# A Dependable Architecture for COTS-Based Software Systems using Protective Wrappers

Paulo Asterio de C. Guerra[1], Cecília Mary F. Rubira[1],
Alexander Romanovsky[2], Rogério de Lemos[3]

[1]Instituto de Computação Universidade Estadual de Campinas, Brazil
{asterio,cmrubira}@ic.unicamp.br
[2]School of Computing Science, University of Newcastle upon Tyne, UK
alexander.romanovsky@ncl.ac.uk
[3]Computing Laboratory, University of Kent at Canterbury, UK
r.delemos@ukc.ac.uk

**Abstract.** Commercial off-the-shelf (COTS) software components are built to be used as black boxes that cannot be modified. The specific context in which these COTS components are employed is not known to their developers. When integrating such COTS components into systems, which have high dependability requirements, there may be mismatches between the failure assumptions of these components and the rest of the system. For resolving these mismatches, system integrators must rely on techniques that are external to the COTS software 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.

## 1 Introduction

A commercial off-the-shelf (COTS) software component is usually provided as a black box to be reused "as it is". Most of the time these components do not have a rigorously written specification, hence there is no guarantee that their description is correct (very often, it is ambiguous and incomplete). Moreover, these components may have faults, and the specific context in which they will be used is not known during their development. Once they are created, they can evolve over time through different versions. When an integrator builds a system out of COTS components, she/he can be forced to evolve the system whenever a new version of these COTS components is released. These new versions can be sources of new faults. When integrating such components into a system, solutions for meeting its overall dependability requirements should be envisaged at the architectural level, independently of the particular COTS versions. These solutions should ensure that the system delivers the service despite the presence of faults in the COTS component and how it interacts with other system components. In this paper, we focus on COTS software components that are integrated in a system at the application level and provide their services to other components and, possibly, use services provided by

them. We assume that these application level software components are deployed in a reliable runtime environment that may include other COTS software components at the infrastructure level, such as operating systems, distribution middleware and component frameworks.

Research into describing software architectures with respect to their dependability properties has recently gained considerable attention [20,24,25]. In this paper, we focus on the architectural description of fault-tolerant component-based systems that provides an effective and systematic way for building dependable software systems from COTS software components. For that, we combine the concepts of an idealised architectural component [8], which is based on the idealised fault-tolerant component [2], and protective wrappers [15], known to be the most general approach to developing dependable software systems based on COTS components. While in previous work we have described the basis of the proposed approach [9,10], in this paper we elaborate on that work by discussing guidelines for specifying and implementing protective wrappers and by demonstrating our ideas using a case study.

The rest of the paper is organised as follows. In the next section, we briefly discuss background work on architectural mismatches, wrapper protectors, and the C2 architectural style. Section 3 describes the architectural representation of idealised fault-tolerant COTS, in terms of the idealised C2 component (iC2C), the idealised C2 COTS component (iCOTS), and the process of architecting fault-tolerant systems using iCOTS components. The case study demonstrating the feasibility of the proposed approach is presented in section 4. Related work on how to build dependable software systems based on COTS components is discussed in section 5. Finally, section 6 presents some concluding remarks and discusses future work.

## 2 Background

When integrating COTS components into a software system, the architect needs to develop glue code [18] that links various components together and includes new architectural elements, or *adaptors*, to resolve the different kinds of incompatibilities that may exist. A protector is a special kind of adaptor that deals with incompatibilities in the failure assumptions.

### 2.1 Architectural Mismatches and COTS Component Integration

Dealing with architectural mismatches [7] is one of the most difficult problems system integrators face when developing systems from COTS components. An *architectural mismatch* occurs when the assumptions that a component makes about another component or the rest of the system (ROS) 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 [13]. When building systems from existing components, it is inevitable that incompatibilities between the service delivered by the component and the service that the ROS 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 dependability attributes related, for example, to the component failure mode assumptions or its safety integrity levels.

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

### 2.2 COTS Component Protectors

Component wrapping is a well-known structuring technique that has been used in several areas. In this paper, we use the term "wrapper" in a very broad sense, incorporating the concepts of wrappers, mediators, and bridges [6]. 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) 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 properties such as security and transparent component replication.

A systematic approach has been proposed for using protective wrappers, known as *protectors*, that can improve the overall system dependability [15]. This is achieved by protecting both the system against erroneous behaviour of a COTS component, and the COTS component against erroneous behaviour of the rest of the system (ROS). As a protector has this dual role we call the interface between the COTS and the ROS the *protected interface*. The protectors are viewed as redundant software that detects errors or suspicious activity on a protected interface and executes appropriate recovery.

