



# Incorporating Fault Tolerance into Component-based Architectures for Embedded Systems

Shourong Lu and Wolfgang A. Halang

Fernuniversität in Hagen  
Chair of Computer Engineering  
58084 Hagen, Germany  
{Shourong.Lu, Wolfgang.Halang}@FernUni-Hagen.de

**Abstract.** A component-based software architecture is presented to support the process of designing and developing fault-tolerant computerised control systems. To this end, we combine an idealised fault-tolerant component, the C2 architecture style and protective wrappers, and embed fault tolerance techniques into component definitions. The resulting architecture is described by normal- and abnormal-activity components aiming to support a wide range of fault tolerance features. Use of this architecture enables to reason about system dependability already from the earliest development stages on, and to customise fault tolerance strategies according to application characteristics.

## 1 Introduction

Fault prevention, fault tolerance, fault removal, and fault forecasting are the four main means to attain the various attributes of dependability [4]. Fault prevention and fault tolerance aim to prevent introducing faults, or to avoid service failures when faults occur. Fault removal and fault forecasting, instead, mean to reduce number and severity of faults, and to estimate present and future incidences and consequences of faults. Among them, fault tolerance is the most promising mechanism to meet the dependability requirements. Fault-tolerant systems work under the assumption that they contain faults (e.g., made by humans while developing or using systems, or caused by aging hardware), and aim to provide specified services in spite of faults being present. Fault tolerance depends on redundancy, fault detection, and recovery. Many fault-tolerant systems are complex because of redundancy, re-configurability, and various interactions between their components. This complexity has a strong impact on system architecture, as fault occurrences have to be taken into account from the earliest design stages on of systems required to be dependable. Therefore, architecture design is a crucial aspect for fault-tolerant systems, as indicated by well-known safety mechanisms such as masking, dynamic redundancy, and design diversity (e.g., N-version programming, recovery blocks) [8].

Component-Based Software Engineering (CBSE) focuses on producing software systems by composing prefabricated components, which can improve the

productivity and quality of target systems. In recent years, the emphasis of research on and practice in CBSE has changed from functional aspects to non-functional ones. Particularly, dependability of component-based systems is considered as one of the most crucial non-functional properties in CBSE. To cope with complexity and fault tolerance requirements, it may be beneficial to combine in their development process well-established fault tolerance techniques with component techniques, as well as to employ the Unified Modeling Language (UML) [10] to describe component-based software architecture models providing fault tolerance.

In a component-based system, the software architecture specification captures system structure, by identifying architectural components and connectors, and required system behaviour, by specifying how components and connectors are intended to interact. It is desirable that all components show only their normal behaviours. However, certain abnormal behaviours exist when some unexpected condition occur. In CBSE, the abnormal behaviours of components are usually represented by a set of exceptions defined by the component developers. It is impossible to eliminate all abnormal behaviours, but one can reduce them, and handle them properly after their occurrences. The main goal of this paper is to design a comprehensive architecture to develop fault-tolerant systems from components. By embedding fault tolerance mechanisms into an integration architecture, normal and exceptional behaviours of system components are specified.

The body of this paper is organised as follows. Section 2 briefly introduces an idealised fault-tolerant component meeting the architecture style C2 as defined in [7]. Section 3 describes the integration of idealised fault-tolerant components and the C2 architecture. Section 4 presents the main activities of designing fault-tolerant systems out of components.

## 2 An Idealised Fault-tolerant Component and C2 Architecture Style

An idealised fault-tolerant component [5] is a structuring concept for coherent provision of fault tolerance in a system as shown in Fig. 1(a). It includes both normal and abnormal responses in the interface between interacting components. Idealised fault-tolerant components communicate through request or response messages, only. On receiving a service request, an idealised component will react with a normal response if the request is successfully processed, and with an external exception, otherwise. Such an external exception may be due to an invalid service request, in which case it is called interface exception, or due to a failure in processing a valid request, in which case it is called 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.

