Integrating COTS Software Components into Dependable Software Architectures

Paulo Asterio de C. Guerra
Cecília Mary F. Rubira
Instituto de Computação
Universidade Estadual de Campinas, Brazil
{asterio,cmrubira}@ic.unicamp.br

Alexander Romanovsky
School of Computing Science
University of Newcastle upon Tyne, UK
alexander.romanovsky@ncl.ac.uk

Rogério de Lemos
Computing Laboratory
University of Kent at Canterbury, UK
r.delemos@ukc.ac.uk

Abstract

This paper considers the problem of integrating commercial off-the-shelf (COTS) software components into systems with high dependability requirements. These components, by their very nature, are built to be reused as black boxes that cannot be modified. Instead, the system architect has to rely on techniques external with respect to the component for resolving mismatches of the services required and provided that might arise in the interaction of the component and its environment. An approach is described in this paper to how these techniques should be structured around the COTS component to obtain an idealised fault-tolerant component. The approach employs the layer-based C2 architectural style for structuring mechanisms of error detection and recovery that should be integrated into the software architecture. The feasibility of the proposed approach is presented in the context of a steam boiler system which contains a COTS controller.

1. Introduction

Component-based software development (CBSD) is recognized today as an effective way to reduce development costs and time-to-market [S98]. Until recently, the majority of CBSD uses was primarily for client-tier applications, with little attention paid to server-tier components [BW98]. Components at the client side are usually fine-grained classes of simple objects, such as boxes and buttons on the user's screen. Their counterparts at the server side are large-grained components

encapsulating complex business rules or infrastructure services, usually comprising a set of related components such as commercial off-the-shelf (COTS) application frameworks and data base managers.

Today the main challenges of component-based software engineering (CBSE) are in guaranteeing system safety, reliability, and security [V98]. Although the fundamental principles of CBSD apply to both client-side and server-side portions of a system equally, their dependability requirements are substantially different. A COTS component is usually provided as a black box to be reused "as it is", which can independently evolve after it was integrated. These components usually do not have complete rigorously-written specification, there is no guarantee that the description the integrators have in their disposal is correct (very often it is ambiguous). These components can have bugs, moreover, the specific context in which they are used is not known at their development time. When integrating such a component into a system with high dependability requirements we should employ solutions at the architectural level to ensure that these requirements are met, irrespective of faults in the COTS component itself or in the way it interacts with the other system components.

Research into describing software architectures with respect to their dependability properties has recently gained considerable attention [SI99, S01, S98]. In [GRL02] the idealised fault-tolerant component concept [AL81] is applied in the architectural description of fault-tolerant component-based systems. [PR01] puts forward a general approach to developing protective wrappers to be used for building dependable software systems based on COTS components. In this paper we combine the concepts of an idealised architectural component and protective wrappers to develop an architectural solution that provides an effective and systematic way for building dependable software systems from COTS software components. The rest of the paper is organised as follows. In the next section, we briefly discuss background work on the idealised fault-tolerant component, architectural mismatches and the C2 architectural style. Section 3 describes the architectural representation of idealised fault-tolerant COTS. The case study demonstrating the feasibility of the proposed approach is presented in section 4. Related work on how to build dependable software system based on COTS components is discussed in section 5. Finally, section 6 presents some concluding remarks and discusses our future work.

2. Background

2.1. Architectural Mismatches and COTS Component Integration

Dealing with architectural mismatches [GAO95] is one of the most difficult problems system integrators face when developing architectural approaches to integrating systems with COTS components. An *architectural mismatch* occurs when the assumptions that a component makes about another component or the rest of the system do not match. That is, the assumptions associated with the service provided by the component are different from the assumptions associated with the services required by the component for behaving as specified [OWZ98]. When building systems from existing components, it is inevitable that incompatibilities between the service delivered by the component and the service that the rest of the system expects from that component, give rise to such mismatches. These mismatches are not exclusive to the functional attributes of the component; mismatches may also include quality attributes, such as dependability, which can be related to the component failure mode assumptions or its safety integrity levels.

We view all incompatibilities between a COTS component and the rest of the system as architectural mismatches. This, for example, includes internal faults of a COTS component that affect others system components or its environment, in which case the failure assumptions of the component were wrong.

2.2. The C2 Architectural Style

Architectural structures of a system tend to abstract away the details of the system, but assist in understanding broader system-level concerns [SG96]. This is achieved by employing architectural styles that are appropriate for describing the system in terms of its components, the interactions between these components - connectors, and the properties that regulate the composition of components - configurations.

The C2 architectural style is a component-based style that supports large grain reuse and flexible system composition, emphasizing weak bindings between components [TMA+96]. In this style, components of a system may be completely unaware of each other, as when one integrates various COTS components, which may have heterogeneous styles and implementation languages. These components communicate only through asynchronous messages mediated by connectors that are responsible for message routing, broadcasting and filtering. Interface and architectural mismatches are dealt with by means of wrappers that encapsulate each component.

In the C2 architectural style both components and connectors (Figure 1) have a *top interface* and a *bottom interface*. Systems are composed in a layered style, where the top interface of a component may be connected to the bottom interface of a connector and its bottom interface may be connected to the top interface of another connector. Each side of a connector may be connected to any number of components or connectors.