The development of protectors occurs during the assembly stage of the development process of a COTS-based system, as part of the system integration activities [15]. The approach consists of rigorous specification of the protector functionality, in terms of error detection and associated recovery actions, and in their integration into the software architecture. The protector error detection capabilities are specified in the form of *acceptable behaviour constraints* (ABCs) that ensure the normal behaviour of the protected interface. The protector recovery actions are specified in the form of exception handlers associated with the erroneous conditions that may arise in the protected interface. The protector specification is based on a set of *blueprints* and *safety specifications* that are produced during the earlier stages of the development process. A blueprint is a documented entity that specifies the overall architecture and external behaviour of a piece of software [3]. Safety specifications are derived from the system's safety requirements [5], which focus on reducing the risk associated with hazards and on limiting damage when an accident occurs. 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:

1. The behaviour specification of COTS components as specified by the COTS's developers. This specification is materialized in the form of a *COTS blueprint* that is provided to the system designers as part of the COTS documentation.
2. The behaviour specification of a COTS component as specified by the system designers. This specification is materialized in the form of a *component blueprint* that is produced by the system designers during the specification phase of the system's development process. The component blueprint and the COTS blueprint 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.
3. The description of the actual behaviour that the system designer expects from a COTS component (not necessarily approving it) based on previous experiences, i.e., he/she may know that it often fails in response to certain legal stimuli. The system designers describe this behaviour in an *annotated COTS blueprint*.
4. The behaviour specified for the ROS. This specification is materialized in a *system blueprint*.
5. The behaviour specification of the undesirable behaviour, especially unacceptable, of the component and the rest of the system, respectively, the *component safety specifications* and the *system safety specifications*. The system designer produces these during the specification stage of the development process.

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 ROS). The statements are expressed as a set of assertions on the states of input and output parameters. In addition to that, they may include assertions on the histories (sequences of calls) and assertions on the states of the system components.

### 2.3 The C2 Architectural Style

The C2 architectural style is a component-based style that supports large grain reuse and flexible system composition, emphasizing weak bindings between components [26]. In this style, components of a system are 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 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.

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.

While in this section we have introduced a background on protectors and iC2C, in the next section we propose an architectural solution for turning COTS components into idealised fault-tolerant COTS components (iCOTS) by adding protective wrappers to them. Although in previous work we introduced the iCOTS concept [9, 10], in this paper we provide a detailed description of the iCOTS concept, a systematic description of the engineering steps to be used when applying the proposed solution, and a description of a case study used to evaluate this solution.

## 3 Idealised Fault-Tolerant COTS Component

Modern 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 their internal states 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) [8] that 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 [2]; it was proposed to allow structuring of software architectures compliant with the C2 architectural style [26]. The C2 style was chosen for its orientation towards independent components that do not communicate directly. This makes it easier for the system developers to isolate critical components from the ROS.

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 to be 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 has lead us to developing an overall structure for the iC2C that has two distinct components and three connectors, as shown in Figure 1.

The iC2C NormalActivity component implements the normal behaviour, and is responsible for error detection during normal operation and for signalling the interface and internal exceptions. The iC2C AbnormalActivity component is responsible for error recovery and for signalling the failure exceptions. For consistency, the signalling of an internal exception by an idealised fault-tolerant component is viewed as a subtype of notification, and, the "return to normal", flowing in the opposite direction, is viewed as a request. During error recovery, the AbnormalActivity component may also emit requests and receive notifications, which are not shown in Figure 1.

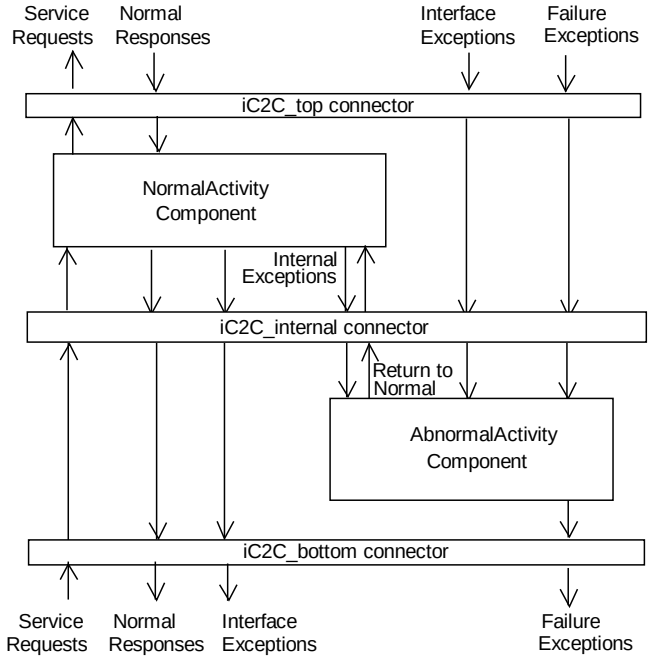


**Fig. 1.** Idealised C2 Component (iC2C)

The iC2C connectors are specialized reusable C2 connectors, which have the following roles:

1. 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, it 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.
2. 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;

3. The iC2C_top connector connects the iC2C with the upper components of a C2 configuration.

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 to 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.

### 3.2 Guidelines for Turning COTS into iC2C

In this section we show how the development of protectors can be included in the development process of a COTS-based software system. Typically, a COTS-based development process can be divided into six stages [4]: requirements, specification, provisioning, assembly (or integration), test and deployment. The requirements stage aims to identify the system's requirements. During the specification stage the system is decomposed in a set of components with specific responsibilities that interact to fulfil the system requirements. These components are instantiated during the provisioning stage. During this stage the system integrator decides if a component can be instantiated by an existing 'off-the-shelf' component, herein called a *COTS instance,* or if it will require an implementation effort, in which case it is called an *in-house instance*. During the assembly stage the system designer integrates COTS and in-house instances to build the whole system. This integration effort includes the development of glue code necessary to connect the various components, which include the specification and implementation of protectors. During the test stage the integrated system is tested and corrections may be made to ascertain that it fulfils its requirements and conforms to its specification. During the deployment stage the final system is installed in the user's environment.

