A Reflective Architecture for Process Control Applications

Charlotte Pii Lunau

Department of Computer Science, Aalborg University
Fredrik Bajers vej 7E, 9220 Aalborg, Denmark
lunau@cs.auc.dk

ABSTRACT

This paper presents a reflective architecture for process control applications with composition of metaobjects. Reflection is used to separate a model of physical entities from the monitoring and controlling part of the application and to exchange metaobjects dynamically. Dynamic exchange of metaobjects is used to implement context sensitive monitoring. Being able to program a monitoring strategy in a single metaobject, and to exchange a metaobject depending on plant state, significantly ease the programming of the monitoring and controlling part. Composition of metaobjects makes the architecture extensible and avoids to program metaobjects for all possible combinations of behaviour. A diagnosis system based on the proposed architecture has been implemented.

Topics: Reflection, Composability, Adaptability

1 INTRODUCTION

This paper presents an architecture for process control applications which makes it easier to develop applications and to adapt the applications to changing environments. The architecture is based on computational reflection and composition of metaobjects. Computational reflection separates the base level of a computation from the metalevel. The metalevel is monitoring and influencing the base level. This separation is used to obtain a structured architecture where a model of the physical system is located at the base level and the monitoring and controlling part constitutes the metalevel. A model of a physical system is relatively static, it only requires changes when new hardware, in form of sensors are added or removed. The monitoring and controlling part of a process control application must detect and react to state changes and faults in the process. In complex systems, the number of state changes and the combination of faults are large. In traditional architectures, the

monitoring and controlling part is programmed as one module containing a number of large case statement. It is difficult to ensure that all cases and combinations of faults are covered and it is difficult to extend and maintain the software. By placing the monitoring and controlling part at the metalevel, this part can be divided into a number a independent metaobjects. Each metaobject is responsible for one fault or for a combination of faults. The metaobjects are dynamically exchanged at run time dependent on developments in the monitored process.

Main contributions of the paper include definition of requirements to reflection in process control applications and a definition of a reflective architecture with composition of metaobjects, suitable for process control applications.

The proposed architecture has been evaluated by implemented a diagnosis system, which handles fire and damages to the hull onboard ships. The diagnosis system is a prototype which has been demonstrated to ship owners and to classification societies.

Experience from implementing the prototype demonstrated the following main advantages of using a reflective architecture:

- The physical layer is separated from the monitoring and controlling part
- Monitoring strategies are separated into metaobjects
- Metaobjects are dynamically exchanged as faults are occurring
- Adaptability is obtained by adding new metaobjects

The advantages lead to a modular design with a clear separation of issues. This separation makes it easier to implement, maintain, and adapt process control applications.

To build an application with a reflective architecture, we need a reflective programming language. Commercial available object-oriented languages, such as Smalltalk-80 [Goldberg and Robson 83], Objective-C [Cox 86],[Next 92], and Clos [Kiczales et al. 91] contain structural reflection, where changes are applied at compile time. Our analysis of the requirements to reflection for process control applications, presented in Section 4, require computational reflection, where changes take place at run time. We therefore had to extend the object-oriented language Objective-C with computational reflection and composition of metaobjects. Computation reflection is obtained by intercepting all messages to objects, which has a metaobject attached. The message is redirected to the metaobject either before or after the method is invoked in the base level object. Extension of Objective-C with computational reflection is described in Appendix A.

1.1 Related Work

A number of papers discuss reflection seen from a programming language point of view. Computational reflection is discussed in [Smith 82], [Maes 87] and [Ferber 89]. Structural reflection is discussed in [Kiczales et al. 91] and [Cointe 87]. However, only few papers address the advantages of applications based on a reflective architecture. [Rao 91] reports on the design and implementation of a window system, called Silica. Silica's design separates parts of the basic functionality of window systems into independent objects called contracts. The contracts are reified through a protocol and constitute the reflective level. The contracts can be selected for each window independently and they can be specialized by a user. Silica's design is radically different from ours. Silica's reflective level is obtained by reifying implementation details through an interface, without the use of neither metaclasses nor computational reflection.

[Agha and Sturman 94] use computational reflection to dynamically install dependable protocols in distributed system using an Actor based language. We do share the goals of dependability, modularity, and composition with [Agha and Sturman 94], but our application areas and programming language environments differ significantly. [Agha and Sturman 94] focus on dependable services in a distributed system, while our focus is on context sensitive monitoring driven by requirements from an industrial application.

[Aksit et. al 93] proposes composition filters to extend a language and to allow independent extensions to be composed. Composition filters are implemented using message interception and come in two variants input filters and output filters. All input filters are invoked before the method on the object. Output filters deal with messages sent from an object. Input filters and our composition of metaobjects are similar in that they both compose behaviour. However, input filters are allowed to reject the method invocation on the object, while our metaobjects always invoke the method. Input filters are always invoked before the method, while our metaobjects can be invoked before or after method invocation. Our metaobjects does not support the equivalence of output filters. The main focus of composition filters is language extensions, while our focus is to build an architecture for a reflective application.

