FaTC2: An Object-Oriented Framework for Developing Fault-Tolerant Component-Based Systems

Fernando J. Castor de Lima Filho Paulo Asterio de C. Guerra
Cecília Mary F. Rubira
Instituto de Computação
Universidade Estadual de Campinas, Brazil
{fernando.lima, asterio, cmrubira}@ic.unicamp.br

Abstract

Component-based systems built out of reusable software components are being used in a wide range of applications that have high dependability requirements. In order to achieve the required levels of reliability and availability, it is necessary to incorporate into these complex systems means for coping with software faults. In this paper we present FaTC2, an object-oriented framework which facilitates the construction of fault-tolerant component-based systems by giving support to fault tolerance techniques. FaTC2 is an extension of C2.FW, an OO framework which provides an infrastructure for building applications using the C2 architectural style. More specifically, FaTC2 extends C2.FW in order to introduce a forward error recovery mechanism by means of an exception handling system. Our main contribution is to provide a framework which gives support to a software architectural level exception handling system. We also present a case study showing how our framework can be employed for building a fault-tolerant component-based application.

1. Introduction

Modern computing systems require evolving software that is built from existing software components, developed by independent sources[2]. Hence, the construction of systems with high dependability requirements out of software components represents a major challenge, since few assumptions can generally be made about the level of confidence of third party components. In this context, an architectural approach for fault tolerance is necessary in order to build dependable software systems assembled from untrustworthy components[8].

Fault tolerance at the architectural level is a young research area that has recently gained considerable attention[7]. Most of existing works in this area emphasize the creation of fault tolerance mechanisms[9, 11] and description of software architectures with respect to their dependability properties [12, 14].

The work of Guerra et al[6] presents a structuring concept for the incorporation of an exception handling mechanism in component-based systems, at the architectural level. This notion is based on the concept of the Idealised Fault-Tolerant Component(IFTC)[1]. The IFTC separates the abnormal (fault tolerance measures) activities of a system from its normal activity. Upon the receipt of a service request, an IFTC produces three types of responses: *normal responses* in case the request is successfully processed, *interface exceptions* in case the request is not valid, and failure exceptions, which are produced when a valid request is received but cannot be correctly processed.

In this paper we present an object-oriented framework, called FaTC2, for building fault-tolerant component-based systems based on the IFTC. Our framework is an extension of C2.FW[10], an OO framework which provides an infrastructure for building applications using the C2 architectural style[15]. FaTC2 introduces forward error recovery in the original framework by means of an exception handling system (EHS). An EHS offers control structures which allow developers to define actions that should be executed when an error is detected. This materializes by the capability to signal exceptions and, in the code of the handler, to put the system back in a coherent state. A forward error recovery mechanism manipulates the state of a system in order to remove errors and enable it to resume execution without failing. Forward error recovery is usually implemented by means of exception handling.

The C2 architectural style[10, 15] is a component-based architectural style which supports large grain reuse and flexible system composition, emphasizing weak bindings between components. The C2 style has been chosen due to its ability to compose heterogeneous off-the-shelf

components[10]. The work of Rakic and Medvidovic[11] is the only one we know of which describes means for supporting the construction of fault-tolerant C2 applications. It presents the concept of *Multi-Version Connector*, a mechanism created to permit the reliable upgrade of software components in a configuration, by means of design diversity[1].

Our main contribution is the construction of a framework which supports an architectural level EHS. In component-based development, source code for the components which make up a system might not be available, specially if third party components are employed. Hence, it is not possible to introduce exception handling directly in the component. An architectural level EHS deals with this kind of problem by providing an infrastructure for defining exceptions and attaching the corresponding handlers to components without the need to modify them.

The rest of this paper is organized as follows. Section 2 provides some background information. Section 3 presents the proposed framework, FaTC2, describing its most important elements. An example application is presented in Section 4. Final conclusions are given in Section 5.