C2 is a software architecture style for systems with intensive user interfacing. The style is component-based, and supports large-grain re-use and flexible

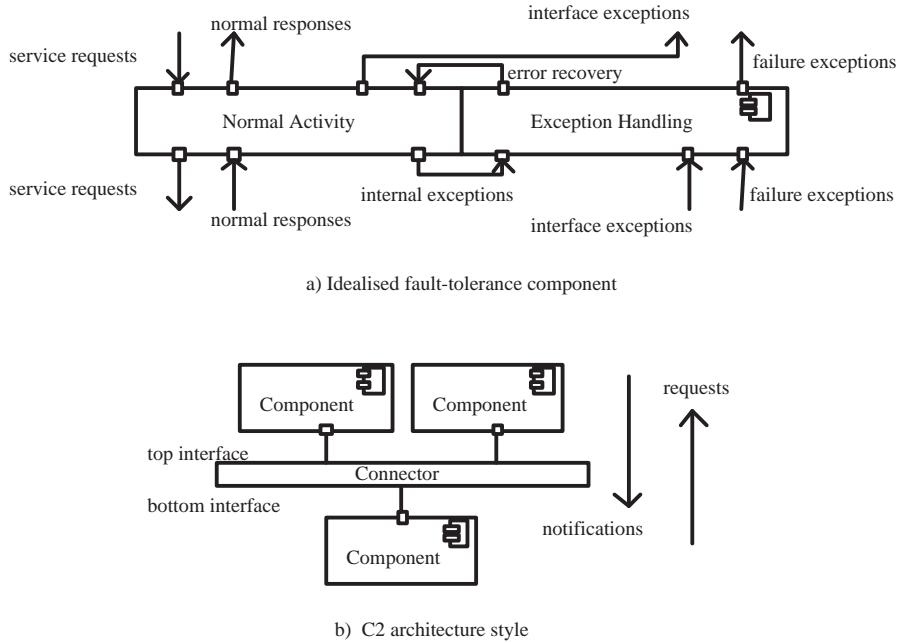


Fig. 1. Idealised fault tolerance component and C2 architecture style

system composition, emphasising weak bindings between components [7]. A C2 architecture as shown in Fig. 1(b) consists of software components and connectors, which transmit messages between components. Components maintain state, perform operations, and exchange messages with other components via two interfaces (named top and bottom). Each interface consists of a set of messages that may be sent or received. Intercomponent messages are classified into two types, viz., requests to a component to perform an operation, and notifications that a given component has performed an operation or changed state. In the C2 architectural style, both components and connectors have a top and a bottom interface. Systems are composed in a layered style, where a component’s top interface 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.

### 3 Integration of Idealised Components into C2

Guerra et al. [3] introduced the concept of the “idealised C2 Component” (iC2C), depicted in Fig. 2, as an equivalent, in structure and behaviour, to the idealised fault-tolerant component. The latter’s purpose is to structure the architectures of component-based software systems compliant with the C2 architectural style. Service requests and normal responses of an idealised fault-tolerant component

are mapped as requests and notifications in the C2 architectural style. Interfaces and failure exceptions of an idealised fault-tolerant component are considered subtypes of notifications. An iC2C is composed of five elements: NormalActivity and AbnormalActivity components, and iC2C top, iC2C internal, and iC2C bottom connectors.