1.2 Outline of the Paper

The paper starts the background for advocating new architectures for software development. Next, the application area is described and a set of requirements to reflection in process control applications is identified. Based on the requirements, a reflective architecture with dynamic exchange of metaobjects and composition of metaobjects is defined. The next section presents a diagnosis system which handles fires and damages onboard a ship. The diagnosis system is implemented using the reflective architecture. Finally, the conclusions are presented. The extensions to

Objective-C are described in Appendix A. Appendix B contains the metaobject class hierarchy used by the diagnosis system.

2 BACKGROUND

The production of adaptable software systems has become an important goal of software engineering. In general, adaptability ensures that systems can evolve over time by adding new functionality and changing existing services.

At present a number of mechanisms for obtaining adaptability coexist. One such mechanism is design patterns [Gamma et. al. 94] where each pattern defines aspects that vary over time. Examples of aspects are:

Pattern	Aspect that vary
Adapter	interface to an object
Bridge	implementation of an object
Composite	structure and composition of an object
Observer	objects that depend on another object
Strategy	an algorithm

Table 1 Design patterns and their variations

Besides design reuse, a goal of design patterns is to obtain a more flexible and dynamic design with greater potential for reuse. Many design patterns emphasises object composition, where an object can be substituted by another object, at run-time, if they have identical interfaces.

Another mechanism for obtaining adaptability is reflection. In object-oriented programming two kinds of reflection exist with different purposes: Structural and computational.

Structural reflection is obtained by metaclasses where a metaclass defines the behaviour and structure of it class. Changes to a metaclass influences its class and all objects of the class. Using structural reflection it is not possible to make adaptations to individual objects; adaptations apply to all objects of a class. Using structural reflection adaptations take place at compile time because a class needs to be defined and compiled before objects can be created.

Computational reflection has a close relation between an object and its metaobject. Each individual object can have a metaobject attached and objects of the same class can be attached to metaobjects of different classes. Furthermore it is possible to dynamically remove or attach metaobjects. A metaobject monitors and affects its base level object, dependent on the base level object's state. A base level object and its

metaobject are co-ordinated through a causal connection. A causal connection implies that any changes made in a metaobject are immediately reflected in the base level object's state and behaviour.

In our implementation of computational reflection all messages sent to an object, with an attached metaobject, are intercepted and redirected to the metaobject. The same behaviour can be obtained using an event/notification mechanism, such as the Observer design pattern [Gamma et.al.95] or the Observable/Observer classes in Java [Chan and Lee 96]. However, computational reflection has a number of advantages over event/notification mechanisms.

In event/notification mechanisms all objects reside at the same level and therefore no division of an application into separate levels exist. Furthermore, all observable or subject objects must share a common superclass, which maintains a list of current observers and implements a notification method. Each observable/subject object must explicitly invoke the notification method, whenever a change has taken place. This implies that monitoring cannot be extended to include new objects without changing the objects, so they inherit from the common superclass and explicit invoke the notification method.

Using computational reflection, there is a clear separation between monitored and monitoring objects, because they reside at different levels. Any object can, at any time, have a metaobject attached or detached and notification is implicit for all objects with an attached metaobject. Thus, computational reflection provides better support for adaptive software than the simpler event/notification mechanism.

3 PROCESS CONTROL APPLICATIONS

A process control application monitors and controls a physical system and it must be able to react to changes and faults in the physical system. Changes and faults occur randomly and in bursts. The software must be able to react to the changed situation dynamically. This implies that the process control software should be able to change its structure and behaviour while it is running.

A physical system can be anything from a fairly simple thermostat regulating the temperature to complex production lines in a factory. Monitoring is performed by collecting input from sensors and evaluating it against an expected behaviour. Sensors are connected to physical entities and can be either binary, measuring an on/off state or analogue, measuring a continuos value. For instance, a sensor connected to the outlet of a pump measures the pressure in the connected tube. Interpretation of the pressure informs about the state of the pump; states may include pump running, pump stopped, pump running at half speed. Advanced diagnosis systems use development of sensor values to predict whether a failure is about to occur. This requires that value and time are logged, statistical signal processing is used, and fault detection

techniques applied. Detection and isolation of state changes in a plant usually require fusion of data from several sensors and a mathematical model of the physics involved. Output from such detectors can be considered as virtual sensor signals which must be treated along with the physical signals.

Control actions are performed to prevent faults from occurring or to remedy the situation when a fault has occurred. Control is performed by actuators, which for instance turn a valve or start/stop a pump. Connection to sensors and actuators is obtained through a real time local area network.

Process control applications are usually designed as closed-loop systems. Closed-loop implies that sensor measurements are used to control the physical process by changing the settings of entities, such as valves and pumps via actuators. A traditional architecture for a process control application is shown in Figure 1.

