An Aspect Oriented Programming-based approach to software development for measurement system fault detection

Pasquale Arpaia$^{1,2}$, Mario Luca Bernardi$^1$, Giuseppe Di Lucca$^1$

Vitaliano Inglese$^{2,3}$, Giovanni Spiezia$^{2,3}$

$^1$Department of Engineering, University of Sannio, Corso Garibaldi 107, 82100 Benevento, Italy.
Ph: +39 0824 305804-17, Fax: +39 0824 305840, E-mail: {arpaia-dilucca-mlbernar}@unisannio.it

$^2$CERN, Dept. AT (Accelerator Technology), Group MEI, CH 1211, Genève 23, Switzerland.
Ph: +41 22 76 76635, Fax: +41 22 76 76230, E-mail: {Vitaliano.Inglese-Giovanni.Spiezia}@cern.ch

$^3$Department of Electrical Engineering, University of Naples, Federico II, Via Claudio, Napoli, Italy;

Abstract – An Aspect Oriented Programming-based approach to the development of software components for fault detection in automatic measurement systems is proposed. Faults are handled by means of specific software units, the “aspects”, in order to better modularize issues transversal to several components. Once a modification of the fault detection policy occurs, only the related aspects have to be modified. In this way, maintainability and reusability of the measurement software are improved. As a case study, this approach was applied to the design of the fault detection software inside a flexible framework for magnetic measurements, developed at the European Organization for Nuclear Research (CERN). Experimental results of software modularity and performance measurements for comparing aspect- and object-oriented solutions in rotating coils tests on superconducting magnets are reported.

Keywords - Automatic measurement systems, Fault detection, Aspect Oriented Programming, Software development.

I. INTRODUCTION

Nowadays, test automation is a must for product quality and reliability, both in industry and in research. Intelligent measurement systems are used deeply in delicate tests involving several instruments. One of their key issues is the capability of assuring a proper termination to the test
process. With this aim, a suitable fault self-detection software turns out to be an adequate reaction to anomalous working [1]. Devices provide information about their status continuously and, in case of abnormal working, a fault condition is pointed out.

Software implementation of fault detection is a well-known strategy for dealing with failures caused by both hardware and software faults [2]. Compared to hardware implementation, it has the advantage of higher flexibility and cost effectiveness. Today, it is a widely used technique, and emerging application areas for cost-effective dependable systems will further increase its importance [3]. Thus, the software implementation of a fault detector affects the overall system quality, in particular maintainability and reusability. The implementation analysis of state-of-the-art automatic measurement systems highlighted that fault detection is usually scattered all over different software components, mainly with reference to devices’ hierarchy. This means that often the concrete classes of virtual devices contain duplicated code for fault detection, thus making harder their comprehension, testing, and maintenance.

Nowadays, the development of the software for an automatic system exploits usually such as Object-oriented [4]-[5], component-based [6]-[7], and agent-based development techniques [8]. They aim at organizing the software system in modules by reducing their coupling, and maximizing their internal cohesion. Anyway, crosscutting concerns can negatively affect the quality of even well modularized systems [9]. Crosscutting concerns are related to issues, such as the fault detection in an intelligent measurement system, transversal to many modules. This causes the duplication of parts of very similar code in several different modules, by compromising maintainability and reusability.

In this paper, the Aspect Oriented Programming (AOP) is proposed for the development of software components for fault detection implementation in order to overcome such drawbacks. The cross-cutting concerns related to fault self-detection of a large measurement software project are separated and handled better by encapsulating them into specific modules called aspects. In this way, the reusability of system modules improves. As experimental case study, the development of an AOP-based fault self-detection in the Flexible Framework for Magnetic Measurements (FFMM) [10] at the European Organization for Nuclear Research (CERN) is presented. In particular, in the following Sections, (i) an AOP background, (ii) the proposed AOP-based fault detector for automatic measurement systems, (iii) the fault detector design for the
flexible framework for magnetic measurements at CERN, and (iv) a case study on rotating coils fault detection are described.