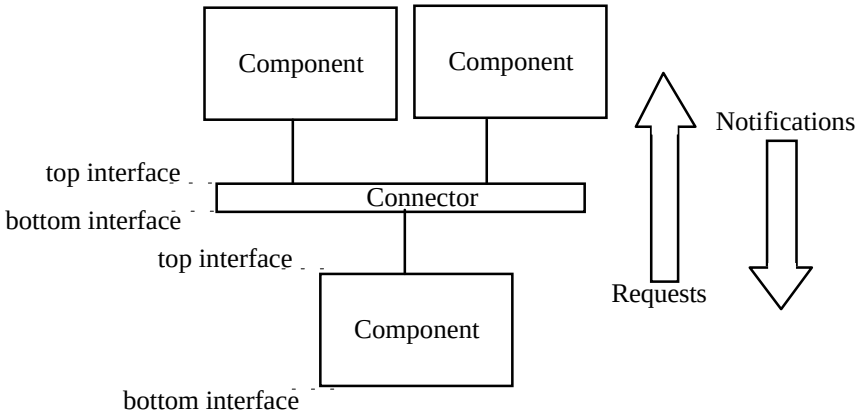

Figure 1. C2 Style Basic Elements

There are two types of messages in C2: requests and notifications. *Requests* flow up through the system layers and *notifications* flow down. In response to a request, a component may emit a notification back to the components below, through its bottom interface. Upon receiving a notification, a component may react with the *implicit invocation* of one of its operations.

3. Idealised Fault-Tolerant COTS Component

Current large scale systems usually integrate COTS components which may act as service providers and/or service users. Since, there is no control, or even full knowledge, over the design, implementation and evolution of COTS components, the evolutionary process of a COTS component should be considered as part of a complex environment, physical and logical, that might directly affect the system components. In order to build a dependable software system from untrustworthy COTS components, the system should treat these components as a potential source of faults. The overall software system should be able to support COTS components while preventing the propagation of errors. In other words, the system should be able to tolerate faults that may reside or occur inside the COTS components, while not being able to directly inspect or modify its internal state or behaviour.

In this paper, we present the concept of an *idealised fault-tolerant COTS component*, which is an architectural solution that encapsulates a COTS component adding fault tolerance capabilities to allow it to be integrated in a larger system. These fault tolerant capabilities are related to the activities associated with error processing, that is, error detection and error recovery. The idealised fault-tolerant COTS component is a specialization of the idealised C2 Component (iC2C) [GRL02], which is briefly described in the following section.

3.1. The Idealised C2 Component (iC2C)

The idealised C2 component (iC2C) is equivalent, in terms of behaviour and structure, to the idealised fault-tolerant component [AL81] and designed to allow the structuring of software architectures compliant with the C2 architectural style [TMA+96]. Service requests and normal responses of an idealised fault-tolerant component are mapped as requests and notifications in the C2 architectural style. Interface and failure exceptions of an idealised fault-tolerant component are considered subtypes of notifications. In order to minimize the impact of fault tolerance provisions on the system complexity, we have decoupled the normal activity and abnormal activity parts of the idealised component. This outcome has lead to an overall structure for the iC2C that has two distinct components and three connectors, as shown in Figure 2.

The iC2C NormalActivity component implements the normal behaviour, and is responsible for error detection during normal operation, and the signalling of interface and internal exceptions. The iC2C AbnormalActivity component is responsible for error recovery, and the signalling of failure exceptions. For consistency, the signalling of an internal exception by an idealised fault-tolerant component was mapped as a subtype of notification, and, the “return to normal”, flowing in the opposite direction, was mapped as a request. During error recovery, the AbnormalActivity component may also emit requests and receive notifications, which are not shown in Figure 2.

The connectors of our iC2C (Figure 2) are specialized reusable C2 connectors with the following roles.

(i) The iC2C_bottom connector connects the iC2C with the lower components of a C2 configuration, and serializes the requests received. Once a request is accepted, this connector queues new requests that are received until completion of the first request. When a request is completed, a notification is sent back, which may be a normal response, an interface exception or a failure exception.

(ii) The iC2C_internal connector controls message flow inside the iC2C, selecting the destination of each message received based on its originator, the message type and the operational state of the iC2C (either under normal or abnormal operation).

(iii) The iC2C_top connector connects the iC2C with the upper components of a C2 configuration, which may provide services to the NormalActivity and/or AbnormalActivity components.

The overall structure defined for the idealised C2 component is fully compliant with the component rules of the C2 architectural style. This allows an iC2C to be integrated into any C2 configuration and interact with components of a larger system. When this interaction establishes a chain of iC2C components, the external exceptions raised by a component can be handled by a lower level component (in the C2 sense of "upper" and "lower") allowing hierarchical structuring of error recovery activities. An iC2C may also interact with a regular C2 component, either requesting or providing services.

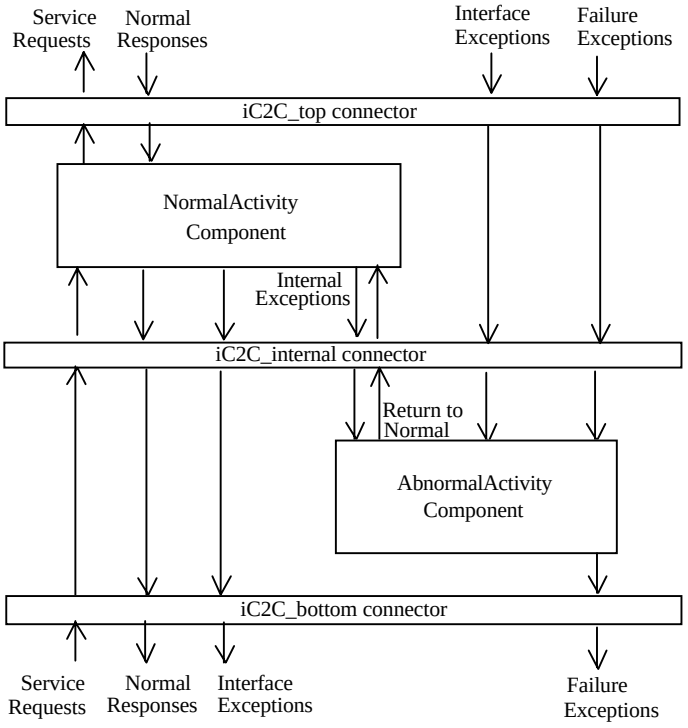

Figure 2. Idealised C2 Component (iC2C)

3.2. COTS Component Protectors

Component wrapping is a well-known structuring technique that has been used in several areas. In this paper, we use term “wrapper” in a very broad sense, incorporating the concepts of wrappers, mediators, and bridges [D99]. A *wrapper* is a specialised component inserted between a component and its environment to deal with the flows of control and data going to and/or from the wrapped component. The need for wrapping arises when (i) it is impossible or expensive to change the components when reusing them as parts of a new system, or (ii) if it is easier to add new features by incorporating them into wrappers. Wrapping is a structured and a cost-effective solution to many problems in component-based software development. Wrappers can be employed for improving quality properties of the components such as adding caching and buffering, dealing with mismatches or simplifying the component interface. With respect to dependability, wrappers are usually used for ensuring security, transparent component replication, etc.