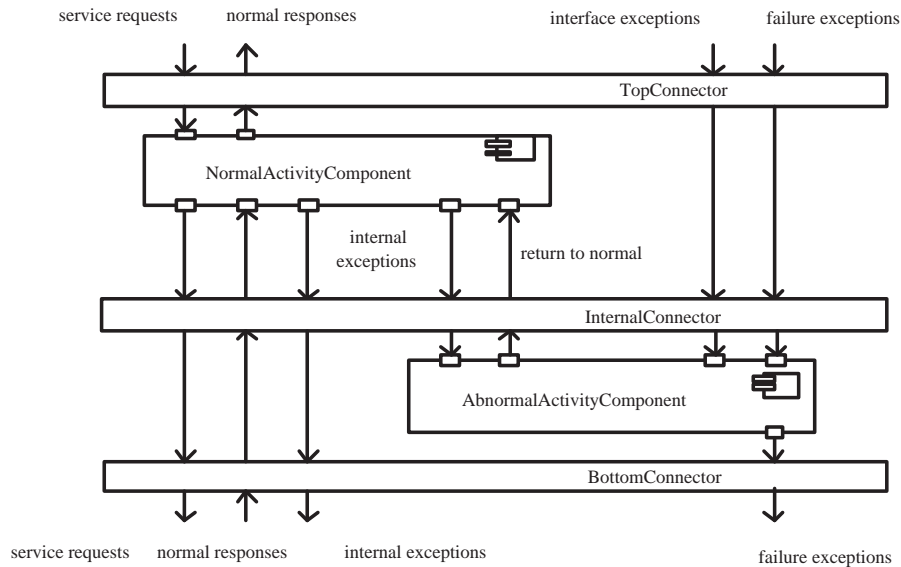


Fig. 2. An idealised C2 component

The NormalActivity component processes service requests and answers them through notifications. It implements the normal behaviour, responsible for error detection during normal operation, and the signaling of interface and internal exceptions. The AbnormalActivity component is responsible for the exception handlers (error recovery) of the iC2C, and the signaling of failure exceptions. While an iC2C is in its 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 iC2C returns to its normal state and the NormalActivity component resumes processing. Otherwise, a failure exception is signaled to components in lower layers of the architecture, which become responsible for handling it.

The iC2C top connector encapsulates the interaction between the iC2C and components located in upper levels of an architecture. It is responsible to guarantee 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 des-

tinations. The iC2C top connector also performs domain translation, converting incoming notifications to a format the iC2C understands, and outgoing requests to a format the application understands. The iC2C internal connector is responsible for message routing inside the iC2C. The destination of a message sent by the internal elements of the iC2C depends on its type, and whether the iC2C is in a normal or abnormal state. The iC2C's bottom connector connects it with the lower components of a C2 configuration, and serialises the requests received. Once a request is accepted, this connector queues new requests received until completion of the first one. When a request is completed, a notification is returned, which may be a normal response, an interface exception, or a failure exception.

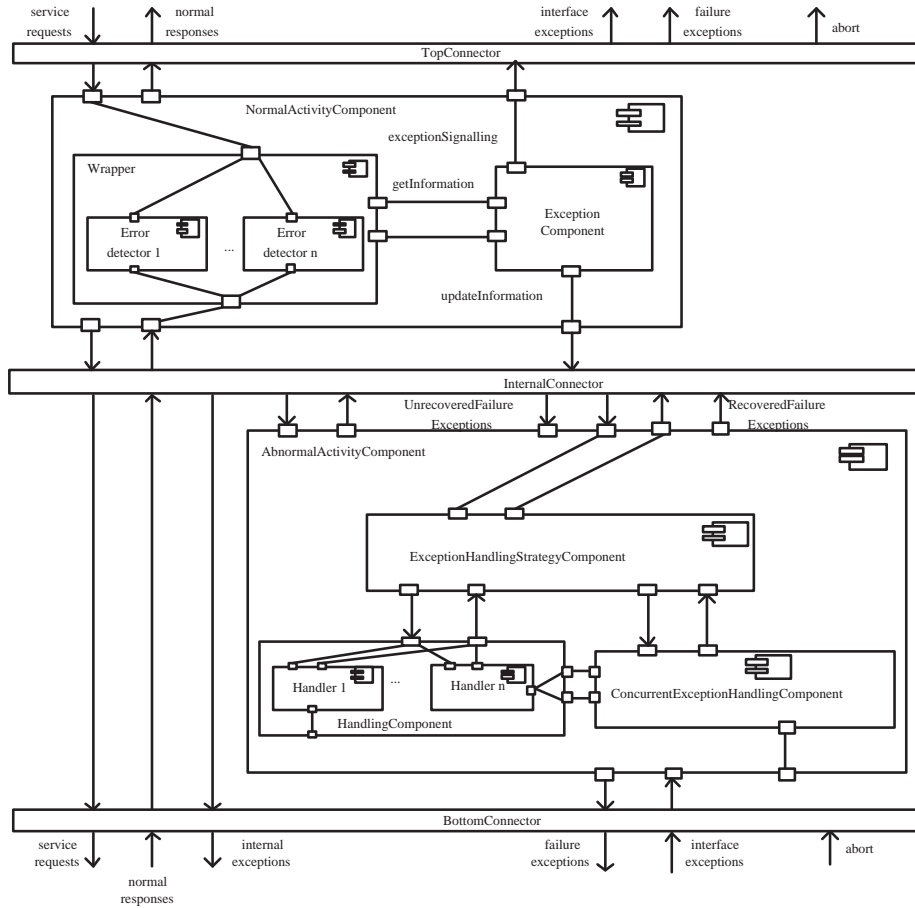
## 4 Extending the iC2C Architecture by Fault Tolerance Mechanisms