II. AOP BACKGROUND

Aspect-Oriented Programming (AOP) [11] is an extension of the object-oriented paradigm that provides new constructs for improving the separation of concerns and supporting 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: (i) one consisting of traditional modularization units (e.g. classes, functions) and referred as the base system or core concern, and (ii) 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 behaviour of a program;

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

Dynamic crosscutting is implemented by using pointcuts and advice. An advice is a code fragment executed in specified points at the program runtime. The points in the dynamic control flow where 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 matched during execution to join points of interest. Every advice is associated to a pointcut defining the joinpoint(s) at which it must be applied. Advice code can be executed either before, after, or around the intercepted joinpoint. A join point shadow is the static counterpart, in the code, of a join point; equivalently, a joinpoint is a particular execution of a joinpoint 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 [12]) responsible for the weaving process. The weaver inserts instructions at join point shadows to execute the advice to be applied 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.

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 super-class or implement a new interface. These features are called intertype declarations.

An example of a straightforward AOP program (an AO version of the ‘Hello World’ program implemented in AspectJ [13]), enlightening AO basic working, is reported in Fig. 1. In the figure, 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. 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.

III. AOP-BASED FAULT DETECTOR FOR AUTOMATIC MEASUREMENT SYSTEMS

In the following, the proposed AOP-based approach to the development of software for fault detection in automatic measurement systems is described. In particular, (i) the measurement fault analysis, and (ii) the fault detector architecture are highlighted.

A. Measurement Fault Analysis

Most common faults in an automatic measurement system can be classified according to the sources and the synchronization of the related handling operations.

According to the sources, faults can be classified as arising from:

P. Arpaia et al., “An Aspect Oriented Programming-based approach to software development for measurement system fault detection”
- **hardware devices**, when devices are in a faulty internal state due to hardware anomaly or to an external condition. Device internal fault detection can scale from very basic internal information to very complex routines forcing the device in different states. Correspondingly, concrete aspects of the fault detection subsystem must be capable of intercepting relevant changes in the device status, decoding them, and broadcasting high-level faults description to the interested components.

- **the measurement environment**, when the measurement environment is compromised by external or internal alterations.

- **software components**, when software components are in any non consistent state, owing to an incorrect use violating pre-conditions and/or post-conditions, or to the presence of unresolved or undiscovered bugs.

According to the *synchronization of the related handling operations*, faults can be classified as:

- **synchronous**, when an anomalous operation is attempted. In this case, the following policies can be applied, according to the criticality level and the kind of the fault:
  - **k-times retry**: some operations are retried until the device goes back in a consistent state, without any performance constraint on the operation. As an example, an initialization reset tried several times during a slow start up of a multimeter.
  - **multicast warning and continue**: for operations requested in wrong conditions, requests can be ignored by issuing only a notification warning. As an example, a digital scope is triggered when previous data digitization is not ended, or when a stop or an abort is issued on a already stopped instrument.
  - **multicast fault and deny operation**: for operations to not be executed when specified faults occur. In this case, the operation is denied and the fault information is sent to the pertinent components in order to be properly handled.
  - **multicast an immediate shutdown request and deny operation**: for the most critical situation when a fault on an critical operation in a risky device should be blocked at the lowest level. Moreover, since the system as a whole is to be shutdown gracefully as fast as possible, a high priority request of system shutdown is sent to the fault handler component. These faults are handled suitably by wrapping
operations through concrete pointcuts, bounded to around advices defined by abstract aspects of the fault detector component,

- **asynchronous**: when hardware or environment anomalies, in a whatever moment not synchronized with the measurement operations generate faults forcing devices in faulty states usually detected by suitable monitoring. The detection is based on field access pointcut expressions bound to the decoding logic used to detect changes in the status of devices.

**B. Fault Detector Architecture**

The proposed architecture 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 soon as they occur;

- a **fault notification subsystem**, responsible for
  - receiving the sequence of occurring faults from all the system components constantly;
  - storing the diagnostic history and providing access to other components or to external humans in order to react to faulty events adequately.

These two subsystems exploit three key components: (i) a *FaultDetector* aspect hierarchy, allowing the code related to the fault detection logic to be removed from the modules implementing the virtual devices; (ii) *fault decoder tables*, needed by concrete aspects for decoding status representation specific of concrete *VirtualDevices*; (iii) *FaultListeners* in order to dynamically and to bind (obliviously) components responsible for the fault management to the ones acting as fault sources.