The presented guidelines are applied to the provisioning and assembly stages of the development process. We assume that the following artefacts, as described in Section 2.2, have already been produced: (i) a system blueprint describing the initial software architecture and the system's safety specifications; and (ii) a set of component's blueprints specifying the components' interfaces and their safety specifications.

#### 3.2.1 Steps for the Provisioning Stage

*Step 1. Develop a basic test plan for the component*. This test plan should be based on the expected operational profile [11] of the component in the system being developed.

*Step 2. List candidate COTS components.* One should obtain a list of candidate COTS software components that could be used to instantiate the provided interfaces specified in the component's blueprint.

For each candidate COTS component, execute steps 3 to 6, as below.

*Step 3. Consolidate COTS blueprint.* One should obtain from the COTS vendor (or developer) the following information, which are consolidated in the COTS blueprint.

a) The specification of the COTS provided interfaces, which is commonly referred to as the COTS API (Application Programming Interface).
b) The specification of the COTS required interfaces, which is commonly referred as the COTS System Requirements.
c) Information about known design faults in the COTS, which is usually found in sections called 'Known Bugs and Problems'.
d) Any information that may give a "grey-box" view of the COTS, with selected details visible only [14]. Usually, this information may be found in technical articles and white papers from the COTS developers.

*Step 4. Test the COTS.* One should test the COTS instance applying the basic test plan previously developed. The results obtained from these tests should be documented with information about:

a) The subset of the COTS interfaces (provided and required) activated during the tests.
b) The input domains covered by the tests.
c) The erroneous conditions detected and the observed COTS behaviour under those conditions.
d) Discrepancies between the behaviour specified in the COTS blueprint and its actual observed behaviour.

*Step 5. Enhance test coverage.* One should revise the test plan and repeat the testing procedure until adequate test coverage is attained. Test coverage influences reliability, as higher test coverage is more likely to remove a greater number of software faults, leading to a lower failure rate and higher reliability [19]. The final tests should detect all known design faults in the COTS that can be activated under the component's expected operational profile. The test plan should also include test cases based on the "grey-box" view of the COTS.

*Step 6. Produce the annotated COTS blueprint.* The annotated COTS blueprint consolidates the information obtained about the COTS actual behaviour. This annotated COTS blueprint is based on the COTS blueprint, the system's safety specifications and the results of the final tests and should include:

a) Detailed specifications of the actual behaviour of the interfaces that were activated during the tests, under both normal and erroneous inputs.
b) Specification of potentially dangerous conditions associated with the interfaces that were not activated during the tests.
c) Additional information that may be available from previous use of the same COTS instance.

*Step 7. Select COTS instance.* If there are two or more candidate COTS instances being considered, select the one that fits best in the system. This selection is based on the information contained in the system blueprint and the various annotated COTS blueprints. For this selection, it may be necessary to develop alternate versions of the system blueprint adapted to limitations and requirements specific to each COTS instance. The result of this step is a revised system's blueprint with the version of the software architecture that includes the selected COTS instance wrapped by a protector (to be developed during the assembly stage).

*Step 8. Decide COTS integration.* At this point, it should be decided between the system integration using the selected COTS instance or, alternatively, using a new component to be developed in-house.

### 3.2.2 Steps for the Assembly Stage

*Step 9. Classify erroneous conditions.* One should define a set of generalised erroneous conditions that may arise in the protected interface. The erroneous (or dangerous) conditions specified in the annotated COTS blueprint (Step 6) are analyzed in view of the system's safety specification and classified according to how and in what extent they may affect the system's dependability requirements. For each resulting class it is defined a generalised exceptional condition.

*Step 10. Specify acceptable behaviour constraints (ABCs) associated to the erroneous conditions.* This specification is based on the information contained in the annotated COTS blueprint. The ABCs may include assertions on:

a) The domain of parameters and results of the requests that flow between the COTS instance and the ROS.
b) The history of messages exchanged through the interface.
c) Portions of the internal state of system's components that can be inspected by calling side-effect-free functions.

*Step 11. Specify the desired system's exceptional behaviour.* This exceptional behaviour defines the error recovery goals, which may depend on the type and severity of the errors detected. The main source of this specification is the system's safety specifications

*Step 12. Allocate error recovery responsibilities.* The system's exceptional behaviour specified in the preceding step is decomposed in a set of recovery actions assigned to specific components in the software architecture. Some of these responsibilities will be allocated to the protector associated to the selected COTS instance (Step 7). These recovery actions are also specified during this step.

*Step 13. Refine the software architecture.* This refinement decomposes the components involved with error processing (Step 11) into new architectural elements that will be responsible for error processing.

*Step 14. Implement error detectors.* The specified ABCs (Step 10) are implemented as the executable assertions encapsulated in two error detectors that act as message filters. The first error detector intercepts and monitors the service requests that flow from the ROS to the COTS and the corresponding results that flow back to the ROS. The second error detector intercepts and monitors the service requests that flow from the COTS to the ROS and the corresponding results that flow back to the COTS. The error detector intercepts these messages and verifies their associated assertions before delivering the message. When an ABC violation is detected the error detector raises an exception of a specific type associated with this ABC. The exception raised contains the pending message that, in this case, is not delivered to its recipient. Messages that do not cause an ABC violation are delivered to their recipients without change.