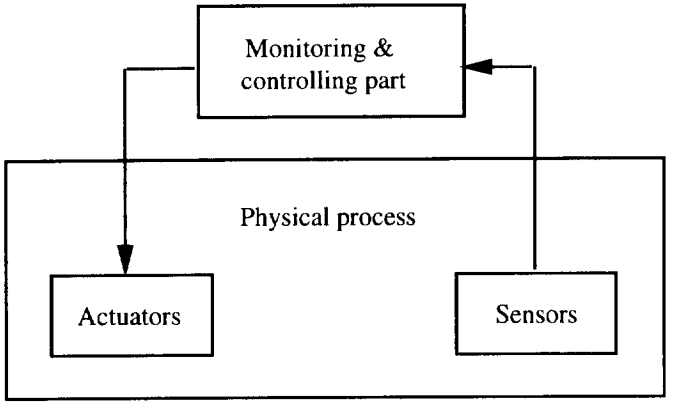

Figure 1 A traditional architecture for a process control application

In traditional architectures, the controller can be rather complex to design and implement, due to a large set of possible faults and the complexity of the underlying physical system. It is difficult to ensure that all combinations of faults are covered and such systems are difficult to extend and adopt to changes in an industrial plant.

4 Requirements to Reflection in Process Control Applications

In this section, we identify a number of requirements to reflection in process control applications. The requirements are identified based on our experience designing and implementing a number of alarm and diagnosis systems for use onboard ships and we

believe the requirements are applicable to sensor based process control applications in general.

In process control applications, the same type of sensors are measuring on a variety of different physical entities. For instance, analogue sensors measure pressures and temperatures. They are connected to units, such as water tanks, outlet from pumps, accommodation areas, and refrigerators. The monitored entities have different functions and behaviour and need individual monitoring strategies. A monitoring strategy is related to the monitored entity and not to the type of sensor used to perform the monitoring. This implies that sensor objects of the same type need different monitoring strategies, i.e. different metaobjects attached.

In case of a failure in one of the monitored entities, it must be possible to exchange the monitoring strategy for the failed entity. This implies that sensors of the same type measuring on the same type of entities need individual monitoring strategies and it must be possible to dynamically exchange the monitoring strategy, when changes occur in the physical process.

In many cases, it is insufficient to look at one sensor value at a time. Instead it is necessary to look at a set of sensor values to get an overview of the state of a process. One example is a fire onboard a ship. Here the monitoring and controlling part needs to know all compartments on fire, to propose a suitable fire fighting strategy. A fire fighting strategy typically includes decreasing the temperature in a compartment by cooling the boundaries using water. The monitoring and controlling part must determine the boundaries of all compartments on fire and propose to cool those boundaries. This is obtained by letting all compartments on fire share a single meta object, which monitors the fire compartments' sensors. This leads to a requirement that a single meta object must be able to monitor several sensor objects.

Another example, where more than one sensor value is needed, is when a sensor value is not available, either because the sensor has failed or because a sensor is not attached to a physical entity. Here the process control application needs to collect values from other sensors to calculate the state of a physical entity or a process. This leads to a requirement that monitoring strategy metaobjects know each other and must be able to communicate.

A common requirement is to log the values of sensors. Logging is a reflective activity and is naturally done in a logging meta object. But most sensors already have a monitoring meta object attached. Although the number of monitoring strategies is known it is impractical to predict all possible needed combinations. Furthermore, it is desirable to be able to extend the meta level with new functionalities without having to incorporate this functionality in all existing metaobjects. A general solution to the problem is to compose metaobjects. Fortunately, metaobjects implement independent behaviour and composition is possible and easy. Thus, it is a requirement to compose metaobjects attached to a base object.

The following list summarises the requirements, identified above:

- Sensor objects of the same type must be able to attach different metaobjects .
- Each sensor object may need an individual metaobject
- Metaobjects must be changed dynamically when changes in the process occur.
- A metaobject must be able to monitor more than one base object
- Metaobjects must be able to communicate with each other
- It must be possible to compose metaobjects attached to a base object

The next section presents a reflective architecture, which fulfils these requirements.

5 A Reflective Architecture

Based on the requirements from the previous section, a reflective architecture for process control applications is presented. The architecture is implemented in Objective-C extended with computational reflection and composition of metaobjects. A reflective architecture separates an application into a base-level and a reflective-level. First, we discuss the objects at the two levels.

The base-level objects are all the physical entities, such as pumps, valves, tubes, tanks etc., depending on the actual application. Sensors and actuators are also part of the base-level. Objects at the base-level need to be connected using relations, because we need to be able to deduce which physical entities a sensor object actually is measuring. For instance, measuring if a pump is running is usually done by attaching a flow sensor to the outlet tube of the pump. Changes in the physical entities are reflected in the sensor objects only. The actuators are used to influence the process in critical situations by operating on the physical entities, such as opening a valve.