**IV. FAULT DETECTOR DESIGN FOR THE FLEXIBLE FRAMEWORK FOR MAGNETIC MEASUREMENTS AT CERN**

The design of proposed AOP-based architecture for developing fault detection software components in automatic measurement systems is presented in the context of the Flexible
Framework for Magnetic Measurement (FFMM) [10], under development at CERN in cooperation with the University of Sannio. FFMM is based on Object Oriented Programming (OOP) and Aspect-Oriented Programming (AOP), and aims at supporting the user in developing software for the test of superconducting magnets for particle accelerators, maximizing quality in terms of flexibility, reusability, maintainability, and portability, without neglecting efficiency, vital in test applications. FFMM gathers from the user the requirements about a measurement task to be accomplished, and helps produce software capable of carrying out the requested task.

In Fig. 2, the proposed FaultDetector hierarchy is reported, by illustrating the static relationships among Virtual_Device classes, the FaultDetector aspect with its sub-aspects, and some concrete virtual devices (DigitalIntegrator and EncoderBoard). In the figure, the role played by the classes FaultDecoder and FaultTable is highlighted.

For the sake of the clarity, an example related to two devices, typical of magnetic measurement applications, a digital integrator (namely the Fast Digital Integrator, FDI [14]), and an encoder board, with the corresponding classes FastDI and EncoderBoard, respectively, is reported in the same figure. Encoded fault information is extracted from the FastDI device by context interception and is decoded by means of a concrete class DigitalIntegrator_FaultDecoder. The aspect DigitalIntegrator_FaultDetector is hence responsible for enforcing fault management policies according to the fault kind. It defines the appropriate listener and, when a device is registered in the FaultDetector, an instance of the listener is subscribed to the concrete instance of the device. The FaultDetector is responsible for defining pointcuts capturing creation and destruction of devices. The Virtual_Device hierarchy models and organizes all the physical devices involved in the measurement process. Each device has an internal status; modifications to such a status are captured by means of concrete FaultDetector sub-aspects. The sub-aspects execute the logic needed to decode the modification, as well as an appropriate method (i.e. fireFault, fireBadParameter, fireError,...) to broadcast fault information to the concrete FaultListener registered during the device creation and to all the other interested components.

Each FaultDetector sub-aspect is associated to the main devices categories and defines the mapping logic towards concrete device classes belonging to the same family. Indeed, the coarseness of the mapping between aspects and concrete devices allows the fault detection logic to
be reused for similar devices very flexibly, as well as to be encapsulated in a few of modules (instead to be spread all over the device classes).

In Fig. 3, the different levels of fault interceptions, according to the fault types, are depicted. The bottom level takes care about very specific issues and features of concrete devices to be encapsulated in dedicated sub-aspects. At the middle level, concrete aspects perform continuous monitoring of devices’ status, by means of appropriate pointcut expressions and decoders. The top level includes abstract aspects, implementing the fault detection logic, reusable in concrete sub-aspects.

The fault notification strategy has been implemented by means of a publish-subscribe architecture, such as shown in Fig. 4, in the case of the device EncoderBoard. In particular, the cooperation among FaultDetector and its sub-aspects (in order to associate handlers to fault sources in the measurement system dynamically) is highlighted. The sub-aspects of the aspect FaultHandler have the responsibility to make aware the concrete classes of the faults occurring in the system (e.g. the TestManger is responsible for supervising the test session).

The aspect EncoderBoard_FaultHandler defines a concrete implementation for the abstract slice class specific for the EncoderBoard devices. Such an implementation is responsible for registering/deregistering the EncoderBoardFaultListener when an EncoderBoard device is created/destructed. Moreover, the sub-aspect also defines the pointcut expressions to intercept and decode faults calling the appropriate fault broadcasting methods.

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 occurring anywhere in the system by carrying out the related handling actions.

The concrete classes (TestManager or any other components interested in monitoring faults) are oblivious of being faults’ handlers, thus the monitoring relationships can be changed by acting simply 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 by defining several listeners for a single concrete class.

In the following, two typical scenarios are discussed in order to highlight the system behaviour at run-time. In Figs. 5 and 6, a class and a communication diagram, showing the interception of a
device creation and the messages exchanged by key involved components, respectively, are depicted.

In Fig. 7, a sequence diagram, modelling (i) the typical behaviour after listener registration, when faults can happen during normal device operations, and (ii) another typical scenario, when a device accesses internal state by means of read/write operations, is reported.