*Step 15. Implement error handlers.* The specified recovery actions (Step 12) are implemented in error handlers associated with the various exception types. These error handlers can be attached to the respective components of the architecture.

This placement depends on the scope of the recovery action, which may vary from a single component instance to the whole system.

*Step 16. Integrate the protectors.* During this step, the COTS instances are integrated with their associated error detectors (Step 14) and errors handlers (Step 15) as specified by the refined software architecture (Step 13). The result of this step is a set of COTS instances in the form of iC2C.

*Step 17. Integrate the system.* During this step, the COTS instances are integrated with the in-house instances to produce the final system.

The integration of COTS instances in the form of iC2C into a C2 architectural configuration will be the topic of the next section. Such architectural configuration will contain iC2Cs for structuring in-house instances, and idealised C2 COTS component (iCOTS) for structuring COTS instances and their respective protective wrappers.

### 3.3 Idealised C2 COTS (iCOTS) Overall Architecture

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 [8]. 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 component (iCOTS) is a specialization of the iC2C that employs protective wrappers for encapsulating a COTS component. In our approach, the COTS component is connected to two specialized connectors acting as error detectors (Figure 2) to compose the NormalActivity component of the iCOTS. These detectors are responsible for verifying that the messages that flow 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. In the context of the overall diagram, the iC2C_bottom connector connects the iCOTS with the lower components of a C2 configuration, and serializes the requests received. The iC2C_internal connector controls message flow inside the iCOTS. The iC2C_top connector connects the iCOTS with the upper components of a C2 configuration.

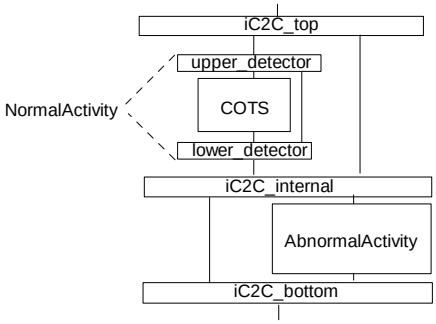


**Fig. 2.** Idealised C2 COTS (iCOTS) Overall Structure

When a constraint 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 that, in their turn, are wrapped by a pair of connectors. For example, Figure 3 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.

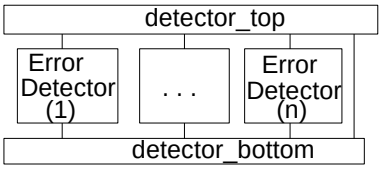


**Fig. 3.** Decomposition of a Detector

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 shown in Figure 4. In this design, the ErrorDiagnosis component is able 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.

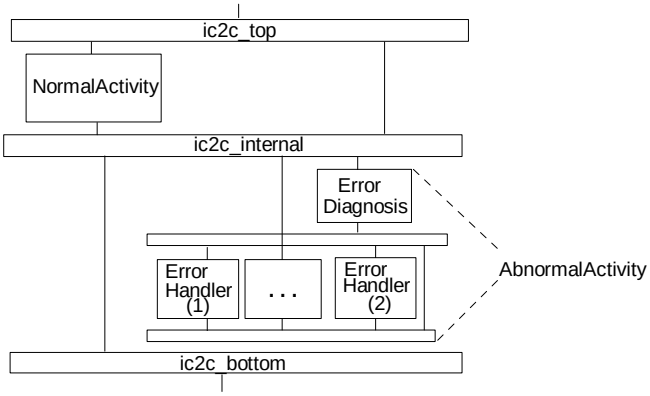


**Fig. 4** . Decomposition of the AbnormalActivity

## 4 Case Study

In this section, we present a case study that demonstrates the applicability of the iCOTS architectural solution when dealing with mismatches in the failure assumptions of COTS software components.

### 4.1 Problem Statement

Anderson et. al. [1] present the results of a case study in protective wrapper development [15], 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 (ROS). In turn, the control system consists of PID controllers, which are the COTS components, and the ROS consisting of:

1. 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.
2. 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.
3. 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 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), $O_2$ set-point (O2_ref), drum level (D_l), steam flow (S_f), steam pressure/drum (P_d), steam pressure/bus (P_b), $O_2$ concentration at economizer (O2eco), CO concentration at economizer (Coeco), and NOx concentration at economizer (Noxeco).

### 4.2 The Provisioning Stage

In this case study, we assume that the provisioning stage has been completed with the selection of a COTS PID Controller instance, as mentioned in Step 7 (Select COTS instance) of Section 3.2.1 (Steps for the Provisioning Stage). Anderson et. al. [1] summarise the available information describing the correct COTS component behaviour to be used in developing the protective wrappers. This document play the role of the annotated COTS blueprint mentioned in Step 6 (Produce the annotated COTS blueprint). Figure 5 shows the initial software architecture that is part of the system blueprint (Section 3.2). This architecture, which is based on the C2 architectural style, 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 5.