The reflective-level consists of metaobjects performing a number of different functions. One set of metaobjects is responsible for monitoring the physical process. Monitoring is performed by attaching metaobjects to sensor objects. With the possibility of dynamically exchanging metaobjects, the monitoring part can be divided into a number of independent metaobjects, each performing a specific and limited part of the overall monitoring. Typically, the monitoring will be divided into three major areas: Monitoring during normal operation, monitoring when there is a suspicion that a fault is about to happen, and monitoring when a fault has occurred.

During normal operation simple checks of the values of the sensor objects are performed. As soon as there is a suspicion that a fault is about to happen, because

some values are exceeding their normal operation range, the monitoring metaobject exchanges itself with a fault detection metaobject. A fault detection metaobject decides if a fault is about to happen or if it was a false alarm. This decision can be made by calculating increase/decrease rates for the values of a sensor object and by attaching metaobjects to an extended set of sensor objects. A fault detection metaobject can issue control commands to prevent a fault from occurring. If a fault occurs the fault detection metaobject exchanges itself with a fault handling metaobject. A fault handling metaobject has two responsibilities: To remedy the fault by issuing control commands in co-operation with the control part and to monitor the effects of the remedy actions. To fulfil the last responsibility, the fault handling metaobject may need to add metaobjects to an extended set of sensor objects. When a fault has been circumvented the fault handling metaobject exchanges itself with a monitoring metaobject. The set of state changes for the monitoring metaobjects is shown in Figure 2.

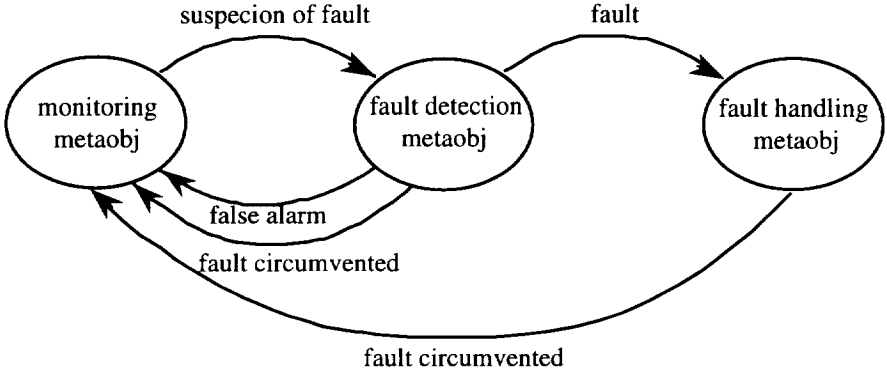

Figure 2 Exchange of monitoring metaobjects.

Another set of objects residing at the reflective level is responsible for controlling the physical process. The controlling objects are not attached to sensor objects, but are invoked from the monitoring metaobjects. The controlling objects perform control actions by communication with actuators at the base-level. A control action can be to move a specified amount of water from one tank to another. This requires starting of a pump and calculating which valves to open and which to close. Tanks are connected through a common tube system and a connection between two tanks requires opening/closing of a set of valves.

5.1 Composition of Metaobjects

Traditional reflective architectures allow each base object to have one metaobject attached. In process control applications, several independent aspects of behaviour need to be monitored simultaneously, as discussed in Section 4. The monitoring can

be performed by having a large set of metaobjects which contain combinations of the different aspects to be monitored. However, this will soon be incomprehensible. Composing metaobjects are a much more structured approach, where each metaobject is responsible for one aspect. Composition of metaobjects is obtained by attaching one system metaobject to a base object. The system metaobject administrates the composed metaobjects, and invoke them in turn. The system metaobject contains two lists. On the pre list are all metaobjects that should be notified before the method is performed in the base object. The post list contains the metaobjects that need to be notified after the performance of the method in the base object. A meta object may be in both lists at the same time. The architecture is shown in Figure 3.