2. Background

2.1. The C2 Architectural Style

In the C2 architectural style components communicate by exchanging asynchronous messages sent through connectors, which are responsible for the routing, filtering, and broadcast of messages. Figure 1 shows a Software Architecture using the C2 style where the elements A, B, and D are components, and C is a connector.

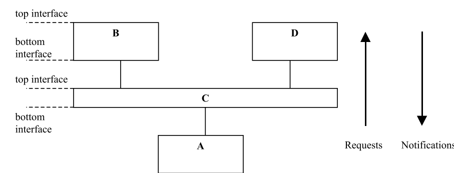

Figure 1. An example architecture using the C2 style.

Components and connectors have a *top interface* and a *bottom interface*(Figure 1). 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. Two types of messages are defined by the C2 style: requests, which are sent upwards through an architecture, and notifications, which are sent downwards. Requests ask components in upper layers of the architecture for some service to be provided, while notifications signal a change in the internal state of a component.

2.2. C2.FW Framework

The C2.FW framework[10] provides an infrastructure for building C2 applications. It is part of the ArchStudio[16] environment, which is an architecture-oriented integrated development environment which comprises a collection of tools to help in the development of applications based on the C2 style. C2.FW has been implemented in C++, Java, Python and Ada.

The C2.FW Java[5] framework comprises a set of classes and interfaces which implement the abstractions of the C2 style, such as components, connectors, messages, and interconnections. The framework provides various features, such as support to different threading models and queuing policies, and sophisticated message processing and event propagation mechanisms. It does not, however, implement any mechanisms for the provision of error recovery.

2.3. Idealised C2 Component

The work of Guerra et al[6] uses the concept of Idealised Fault-Tolerant Component (IFTC) to structure the architecture of component-based software systems compliant with the C2 architectural style. It introduces the Idealized C2 Component(iC2C), which is equivalent, in structure and behavior, to the IFTC. Service requests and normal responses of an IFTC are mapped as requests and notifications in the C2 architectural style. Interface and failure exceptions of an IFTC are considered subtypes of notifications.

The iC2C is composed of five elements: NormalActivity and AbnormalActivity components, and iC2C_top, iC2C_internal, and iC2C_bottom connectors. Its internal structure is presented in Figure 2.

The NormalActivity component processes service requests and answers them through notifications. It also implements the error detection mechanisms of the iC2C. The AbnormalActivity component encapsulates the exception handlers of the iC2C. While a system is in a normal state, the AbnormalActivity component remains inactive. When an exceptional condition is detected, it is activated to handle the exception. In case the exception is successfully handled, the system enters a normal state and the NormalActivity component resumes processing. Otherwise, a failure exception is sent and components in lower layers of the architecture become responsible for handling it.

The iC2C_bottom connector is responsible for filtering and serializing requests received by the iC2C. This conservative policy aims at guaranteeing that requests are always received by the NormalActivity component in its ini-

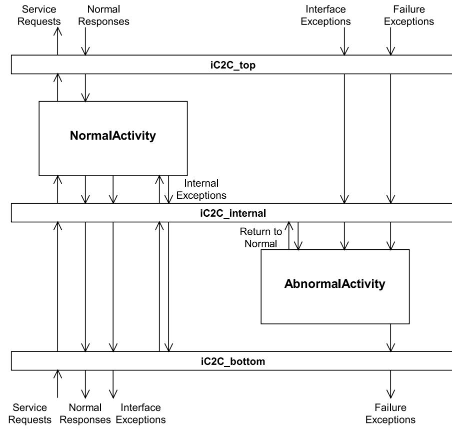

Figure 2. Internal structure of an iC2C.

