2^{nd} Component User's Conference
CUC'97

On the Aspect of Fault Tolerance in Component–Oriented Systems

Stefan Schreyjak

Stefan.Schreyjak@informatik.uni-stuttgart.de

To be published in:
Thomas Jell (Ed.), *Component–Based Software Engineering (CUC '97)*,
SIGS, 1998,

The use of applications built of components in the activities of a workflow solves several problems current workflow systems are confronted. The aspect of fault tolerance in this system architecture is discussed in this paper. Several basic fault handling concepts are presented and it is showed how to use these concepts in this system architecture. One result is that the property of fault tolerance cannot always realized by adding independent fault handling components.

Keywords: extendable systems, components, workflow, fault tolerance, transactions

1 Introduction

Fault tolerance is an important requirement in a workflow system. Using a workflow system for controlling business processes makes a company dependent on the stable working of the software. If the system is not working every computer supported business process is stopped. Proceeding manually with the process is difficult because this results in inconsistencies between the stored data and the reality. A workflow system is similar to a backbone — a break is lethal.

Prime directive of such a system must be fault avoidance. If it is not possible flexible and powerful fault handling is necessary.

In order to realize fault tolerant workflows all applications executed in the workflow must also be fault tolerant. Using an application in a workflow has the effect that faults can affect under certain circumstances all people involved with the process. A fault in an application supporting functional–oriented work of a person (e. g. a word processing program) affects only one person.

After introducing terms and basics in the area of fault tolerance and component–oriented programming a system architecture combining a workflow system and a component–oriented system is presented and motivated. The special problems of such a coupled system concerning fault tolerance aspects are discussed. This results in a list of requirements that fault handling must fulfil. Afterwards, several fault handling concepts are introduced ordered after their implementation place. In the end related work is considered and the essential ideas are summed up.

2 Basics of Fault Handling

Objective of fault handling is to insure the integrity of an application after the occurrence of a fault in the system. Objective of fault prevention is to avoid that faults are even occurring. That point is not discussed further in this paper.

Two essential types of faults are possible: faults are *transient* if they occur occasionally [GR93]. It is difficult or even impossible to reproduce the fault. If the system is put back in the same state where the fault occurred, the fault does not have to occur again. A cause for transient faults can be timing problems in situations with high system load or e. g. memory overflows.

Faults occurring reproducible in a certain system state are called *persistent* faults. Programming typos like wrong parameters in function calls or procedures differing from their specification belong to this kind of fault.

Fault handling can be divided in the following phases dependent on the kind of fault:

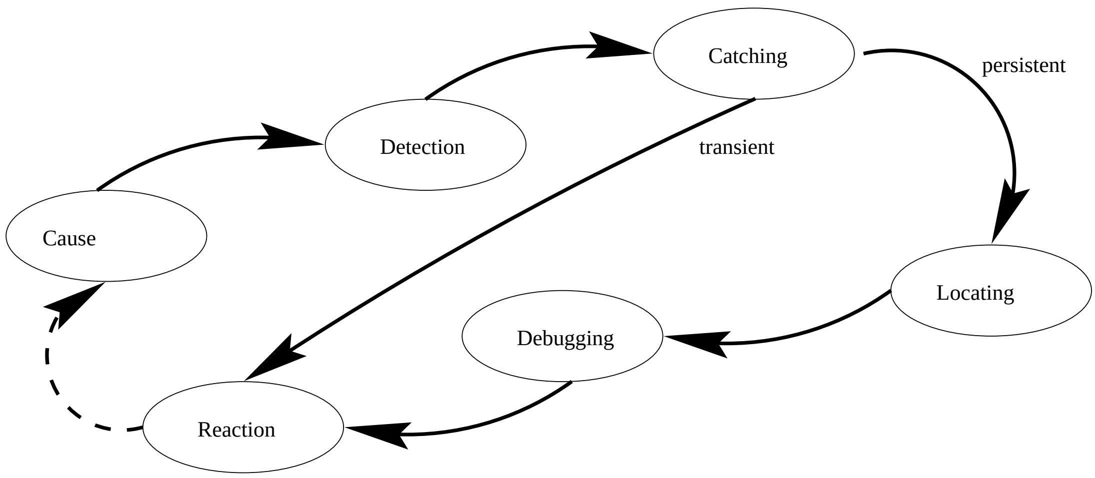

Figure 1: The phases fault handling can be divided.

