An Idealized Fault-Tolerant Architectural Component

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

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

ABSTRACT

Component-based systems built from existing 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. However, the problem is exacerbated if we consider the current trend of integrating third-party software components, which allow neither code inspection nor changes. To leverage the reliability properties of these systems, we need solutions at the architectural level that are able to guide the structuring of unreliable components into a fault tolerant architecture. In this paper, we present an approach for structuring fault tolerant component-based systems based on the C2 architectural style.

1. INTRODUCTION

Modern computer systems require evolving software that is built from existing software components, developed by independent sources [6]. Instead of relying on traditional software assurance technology that has shown not to be effective for this kind of systems [24], alternative approaches have to be sought in order for obtaining trustworthy systems. One of these approaches is fault-tolerance, which is associated with the ability of a system to deliver services according with its specification in spite the presence of faults [12]. In this paper, we employ the concept of idealized fault tolerant component [1] for describing fault-tolerant component-based systems, at the architectural level.

For representing software systems at the architectural level, we have chosen the C2 architectural style for its ability to incorporate heterogeneous off-the-shelf components [15]. However, this ability of combining existing components is achieved through rules on topology and communication between the components (communication through broadcasting of asynchronous messages) that complicate the incorporation of fault-tolerance mechanisms into C2 software architectures, especially those mechanisms for error detection and fault containment [6, 9].

Research into describing software architectures with respect to their dependability properties has gained attention recently [17,20,21]. Nonetheless, rigorous specification of exception handling models and of exception propagation at the architecture level remains an open issue [11].

Particularly related to the architectural approach presented in this paper, there has been work on exception handling and software fault tolerance. The work on exception handling has focused on *configuration exceptions*, which are exceptional events that have to be handled at the configuration level of architectures [11]. In terms of software fault tolerance, the principles used for obtaining software diversity have also been employed in the reliable evolution of software systems, specifically, the upgrading of software components. While the core idea of the *Hercules framework* [8] is derived from concepts associated with *recovery blocks* [17], the notion of *multi-versioning connectors* (MVC) [16], in the context of C2 architectures, is derived from concepts associated with *N-version programming* [3]. The architectural approach presented in this paper is distinct from the work referred above since its focus is on structuring concepts to be applied in a broader class of exceptional conditions and fault-tolerance mechanisms. The aim is to structure, at the architecture level, fault-tolerant component-based systems that use off-the-shelf components. For that, we define an idealized C2 component with structure and behaviour equivalent to the idealized fault-tolerant component [1]. This idealized C2 component can then be used as a building block for a system of design patterns that implement the idealized fault-tolerant component for concurrent distributed systems [5].

The rest of this paper is structured as follows. Section 2 gives a brief overview of fault-tolerance and the C2 architectural style. Section 3 describes the proposed architectural solution of the idealized component, along with an small illustrative example. Final conclusions are given in section 4.

2. BACKGROUND

The capability of a system to tolerate faults is highly dependent on the software architecture [4]. Though, the structure of the system should allow fault tolerant mechanisms to operate in an orchestrated way with the system functions, without unnecessarily increasing the complexity of the system [17].

2.1. Fault Tolerance

The basic strategy to achieve fault tolerance in a system can be divided into two steps [13]. The first step, called *error processing*, is concerned with the system internal state, aiming to: detect errors that are caused by activation of faults, the diagnosis of the

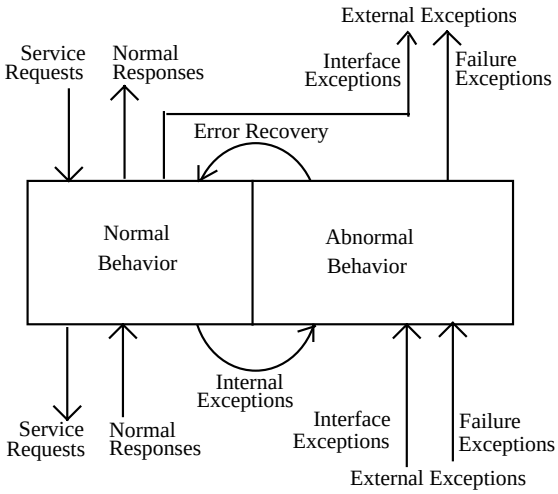

Figure 1. Idealized Fault-Tolerant Component

erroneous states, and recovery to error free states. The second step, called *fault treatment*, is concerned with the sources of faults that may affect the system and includes: fault localization, and fault removal.