A systematic approach has been proposed for using protective wrappers, known as *protectors*, that can improve the overall system dependability. This is achieved by protecting both the system against erroneous behaviour of a COTS component, and the COTS component against erroneous requests from the rest of the system. The wrappers are viewed as redundant software that detects errors or suspicious activity and executes appropriate recovery when possible.

The development of protectors is considered as a part of system integration activities [PR01]. The approach consists of rigorous specification of the wrapper functionality in forms of *acceptable behaviour contraints* (ABCs) and in their execution at run time in the form of executable assertions detecting a contraint violation and of exception handlers recovering after it. The general sources of information to be used in developing both ABCs and possible actions to be undertaken in response to their violations are the following:

(i) The behaviour specification of COTS components as specified by the COTS’s developers.

(ii) The behaviour specification of a COTS component as specified by the system designers. This description and the previous one must satisfy certain mutual constraints for the system design to be correct, but they will not be identical. E.g., the system designer's description requires the COTS component to be able to react to a set of stimuli that is a subset of the set specified by the COTS’s developers.

(iii) The behaviour that the system designer expects from a COTS component (not necessarily approving it), based on previous experiences with it, i.e., he/she may know that it often fails in response to certain legal stimuli.

(iv) Component (COTS or part of the rest of the system) behaviour that system designers considers especially unacceptable, without knowing whether it is likely or not.

(v) The behaviour specifications of the rest of the system.

ABCs are represented as executable assertions, the main elements of which are system states and sequences of system events as seen by the wrapper. The sources of information above allow the developer to formulate a number of statements describing the correct behaviour of the system (consisting in this case of the COTS component and of the rest of the system). The statements are expressed as a set of assertions on the states of input and output parameters. In addition to that they can include assertions on the histories (sequences of calls) that the protector has to collect and assertions on the states of the system components which are to be retrieved by the protector by calling side-effect-free functions returning the states of these components. While [PR01] deals with the design of COTS protectors and its development process, in this paper we are mainly concerned with architectural issues related to their integration in a fault-tolerant component-based system.

3.3. Idealised C2 COTS (iCOTS)

A protective wrapper for a COTS software component is a special type of application-specific fault-tolerance capability. To be effective, the design of fault-tolerance capabilities must be concerned with architectural issues, such as process distribution and communication mode, that impact the overall system dependability. Although the C2 architectural style is specially suited for integrating COTS components into a larger system, its rules on topology and communication are not adequate for incorporating fault tolerance mechanisms into C2 software architectures, especially the mechanisms used for error detection and fault containment [GRL02]. The idealised C2 fault-tolerant component (iC2C) architectural solution (section 3.1) overcomes these problems leveraging the C2 architectural style to allow such COTS software components to be integrated in dependable systems.

The idealised C2 COTS (iCOTS) is a specialization of the iC2C aiming to add protective wrappers to a COTS component to be integrated in a software system. In our approach, the COTS component is encapsulated into the NormalActivity component of an iC2C, wrapped by two specialized connectors acting as error detectors (Figure 3). These detectors are responsible for verifying that the messages that flows to/from the COTS being wrapped do not violate the acceptable

behaviour constraints specified for that system. The lower_detector inspects incoming requests and outgoing responses (C2 notifications) from/to the COTS clients while the upper_detector inspects outgoing requests and incoming responses to/from other components providing services to the COTS.

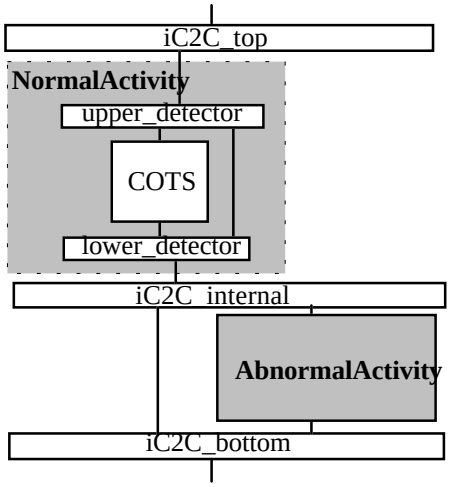

Figure 3. Idealised C2 COTS (iCOTS) Overall Structure

When a contraint violation is detected, the detector sends an exception notification which will be handled by the AbnormalActivity component, following the rules defined for the iC2C. Any of these detectors may be decomposed in a set of special purpose error detectors which, in their turn, are wrapped by a pair of connectors. For example, Figure 4 shows an upper_detector decomposed into a number of error detectors. The detector_bottom coordinates error detection, and the detector_top connects the whole detector either to the COTS or to the iC2C top_connector.

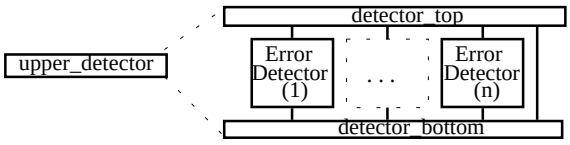

Figure 4. Decomposition of a Detector

The AbnormalActivity component is responsible for both error diagnosis and error recovery. Depending on the complexity of these tasks, it may be convenient to decompose it into more specialized components for error diagnosis and a set of error handlers, as illustrated by Figure 5. This design allows the ErrorDiagnosis component to react directly to exceptions raised by the NormalActivity component and send notifications to activate the ErrorHandlers or, alternatively, to stand as a service provider of requests sent by the ErrorHandlers.