Cause: The event causing the fault is occurring. The cause determines the kind of fault. An example for a cause of a persistent fault is an incorrect source code line.

Detection: The effect caused by the fault occurs. The effect (symptom) of a fault enables the system or the user to detect the occurrence of a fault. Thereby a delay between cause and effect is possible. Writing an incorrect program statement and the execution of this statement is a good example for a delayed detection of the fault.

Catching: The effects of a fault must be isolated as good as possible to prevent further contamination. Afterwards, the system must reach a secure state, where the integrity of the system is not violated. Either, an old state is reenter after clearing all effects of the fault or a new secure state is created, where the effects does not affect the integrity anymore.

(Locating:) In the case of persistent faults the cause of the fault must be located.

(Debugging:) Than, also in the case of persistent faults, the cause of the fault must be removed.

Reaction: Now, in both cases, a reaction to the fault can be done. Possible reactions are repeating the failed function, executing an alternate function or delegating the responsibility to the next higher system instance[1]. There, the same phases can be passed through.

The fault handling concepts described in this paper take place mostly in the phases *Catching, Debugging* and *Reaction*.

3 Definitions

Since no consistent terminology is used in the area of component–oriented programming, all essential terms used in this paper are defined here to clarify their usage.

Components are reusable software building blocks which can be assembled or disassembled in an application without violating its integrity. There are atomic and composed components. Atomic components offer functionality of one specific domain, i. e. functionality must be distributed among components in a way that interfaces are of minimal size.

A *component model* specifies a set of protocols which enable components to interact and defines the structure of components.

A *component–oriented system* is an architecture for a software system which provides the infrastructure for components. An implementation of this architecture consists of an application development environment and a runtime environment for components. A *component–based system* uses components only in the development environment. In the runtime environment (respectively the compiled application) components do not exist anymore.

The term *component–oriented application* defines an application which is built using components as building blocks and is used to distinguish itself from applications conventionally built.

[1] see figure 3

4 The Coupling of a Workflow and a Component–oriented System

In this section a coupling of these two systems is motivated because it will solve some problems today's workflow systems are confronted.

Workflow systems and component–oriented systems have similar assignments and use similar technical approaches to facilitate the development of applications. But they differ in the application domain and in the magnitude of the application. Workflow systems control enterprise–wide applications consisting of activities. Component–oriented systems control local applications consisting of software building blocks.

Using today's workflow systems reveal some problems, which can be solved or reduced by using component–oriented systems as application programs in activities [Sch97].

Neither workflow systems nor component–oriented systems do know anything about the existence or use of the other system. Therefore, there are only few possibilities for cooperation, even more, double implementations of services are possible. Creating interfaces enables the partitioning of responsibilities and allows more cooperation.

Using a modelling tool an user can easily adapt a workflow to changing market situations. But the adaption ends in the activities. The programs executed in activities can only be adapted by software vendors, which is expensive and time–consuming. Component–oriented systems allow the user to adapt the component application. Indeed, this feature is not supported very well in current systems. A possibility to improve this support is to offer visual interactive changeable views on different aspects of the component application.

Workflow systems are used in a very heterogenous infrastructure of computer equipments, which is grown by and by. Workflows are platform independent because of their platform independent specification. Typically, the application in the workflow are platform dependent. This problem can be solved by using platform independent components.

In order to reuse a workflow in another company, it must be adaptable and the applications used in the workflow must be runnable on the platforms the other company. Therefore, the problem of reuse is coupled to the solution of the adaptation and platform independence problem.

By modifying a component–oriented system with respect to the problems mentioned and by modifying a workflow system to support component applications a system coupling is possible which solves all problems mentioned [Sch97].

4.1 The Architecture of the System Coupling

In this subsection an architecture for a component–oriented system is presented which can be used in a workflow system. The presentation is necessary for the understanding of fault handling concepts which are described in later sections. Descriptions of the architecture of workflow systems can be found in literature (e. g. [SB96a, Jab95, Wor94, LR94]).