Our work mainly concentrates on providing error processing at the architectural level of software systems. The *idealized fault-tolerant component* [1] is a structuring concept for the coherent provision of fault tolerance in a system (Figure 1). Through this concept, we can allocate fault-tolerance responsibilities to the various parts of a system in an orderly fashion, and model the system recursively, such that each: component can itself be considered as a system on its own, which has an internal design containing further sub-components [1].

The communication between idealized fault-tolerant components is only through request/response messages. Upon receiving a request for a service, an idealized component will react with a *normal response* if the request is successfully processed or an *external exception*, otherwise. This external exception may be due to an invalid service request, in which case it is called an *interface exception*, or due to a failure in processing a valid request, in which case it is called a *failure exception*. *Internal exceptions* are associated with errors detected within a component that may be corrected, allowing the operation to be completed successfully; otherwise, they are propagated as external exceptions.

An idealized component must provide appropriate handlers for all exceptions it may be exposed to. Thus, the internal structure of an idealized component has two distinct parts: one that implements its *normal behaviour*, when no exceptions occur, and another that implements its *abnormal behaviour*, which deals with the exceptional conditions. This separation of concerns, applied recursively to components, subsystems and the overall system, greatly simplifies the structuring of fault tolerance systems, allowing their complexity to be manageable.

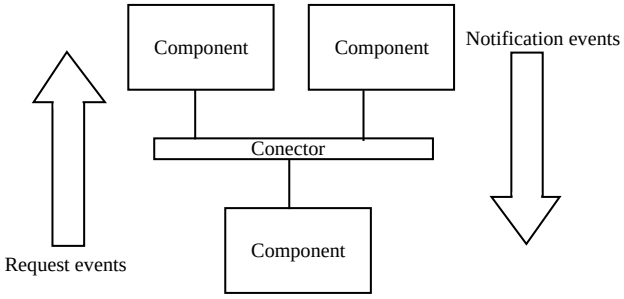

Figure 2. C2 style basic elements.

2.2. The C2 Architectural Style

The C2 architectural style is a component-based style directed at supporting large grain reuse and flexible system composition, emphasizing weak bindings between components [23]. In this style components of a system may be completely unaware of each other, as when one integrates various commercial off-the-shelf components (COTS), which may have heterogeneous style and implementation language. 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 using wrappers for encapsulating each component [9].

Both components and connectors in the C2 architectural style (Figure 2) 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's 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, as if a service was requested, with the *implicit invocation* of one of its operations.

3. PROPOSED ARCHITECTURE

3.1. Overall Structure of the Idealized C2 Component

The objective of this section is to define an idealized C2 component (iC2C), which should be equivalent, in terms of behaviour and structure, to the idealized fault-tolerant component (iFTC) [1]. The implementation of an iC2C should be able to use any C2 component without any restrictions. Furthermore, it should also be possible for integrating idealized C2 components into any C2 configurations, thus allowing the interaction of iC2Cs with other idealized and/or regular C2 components.

The first task was to extend the C2 message type hierarchy to allow for the various message types defined for the iFTC. This was a relatively simple task, since service requests and normal

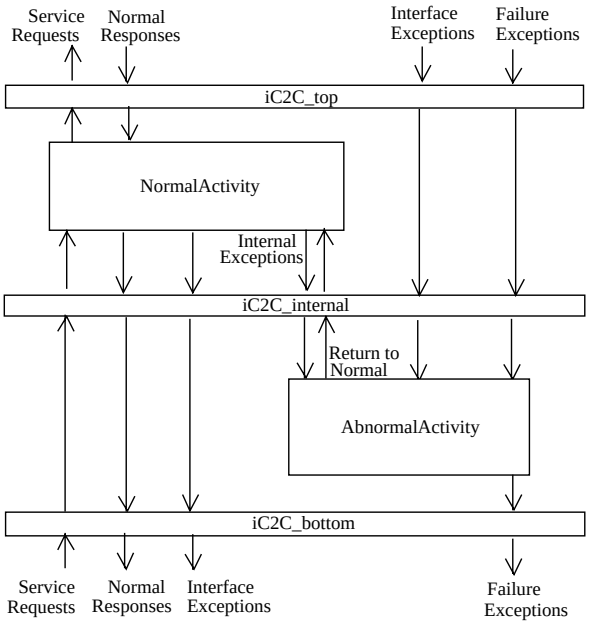

Figure 3. Idealized C2 Component (i2C2)

responses of an iFTC were directly mapped as requests and notifications in the C2 architecture. As interface and failure exceptions of an iFTC flow in the same direction as a normal response, they were considered subtypes of notifications in the C2 architecture.

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 idealized component. This outcome has lead to an overall structure for the iC2C that has two distinct components and three connectors, as shown in Figure 3.