The proposed architecture is constructed by adding fault tolerance mechanisms to normal- and abnormal-activity components as shown in Fig. 3. The fault tolerance mechanisms are responsible for detecting errors or suspicious activities, and for executing appropriate recoveries whenever possible. The incorporation of fault tolerance mechanisms into the iC2C architecture requires to re-define normal and abnormal activities.

### 4.1 Definition of the NormalActivity Component

In the iC2C architecture, the NormalActivity component encapsulates a desired functionality (normal activity). It may represent both a single component and a configuration established through connectors. To embed error detection mechanisms into the NormalActivity component, the concept of protective wrappers, known to be the most general approach to develop fault-tolerant systems based on COTS components, is employed. Component wrapping is a well-known structuring technique, and a cost-effective solution to many problems in component-based software development [2].

In our extension, a set of detectors is wrapped as iC2C connectors connecting the normal activity component, which is viewed as redundant software, and responsible for (1) detecting exceptional conditions anticipated by the specification and signalling them by raising exceptions in the provided interface of the exception component, and (2) signalling other exceptional conditions specific to the implementation of the component, by raising exceptions. This set consists of well-defined error detectors concerned with special purposes such as “Fail-Stop Processor”, “Acknowledgment”, “I Am Alive” — “Are You Alive” [6]. When a constraint violation is detected, the detector sends an exception notification to the AbnormalActivity component. In addition to error detectors embedded in the NormalActivity component, an exception component is required to specify the raising of local and co-operating exceptions, i.e., signalling of an appropriate



**Fig. 3.** Extending the iC2C architecture by fault tolerance mechanisms

exception. Hence, an exception component is designed working as an extra information holder, and keeping information about application exceptions which are used by the other components. In other words, the exception component interacts with the handler, concurrency and strategy component located in the AbnormalActivity component in order to obtain and update information about exception occurrences and handling. The exception component should have the required interface for getting information and the one for updating information. The former allows the application and other components to obtain information about exception occurrences, whereas the latter allows its clients to update information about exceptions.

## 4.2 Definition of the AbnormalActivity Component

Raising an exception results in an interruption of a component's normal activity, followed by a search for an appropriate exception handler to deal with the exception signalled. Therefore, the AbnormalActivity component can be defined as an exception handling mechanism as shown in Fig. 3 aiming for error recovery. When an exception is raised, the exception handling mechanism begins to work. It should be able to find and activate an exception handler to recover from errors, and to put the system back into a coherent state.

Abnormal behaviour is not restricted to failures of a single component, but also associated with invalid interactions between two or more components, and with combinations of component failures. An intra-component exception is raised by an individual component. The strategy to deal with it is based on local exception handlers, which are associated to a component's implementation. Their main responsibility is to cope with anticipated exceptions, which are more related to the component's application domain, and are declared in its required interfaces, i.e., they are responsible to handle the external exceptions of the component's required interfaces and the internal exceptions raised by the component's implementation. When possible, the handlers should implement forward error recovery to mask the exceptions.