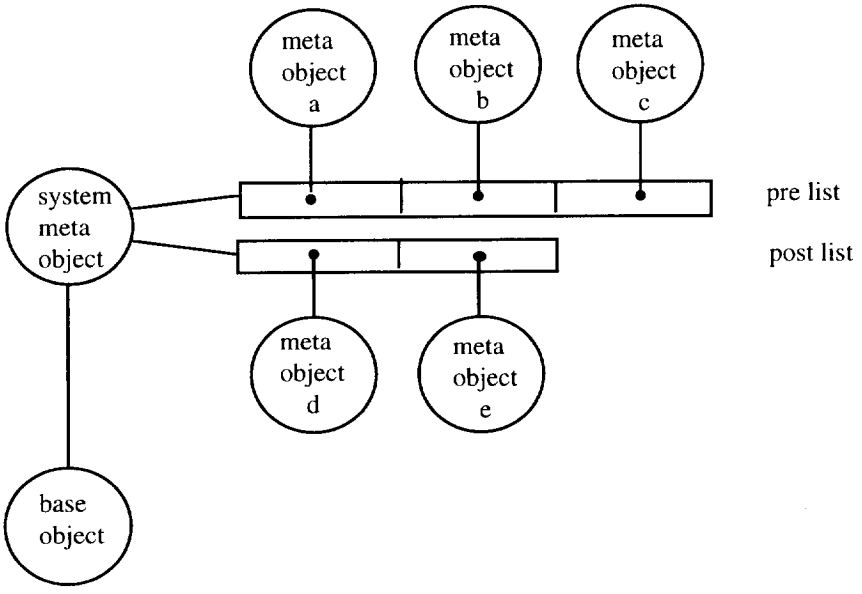

Figure 3 A meta architecture with composition of metaobjects

The system metaobject is invisible to the application. Meta objects are attached and removed using the following statements:

```
[aBaseObject insertPreMetaObject: aMetaObject]
[aBaseObject insertPostMetaObject: aMetaObject]
[aBaseObject withdrawPreMetaObject: aMetaObject]
[aBaseObject withdrawPostMetaObject: aMetaObject]
```

When a metaobject is attached to a base level object, all messages send to the base object are intercepted. A metaobject decides which messages to handle and metaobjects attached to the same base object can handle different messages.

5.2 Causal Connection

In process control applications, it does not make sense to talk about a causal connection between a base-level object and a metaobject. The directly monitored base-level objects are sensor objects and their is no way to dynamically change their behaviour. However, if we look at a physical system in a greater perspective, we are monitoring the entire process by selecting a set of sensor objects and we are affecting its behaviour through actuators. The way the physical process is affected is dependent on the metaobjects. However, instead of changing a metaobject, we exchange it with another metaobject, which has a different affect on the physical process.

5.3 Architecture Summary

The proposed architecture separates an application into a base level and a reflective level. The base level consists of a set of interrelated physical entities. The reflective level consists of a set of independent metaobjects, each responsible for a limited part of the overall functionality.

The architecture is highly dynamic. Monitoring metaobjects exchange themselves as a situation develops. This implies that a monitoring strategy is selected dependent on the current context. Furthermore, it is possible to develop new metaobjects or to correct errors in existing metaobjects and to install them on the fly. For a process control application, this is an important feature because it is usually expensive to close down the system for software updates.

The possibility for composing metaobjects makes the architecture extendible. New metaobjects can be developed and installed without affecting already existing metaobjects.

6 A Diagnosis System with a Reflective Architecture

This section describes a diagnosis system which we have implemented using the reflective architecture proposed in Section 5.

The specific diagnosis system is an emergency management system for use on board ships. The system detects fires and damages to the hull, based on around 500 sensors and provides decision support for handling of emergencies. On board a ship, the master is responsible for safety and for the operation of the ship. The diagnosis system must not automatically control the ship for reasons of legal responsibility. Instead, the system proposes which actions to take. The actions are carried out when

they have been acknowledged at a graphical user interface, so the diagnosis system is carrying out control through the actuators although it does not do it automatically.

The diagnosis system is described by its base level objects and by objects at the reflective level, which handle the two kinds of emergencies.

6.1 THE BASE LEVEL

The base level models the ship and consists of a variety of ship objects, such as decks, compartments, tanks, fire equipment, pumps, sensors, and actuators. All together 49 different ship objects are defined. All ships consists of the same type of objects, but they differ in the number and placement of the objects. To model a specific ship, a set of ship objects are connected using relations. In a shipmodel, it is possible, given one ship object, to get to any other ship object via the relations. A shipmodel expresses the static structure of the ship including the location of sensor objects. Two different types of sensor objects exist, analogue sensors which measure a value and binary sensors which measure an on/off state. Analogue sensors are measuring contents in tanks, pressures in tubes, and temperatures in compartments. Binary sensors are measuring the state of valves, smoke detectors, fire doors, and watertight doors. When physical entities, such as fire doors or valves, change their states, the changes are reflected by a change of state in their corresponding sensor objects.

To demonstrate the diagnosis system on land a sensor simulator was developed. The simulator constantly update all sensor objects and it can be used to generate faults. When faults are simulated mathematical models are used to calculate the changes of sensor values. On board a ship the diagnosis system will be connected to a process net, which collects values from the physical sensors and updates the sensor objects in the diagnosis system.

6.2 THE REFLECTIVE LEVEL

The reflective level consists of 10 different metaobject classes, whose objects are dynamically attached, replaced, and removed from sensor objects depending on the development of an emergency. Composition of metaobjects is used to attach a number of metaobjects to sensor objects. It is a requirement to log all sensor values and logging is performed by attaching an event logging metaobject. The event logging metaobject is usually attached permanently to all sensor objects. The class hierarchy for the metaobjects are shown in Appendix B. The diagnosis system handles two types of emergencies: fire and damages to the hull. These are described below.