While the proposed architecture is suitable for removing the fault detection code from the devices completely, in a first pilot implementation, the design team decided for not affecting the event handling protocol of the framework. Therefore, the code of fault and error broadcasting routines still lies in the component implementing the devices. This approach was aimed at carrying out a concern-driven adoption of AOP. From this point of view, the event handling is itself a concern to be migrated to the AOP paradigm in a future work.

V. A CASE STUDY ON ROTATING COILS FAULT DETECTION

An experimental case study of AOP fault detection, aimed at handling the exception events in a measurement application based on the method of rotating coils [15] for testing superconducting magnets, was conceived at CERN. Such an application was tailored on the FFMM, by specifying: device under test, quantity to be measured, measurement instruments, measurement circuit configuration, measurement algorithm, and data analysis. The case study was aimed at verifying experimentally if the software quality was improved actually with respect to the previous existing OOP version, and if the new AOP architecture had a negative impact on run-time performance of the overall system (due to aspect runtime interception overhead).

In the following, (i) the rotating coils measurement, (ii) the analysis and design of fault detection software, (iii) the modularity comparison, and (iv) the performance verification, are illustrated.

A. The Rotating Coils Measurement

The measurement method is based on the “rotating coil” (Fig. 8 [15]): A set of coil-based transducers are placed in the magnet bores, supported by a shaft turning coaxially inside the magnet. The coil signal is integrated according to the Faraday’s law in the angular domain, by exploiting the pulses of an encoder mounted on the shaft, in order to get the induction field. Several coil segments are placed on the shaft by covering the length of the magnet. Each segment,
in turn, is made up by three overlapped coils: the external one measures the mean field (absolute signal), while the series connection of the external and the central coils in opposition of phase allows the main field to be deleted in order to measure the field harmonics only (compensated signal). The field quality in accelerator magnets is expressed in terms of the magnitude of undesirable harmonics.

The coil shaft inside the magnet is turned by the Rotating Unit (RU) whose motor is driven by a controller (Maxon Epos 24). The magnet under test is supplied by power converters with digital control, with very different capacity depending on the test conditions: tests are carried out at cold (up to 1.9 K) and warm (room temperature) conditions by using a 14 kA 15 V and a 20 A, 135 V power converter, respectively. The current is read by a digital multimeter through a high-accuracy Direct Current-Current Transformer (DCCT). The coil signals are integrated in the angular domain by digital integrators (i.e., a Fast Digital Integrator [14], FDI, implemented by the FastDI class), by exploiting the trigger pulses coming out from a conditioning board (developed at CERN, implemented by the EncoderBoard class), suitably processing the output of the encoder mounted on the RU. The FDI boards, the encoder conditioning board, and the motor controller are remotely controlled by a PC running the test program output by FFMM, produced according to a suitable script [10].

B. Analysis and Design of Fault Detection Software

The rotating coil testing technique is a typical application involving several different devices, each one with its own state and error messages. During the operation of the measurement station of Fig. 8, one or more faults can affect the devices at different levels. At the lowest level, a fault can influence one of the communication buses (e.g. PXI, RS-232, IEEE-488). At this level, possible faults on the buses are: (i) communication timeout; (ii) device not found on the bus; (iii) error on an open, read, write, or close command. All these kinds of faults require the data acquired up to the fault occurrence to be saved, some diagnostic information about the state of the measurement station to be logged, and the devices to be reset in order to return back them to a consistent state.

At a higher level, some faults can involve the devices controlled trough the communication buses, namely the FDI, the encoder board, and the motor controllers. In particular, the FDI can be affected by the following faults: (i) a timeout can occur during the time interval between the transmission and the execution of a command, or when the measurement starts and the integrator...
waits for the trigger pulses from the encoder; (ii) an inconsistency of the internal status of the instrument; (iii) a wrong parameter value is set. The two last fault conditions can occur also for the encoder board and the motor controllers analogously. For the motor controller, another fault can arise from the handshake procedure on the communication bus (RS-232).

Within the proposed fault detection architecture, faults of the type (i) and (ii) are detected by means of specific pointcut expressions associated to around advices capturing the access to internal devices’ status and eventually sending fault events to interested components. Conversely, the faults of type (iii) are detected by means of pointcut expressions, associated to device operations’ signatures in order to enforce the constraint related to incoming parameter settings and eventually configured to enforce a retry policy.