A component consists of glue code which is changeable by the end user and of unchangeable binary code. A component offers its functionality in an export interface

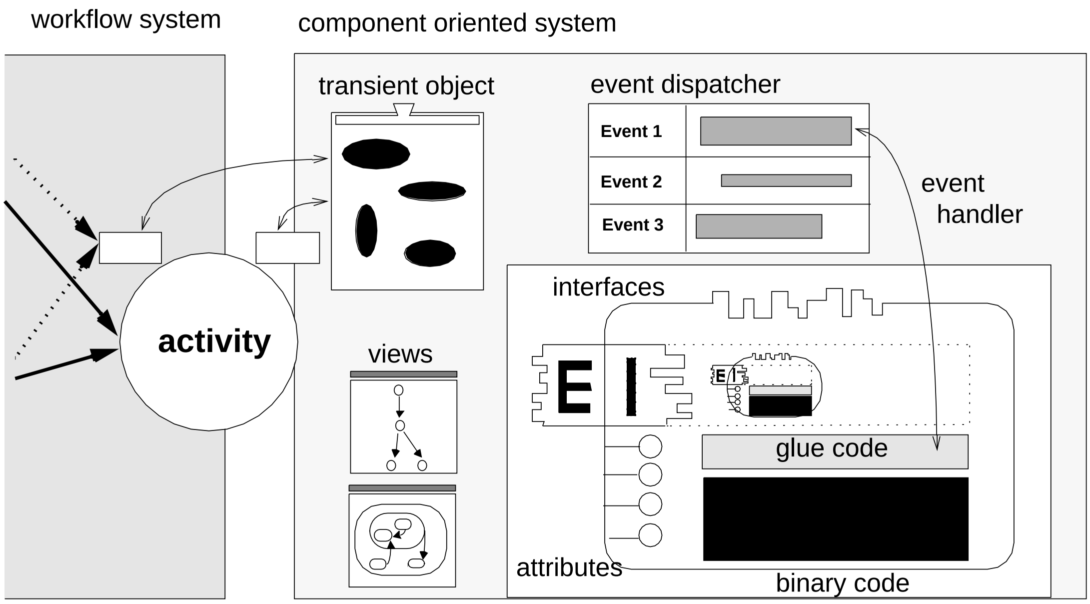

Figure 2: The architecture of the system coupling

to other components. The import interface determines which functionality is used. Components can be composed to more complex components. They offer attributes for the purpose of configuration which can be changed by users. The reaction of a function call consists of side effects, events and transient data objects.

If a component is incorporated into an application an event handler written in glue code is installed in the event dispatcher. The dispatcher executes this code after the occurance of the event. The event handler can consist of function calls to other components. Transient data objects are used to exchange temporary data between components. They are stored in an application global storage (the clipboard). Each component possesses a mandatory interface to the component–oriented system. Interactive views visualize different aspects on the application. The user can change these aspects interactively. Figure 2 shows the described architecture.

5 Problem Description

In this section it will be described which special problems concerning fault tolerance arise in component–oriented systems.

The term "application program" will change in component–oriented systems. Identical programs, which can be identified e.g. by a version number, will vanish because the user himself can now create new versions. Instead, a lot of *variants* of a program will exist. A component application is never "finished" or "completed". Therefore, a final test of the finished program and selling this program under a version number will not be possible anymore. Vendor support will be difficult if the application was changed by the user. Also, the integration test will be shifted into the responsibility of the end user.

The user can enhance the program anytime with new components or he can modify the glue code. This means in particular he can insert new faults in the application.

Components are developed from software vendors working independently from each other. The specification of the component model is the solely common developing basis. Beyond the specification no common context or arrangement is possible which could help avoiding misunderstandings of the specification leading to faults. This could result in the fact, that components fit together only syntactically but not semantically. Using such a composed component will result in a runtime error presented to the end user.

Therefore, faults will occur more often in component–oriented applications than in traditionally built applications.

6 Requirements for Fault Handling

The following requirements for fault handling result from the problems mentioned and the application domain. The requirements need to be complied by the coupled system already introduced.