6.2.1 FIRES

A fire emergency goes through three steps: monitoring, detection, and fire fighting. During the monitoring phase one metaobject, called the FireMonitoringMetaObject is monitoring all temperature sensors in accommodation compartments. If one sensor is exceeding its normal operation value, individual monitoring of that sensor is performed and the shared FireMonitoringMetaObject exchanges itself with a FireDetectionMetaObject.

To give an example of a metaobject, we will present the simple FireMonitoringMetaObject. All sensor objects in accommodation compartments share one FireMonitoringMetaObject. Each time one of the sensor objects is updated, by a SetValue:atTime message, the FireMonitoringMetaObject is invoked. A metaobject is invoked by the message handleMsg: aReceiver selector: (SEL)aSelector args: arguments, where aReciever is the base level object. The FireMonitoringMetaObject make a simple check to test whether the sensor value, i.e. the temperature is greater than an upperlimit. If the upperlimit is exceeded the FireMonitoringMetaObject exchanges itself with a FireDetectionMetaobject. Here is the code:

```
-handleMsg: aReceiver selector: (SEL)aSelector
                                  args: arguments
{
   id   fireDetectionMetaObject;


   struct sensorObjectDef {
     @defs (SensorObject);
   } *public;

if (aSelector == @selector(setValue:atTime:)){
   public = (struct sensorObjectDef *) aReceiver;

   if (public->value > upperLimit) {
    fireDetectionMetaObject =[[FireDetectionMetaObject
                                alloc] init: aReciever];
     [aReceiver withdrawPostMetaObject:self];
     [aReceiver insertPostMetaObject:
                              fireDetectionMetaObject];
   }

}
return self;

}
```

Figure 4 The code of the FireMonitoringMetaObject.

The FireDetectionMetaObject is responsible for intensified monitoring in a possible fire situation, for issuing a fire warning, and to continue monitoring the situation until a fire is detected or until it has been determined that it was a false alarm. A fire warning is issued, when it is suspected that a fire will burst out. Issuing of fire warnings are based on interpretation of sensor values from smoke detectors, current temperature, and a temperature increase rate in the compartment. The FireDetectionMetaObject calculates the temperature increase rate and attach a metaobject to the smoke detector in the compartment.

When a fire is detected, the FireDetectionMetaobject on the sensor object in the compartment on fire is exchanged with a FireHandlingMetaObject. The exchange is done by the FireDetectionMetaObject. One FireHandlingMetaObject monitors and controls all compartments on fire, so the FireDetectionMetaObject checks if a FireHandlingMetaObject already exists. If it does not exist, it is created and the exchange takes place. Figure 5 shows a situation where the temperature is normal in eleven compartments, temperature is increasing in three compartments, and one compartment is on fire.

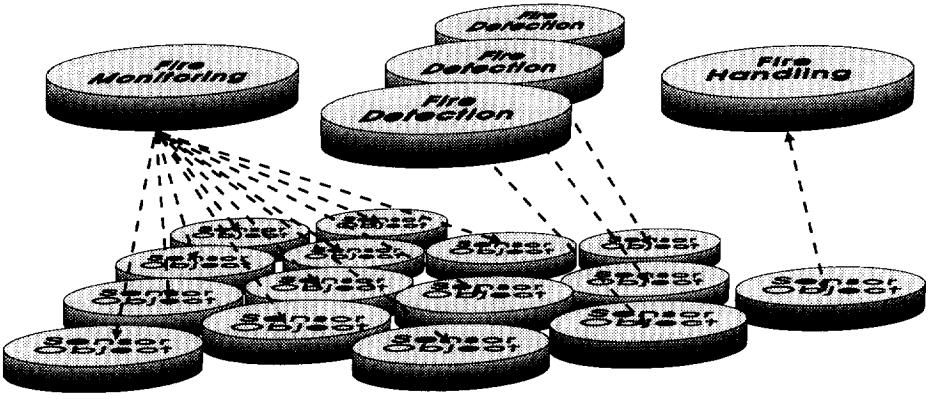

Figure 5 Eleven normal compartments, three compartments with increasing temperature, and one compartment on fire.

The FireHandlingMetaObject is responsible for issuing a fire alarm at the graphical user interface and for monitoring the development and the spreading of the fire. Furthermore, the FireHandlingMetaObject communicates with a mathematical fire model, which predicts how the fire will spread throughout the ship. This information is presented at the graphical user interface together with a strategy for fire fighting.

When the fire is spreading and adjacent compartments catch fire, their metaobjects are exchanged with the FireHandlingMetaObject. The FireHandlingMetaObject is complex, it monitors more than one object and has knowledge of which compartments are on fire. This situation is shown in Figure. 6.