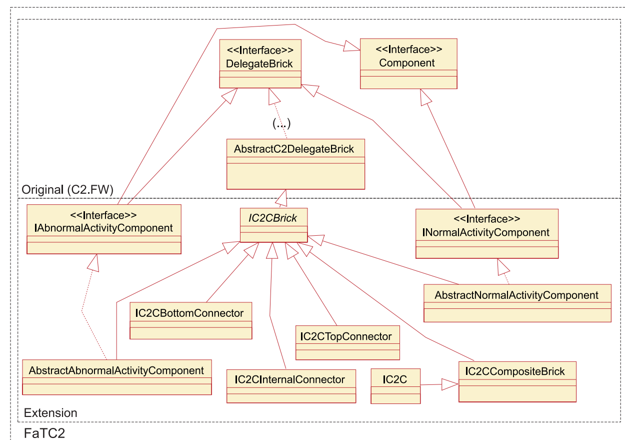

Figure 3. A summarized class hierarchy for C2.FW and FaTC2.

tial state, to avoid possible side-effects of an exceptional condition caused by a concurrent service request. The iC2C_internal connector is responsible for the routing of messages inside the iC2C. The destination of the messages sent by the internal elements of the iC2C depends on the message type and whether the iC2C is in a normal or abnormal state.

The iC2C_top connector encapsulates the interaction between the iC2C and components located in upper levels of the architecture. It is responsible for guaranteeing that service requests sent by the NormalActivity and AbnormalActivity components to other components located in upper levels of the architecture are processed synchronously. And that response notifications reach the intended destinations. The iC2C_top connector also performs domain translation, converting incoming notifications to a format which the iC2C understands and outgoing requests to a format which the application understands.

The structure of the iC2C makes it compatible with the constraints imposed by the C2 architectural style. Hence, an iC2C may be incorporated into an existing C2 configuration. Previous experiments[6, 8] with the IC2C model have shown its adequacy for the construction of component-based systems, including systems built from off-the-shelf components[7].

3. Description of the Framework

In order to facilitate the development of fault-tolerant applications using the C2 style, we have extended the Java[5] version of C2.FW with the concept of iC2C. The original C2.FW framework does not provide adequate support for the construction of fault-tolerant systems. Our aim is to provide the support for error recovery, more specifically, forward error recovery, by means of an EHS.

The extended C2.FW framework has been baptized *FaTC2*, which is an abbreviation for Fault-Tolerant C2. FaTC2 allows fault-tolerant systems to be built in a well-organized manner, using iC2Cs as structural units. The main advantage of this approach is the fact that framework users do not need to implement an EHS in order to create fault-tolerant applications. Only the normal activity(functional requirements) and abnormal activity(exception handling) of the component should be defined. Connections between normal and abnormal parts are managed by FaTC2.

Figure 3 presents a summarized class hierarchy for FaTC2, and its intersection with C2.FW. In the following sections we describe FaTC2, based on the elements which compose an iC2C(Figure 2).

3.1. IC2C

The creation of an iC2C is encapsulated by the **IC2C** class. Instances of **IC2C** are created by a factory method[3] which takes as arguments the name of the iC2C to be created and objects representing the NormalActivity and AbnormalActivity components(Figure 2). Optionally, it may also receive objects representing the iC2C_top and iC2C_bottom connectors as arguments, in case filtering or domain-translation are required. If these arguments are omitted, default implementations are employed.

Although the **IC2C** class may be used directly in an application, it is recommended that developers create subclasses of it, specifying the NormalActivity and AbnormalActivity components, and iC2C_top and iC2C_bottom connectors which are to be used.

3.2. NormalActivity Component

The NormalActivity component is one of the elements of the iC2C which must be implemented by developers employing FaTC2. In order to define a NormalActivity component, a developer must provide a class that implements the **INormalActivity** interface. This interface declares three operations which define the application-dependent behavior of the component: `handleRequest()`, `returnToNormal()`, and `reset()`. These operations must be implemented by the developer. The **AbstractNormalActivityComponent** abstract class should also be extended. This class implements the internal protocol of the iC2C, which is application-independent.

The `handleRequest()` method is responsible for processing service requests. It takes as argument a message corresponding to the request to be executed, and returns a response notification to be delivered to the client component. It is important to note that the framework provides the reusable code which actually sends the response notification and receives the service request. The code responsible for these tasks is implemented by **AbstractNormalActivityComponent**.