Inter-component exceptions are raised by component configurations. The strategy to deal with them has to consider the integration of pre-existing components into a new configuration, and is based on co-operating exception handlers. These are associated with hierarchical structure, and deal with exceptions that could not be handled within the single components. The handlers are responsible for providing error recovery and masking by means of redundant components in the configuration, and resolving failure semantics mismatches between server components and their clients. They should be capable of dealing with all exceptions signalled by a server component. Upon receiving an exception from a server component, the handlers should try to mask it invoking the same operation on a redundant (replicated or diversely designed [1]) server component, in case it is available. The handlers are also responsible for translating unmasked exceptions from the server component's domain to the client component's domain, before propagating them to client components. Exceptions requiring no further adaptation are automatically propagated. When direct propagation is not possible, a new exception is created wrapping the unmasked exceptions raised by the server component.

As there are intra- and inter-component exceptions, the architecture is designed to consist (see Fig. 3) of

- HandlerComponent (HC),
- ExceptionHandlingStrategyComponent (EHSC), and
- ConcurrentExceptionHandlingComponent (CEHC).

The *ExceptionHandlingStrategyComponent* is responsible to locate the different handlers required to resolve an exception. It implements services related to the strategy for exception handling. Hence, its responsibilities are deviation of

control flow and search for handlers. It plays a central rôle in the architecture, and interacts with all other components. From `ExceptionComponent` it receives information about an exception occurrence while searching for its corresponding handler. Its provided interface provides the service for handler search.

The handling and propagation of an exception depends on whether it is an intra-component exception or an inter-component one. For the former, the handling is limited within the component's `AbnormalActivity`. If the exception cannot be handled, then it is propagated. A handler may also explicitly re-signal the exception to a component in a layer of the architecture. An inter-component exception raised on receiving a service request by a component is not handled there, since it does not indicate that the component is faulty. Therefore, the exception is propagated to the client component which issued the request. The client component handles the exception in the same manner as an intra-component one, because it possibly indicates a fault within the client component. The strategy to search for handlers of intra- and inter-component exceptions can be formulated in pseudo-code as follows:

```

if exception is propagated from intra-component then
  if local handler exists then
    call local handler;
    if not handled then
      go to a higher (action) level handling;
    end if;
  else
    go to a higher (action) level handling;
  end if;
  default handler;
else
  error detection in client component;
  if exception is raised (error has been found) then
    if local handler exists then
      call local handler;
      if not handled then
        go to a higher (action) level handling;
      end if;
    else go to a higher (action) level handling;
    end if;
    default handler
  end if;
end if;

```

The *HandlerComponent* is constructed with a set of handlers to cope with abnormal activities, i.e., it is responsible for fault masking and recovering. After a handler is found, EHSC asks the *HandlerComponent* to invoke the exception handler. According to the hierarchical structure, handlers may be associated with a component, a connector, or a configuration as well as with exceptions themselves (default handlers). A default handler is executed when there is not a more specific handler in an application. The *HandlerComponent* has a required interface which allows the EHSC to invoke a handler when the appropriate handler has been found. If an exception reaches the lowest level of an architecture, the handler for the entire system should be executed.

The *ConcurrentExceptionHandlingComponent* is designed for concurrent cooperative actions which use the services provided by the exception handling strategy in order to carry out the strategy for concurrent exception handling.

When co-operating exceptions are raised during an action, the exception resolution is accomplished by this component. It has a provided interface which is accessible by applications to create concurrent co-operative actions.

To design co-operative component activities using nested atomic actions, it is described how each individual component is involved in such activities, and co-operative exception handling for all participants in each action is developed. We employ the Coordinated Atomic (CA) action scheme [9] in which components take part, by defining a group as an action, and participants are components. Then, only co-ordinated recovery needs to be activated within the participant tasks of a group. This obviously restricts system design, but enables to regard each group as a recovery region, and to attach fault tolerance activities to each group participant.

Each group has participants which may be activated by some external activities, e.g., tasks, and which co-operate within the group's scope. Participants execute object methods that should have been designed to work co-operatively by means of shared objects. Participants may enter asynchronously into a group activity, but should exit in a synchronised way. Each group participant has a set of exception handlers that are designed to recover the group co-operatively from eventual errors. If any suitable handler has not been defined at least in one of the group participants, an "abort exception" is raised and, then, the group activity must be undone (backward error recovery), and such an exception must be signalled to the enclosing group. If the backward error recovery is not executed successfully within the group, a "failure exception" is signalled to the enclosing group.