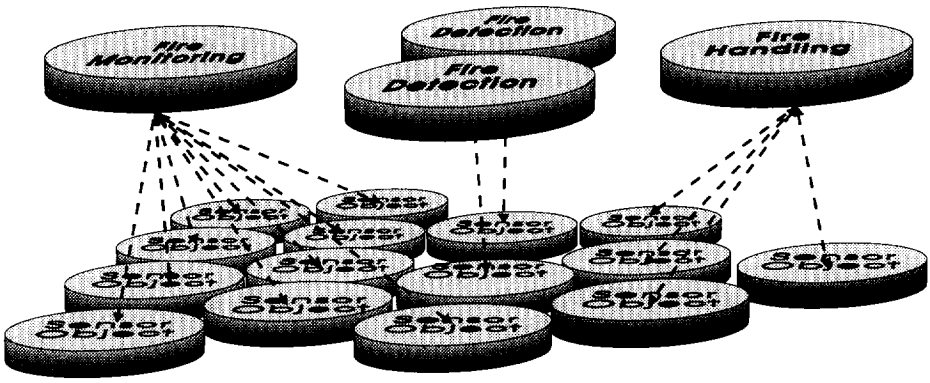

Figure 6 Nine normal compartments, two compartments with increasing temperature, and four compartments on fire.

When a fire is detected all relevant state changes are shown at the graphical user interface. Relevant state changes include sprinklers, valves, fire doors, dampers, and ventilation. Monitoring of state changes is done by dynamically connecting a metaobject to the relevant binary sensor objects.

6.2.2 DAMAGES TO THE HULL

Damages to the hull use the same principle of exchanging metaobjects, depending on the development of the emergency. A difference is that individual monitoring of sensor objects measuring contents in tanks takes place from the beginning. Individual monitoring is needed because tanks do not have the same size and their contents differ. The attached DamageMonitoringMetaObjects monitor increase/decrease rates in a tank. If the level is suddenly increasing/decreasing and no pumping operations take place, the reflective level interprets it as a damage. The user is informed and the DamageMonitoringMetaObject is exchanged with a DamageHandlingMetaObject. The DamageHandlingMetaObject communicates with a mathematical stability module, which calculates the stability of the ship and proposes pumping of ballast water into or out of tanks. When the user starts a pumping operation moving a specified amount of ballast water from one tank to another, this operation is also monitored by an AutomaticPumpTransferMetaObject. The metaobject monitors when the specified amount of water has been moved and stops the pumping operation automatically Starting and stopping an operation is done by sending control commands to actuators.

The handling of a fire emergency and a damage to the hull follow the same basic principle of dynamically exchanging metaobjects depending on the development of the emergency. Furthermore, the set of entities that are monitored is changing dynamically, depending on the situation. The reflective level is also able to monitor operations and to stop them automatically when they are finished.

7 CONCLUSION

To obtain dynamically adaptable process control applications, we have suggested a reflective architecture which is implemented using computational reflection. The architecture fulfils the requirements we have identified based on our experience developing several diagnosis and alarm systems and the architecture should thus be useful for a range of process control applications. We have implemented one specific diagnosis system based on the proposed architecture and a number of important benefits were achieved.

The benefits include: Separation of a model of a physical process and the monitoring and controlling part. This allows the two parts to be programmed and maintained independently. Dynamic exchange of metaobjects allows monitoring strategies to be programmed as single comprehensible entities. The entities are dynamically exchanged to obtain the equivalence of an overall monitoring strategy. Composition of metaobjects avoids to program numerous metaobjects with all possible combinations of behaviour. Instead each behaviour is programmed in one metaobject and the metaobjects are composed by attaching a number of metaobjects to the same base level object.

Using the proposed reflective architecture, we have obtained a highly modular system which was easy to implement and test.

8 ACKNOWLEDGEMENTS

The definition of the reflective architecture and the implementation of the diagnosis system were developed as part of the EEC project ATOMOS under the EURET programme. The functional specification of the system was developed in co-operation with The Danish Maritime Institute and Lloyds Register of Shipping, London. The research was also supported by the Danish Research Council under grant number 9500756. This support is gratefully acknowledged.

REFERENCES

[Agha and Sturman 94] Gul Agha, Daniel Sturman:
A Methodology for Adapting to Patterns of Faults
To appear in G.Koob(ed):
Foundations of Ultradependability
Vol. 1, Kluwer Academic 1994

[Aksit et.al 93] M. Aksit; K. Wakita; j. Bosch, L. Bergmans; A. Yonezawa:
Abstracting object-interactions using composition-filters.
Object-based Distributed Processing
Lecture Notes in Computer Science Vol. 791.
Springer-Verlag 1993.