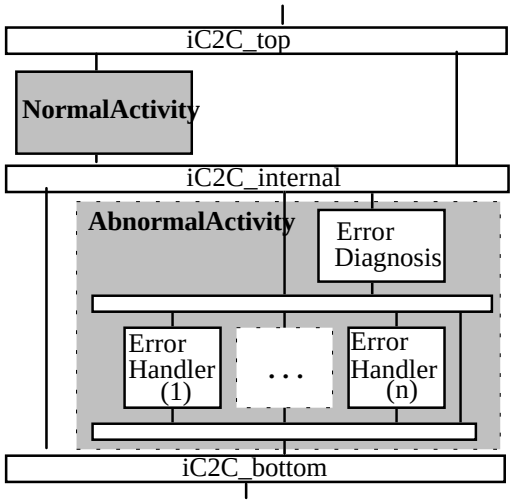

Figure 5. Decomposition of the AbnormalActivity

4. Case Study

4.1. Problem Statement

Anderson et. al. [AF03] present the results of a case study in protective wrapper development [PR01], in which a Simulink model of a steam boiler system is used together with an off-the-shelf PID (Proportional, Integral and Derivative) controller. The protective wrappers are developed to allow detection and recovery from typical errors caused by unavailability of signals, violations of limitations, and oscillations.

The boiler system comprises the following components: the physical boiler, the control system and the rest of the system. In turn, the control system consists of PID controllers, which are the COTS components, and the rest of the system consisting of:

(i) Sensors - these are "smart" sensors that monitor variables providing input to the PID controllers: the drum level, the steam flow, the steam pressure, the gas concentrations and the coal feeder rate.

(ii) Actuators - these devices control a heating burner that can be ON/OFF, and adjust inlet/outlet valves in response to outputs from the PID controllers: the feed water flow, the coal feeder rate and the air flow.

(iii) Boiler Controller - this device allows to enter the configuration set-points for the system: the steam load and the coal quality, which must be set up in advance by the operators.

The Simulink model represents the control system as three PID controllers dealing with the feed water flow, the coal feeder rate and the air flow. These three controllers output three eponymous variables: feed water flow (F_wf), coal feeder rate (C_fr) and air flow (Air_f), respectively; these three variables, together with two configuration set-points (coal quality and steam load) constitute the parameters which determine the behaviour of the boiler system. There are also several internal variables generated by the smart sensors; some of these, together with the configuration set-points, provide the inputs to the PID controllers, in particular: bus pressure set-point (P_ref), O2 set-point (O2_ref), drum level (D_l), steam flow (S_f), steam pressure/drum (P_d), steam pressure/bus (P_b), O2 concentration at economizer (O2eco), CO concentration at economizer (Coeco), and NOx concentration at economizer (Noxeco).

Anderson et. al. [AF03] summarise the available information describing the correct COTS component behaviour to be used in developing the protective wrappers. The following analysis helped the authors to formulate three types of erroneous conditions that can be detected by the protective wrappers positioned between the COTS components and the rest of the system: (i) Unavailability of inputs/outputs to/from the PID controllers; (ii) Violation of specifications of monitored variables; and (iii) Oscillations in monitored variables.

This information served as a base for formulating a number of acceptable behaviour constraints that were used in defining error detection features of the protectors. Depending on the severity of the errors and on the specific characteristics of the system, two types of recovery are used in the case study: raising an alarm and safe stop.

4.2. Architectural Solution

In this section, we describe an architectural solution for the steam boiler system conforming to the C2 architectural style and applying the idealised C2 COTS (iCOTS) previously described (section 3.2).

4.2.1. System Configuration

Figure 6 shows the proposed C2 architectural configuration which is organized in four layers: (i) the BoilerController component; (ii) the WaterFlowController and CoalFeederController; (iii) the AirFlowController, which has as input the CoalFeederRate from the CoalFeederController; and (iv) the sensors and actuators required by the system.

Table 1 specifies the operations provided by some key components that appear in Figure 6.