If an error occurs during the processing of a service request, an exception is thrown, which may be a failure exception (class **IC2CFailureException**) or an interface exception (class **IC2CInterfaceException**). These are caught by the framework and packaged as exception messages, which are sent to the AbnormalActivity component. It is important to note that the application code only throws language-specific exceptions. Architecture-level exceptions are managed by the framework itself.

In case the handling of a request demands the NormalActivity component to request services from components located in upper levels of the architecture, the **AbstractNormalActivityComponent** class provides a utility method, `requestService()`, which may be used to send synchronous(request/response) requests transparently, upwards the architecture.

The `returnToNormal()` and `reset()` methods are related to the abnormal activity of the iC2C. The former is called when the iC2C has successfully handled an exception, and should return to normal activity. The latter is called when the iC2C is unable handle an exception, and should return to its initial state so that the erroneous state does not affect subsequent requests.

3.3. AbnormalActivity Component

In order to implement an AbormalActivity component, a developer must provide a class that implements the **IAbnormalActivityComponent** interface and extends the **AbstractAbnormalActivityComponent** abstract class. Similarly to the NormalActivity component, the AbnormalActivity component has both application-dependent and application-independent behaviors. **AbstractAbnormalActivityComponent** implements the application-independent behavior of the AbnormalActivity component, while **IAbnormalActivityComponent** specifies the application-dependent behavior.

A single operation is defined by the **IAbnormalActivityComponent** interface: `handleException()`. This operation must be implemented by the developer and defines the exception handler of the iC2C. This operation takes the exception to be handled as argument. If an exception is successfully handled, `handleException()` returns a message object which is sent to the NormalActivity component. Processing is then resumed. Otherwise, an exception is thrown from the body of `handleException()`. This exception is caught by FaTC2 and a failure exception message is sent to the components in the lower levels of the architecture. In case the exception handler for a component which is in the lowest level of an architecture is unable to handle a given exception, it should notify an external user about this fact.

In case the handling of an exception requires the AbnormalActivityComponent to request services from other components, or from the NormalActivityComponent in the same iC2C, class **AbstractAbnormalActivityComponent** provides methods which allow synchronous requests to be carried transparently, similarly to the **AbstractNormalActivityComponent** class.

3.4. iC2C_top, iC2C_bottom and iC2C_internal Connectors

The **IC2CTopConnector**, **IC2CBottomConnector**, and **IC2CInternalConnector** classes are default implementations for the iC2C_top, iC2C_bottom, and iC2C_internal connectors, respectively.

IC2CTopConnector and **IC2CBottomConnector** may be extended in order to implement filtering of notifications in the top domain of an iC2C, or requests in its bottom domain, respectively. A filtering scheme is defined by implementing the `accept()` method in a subclass of **IC2CTopConnector** or **IC2CBottomConnector**. A message *m* is processed only if `accept(`*m*`) == true`.

Subclasses of **IC2CTopConnector** may also implement domain translation in the top domain of the iC2C. The methods `translateIncomingMessage()` and `processOutgoingMessage()` are responsible for this task and are called by FaTC2, respectively, immediately after a message has been *accepted* by the iC2C_top connector, and immediately before a given message is sent by it.

The iC2C_bottom connector is not expected to perform

domain translation. In the C2 architectural style, an element placed in an upper layer of an architecture should make no assumptions about elements in the lower layers[15].

In case no filtering or domain translation is necessary, the default implementations for the iC2C_top and iC2C_bottom connectors may be used.

The **IC2CInternalConnector** class is reused without needing any specialization, since its only task is to route messages inside an iC2C.

4. An Application Example

In order to show the usability of FaTC2, we present a small example extracted from the Mine Pump Control System[13]. The problem is to control the amount of water that collects at the mine sump, switching on a pump when the water level rises above a certain limit and switching it off when the water has been sufficiently reduced. In this section, we describe an implementation for the example application which uses the infrastructure provided by FaTC2.