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 iFTC was mapped as a subtype of notification, and, the "return to normal" , flowing in the opposite direction, was mapped as a request. In the course of error recovery, the AbnormalActivity component may also emit requests and receive notifications, which are not represented in Figure 3. More specifically, this design allows the AbnormalActivity component to be notified about state changes of the NormalActivity component and request operations which may change that state.

The connectors of our iC2C shown in Figure 3 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,

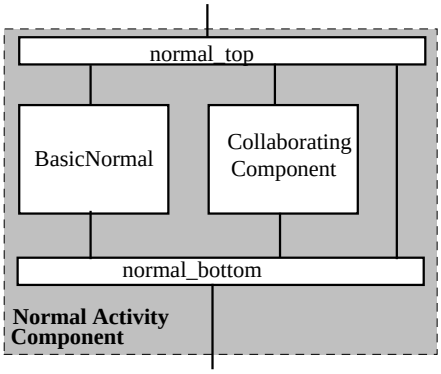

Figure 4. Normal Activity Component

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 idealized C2 component makes it fully compliant with the component's 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.

3.2. Structuring the Normal Activity Component

In this section, we describe in more detail how the NormalActivity component can be implemented from existing C2 components.

As previously mentioned, the NormalActivity component is responsible for the implementation of the normal behaviour of the idealized C2 component, and the detection of errors that may affect the normal behaviour. Since a NormalActivity component should be built from existing C2 components, and these components might not have error detection capabilities, there is the need to add error detection capabilities to the existing C2 component. The architectural solution for implementing a NormalActivity component is shown in Figure 4, for a particular configuration of two components. The existing C2 component, identified as the BasicNormal component, and any other component required for the provision of additional error detection capabilities, are wrapped by a pair of special-purpose connectors

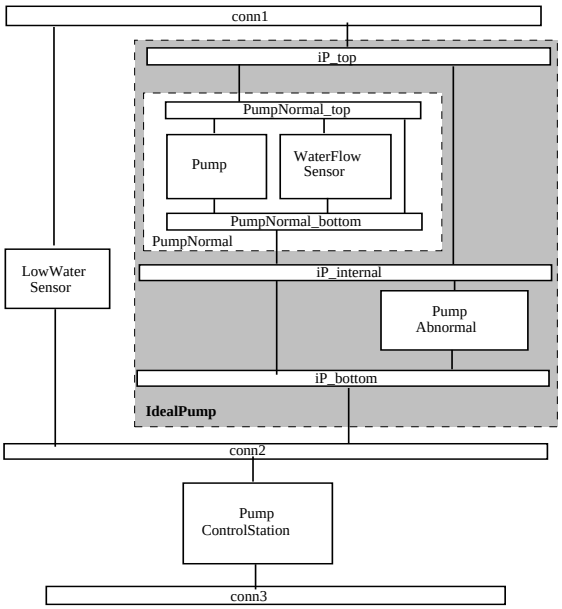

Figure 5. C2 Configuration for Fault Tolerant PumpControlStation

(normal_top and normal_bottom), following the pattern of the multi-versioning connector (MVC) [16]. These connectors coordinate the collaboration between the components, and provide the NormalActivity component with the capabilities for error detection. These capabilities can be associated to the operations provided either by the BasicNormal component or the other collaborating components. Errors are detected by checking the pre- and post-conditions, and invariants associated to the operations [21]. The proposed approach was inspired by the concepts of *coordination contracts* [2] and *co-operative connectors* [14].

On top of the above architecture for an ideal C2 component (iC2C), the Normal Activity component could also interact with other components outside the scope of the iC2C. In this case, the component should be placed higher in the C2 configuration, and the normal_top connector should act as a proxy of the component in the context of the NormalActivity component.

Another special case is when components placed at lower levels of a C2 architecture require to access services provided by other collaborating components wrapped into the NormalActivity component. In this case, the interface of the iC2C can extend that of the BasicNormal for including the required services.

3.3. A Small Example

In order to illustrate the structuring concepts presented in this paper, we refer to a small example extracted from the Mine Pump Control System [20]. The subsystem that we consider is responsible for draining the sump of the mine, and contains the following existing C2 components:

(i) PumpControlStation - controls the draining of the sump by turning on/off a physical pump according to the level of the water in the sump.

(ii) LowWaterSensor - signals when the level of water is low.

(iii) Pump - commands the pump to be turned on/off.

(iv) WaterFlowSensor - signals whether water flows from the sump.

The fault model for the above subsystem assumes that transient faults can affect the operation of the physical pump when reacting to commands from Pump.

