc. First International Component Users Conference (CUC),* Jul. 1996.

[7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.M. Loingtier, J. Irwin, “Aspect-Oriented Programming”, in *Proc. of the 11th European Conf. on Object-Oriented Programming (ECOOP*), Vol. 1241, pp. 220-242, Springer-Verlag, 1997.

[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. “An Overview of AspectJ”, in *Proceedings of the 15th European Conf. on Object-Oriented Programming (ECOOP),* Vol. 2072 of *Lecture Notes in Computer Science,* Budapest, Hungary, 2001.