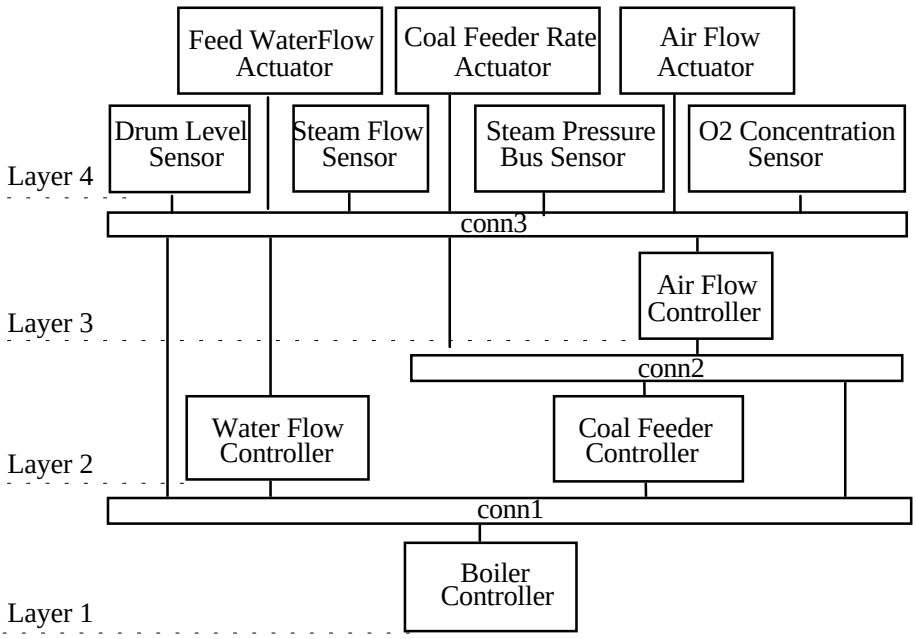

Figure 6. C2 Configuration for the Boiler System

Table 1. List of Operations

Operation	Provider Component
readDrumLevel() : D_l	Drum Level Sensor
readSteamFlow() : S_f	Steam Flow Sensor
readBusPressure() : P_b	Steam Pressure Bus Sensor
readO2Concentration() : O2eco	O2 Concentration Sensor
setFeedWaterFlow(F_wf)	Feed Water Flow Actuator
setCoalFeedRate(C_fr)	Cool Feeder Rate Actuator Air Flow Controller
setAirFlow(Air_f)	Air Flow Actuator
setConfiguration(P_ref, O2_ref)	Coal Feeder Controller Air Flow Controller

4.2.2. The Idealised AirFlowController

We assume that the three controllers are implemented reusing a COTS PID controller. In this section, we describe how we can build an iCOTS AirFlowController encapsulating the COTS PID controller with protectors. This solution equally applies to the other two controllers.

Figure 7 shows the internal structure of the iCOTS for the AirFlowController, based on Figure 3. The COTS PID controller is wrapped by a pair of error detectors (upper_detector and lower_detector) and inserted into an iC2C as its NormalActivity component. Both detectors use OscillatorChecker, which is responsible for checking whether oscillating variables revert to a stable state before a maximum number of oscillations. Table 2 specifies, for each detector, the message types to be inspected, their corresponding assertions that guarantee the acceptable behaviour constraints (section 4.1) and the type of the exception notification that should be generated when a contraint is violated. Table 3 summarises these exception types, grouped by their generalised types. Two of these exception types are interface exceptions that are sent directly to the next lower level in the architectural configuration. The other exceptions types are internal exceptions, to be handled by the AFCErrorHandler.

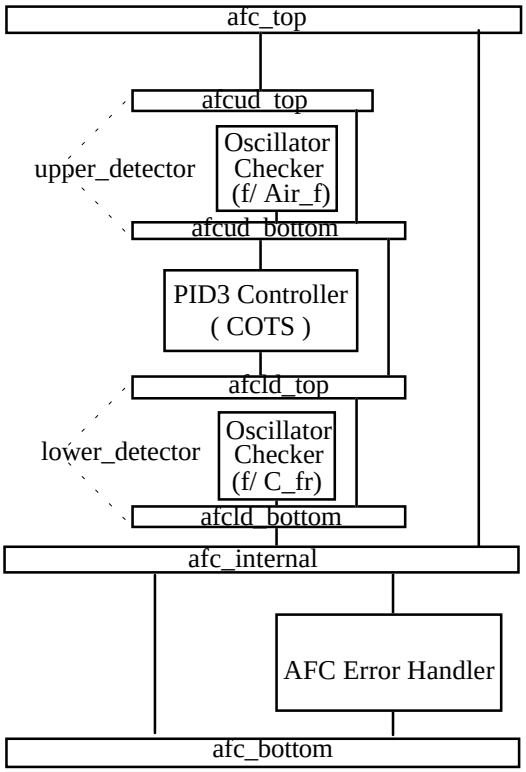

Figure 7. Decomposition of the AirFlowController

Table 2. Error Detection Specifications

Message Type	Constraints to be checked	Exceptional Notification
lower_detector		
Request setConfiguration(P_ref, O2_ref)	0 <= O2_ref <= 0.1	InvalidConfigurationSetpoint
	the corresponding notification must be received within a specified time interval	PIDTimeout
Request setCoalFeeder(C_fr)	0 <= C_fr <= 1	InvalidCoalFeederRate
	check_oscillate(Air_f)	CoalFeederRateOscillating
	the corresponding notification must be received within a specified time interval	PIDTimeout
upper_detector		
Request setAirFlow(Air_f)	0 <= Air_f <= 0.1	InvalidAirFlowRate
	check_oscillate(Air_f)	AirFlowRateOscillating
	the corresponding notification must be received within a specified time interval	AirFlowActuatorTimeout
Notification from readO2Concentration()	0 <= O2eco <=1	InvalidO2Concentration

Table 3. Summary of Exceptional Notifications

Exception Notification	Generic Exception Type
PIDTimeout	NoResponse (Unavailability of inputs/ outputs to/from the PIDController)
AirFlowActuatorTimeout	
InvalidConfigurationSetpoint*	OutOfRange (Violation of specifications of monitored variables)
InvalidCoalFeederRate*	
InvalidO2Concentration	
InvalidAirFlowRate	
CoalFeederRateOscillating	Oscillation (Oscillations in monitored variables)
AirFlowRateOscillating	
* Interface exceptions.	

In this example, the recovery actions are delegated to the BoilerController component, which may either sound an alarm or shut down the system, depending on the exception type. Thus, internal exceptions are propagated by the AFCErrorHandler as failure exceptions of the generic type of the corresponding internal exception, using the mapping shown in Table 3. A PIDTimeout exception, for example, will generate a NoResponse failure exception.

4.2.3. The BoilerController Component

This component is responsible for:

(i) configuring the boiler system, sending setConfiguration requests when appropriate;

(ii) handling interface exceptions of type InvalidConfigurationSetpoint, which may be raised in response of a setConfiguration request;

(iii) handling failure exceptions of type NoResponse, OutOfRange or Oscillation, that may be raised by the three controllers (WaterFlowController, CoalFeederController, AirFlowController).

We structured the BoilerController component as an iC2C, to cope with fault-tolerance responsabilities - itens (ii) and (iii) above, apart from its main functional responsability - item (i), by two distinct internal components: the AbnormalActivity and NormalActivity, respectively.