Independence: The fault handling should be organized in a way that functionality code and fault handling code are well separated. Thus, fault handling will be more flexible.

Reuse: The separation of fault handling code in components facilitates the reuse of fault handling. Therefore, it will be easier and less time–consuming to create fault tolerant applications.

User Configuration: The user should be able to add additional fault tolerance properties to applications as far as the components will allow. In particular, he should be able to select the grade of fault tolerance by considering performance against fault tolerance aspects.

Coupling: The fault handling of the applications should be coupled with the fault handling of the workflow system. Thus, the fault tolerance of the whole system can be more flexible compared to fault handling restricted to one system because the priority of a fault can be considered better due to the process context.

7 Concepts for Improving Fault Tolerance

In this section concepts for improving fault tolerance in component–oriented applications are described.

In the coupled system presented three starting points for fault handling exists. In figure 3 the layering of the fault handling in such a system is shown. If one system layer cannot cope with a fault, it passes the fault to the next layer. This is done until a layer can do fault handling or the user must interfere as the last instance.

First, fault handling or parts of it can take place in the component itself. This fault handling must be programmed by the software vendor of the component. The user of the component does not have influence on the capability of the component.

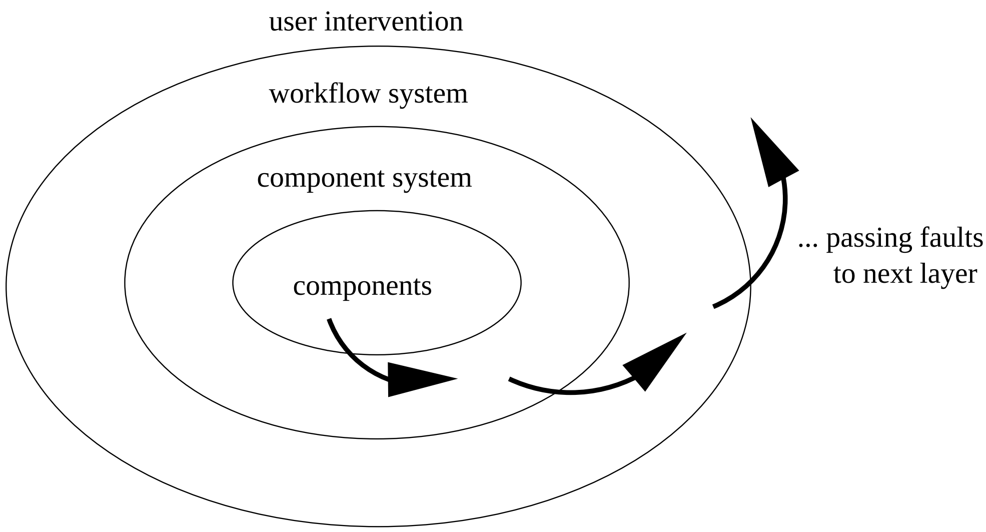

Figure 3: The layers of fault handling in the coupled system

Certain functionality and interfaces are often prerequisite to the fault handling in an outer layer. If components cannot fulfil these properties, some fault handling concepts cannot be used.

The second starting point is the component–oriented system. Fault handling can be inserted by the user using fault handling components. There are two kinds of fault handling components possible. A *wrapper component* wraps inner components and enhances them with fault tolerance properties. A *additional fault handling component* interferes in the communication of all other components by modifying the control flow for the purpose of fault tolerance.

Third, the workflow system can do the same as an additional fault handling component but it can enhance the fault handling to the whole workflow.

In this paper four basic fault tolerance concepts are being discussed. The concept mostly used is called *ACID–transaction* [GR93]. A transaction merges several operations into one atomar operation. Either the transaction will be done successfully as a whole or the transaction will fail and than no effect on the data is visible. The data is either consistent or isolated in the transaction. Data changes are durable if they were successful. In the case of a fault all changed data are rolled back into the originally data value. In the *compensation* concept for each operation executed a compensation operation will be executed if a fault occurred. The concept of *alternative actions* catches a fault and reacts with the execution of alternative operations. In the concept of *manual intervention* the system catches a fault,too and waits then for some action by the user fixing the problem. These concepts can be used on all layers of the proposed architecture in different peculatity.