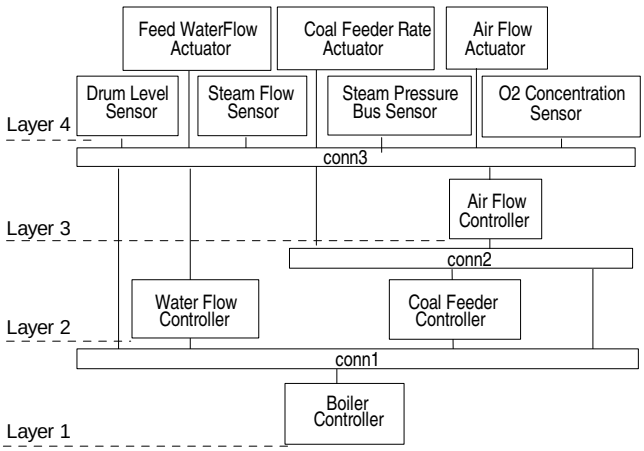


**Fig. 5.** C2 Configuration for the Boiler System

**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 | $O_2$ Concentration Sensor |
| setFeedWaterFlow(F_wf) | Feed Water Flow Actuator |
| setCoalFeedRate(C_fr) | Cool Feeder Rate Actuator<br>Air Flow Controller |
| setAirFlow(Air_f) | Air Flow Actuator |
| setConfiguration(P_ref, O2_ref) | Coal Feeder Controller<br>Air Flow Controller |

### 4.3 The Assembly Stage

The following paragraphs illustrate the assembly stage, starting from Step 9 of Section 3.2.2 (Steps for the Assembly Stage).

*Step 9. Define a set of generalised erroneous conditions that may arise in the protected interface.* Three types of such erroneous conditions are considered: (i) unavailability of inputs/outputs to/from the PID controllers; (ii) violation of specifications of monitored variables; and (iii) oscillations in monitored variables.

*Step 10. Specify acceptable behaviour constraints (ABCs) associated to the erroneous conditions.* These ABCs are summarized in the second column of Table 2 (ABC to be checked).

*Step 11. Specify the desired system's exceptional behaviour.* 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.

*Step 12. Allocate error recovery responsibilities.* The AirFlowController, WaterFlowController and CoalFeederController components are responsible for error detection (ABCs violations). The BoilerController component is responsible for error recovery, which may be either to sound an alarm or to shut down the system, depending on the exception type.

*Step 13. Refine the software architecture.* The proposed solution applies the concepts of iCOTS and iC2C for structuring four components. The AirFlowController, WaterFlowController and CoalFeederController components are structured as iCOTS components that encapsulate a COTS instance (the COTS PID controller) wrapped by a protector. The BoilerController component is structured as an iC2C, to be provided as an in-house instance. Next we describe how we can build an iCOTS AirFlowController encapsulating a COTS PID controller wrapped by a protector. This solution equally applies to the WaterFlowController and CoalFeederController components.

**Table 2.** Error Detection Specifications

| Message Type | ABC to be checked | Exceptional Notification |
|---|---|---|
| **Lower Detector** | | |
| Request setConfiguration (P_ref, O2_ref) | 0 <= O2_ref <= 0.1 | InvalidConfigurationSetpoint |
| | 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 |
| | 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 |
| | 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. | |

Figure 6 shows the internal structure of the iCOTS for the AirFlowController, based on the patterns shown in Figures 2 and 3. This solution equally applies to the WaterFlowController and CoalFeederController components.

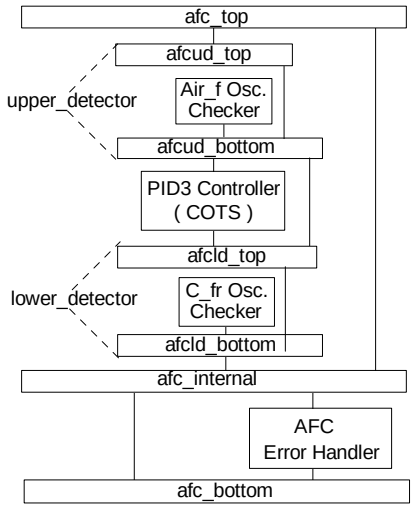


**Fig. 6.** Decomposition of the AirFlowController

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 2.2), and the type of the exception notification that should be generated when a constraint 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 exception types are internal exceptions, to be handled by the AFCErrorHandler. Thus, the AFCErrorHandler propagates internal exceptions 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.

The BoilerController component is responsible for:

1. Configuring the boiler system, sending setConfiguration requests when appropriate.
2. Handling interface exceptions of type InvalidConfigurationSetpoint, which may be raised in response of a setConfiguration request.
3. Handling failure exceptions of type NoResponse, OutOfRange or Oscillation, which may be raised by the three controllers (WaterFlowController, CoalFeederController, AirFlowController).

The BoilerController component was structured as an iC2C to cope with fault-tolerance responsibilities, which are captured by items (2) and (3) above, in addition to its main functional responsibility, which is captured by item (1) above.

Figure 7 shows the resulting fault-tolerant architecture for this system, which is derived from the overall architectural configuration for the boiler system (Figure 5).