[Chan and Lee 96] P. Chan; R. Lee:
The Java Class Libraries: An Annotated Reference.
Addison-Wesley, Reading, Massachusetts, 1996

[Christensen et al. 94] Kim Harding Christensen, Charlotte Pii Lunau, Jeppe Sommer:
Design Specification for the Emergency Management System Emma
Technical Report Atomos Task 2304
Aalborg University 1994

[Cointe 87] Pierre Cointe:
Metaclasses are First Class: the ObjVlisp Model
Proc. of Object-Oriented Programming: Systems, Languages and Applications
October 1987

[Cox 86] Brad J. Cox:
Object-Oriented Programming An Evolutionary Approach
Addison-Wesley, Reading, Massachusetts, 1986

[Ferber 89] Jacques Ferber:
Computational Reflection in Class Based Object Oriented Languages.
Proc. of Object-Oriented Programming: Systems, Languages and Applications
p 317 - 326, October 1989

[Gamma et. al. 94] E. Gamma; R. Helm; R. Johnson; J. Vlissides:
Design Patterns Elements of Reusable Object-Oriented Software
Addison-Wesley, Reading, Massachusetts, 1994

[Goldberg and Robson 83] Adele Goldberg, David Robson:
Smalltalk-80 the language and its implementation
Addison-Wesley, Reading, Massachusetts, 1983

[Kiczales et al. 91] Gregor Kiczales, Jim des Rivieres, Daniel Bobrow:
The Art of the Metaobject Protocol.
MIT Press, Cambridge, Massachusetts, 1991

[Lunau and Nielsen 95] Charlotte Pii Lunau and John Koch Nielsen:
Emma: An Emergency Management System
for use onboard Ships
Proceedings of IFAC Workshop on Control Applications in Marine Systems
p. 164-173 Trondheim May 1995.

[Maes 87] Pattie Maes:
Computational Reflection
Ph D. Thesis
Technical Report 87-2
Artificial Intelligence Laboratory
Vrije University Brussel, 1987

[NeXT 92] NeXTSTEP Object-Oriented Programming and the
Objective C Language: Release 3
Addison-Wesley Publishing Company
Readings, 1992.

[Rao 91] Ramano Rao:
Implementation Reflection in Silica
Proceedings of ECOOP '91
p. 251-267
Lecture Notes in Computer Science,
Springer-Verlag 1991

[Smith 82] Brian C. Smith:
Reflection and Semantics in a Procedural Language
Technical Report TR-272, MIT 1982

APPENDIX A: COMPUTATIONAL REFLECTION IN OBJECTIVE-C

This Appendix describes how Objective-C is extended to allow computational reflection.

IMPLEMENTATION OF COMPUTATIONAL REFLECTION

To extend a class based language with computational reflection, messages need to be reified and reflected upon.

In Objective-C, a message sent is translated into a call of the function objc_msgSend(receiver, selector, arguments). This function need to be changed so that it first checks if the receiver has a metaobject. If the receiver has a metaobject the function must sent a handleMsg to the metaobject, as shown in Figure A.1

```
objc_msgSend(receiver, selector, arguments)
  if [receiver hasMetaObject]
     [objc_msgSend(metaObject(receiver),
       "handleMsg:receiver:args",
              receiver,selector,arguments)]
```

Figure A.1 Changed *Objc_msgSend* function

Unfortunately, it is not possible to change the *objc_msgSend(receiver, selector, arguments)* function because it is part of Objective-C's runtime system. Instead the functionality is obtained by changing class pointers dynamically. In Objective-C each object has a *isa* pointer to its class. The class has a *isa* pointer to its metaclass and a list of methods used by the runtime system when resolving message sending. When an object is sent a message and the runtime system cannot find an implementation of the method the receiving object is sent a *-forward::* message. In order to provoke a *-forward::* message for each message sent to an object with a metaobject we change the *isa* pointer of the object. Instead of pointing at the object's class the *isa* pointer points to a class MessageIntercept that implements the *-forward::* method as it's only method. MessageIntercept's implementation of *-forward::* first have to find the metaobject of the object causing the intercept and then sends it the *-handleMsg:receiver:args:* message.

Because Objective-C is class based, metaobjects are instances of classes. The only responsibility that metaobjects need to fulfil is the ability to respond to the *-handleMsg:receiver:args:* message. This implies that new metaobjects are easy to define and they can be defined using inheritance.

Metaobjects need to be attached and removed dynamically to objects. In order to do this, Object has been extended with a new category. A category is a mechanism for extending the set of methods defined on a class, without having access to the source code of the class. The new category contains the following three methods:

```
@interface Object(metakit)
   -insertPreMetaObject: (id <MetaObject> aMetaObject
   -insertPostMetaObject: (id <MetaObject> aMetaObject
   -withdrawPreMetaObject: (id <MetaObject> aMetaObject
   -withdrawPostMetaObject: (id <MetaObject> aMetaObject
   -(BOOL)hasMetaObject;
   -(id<metaObject>)metaObject;
@end
```

The *-insertPreMetaObject* and *-insertPostMetaObject* methods make the argument object the metaobject of the receiver. The *-withdrawPreMetaObject* and -

withdrawPostMetaObject methods remove the argument object from the list of metaobjects. *-hasMetaObject* returns YES if the receiver has a metaobject. *-metaObject* returns the metaobject of the receiver.

APPENDIX B: THE METAOBJECT CLASS HIERARCHY