In all these cases, the fault turns out to be fatal and requires the component to be reset after saving the data and logging all the additional information, as well as saving the bad parameter setting in order to detect this situation and ask for new parameter values.

In particular, the aspect \textit{FaultDetector} realizes the infrastructure needed to add/remove fault monitoring services on device creation/destruction. Its sub-aspects provide advice logic for different kind of policies of fault handling, such as discussed in Section IV. Furthermore, such sub-aspects define pointcut expressions needed to capture interesting context in which status of device must be checked, perform status decoding, and eventually send a fault event by means of the logic provided by the classes \textit{FaultListener}.

Figs. 9 and 10 show an excerpt of the code of the \textit{FaultDetector} advice (and its sub-aspect related to \textit{FastDI} digital integrator device) and of the \textit{DigitalIntegrator_FaultDetector}.

\textbf{C. Modularity Comparison}

The software quality of the proposed AOP version of the fault detector was evaluated in comparison with the corresponding OOP version previously existing at CERN inside FFMM. With this aim, the software quality attribute of modularity was assessed for both the AOP and OOP versions by evaluating (i) the percentage of lines of source code related to fault detection logic present in each module with respect to the total lines of code (LOC) of the same module, and (ii) the Degree of Scattering (DOS) and Degree of Focus (DOF) metrics [16], for each module and fault detection concern. The analysis was focused on the most relevant fault sources of the FFMM, i.e. the modules implementing devices.

\textit{P. Arpaia et al., “An Aspect Oriented Programming-based approach to software development for measurement system fault detection”}
In Tab. 1, the analysis results along with the ratio of code duplicated in the different software modules (cloned code ratio) are reported. In the OOP version of the fault detector, a high level of cloned code exists, because in each device operation often the same tests against the internal status are requested. Conversely, in the AOP version, this ratio is drastically reduced.

An ideal implementation of the Fault Detection concern would have a null DOS and a DOF equal to 1, for each device module (i.e. each device is focused only on the base concern and does not contribute at all to the Fault Detector concern). Table 1 shows that the OOP version has a very-low value of DOF, for each module (i.e. all modules contribute to the fault detection concern), and a DOS for the Fault Detection concern near to the maximum (uniformly scattered). This means that the fault detection concern in the OOP version has very bad values of modularity. Thus, any not trivial maintenance (or evolution) is very difficult, because each modification could affect and require changes in many different software modules (i.e. mainly all device modules).

Instead, in Table 1, DOS values of the AOP version are near to the minimum: the fault detection concern is well modularized in one module (the FaultDetector aspect), and each device module is marginally involved in the concern (such as said before, this is due to the fault and error broadcasting methods not yet removed from the devices).

This results is better highlighted in Figs. 11 and 12. In particular, in Fig. 11 the percentage LOC (%LOC) of the Fault Detection concern all over the device modules are compared for both AOP and OOP versions. In the figure, the ratio of the cloned LOCs in the OOP implementation, completely removed in the AOP version, is reported. Of course, the cloned code makes worst the maintainability and increases the probability of introducing bugs in the code.

In Fig. 12, the level of DOS (a) and DOF (b) for each device module with respect to Base System and Fault Detection concerns is reported. The results show a radically increased modularity for the AOP version, because each device module is much more focused on the base concern with respect to the OOP version. Moreover, the fault detection concern is highly scattered in the OOP version (high values of DOS), while it is very focused in the AOP implementation (very low values for DOS).

D. Performance Verification

The case study was aimed also at verifying experimentally that the AOP architecture would not have a negative impact on run-time performance of the overall system (due to aspect runtime
interception overhead). With this aim, the AOP system was instrumented in order to gather execution times of the aspect overheads.

In both the versions, fault detection times related to fault decoding and handling, are present. They were filtered out from the analysis. Therefore, main attention was paid to evaluate the overheads added by AOP interception mechanism to the fault detection time in order to assess the effectiveness of the AOP architecture, i.e. that the AOP response times are not worst than the OOP version.