Table 4 specifies the actions to be taken by the AbnormalActivity component of the BoilerController, in response to each exception type that it may receive.

Figure 8 shows the resulting fault-tolerant architecture for this system, which is derived from the overall architectural configuration for the boiler system (Figure 6). Each of its three controllers is structured as idealised C2 COTS (iCOTS) and the BoilerController as an idealised C2 component (iC2C). It is assumed that the sensors and actuators, as well as the connectors, are reliable. Figure 9 illustrates the flow of messages between the various components involved when a PIDTimeout exception occurs, after the BoilerController fails to configure the AirFlowController PIDController. When the detector AirFlowController bottom connector (afcId_bottom) detects that the AirFlowController is not responding, it raises an exception to AFCErrorHandler. As this AFCErrorHandler cannot cope with this exception type, it raises another exception to the BCErrorHandler that shuts down the whole system.

Table 4. BoilerController Recovery Actions

Exception Type	Recovery Action
InvalidConfigurationSetpoint OutOfRange	Alarm and return to normal.
NoResponse Oscillation	Shut down the system.

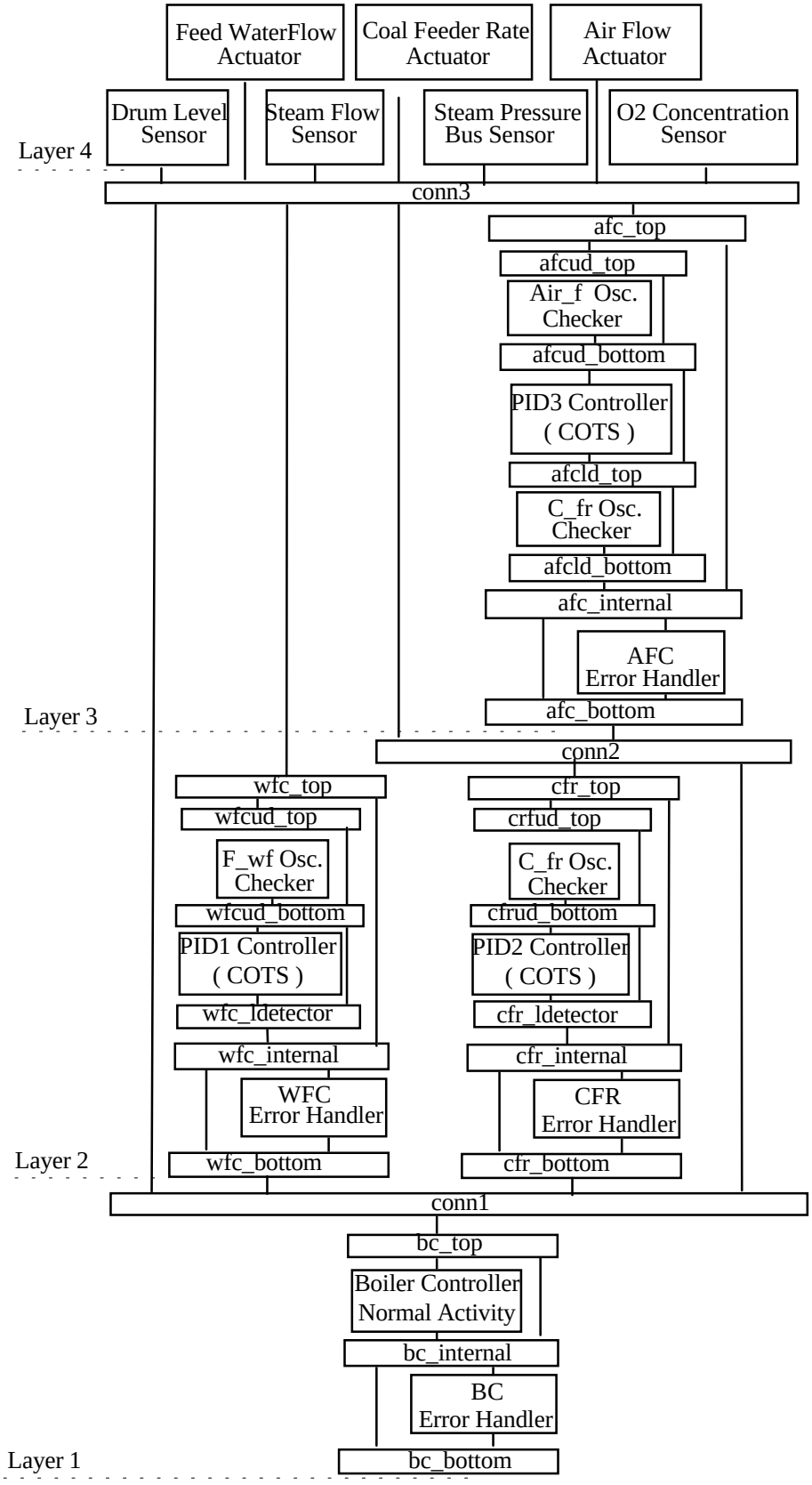

Figure 8. Resulting Configuration for the Boiler System

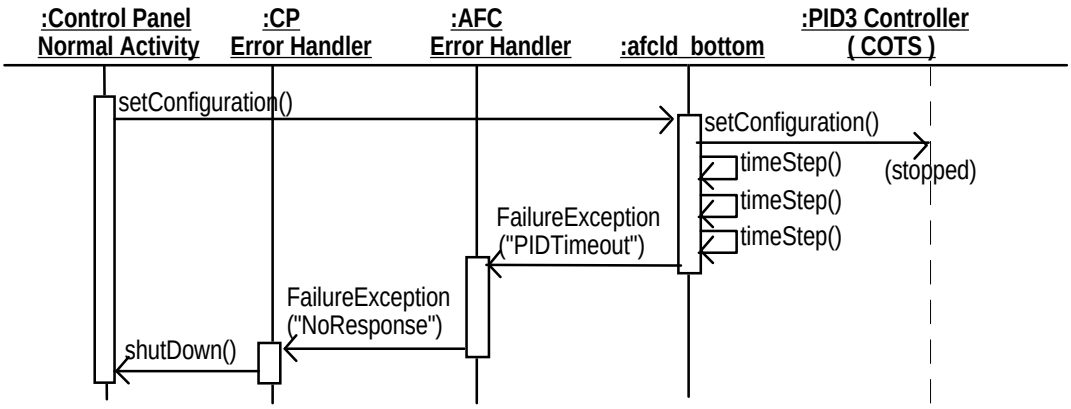

Figure 9. Sequence Diagram for a PIDTimeout Exception

4.2.4. Sample Implementation

Figures 10 and 11 shows a partial implementation of the protector for the AirFlowController. The listing in Figure 10 shows a class definition for its lower detector and how it intercepts a setConfiguration request and its corresponding notification. When this request is intercepted the detector checks if it violates a contraint and may decide to: (i) forward the request to the COTS PID Controller, if it does not violate any contraint; or (ii) send an interface exception notification down to the component that originated this request. If the request is accepted, the lower detector waits for the corresponding notification. If this notification is not received within an specified number of time steps it sends a PIDTimeout failure exception notification, to be handled by the AirFlowController error handler. The listing in Figure 11 shows a class definition for this error handler, which sends a NoResponse failure exception when a PIDTimeout exception is received.

5. Related Work

This section compares our approach with several relevant existing proposals. The main comparison criteria are the types of the components (application-level or middleware/OS level), fault tolerance (error detection and recovery) provided, type of the redundancy, the information used for developing error detection and recovery, phases of the life cycle (at which they are applied).

```
class AFC_LowerDetector extends ICOTSDetector {
  private Request req;
  private boolean waitingSetConfiguration=false;
  private boolean waitingCycles;
  public void handle(Request r) {
    IC2CException exc=null;
    if (r.type().equals("setConfiguration")) {
      try {
        float o2ref=((Float)r.getParameter("O2_ref")).floatValue();
        if (o2ref < 0 || o2ref > 0.1)
             exc=new IC2CInterfaceException("InvalidConfigurationSetpoint", r);
        else {
             waitingSetConfiguration==true;
             waitingCycles==10;
             req=r;
        }
      } catch (Exception e) {
             exc=new IC2CInterfaceException("InvalidConfigurationSetpoint", r);
      }
    }
    if (exc==null)
      send (r);
    else
      send (exc);
  }
  public void handle(Notification n) {
    if (n.type().equals("setConfiguration")) {
      waitingSetConfiguration==false;
    }
    send(n);
  }
  public void timeStep() {
    if (waitingSetConfiguration) {
      if(--waitingCycles==0) {
             send (new IC2CFailureException("PIDTimeout", req));
             waitingSetConfiguration=false;
      }
    }
  }
}
```

Figure 10. AFC Lower Detector Implementation.

```