Exceptions can be raised by participants during an action. Some of them can be handled internally by a local handler attached to the participant that raised an exception. If an exception occurrence is not handled internally by a participant, then it should be handled co-operatively by all action participants. A set of co-operating exceptions is associated with each action. Each participant has a set of handlers for (all or part of) these exceptions. Participants are synchronised, and probably different handlers for the same exception have to be invoked in the different participants. These handlers are executed concurrently, and co-operate in handling the exception in a co-ordinated way. Therefore, we employ local handling at the level of individual components. If local handling does not succeed, an exception is propagated to the level of an action in which this component is involved to be handled co-operatively. When action-level handling is not possible, an exception is propagated to the containing action.

The newly defined `AbnormalActivity` component encapsulates the exception handling mechanisms (intra-component and inter-component exception handling). After implementation, it requires an interface which should have the function `handleException()`. If an exception is successfully handled, `handleException()` returns a message which is sent to the `NormalActivity` component. Then, processing is resumed. Otherwise, an exception is raised in the body of `handleException()`.

## 5 Conclusion

With the concept of the extended iC2C we aim to add fault tolerance mechanisms to the NormalActivity and AbnormalActivity components of an iC2C. In our approach, error detector and exception components are encapsulated into the NormalActivity component, with error detectors contained by a wrapper. These detectors are responsible to verify that messages do not violate the acceptable behaviour constraints specified for a system. The exception handling mechanism is embedded into the AbnormalActivity component.

The advantage of integrating fault tolerance techniques into the process of designing and developing computer control systems required to be dependable is that appropriate fault tolerance techniques can be selected from a set of mechanisms provided and customised according to application characteristics. This approach will enhance the safety of control systems. Employing its built-in extension mechanisms, UML can be extended to suit dependable applications with respect to aspects such as error detection, error recovery, or configuration of redundancy measures. Thus, for each aspect to be modeled, the most expressive techniques can be selected by the user. Furthermore, UML and the here specified extensions constitute an effective environment to design dependable computer control systems in a comprehensive way taking fault tolerance into account throughout the entire development process.

## References

1. Anderson T. and Lee P. A.: *Fault Tolerance: Principles and Practice*, Dependable Computing and Fault Tolerant Systems, Vol 3. Springer-Verlag, 1990.
2. Anderson T., Randell B. and Romanovsky A.: *Wrapping the Future*, Proc. 18th IFIP World Computer Congress, pp. 165–173, 2004.
3. Guerra P. A. de C., Rubira C. M. F. and de Lemos R.: *An Idealized Fault-Tolerant Architectural Component*, in *Architecting Dependable Systems*, pp. 21–41. LNCS, Springer-Verlag, 2003.
4. Laprie J.-C.: *Dependability: Basic Concepts and Terminology*, Proc. 25th IEEE Intl. Symp. on Fault-Tolerant Computing, pp. 42–54. IEEE Computer Society Press, 1995.
5. Lee A. and Anderson T.: *Fault Tolerance: Principles and Practice*, Springer-Verlag, 1990.
6. Saridakis T.: *A System of Patterns for Fault Tolerance*, Proc. EuroPLoP.
7. Taylor R. N., Medvidovic N., Anderson K. M., Whitehead E. J., Robbins J. E., Nies K. A., Oreizy P. D. and Dubrow L. A.: *Component- and message-based architectural style for GUI software*, IEEE Trans. on Software Engineering, 22(6): 390–406, 1996.
8. Torres-Pomales W.: *Software Fault Tolerance: A tutorial*, NASA/TM-2000-210616, 2000.
9. Xu J., Randell B., Romanovsky A. B., Rubira C. M. F., Stroud R. J. and Wu Z.: *Fault tolerance in concurrent object-oriented software through coordinated error recovery*, Proc. Symp. on Fault-Tolerant Computing, pp. 499–508, 1995.
10. Unified Modeling Language: Superstructure. *OMG document ptc/2003-08-02*, 2003.