The above described analysis was carried out by running the two versions of the software in the same conditions. The runs were performed by causing some faults in the measurement station previously described. Those faults were induced intentionally in different ways, for example by providing the devices with wrong parameter values, by interrupting the communication between the PC and the devices (device not found, or communication timeout if the communication with device had already been established), by starting the FDI acquisition procedure without feeding the instrument with the required trigger signal (measurement timeout) and adding a delay in the execution of some commands (command timeout).

The worst average times in several different categories of fault detection pointcut expressions (i.e. device creation/destruction, interception of device operations) were selected, and the time spent in the aspect runtime to jump to fault detection routines were collected. These are reported in the last column of Table 2\(^1\) (in percentage of the total time spent in the aspect).

Times needed to handle creation/destruction of devices (pointcut expressions from rows 1 to 8) are greater than those required to handle faults during measurement tasks (pointcut expressions from rows 9 to 15). In the former cases, the fault detector infrastructure must be set up for devices being created. This requires more time than the other kind of pointcuts expressions, that have only to capture the context of an operation, issuing a fault event if necessary. These times are comparable to those of the OOP version, where listeners are explicitly registered with the created devices to handle faults. In these cases, the aspect overhead is particularly reduced with respect to the entire fault detection tasks.

\(^1\) The times refer to a Pentium IV 1.3GHz machine, with 512Mb of RAM running the instrumented AOP version.
Pointcut expressions ranging from row 9 to row 15 are related to fault detection during normal device operations. Their goal is to capture all the context in which the device state changes to check its validity. In the developed AOP implementation, the worst overheads due to aspect interception mechanism (see the last column in Tab. 2) are always less than 1.5% of the fault detection times (the worst case is for the interception of calls to EncoderBoard device operations with complex arguments matching expression to check preconditions; related to pointcut expression at row 9).

Therefore, the suitability of such performance overhead in the concrete measurement scenario was assessed, where all the timing constraints were satisfied flatly.

VI. CONCLUSIONS

A fault self-detector based on Aspect-Oriented programming was presented. Such a software design integrates the Object-Oriented approach, by adding specific encapsulation of crosscutting concerns. The proposed approach was used for the development of a fault detector in a framework for magnetic measurement, under development at CERN. The advantages of using AOP in the development of a fault detector were verified in the case of a measurement application based on rotating coils for testing a superconducting magnet.

This approach allows uniformly different kinds of faults produced from different components to be handled. The proposed architecture allows a high level of flexibility by performing very complex and bendable run-time binding among sources and handlers of the faults, while keeping the detection code well modularized in its hierarchy.

Another main advantage of such a technique is the maintainability and the reusability of the code: 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.

Finally, the proposed AOP architecture is not targeted at a specific system component, and the
same fault detector architecture can be reused to detect different kinds of faults in different components.
REFERENCES


LIST OF FIGURE CAPTIONS

Figure 1. A simple AO program.
Figure 2. An excerpt of the hierarchy of the proposed fault detector.
Figure 3. Levels of faults interception.
Figure 4. Fault notification publish-subscribe architecture.
Figure 5. Device Creation/Destruction interception.
Figure 6. A collaboration scenario started by createDevice interception on EncoderBoard.
Figure 7. A sequence diagram showing the detection of a wrong parameter configuration of an EncoderBoard instance.
Figure 8. Layout of the rotating coils measurement setup [15].
Figure 9. The abstract FaultDetector aspect.
Figure 10. Excerpt of DigitalIntegrator_FaultDetector.
Figure 11. LOC% of fault detection concern in device modules for OOP and AOP versions.
Figure 12. DOS and DOF comparisons of OOP and AOP versions with respect to Fault Detection concern.

LIST OF TABLE CAPTIONS

Table 1. Fault detection code in each device module and computation of DOF and DOS metric for both OOP and AOP versions (OOP: object-oriented programming; AOP: aspect-oriented programming; LOC: lines of code; DOF: degree of focus; DOS: degree of scattering).
Table 2. Worst average times spent in aspect runtime with respect to device creation/destruction and fault detection pointcuts.
Figure 1. A simple AO program.
Figure 2. An excerpt of the hierarchy of the proposed fault detector.
Figure 3. Levels of faults interception.
Figure 4. Fault notification publish-subscribe architecture.
Figure 5. Device Creation/Destruction interception.

Figure 6. A collaboration scenario started by createDevice interception on EncoderBoard.