class AFC_ErrorHandler extends IC2CErrorHandler {
  public void handle(IC2CException e) {
    if (e.type().equals("PIDTimeout")) {
      send (new IC2CFailureException("NoResponse", e.request());
    }
  }
}
```

Figure 11. AFC Error Handler Implementation.

Ballista [KD99] works with POSIX systems coming from several providers. The approach works under a strong assumption that the normal specification of the component is available, from which error detectors can be specified. In addition to this, the results of fault injection are used for the specification of error detectors. A layer between the applications and the operating system (OS), intercepting all OS calls as well as the outgoing results, implements this error

detection. The recovery provided by this approach is very basic (blocking the erroneous calls) and is not application-specific.

A very interesting approach to developing protective wrappers for a COTS microkernel is discussed in [SR99]. The idea is to specify the correct behaviour of a microkernel and to make the protective wrapper check all functional calls (similar to Ballista, this approach cannot be applied for application-level COTS components that lack a complete and correct specification of the component's behaviour). Reflective features are employed for accessing the internal state of the microkernel to improve the error detection capability. In addition, the results of fault injection are used in the design of wrappers for catching those calls that have been found to cause errors of the particular microkernel implementation. A recent work [RF02] shows how recovery wrappers can be developed within this framework to allow for recovery after transient hardware faults, which is mainly based on redoing the recent operation.

Unfortunately these two approaches focus only on the later phases of the system development, without offering any assistance in developing fault tolerant system architectures. The Simplex framework (the best summary of this work performed in mid 90's can be found in [S01]) proposes an architectural solution to dealing with the faults of the application-level COTS components. The idea is to employ two versions of the same component: one of them is the COTS component itself and another one is a specially-developed unit implementing some basic functions. The second unit is assumed to be bug free as it implements very simple algorithms. The authors call this analytical redundancy. The two units together form a safety unit in which only externally observable events of the COTS component are dealt with. The system architect is to implement a set of consistency constraints on the inputs to the COTS component and the outputs from it, as well as on the states of the device under control. This approach is oriented towards developing fault tolerant architectures of control systems. The disadvantage of this approach is that it is not recursive as it treats the whole control software as one unit and provides fault tolerance at only this level.

Rakic et. al. [RM01] offers a software connector-based approach to increasing the dependability of systems with COTS components. The idea is to employ the new and the (several if available) old versions to improve the overall system dependability. The authors put forward the idea of using a specialised multi-version connector allowing a system's architect to specify the component authority for different operations: a component designated as authoritative will be considered nominally correct with respect to a given operation. The connector will propagate only the results from an authoritative version to the rest of the system and at the same time, log the results of all the multi-versioned components'

invocations and compares them to the results produced by the authoritative version. The solution is mainly suitable for systems in which COTS components are to be upgraded (under the assumption that the interface of the old and new components remain unchanged) so there are several versions of a component in place.