4.1. Description of the Architecture

The C2 architecture of our example is shown in Figure 4. The **Pump** component commands the physical pump to be turned on/off. Component **LowWaterSensor** signals a notification when the water level is low. **WaterFlowSensor** checks whether water is flowing out of the sump. The **IdealPumpControlStation** component controls the draining of the sump by turning on/off the pump, according to the level of the water in the sump. It includes an exception handler which is executed when the pump is turned on but no water flow is detected. The error handler is implemented by the **AbnormalPumpControlStation** component. The **Pump**, **LowWaterSensor** and **WaterFlowSensor** components have been implemented as simple C2 components, while **IdealPumpControlStation** is an iC2C. In order to build the **IdealPumpControlStation**, five classes are implemented: **NormlPumpControlStation**, **AbnormalPumpControlStation**, **PumpControlStationTop**, **IdealPumpControlStation** and **TranslationConnector**.

Class **NormalPumpControlStation** implements the NormalActivity component of **IdealPumpControlStation**, that is, the methods defined by the **INormalActivityComponent** interface(Section 3.2). Due to the support provided by FaTC2, no messages need to be explicitly sent by any of the methods in **NormalPumpControlStation**; that is, the architect does not need to understand the internal protocol of the iC2C or the way it is implemented.

The **AbnormalPumpControlStation** class implements the exception handler of the **IdealPumpControlStation**. When an exception message is received by the

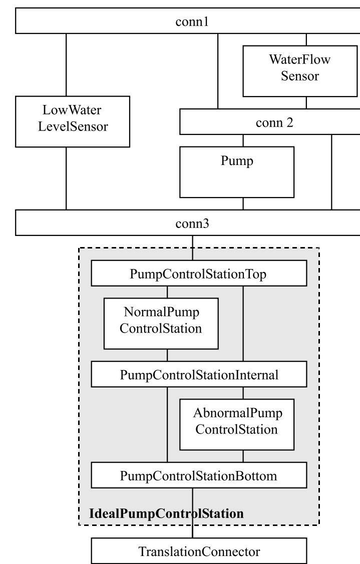

Figure 4. C2 configuration for the Fault Tolerant Mine Pump Control System.

`handleException()` method, the latter keeps sending new requests to **Pump** until either water flow is detected or the maximum number of retries permitted is reached. In the former case, normal activity is resumed(the method simply returns). In the latter, a failure exception message is sent downwards the architecture(the method throws an **IC2CFailureException**). The following code snippet partially illustrates this situation.

```
public Message handleException(Exception e)
 throws Exception {
 (...)
 if(this.retries >= this.MAX_RETRIES) {
   throw new IC2CFailureException(e);
 }
 (...)
```

In order to send an exception message downwards the architecture, the architect should throw a Java exception. In the example above, an exception of type **IC2CFailureException**, a subtype of **Exception**, is thrown.

The **PumpControlStationTop** class provides the **IdealPumpControlStation** component with an extension of the **IC2CTopConnector** class which performs filtering. When a request is issued by the **IdealPumpControlStation**, **PumpControlStationTop** records the type of the request sent, so that only a notification which is a response to that request is allowed to be processed. To build this filtering scheme, two

methods had to be implemented: `accept()` and `processOutgoingMessage()`(Section 3.4).

IdealPumpControlStation is a subclass of **IC2C**. The **IdealPumpControlStation** class defines a public constructor which takes as argument the name of the **IdealPumpControlStation** instance to be created. **TranslationConnector** translates requests and notifications at the bottom interface of the **IdealPumpControlStation** (Figure 4).

It is important to note that *no classes* other than the one which originally implemented the **PumpControlStation** component were modified. Working versions of FaTC2 and the example application can be downloaded at http://www.ic.unicamp.br/˜ra014861/FaTC2.