P. Arpaia et al., “An Aspect Oriented Programming-based approach to software development for measurement system fault detection”
Figure 7. A sequence diagram showing the detection of a wrong parameter configuration of an EncoderBoard instance.
Figure 8. Layout of the rotating coils measurement setup [15].

```cpp
aspect FaultDetector {
    public:
        vector<ffmm::core::devices::Virtual_Device*> _mdevs;

    static vector<std::string> showMonitoredDevices()
    static void showMonitoredDevices()
    pointcut devices() = "ffmm::core::devices::Virtual_Device";

    public abstract pointcut retry(Virtual_Device* dev);
    void around(Virtual_Device* dev) : retry(dev);
    advice devices() : slice class {
        virtual init initFaultDetection();
        virtual init removeFaultDetection();
        virtual bool checkDeviceStatus()
        virtual int decodeError()
        virtual void addListener();
        virtual void removeListener();
    }
    void addToMonitor(ffmm::core::devices::Virtual_Device* m);
    void removeFromMonitor(ffmm::core::devices::Virtual_Device* m);
    void printMonitoredDevices();
    void CheckStatus(ffmm::core::devices::Virtual_Device* m)
    advice device_construction() : after() {this->CheckStatus(tjp->target());}
    advice device_destruction() : before() {this->removeFromMonitor(tjp->target());}
};
```

Figure 9. The abstract FaultDetector aspect.

P. Arpaia et al., “An Aspect Oriented Programming-based approach to software development for measurement system fault detection”
class FDIListener :
    public IFdiListener<ffmm::core::devices::FastDI>
    {
    public:
        virtual void onFdiError(FdiErrorEvent<ffmm::core::devices::FastDI>* evt);
    }

aspect DigitalIntegrator_FaultDetector : public FaultDetector {
    public:
        static FDIListener* _lis;
        static FDIListener* getListener() {
            if (_lis==NULL) _lis = new FDIListener();
            return FastDI_FaultDetection::aspectOf()->_lis;
        }
    private:
        pointcut devices() = "ffmm::core::devices::FastDI";
        // Concrete slice class for Digital Integrators
        advice devices() : slice class FastDISlice {
            virtual void initFaultDetection();
            virtual void removeFaultDetection();
            virtual bool checkDeviceStatus();
            virtual int decodeError();
            virtual void addListener();
            virtual void removeListener();
        };

        pointcut startOnDigitalIntegrator(FastDI _fdi) =
            call("void FastDI.start(...)") && target(_fdi);
        pointcut stopOnDigitalIntegrator(FastDI _fdi) =
            call("void FastDI.stop(...)") && target(_fdi);

        void around(FastDI _fdi) : startOnDigitalIntegrator(_fdi) {
            if (_fdi._status.started) callNotifier(_);
            else proceed(_fdi);
            if (check_postconditions())
                // multicast fault to the listener via fire% methods
                callNotifier(new FastDIFaultDecoder(_fdi).getTable());
        }

        void around(FastDI _fdi) : stopOnDigitalIntegrator(_fdi) {
            if (_fdi._status.stopped) callNotifier(_);
            else proceed(_fdi);
            if (check_postconditions())
                // multicast fault to the listener via fire% methods
                callNotifier(new FastDIFaultDecoder(_fdi).getTable());
        }

    };

Figure 10. Excerpt of DigitalIntegrator_FaultDetector.
Figure 11. Percentage lines of code (LOC%) of fault detection concern in device modules for OOP and AOP versions.
Figure 12. DOS (a) and DOF (b) comparisons of OOP and AOP versions with respect to Fault Detection concern.
Table 1. Fault detection code in each device module and computation of percentage DOF and DOS metric for both OOP and AOP versions (OOP: object-oriented programming; AOP: aspect-oriented programming; LOC: lines of code; DOF: degree of focus; DOS: degree of scattering).