6. Conclusions and Future Work

When building reliable systems from existing components, guarantees cannot be given on the system behaviour, if no guarantees are provided on the behaviour of its individual components. Since no such guarantees are provided for individual COTS components, architectural means at the component level have to be devised for the provision of the necessary guarantees at the system level.. The paper proposes an architectural solution to turning COTS components into idealised fault-tolerant COTS components by adding protective wrappers to them. We demonstrate the feasibility of the proposed approach using the steam boiler system case study, where its controllers are built reusing unreliable COTS components. Although we recognize that the proposed approach can result in incorporating repetitive checks into the integrated system, this is an unavoidable outcome considering the lack of guarantees provided by COTS components. For example, it might be the case that a COTS component has internal assertions checking the validity of an input parameter that is also checked by its protector, or other protectors in other COTS components. However, there are situations in which the system integrator can take care of this by coordinating development of fault tolerance means associated with individual components - we did not address this problem in our paper.

Although the case study has used a single architectural style, software components in the C2 architectural style can be nevertheless integrated into configurations of other architectural styles using simple adapters. This allows the idealised fault tolerant COTS (iCOTS) concept to be applied as a general solution in developing dependable systems from unreliable COTS components. Our future work includes extending an existing C2 Java framework [UCI02] to aid in the implementation of this new abstraction and its integration into C2 architectural configurations.

Acknowledgments

Paulo Guerra is partially supported by CAPES/Brazil. Cecília Rubira and Paulo Guerra are supported by the FINEP/Brazil “Advanced Information Systems” Project (PRONEX-SAI-7697102200). Cecília Rubira is also supported by

CNPq/Brazil under grant no. 351592/97-0. Alexander Romanovsky is supported by IST DSoS and EPSRC/UK DOTS Projects.

References

[AF03] T. Anderson, M. Feng, S. Riddle, A. Romanovsky. Protective Wrapper Development: A Case Study. To be presented at the *2nd Int. Conference on COTS-based Software Systems*. Ottawa, Canada. February, 2003.

[AL81] T. Anderson, P. A. Lee. *Fault Tolerance: Principles and Practice.* Prentice-Hall, 1981.

[BW98] A. W. Brown, K. C. Wallnau. The current state of CBSE. *IEEE Software*, 15(5):37--46, September / October 1998.

[D99] R. DeLine. "A Catalog of Techniques for Resolving Packaging Mismatch". *Proceedings of the 5th Symposium on Software Reusability (SSR'99)*. Los Angeles, CA. May 1999. pp. 44-53.

[GAO95] D. Garlan, R. Allen, J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17--26, November 1995.

[GRL02] P. A. C. Guerra, C. M. F. Rubira, R. de Lemos. An Idealized Fault-Tolerant Architectural Component, in *Proceedings of the ICSE 2002 Workshop on Architecting Dependable Systems*, Orlando, USA, 2002, pp. 15--20.

[KD99] P. Koopman, J. De Vale. Comparing the Robustness of POSIX Operating Systems. In *Proc. Fault Tolerant Computing Symposium (FTCS-29)*, Wisconsin, USA, 1999, 30-37.

[OWZ98] P. Oberndorf, K. Wallnau, A. M. Zaremski. “Product Lines: Reusing Architectural Assets within an Organisation. *Software Architecture in Practice*. Eds. L. Bass, P. Clements, R. Kazman. Addison-Wesley. 1998. pp. 331-344.

[PR01] P. Popov, S. Riddle, A. Romanovsky, L. Strigini. On Systematic Design of Protectors for Employing OTS Items. In *Proc. of the 27th Euromicro conference*. Warsaw, Poland, 4-6 September, IEEE CS, 2001. pp.22-29.

[RF02] M. Rodriguez, J.-C. Fabre, J. Arlat. Wrapping Real-Time Systems from temporal Logic Specification. In *Proc. European Dependable Computing Conference (EDCC-4)*, 2002, Toulouse (France)

[RM01] M. Rakic, N. Medvidovic. Increasing the Confidence in Off-The-Shelf Components: A Software Connector-Based Approach. *Proceedings of {SSR'01} - 2001 Symposium on Software Reusability. ACM/SIGSOFT Software Engineering Notes*, 26,3. 2001. Pp. 11-18.

[S01] L. Sha. Using Simplicity to Control Complexity. *IEEE Software*. July/August, 2001. pp.20-28

[S98] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley. 1998.

[SG96] M. Shaw, D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall. 1996.

[SR99] F. Salles, M. Rodriguez, J.-C. Fabre, J. Arlat. Metakernels and Fault Containment Wrappers. In *Proc. Fault Tolerant Computing Symposium (FTCS-29)*, Wisconsin, USA, 1999, 22-29.

[SI99] T. Saridakis, V. Issarny. Developing Dependable Systems using Software Architecture. In *Proceedings of the 1st Working IFIP Conference on Software Architecture*, pages 83--104, February 1999.

[S01] D. Sotirovski. Towards fault-tolerant software architectures. In R. Kazman, P. Kruchten, C. Verhoef, H. Van Vliet, editors, *Working IEEE/IFIP Conference on Software Architecture*, pages 7--13, Los Alamitos, CA, 2001.

[S98] V. Stavridou, R. A. Riemenschneider. Provably dependable software architectures. In *Proceedings of the Third ACM SIGPLAN International Software Architecture Workshop*, pages 133--136. ACM, 1998.

[TMA+96] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead Jr., J. E. Robbins, K. A. Nies, P. Oreizy, D. L. Dubrow. A component- and message-based architectural style for GUI software. *IEEE Transactions on Software Engineering*, 22(6):390--406, June 1996.

[UCI02] UCI. Archstudio 3 - Foundations - c2.fw, http://www.isr.uci.edu/projects/archstudio/c2fw.html, accessed November, 2002.

[V98] J. M. Voas, The Challenges of Using COTS Software in Component-Based Development. *IEEE Computer*, 31(6):44-45, June 1998.