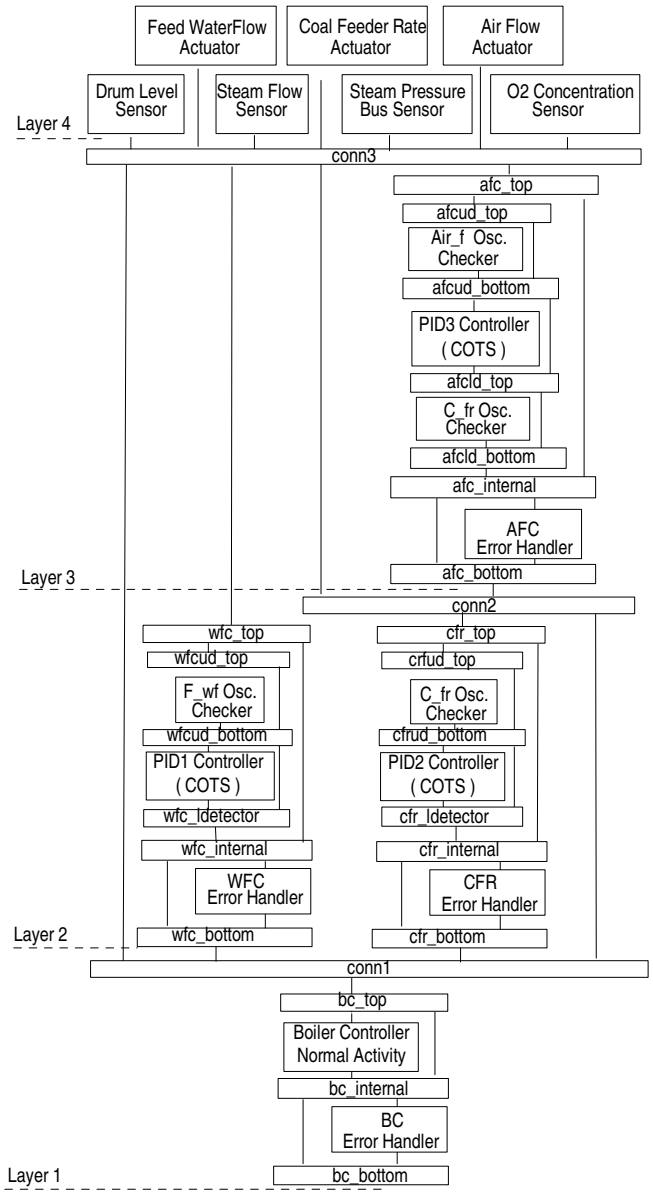


**Fig. 7.** Resulting Configuration for the Boiler System

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, do not fail. Figure 8 illustrates the flow of

messages between the various components involved when a PIDTimeout exception occurs, after the BoilerController fails to configure the AirFlowController, which contains the COTS PID3Controller. When the AirFlowController bottom connector (afcId_bottom) detects that the AirFlowController is not responding, it raises an exception to AFCErrorHandler. Since AFCErrorHandler cannot cope with this exception type, it raises another exception to the BCErrorHandler that shuts down the whole system.

*Step 14. Implement error detectors.* During this and the following steps it was used an objected-oriented framework that provides a generic implementation for the key abstractions of the iC2C and iCOTS structuring concepts. Using this framework, the iCOTS lower and upper detectors are implemented as subclasses of, respectively, LowerDetectorAbst and UpperDetectorAbst. Figure 9 shows the main parts of the AfcLowerDetector class that implements the lower error detector associated with the AirFlowController component. In this code sample, the setConfiguration() and setCoalFeedRate() methods intercept the requests sent to the AirFlowController and check their associated ABCs, based on the specification shown in Table 2. When an assertion is violated an exception is raised by means of the raiseException() method, which is implemented by the abstract classes. Accepted requests are delivered to the AirFlowController that is connected to the wrappedNormal interface.

*Step 15. Implement error handlers.* Using the framework aforementioneted, the error handlers are implented as subclasses of AbnormalActivityAbst. Figure 10 shows the AFCErrorHandler class that implements the error handler associated with the AirFlowController. In this code sample, the handle() method is called when a failure exception is raised by one of the error detectors that wrap the AirFlowController. The exceptions raised are re-signalled with a more generic exception type according to the specifications shown in Table 3.

*Step 16. Integrate the protectors.* The following code snippet creates a component afc that encapsulates the AirFlowController and its associated error detectors and error handler, according to the iCOTS structuring. This new component is an iC2C instance composed by the basic AirFlowController wrapped by the two error detectors, acting as the iC2C normal activity component, and the error handler acting as the iC2C abnormal activity component.

```
Icomponent afc=new iC2C( new AfcWrappedNormal
                              ( new AirFlowController(),
                                new AfcLowerDetector(),
                                new AfcUpperDetector() ),
                             new AFCErrorHandler() );
```

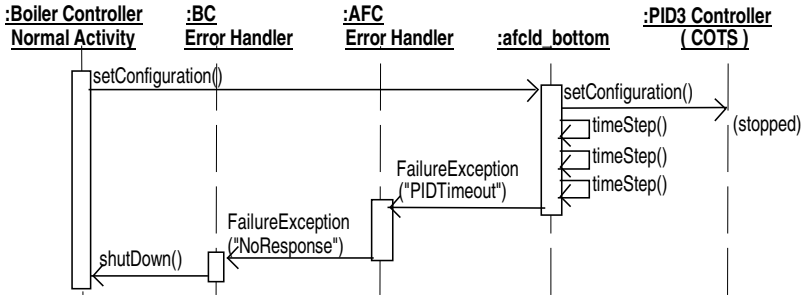


**Fig. 8.** UML Sequence Diagram for a PIDTimeout Exception

*Step 17. Integrate the system.* The integration of the various components and connectors that are composed into the system is coded in the main() method of a StartUp class, based on the configuration shown in Figure7. The code snippet bellow illustrates this method body with: (i) the instantiation of component bc as an iC2C composed by the BoilerController component and its associated error handler; (ii) the instantiation of connector conn1; and (iii) the connection between the top interface of component bc and the bottom interface of connector conn1.

```
IComponent bc=new iC2C
       (new BoilerController(), new BCErrorHandler());
IConnector c1=new Conn1();
bc.connectTop(c1);
```

The resulting system was tested in a simulated environment that allowed us to inject different faults in the system. The final system passed all tests successfully. The system behaved as specified even in the presence of faults, aborting the erroneous operations and either stopping the system or activating an alarm.