<table>
<thead>
<tr>
<th>DEVICE</th>
<th>OOP FD LOC</th>
<th>OOP CLONED LOC</th>
<th>OOP DOF</th>
<th>OOP DOS LOC</th>
<th>AOP %LOC</th>
<th>AOP DOF</th>
<th>AOP DOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>FastDI</td>
<td>15.75</td>
<td>10.93</td>
<td>0.17</td>
<td>0.81</td>
<td>0.97</td>
<td>0.13</td>
<td></td>
</tr>
<tr>
<td>Maxon Epos</td>
<td>18.04</td>
<td>9.78</td>
<td>0.21</td>
<td>0.73</td>
<td>0.97</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Encoder Board</td>
<td>21.53</td>
<td>14.27</td>
<td>0.28</td>
<td>0.62</td>
<td>0.98</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Power Supply</td>
<td>18.36</td>
<td>12.70</td>
<td>0.24</td>
<td>2.64</td>
<td>0.90</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trasducer</td>
<td>21.15</td>
<td>8.24</td>
<td>0.27</td>
<td>1.79</td>
<td>0.93</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Keithley2k</td>
<td>18.48</td>
<td>11.32</td>
<td>0.16</td>
<td>0.77</td>
<td>0.97</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2. Worst average times spent in aspect runtime with respect to device creation/destruction and fault detection pointcuts.

<table>
<thead>
<tr>
<th>Pointcut expressions</th>
<th>Total Time (ms) spent in aspect</th>
<th>Time (ms) spent in matching advices</th>
<th>Time (ms) spent in aspect runtime</th>
<th>%Time spent in aspect runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 EncoderBoard creation</td>
<td>13.029641</td>
<td>13.028840</td>
<td>0.000801</td>
<td>0.006%</td>
</tr>
<tr>
<td>2 FastDI creation</td>
<td>45.912234</td>
<td>45.893478</td>
<td>0.018756</td>
<td>0.044%</td>
</tr>
<tr>
<td>3 FDCluster (1 element) creation</td>
<td>59.063682</td>
<td>59.016519</td>
<td>0.052463</td>
<td>0.089%</td>
</tr>
<tr>
<td>4 Maxon.Epos creation</td>
<td>14.013691</td>
<td>14.011993</td>
<td>0.001898</td>
<td>0.014%</td>
</tr>
<tr>
<td>5 EncoderBoard destruction</td>
<td>10.282863</td>
<td>10.282862</td>
<td>0.00231</td>
<td>0.002%</td>
</tr>
<tr>
<td>6 FastDI destruction</td>
<td>18.511722</td>
<td>18.511302</td>
<td>0.000320</td>
<td>0.002%</td>
</tr>
<tr>
<td>7 FDCluster (1 element) destruction</td>
<td>6.034312</td>
<td>6.034096</td>
<td>0.000216</td>
<td>0.004%</td>
</tr>
<tr>
<td>8 Maxon.Epos destruction</td>
<td>11.045266</td>
<td>11.045013</td>
<td>0.000243</td>
<td>0.002%</td>
</tr>
<tr>
<td>9 within(EncoderBoard) &amp; &amp; call(...) &amp; &amp; args(...)</td>
<td>2.3803261</td>
<td>2.842784</td>
<td>0.037542</td>
<td>1.303%</td>
</tr>
<tr>
<td>10 withincode(FastDI) &amp; &amp; callpix-&gt;write</td>
<td>1.146675</td>
<td>1.144412</td>
<td>0.002263</td>
<td>0.197%</td>
</tr>
<tr>
<td>11 withincode(FastDI) &amp; &amp; callpix-&gt;read</td>
<td>1.48675</td>
<td>1.476575</td>
<td>0.010000</td>
<td>0.673%</td>
</tr>
<tr>
<td>12 withincode(Maxon.Epos) &amp; &amp; execution(set%)</td>
<td>0.982102</td>
<td>0.960102</td>
<td>0.02099</td>
<td>0.204%</td>
</tr>
<tr>
<td>13 withincode(Maxon.Epos) &amp; &amp; execution(get%)</td>
<td>0.902345</td>
<td>0.899015</td>
<td>0.003050</td>
<td>0.359%</td>
</tr>
<tr>
<td>14 withincode(Maxon.Epos) &amp; &amp; execution(start)</td>
<td>2.896736</td>
<td>2.896736</td>
<td>0.010000</td>
<td>3.35%</td>
</tr>
<tr>
<td>15 withincode(Maxon.Epos) &amp; &amp; execution(stop)</td>
<td>2.416875</td>
<td>2.406875</td>
<td>0.010000</td>
<td>0.414%</td>
</tr>
</tbody>
</table>

P. Arpaia et al., “An Aspect Oriented Programming-based approach to software development for measurement system fault detection”