The C2 architecture of the subsystem is shown in Figure 5, where the IdealPump is implemented as an idealized C2 component (iC2C). The NormalActivity component of IdealPump, which is PumpNormal, consists of components Pump and WaterFlowSensor that are joined into a collaboration that is coordinated by the PumpNormal_bottom connector. This same connector is responsible for detecting errors in IdealPump, checking the WaterFlowSensor status after a pump on/off requested, and raising an internal exception when the expected condition is not met.

The AbnormalActivity component (PumpAbnormal) is responsible for processing the error, by issuing retry requests to the Pump until either the normal operation is resumed or the exception is propagated to PumpControlStation.

4. CONCLUSIONS

In this paper, we have investigated the structuring of fault-tolerant component-based systems, at the architectural level. For the purpose of our work we have employed the C2 architectural style [23], which is a style that promotes the development of component-based systems using off-the-shelf components. The intent was to provide an idealized C2 component with structure and behaviour equivalent to the idealized fault-tolerant component [1].

The communication rules between components in the C2 style, namely the synchronicity and broadcasting of messages, although desirable from the point of view of component-based design, they complicate the incorporation of fault-tolerance mechanisms into architectures that are instantiations of this style [6, 9]. Another difficulty that we encountered was the restrictions imposed by the C2 topology rules. For solving these problems we employed constructs similar to multi-versioning connector [16], consisting of pairs of collaborating connectors to define fault containment boundaries within the system, and synchronized communications within the idealized C2 component using notifications as acknowledgments of requests. In addition to the work describe above, we have also defined an idealized C2 connector. This fault tolerant architectural element is especially useful considering that connectors in the C2 architectural style are more than simple communication primitives, and that the architectural approach advocated in this paper requires connectors to be also a place of computation.

Our results demonstrate the feasibility of the proposed approach for the C2 architectural style, and suggest their application to other architectural styles also belonging to the *interacting processes style* category, which are styles dominated by communication patterns among independent, usually concurrent, processes [19].

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.

REFERENCES

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

[2] L. F. Andrade, and J. L. Fiadeiro. Feature modeling and composition with coordination contracts. In *Proceedings Feature Interaction in Composed System (ECOOP 2001)*, pages 49--54. Universitat Karlsruhe, 2001.

[3] A. Avizienis. The N-Version Approach to Fault Tolerant Software. *IEEE Transactions on Software Engineering*, 11(2):1491--1501, December 1995.

[4] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[5] D. M. Beder, B. Randell, A. Romanovsky, and C. M. F. Rubira. On Applying Coordinated Atomic Actions and Dependable Software Architectures for Developing Complex Systems. In *Proceedings of the 4th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2001)*, Magdeburg, Germany, May 2-4, 2001, pp. 103-112, IEEE Computer Society Press.

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

[7] T. D. Chandra. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225--267, March 1996.

[8] J. E. Cook, and J. A. Dage. Highly reliable upgrading of components. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 203--212, New York, NY, May 1999. ACM Press.

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

[10] F. C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, 31(1):1--26, March 1999.

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

[12] J. C. Laprie. *Dependability: A Unifying Concept for Reliable Computing and Fault Tolerance*, chapter 1, pages 1--28. Blackwell Scientific Publications Ltd., 1989.

[13] J. C. Laprie. Dependability: Basic concepts and terminology. In *Special Issue of the Twenty-Fifth International Symposium on Fault-Tolerant Computing (FTCS- 25)*. IEEE Computer Society Press, 1995.

[14] R. de Lemos. Describing evolving dependable systems using co-operative software architectures. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, pages 320--329. 2001.

[15] 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 (SSR'97)*, 1997.

[16] M. Rakic, and N. Medvidovic. Increasing the confidence in o-the-shelf components: A software connector-based approach. In *Proceedings of the 2001 Symposium on Software Reusability (SSR'01)*, pages 11--18. ACM/SIGSOFT, May 2001.

[17] B. Randell, and J. Xu. The evolution of the recovery block concept, In *Software Fault Tolerance*, chapter 1. John Wiley Sons Ltd., 1995.

[18] T. Saridakis, and V. Issarny. Developing Dependable Systems using Software Architecture. Technical report, INRIA/IRISA, 1999.

[19] M. Shaw, and P. Clements. A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. In *Proceedings of the COMPSAC97, First International Computer Software and Applications Conference*, 1997.

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

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

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

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

[24] G. Vecellio, and W. M. Thomas. Issues in the assurance of component-based software. In *Proceedings of the 2000 International Workshop on Component-Based Software Engineering*. Carnegie Mellon Software Engineering Institute, 2000.