5. Conclusions

Component-based systems built out of reusable software components are being used in a wide range of applications that have high dependability requirements. In order to achieve the required levels of reliability and availability, it is necessary to incorporate into these complex systems means for coping with software faults. In component-based development, source code for the components which make up a system might not be available. This motivates the creation of architectural level fault tolerance mechanisms.

In this work, we have presented FaTC2, an object-oriented framework for the construction of fault-tolerant component-based systems. FaTC2 is an extension of C2.FW, a framework which provides an infrastructure for bulding applications in the C2 architectural style, but lacks support for the construction of fault-tolerant systems. FaTC2 extends C2.FW with a software architectural level exception handling system which is based on the concept of idealised C2 component. We have also presented an example demonstrating how to use FaTC2 to make a fault-tolerant system.

We plan to apply our framework to build a more complex case study where some off-the-shelf components are used. In order to meet this goal, it is necessary to expand the implementations of the NormalActivity and AbnormalActivity components of the iC2C, according to the models proposed by Guerra et al[7, 8], so as to deal with the architectural mismatches[4] which usually arise from the integration of COTS components.

Until the present moment, the iC2C has been modeled as a synchronous entity and the implementation of FaTC2 conforms to this model. That means that an iC2C is unable to handle asynchronous notifications and that requests are issued under the assumption that a response will be eventually received. This restriction might be undesirable for some applications, since a large amount of *glue code* may be necessary if a synchronous iC2C needs to interact with asynchronous components. Hence, another future work for FaTC2 is the implementation of an iC2C for which these restrictions are relaxed.

Finally, we also plan to construct a tool that facilitates the incorporation of exception handling into new and existing applications. We plan to integrate this tool with the ArchStudio environment.

References

[1] T. Anderson and P. A. Lee. *Fault Tolerance: Principles and Practice*. Prentice-Hall, 2nd edition, 1990.

[2] A. Brown and K. Wallnau. The current state of CBSE. *IEEE Software*, 15(5):37–46, September/October 1998.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Software Systems*. Addison-Wesley, 1995.

[4] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, 1994.

[5] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[6] P. Guerra, C. Rubira, and R. de Lemos. An idealized fault-tolerant architectural component. In *Proceedings of the 24th International Conference on Software Engineering - Workshop on Architecting Dependable Systems*, May 2002.

[7] P. Guerra, C. Rubira, A. Romanovsky, and R. de Lemos. Integrating COTS software componentes into dependable software architectures. In *Proceedings of the 6th ISORC. To Appear.* IEEE Computer Society Press, 2003.

[8] P. A. C. Guerra, C. M. F. Rubira, and R. de Lemos. *Architecting Dependable Systems*, chapter A Fault-Tolerant Architecture for Component-Based Software Systems. Lecture Notes in Computer Science. Springer-Verlag, 2003.

[9] V. Issarny and J. P. Banatre. Architecture-based exception handling. In *Proceedings of the 34th Annual Hawaii International Conference on System Sciences.* IEEE, 2001.

[10] N. Medvidovic, P. Oreizy, and R. N. Taylor. Reuse of off-the-shelf components in c2-style architectures. In *Proceedings of the 1997 Symposium on Software Reusability*, 1997.

[11] M. Rakic and N. Medvidovic. Increasing the confidence in off-the-shelf components: A software connector-based approach. In *Proceedings of the 2001 Symposium on Software Reusability*, pages 11–18. ACM/SIGSOFT, May 2001.

[12] T. Saridakis and V. Issarny. Fault-tolerant software architectures. Technical Report 3350, INRIA, February 1999.

[13] M. Sloman and J. Kramer. *Distributed Systems and Computer Networks*. Prentice Hall, 1987.

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

[15] R. N. Taylor, N. Medvidovic, K. Anderson, J. E. J. Whitehead, and J. Robbins. A component- and message- based architectural style for GUI software. In *Proceedings of the 17th International Conference on Software Engineering*, pages 295–304, April 1995.

[16] UCI. ArchStudio 3.0 homepage. http://www.isr.uci.edu/projects/archstudio.