A limitation of the case study is that it is based on a simulation of a boiler system, which does not allow an objective performance analysis. During execution time, the main overhead associated to a protector occurs when a service requested passes through a protected interface. This overhead is proportional to the number and complexity of the assertions encapsulated in the protectors. Assuming that this complexity should be much lower than the complexity of the services provided by the COTS, we may infer that the performance impact of the protective wrappers will be low. An ongoing experimental work in the DOTS[1] project is confirming this. The protector associated with the air flow controller, comprising its two error detectors with the eight ABCs and the associated error handler, required about a hundred lines of code. This additional code is added to the system in three new classes, which does not require any changes in the class that implements the base AirFlowController. An ongoing work is applying the proposed approach in a real world system and to more complex COTS software components.

[1] Diversity with Off-The-Shelf Components Project, http://www.csr.ncl.ac.uk/dots/

```
public class AfcLowerDetector extends LowerDetectorAbst
  implements IConnector, IAirFlowController {
  private IAirFlowController wrappedNormal;
  private OscillatorChecker oscillatorChecker;
  public void setConfiguration
              (double P_ref, double O2_ref) {
    if (O2_ref<0 || O2_ref>0.1)
      raiseException
        (new InvalidConfigurationSetpoint(O2_ref));
    try {wrappedNormal.setConfiguration(P_ref, O2_ref);
    } catch (TimeoutException e) {
      raiseException
         (new PIDTimeout("setConfiguration()"));
    } catch (AbortException e) { aborted(); }
  }
  public void setCoalFeedRate(double C_fr) {
    if (C_fr<0 || C_fr>1)
      raiseException(new InvalidCoalFeederRate(C_fr));
      if (oscillatorChecker.check_oscillate(C_fr))
        raiseException
          (new CoalFeederRateOscillating(C_fr));
      try {
        wrappedNormal.setCoalFeedRate(C_fr);
      } catch (TimeoutException e) {
        raiseException
          (new PIDTimeout("setCoalFeedRate()"));
      } catch (AbortException e) { aborted(); }
    } …
```

**Fig. 9.** Implementation of the AirFlowController's lower detector.

```
public class AFCErrorHandler
  extends AbnormalActivityAbst implements IComponent {
  public void handle(Exception exception) {
    try { throw exception; }
      catch (PIDTimeout e) {
      throw new NoResponse(e);
    } catch (AirFlowActuatorTimeout e) {
      throw new NoResponse(e);
    } catch (InvalidO2Concentration e) {
      throw new OutOfRange(e);
    } catch (InvalidAirFlowRate e) {
      throw new OutOfRange(e);
    } catch (CoalFeederRateOscillating e) {
      throw new Oscillation(e);
    } catch (AirFlowRateOscillating e) {
      throw new Oscillation(e);
    } catch (Exception e) {
      throw new FailureException(e); }
  }
```

**Fig. 10.** Implementation of the AirFlowController's error handler.

## 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).

Ballista [12] 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 [23]. 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 [16] 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 do not offer 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 [21]) 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. [17] offer a software connector-based approach to increasing the dependability of systems with components that evolve over time. The idea is to employ the new and the (several if available) old versions of a component to improve the overall system dependability. The authors put forward the idea of using a specialised multi-version connector allowing the system architect to specify the component authority for different operations: a version 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 ROS 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. This 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 dependable systems from existing components, guarantees cannot be given on the system behaviour, if at least guarantees are not provided on the behaviour of its individual components. Since such guarantees are difficult to be obtained for individual COTS components, architectural means have to be devised for the provision of the necessary guarantees at the system level. The paper proposes an architectural solution to transform 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 associated with 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.

The protective wrappers are integrated in the architectural configuration as a set of new architectural elements that are dependent of a runtime environment, which is assumed to be reliable. The proposed approach also does not consider direct interactions between the COTS software component and human users. This implies that the proposed approach may not be effective for protecting COTS software components that either: (i) provide infrastructure services, such as operating systems, distribution middleware and component frameworks; or (ii) interacts intensively with human users, such as word processors.

The effectiveness of the proposed approach depends on the system designer’s ability to anticipate the COTS behaviour when integrating it into a specific system, e.g. using the COTS usage profile. An approach for anticipating undesirable behaviour and, thus, increasing the protector’s coverage factor for error detection is to perform integration tests of the COTS within the system being developed, prior to the specification of the protectors. These testing activities are the harder aspect in the development of the protectors, and they cannot be avoided. Our future work includes evaluating tools for automating these tests, which could be integrated in the protector’s development process.

Although a single architectural style was used in the case study, software components in the C2 architectural style can be nevertheless integrated into configurations of other architectural styles of the independent components family [22], such as client/server and broker styles. This allows the idealised fault tolerant COTS (iCOTS) concept to be applied as a general solution in composing dependable systems from unreliable COTS components.

## Acknowledgments

Paulo Asterio de C. Guerra is partially supported by CAPES/Brazil. Cecília Mary F. Rubira and Paulo Asterio de C. Guerra are supported by the FINEP/Brazil "Advanced Information Systems" Project (PRONEX-SAI-7697102200). Cecília Mary F. Rubira is also supported by CNPq/Brazil under grant no. 351592/97-0. Alexander Romanovsky is supported by EPSRC/UK DOTS and IST/FW6 RODIN Projects.

Special thanks go to Professor Brian Randell for many useful insights and suggestions.

## References

1. T. Anderson, M. Feng, S. Riddle, A. Romanovsky. Protective Wrapper Development: A Case Study. In *Proc. 2nd Int. Conference on COTS-based Software Systems (ICCBSS 2003)*. Ottawa, Canada. Feb., 2003 M.H. Erdogmus, T. Weng (eds.). Lecture Notes in Computer Science Volume 2580 pp. 1-14 Springer-Verlag 2003,
2. T. Anderson, P. A. Lee. *Fault Tolerance: Principles and Practice*. Prentice-Hall, 1981.
3. S. Van Baelen, D. Urting, W. Van Belle, V. Jonckers, T. Holvoet, Y. Berbers, K. De Vlaminck. Toward a unified terminology for component-based development. WCOP Workshop, ECOOP 2000. Cannes, France. Available at: http://www.dess-itea.org/publications/ECOOP2000-WCOP-KULeuven.pdf
4. J. Chessman, J. Daniels. *UML Components: A Simple Process for Specifying Component-Based Software*. Addison-Wesley. 2001.
5. R. De Lemos, A. Saeed, T. Anderson. Analyzing Safety Requirements for Process-Control Systems. *IEEE Software*, Vol. 12, No. 3, May 1995, pp. 42--53.
6. R. DeLine. "A Catalog of Techniques for Resolving Packaging Mismatch". *Proc. 5th Symposium on Software Reusability (SSR'99)*. Los Angeles, CA. May 1999. pp. 44-53.
7. D. Garlan, R. Allen, J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17--26, November 1995.
8. P. A.C. Guerra, C. M. F. Rubira, R. de Lemos. An Idealized Fault-Tolerant Architectural Component, in *Architecting Dependable Systems*. Springer-Verlag, Lecture Notes in Computer Science (LNCS). May, 2003. pp. 21-41.
9. P. A.C. Guerra, C. M. F. Rubira, A. Romanovsky, R. de Lemos. Integrating COTS Software Components Into Dependable Software Architectures. In. Proc. *6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*. Hokkaido, Japan, 2003, pp. 139-142.
10. P. A.C. Guerra, C. M. F. Rubira, A. Romanovsky, R. de Lemos. A Fault-Tolerant Software Architecture for COTS-based Software Systems. In. Proc. *9th European Software Engineering Conference* held jointly with *10th ACM SIGSOFT International*

*Symposium on Foundations of Software Engineering*. Helsinki, Finland, 2003, pp. 375-378.
11. D. Hamlet, D. Mason, D. Woit. Theory of System Reliability Based on Components. In Proc. *2000 International Workshop on Component-Based Software Engineering*. CMU/SEI. 2000.
12. 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.
13. 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.
14. F. Plasil and S. Visnovsky. Behavior Protocols for Software Components. In IEEE Transactions on Software Engineering, 28,11. 2002. pp. 1056-1076.
15. P. Popov, S. Riddle, A. Romanovsky, L. Strigini. On Systematic Design of Protectors for Employing OTS Items. In *Proc. 27th Euromicro Conference*. Warsaw, Poland, 4-6 September, IEEE CS, 2001. pp.22-29.
16. 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)
17. M. Rakic, N. Medvidovic. Increasing the Confidence in Off-The-Shelf Components: A Software Connector-Based Approach. *Proc. 2001 Symposium on Software Reusability (SSR'01). ACM/SIGSOFT Software Engineering Notes*, 26,3. 2001. pp. 11-18.
18. J.-G. Schneider, O. Nierstrasz. *Components, Scripts and Glue*. In: L. Barroca, J. Hall, P. Hall (Eds.) Software Architecture Advances and Applications. Springer-Verlag. 2000. pp. 13-25.
19. S. Sedigh-Ali, A. Ghafoor, R. A. Paul. Metrics and Models for Cost and Quality of Component-Based Software. In. Proc. *6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*. Hokkaido, Japan, 2003.
20. 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.
21. L. Sha. Using Simplicity to Control Complexity. *IEEE Software*. July/August, 2001. pp.20-28
22. M. Shaw, P. C. Clements. A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. In. Proc. *21st International Computer Software and Applications Conference*. 1997. pp. 6-13.
23. 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.
24. T. Saridakis, V. Issarny. Developing Dependable Systems using Software Architecture. *Proc. 1st Working IFIP Conference on Software Architecture*, pages 83--104, February 1999.
25. V. Stavridou, R. A. Riemenschneider. Provably Dependable Software Architectures. *Proc. Third ACM SIGPLAN International Software Architecture Workshop*, pages 133--136. ACM, 1998.
26. 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.