In the following sections concepts for cooperative fault handling are presented. Fault handling within components that take place without interaction to other components or systems will not be considered. The concepts are organized after their mainly place of implementation. It has to be remarked that some concepts cannot be used for their own but only in cooperation with other concepts.

In the Components

Several fault handling concepts require certain properties of components. These properties must be accessible by interfaces.

Manageable: A component is manageable if someone can interfere from outside into the work of the component. It should be able to start, stop, reset, pause and resume the working. An abortive break can be useful for the workflow system, for example.

Persistent: The persistence property means that the state of a component can be stored and reloaded. This is very useful in implementing concepts for fault handling.

Transactional: In order to use a component in a transaction the component must offer an interface for participation in the two–phase–commit protocol. The component must be a resource manager [GR93] or must access data using a resource manager.

Compensatable: A compensatable component must offer for each function an inverse function.

Some properties are already realized as services or frameworks (e. g. in CORBA's common services [Obj97]) and can be used implementing components.

Fault Tolerance Components

A wrapper component can log all function calls and created events of the inner components. This log can be used to understand what happened if a fault occurred. This information is important for debugging purposes or for planning reactions to the fault.

If the inner components are compensatable the logging component can be enhanced to store for each function call the inverse function call in the log. Than, the wrapper component can offer a function ("undo") that compensates the work of the inner components.

A control flow component can catch faults and react with the execution of configurable glue code. An example for a reaction is to redo the function call failed or to call another alternative function. This intervention in the control flow can be done without knowledge of the inner components because the control flow is stored in the event dispatcher and the wrapper component changes this code entry.

A transaction manager, realized as a component, is one of the most important fault handling components. This component can be used for executing transactions within the component–oriented system. The component must create a transaction context and play the role of a coordinator in the two–phase–commit protocol in order to insure that all participants in the transaction end their work unitarily.

The user can use the interactive views of the component–oriented system to react on the occurrence of a persistent fault. For example, he can exchange the faulty component manually if there is a compatible component with the same functionality and a different implementation. This task can be automated with a specialized fault handling component.

The interactive views can be used to change the component–oriented system into a manual working modus. In this modus the automated processing of events is interrupted. Now, the user can interactively type in commands in the glue code language and call functions of the components, similar to a command shell. An experienced user has now the possibility to influence manageable components. He can detour manually around the faulty function or debug incorrect glue code. In particular, repairing corrupt data in transient data objects would be an useful operating area for the manual working modus. Slightly modifying the data can sometimes helps a components to cope with the data. With this little help the processing of data would be possible despite of a faulty function. The interactive views can be enhanced by fault handling components which support the user in the manual working modus. Components can offer a library of support functions or can provide additional graphical views.

Cooperative Work between a Component–oriented System and a Workflow System

A workflow system should offer several concepts for fault handling, like transactions, compensation and alternative paths [SB96b]. The workflow system Surro [SB96a] for example realizes these concepts with transaction and compensation spheres and additional activities. A sphere is a set of activities participating in a transaction or a compensation.

Having components with the properties mentioned above facilitate fault handling in the workflow system. Partly these properties are even precondition for certain concepts. Using compensatable components eases the creating of compensation spheres because the workflow system can automatically construct the compensation activities. Using transaction spheres in the workflow requires to have transactional activities. Those activities can be created more easily by using transactional components.

8 Related Work

Leymann proposes in [Ley95] to use single components in activities. This highly integrative approach compared to the approach in this paper was implemented in Surro [SB96a]. If the components are to small the approach results in high effort for the workflow engine. This integrative approach suffer from bad scalability. Thus, only a few components can be used, which needs to be big and complex. These big components cannot be modified by the user and the programming effort to produce these components is high.

Troya and Vallecillo [TV97] introduce in their component–oriented system an new kind of component, called *properties*. The functionality of properties is similar to the proposed wrapper components, but properties are implemented as message filter in a layered object model. This approach presumes that fault tolerance is totally independent on the functionality. Generally, that is not possible. In particular, transactions cannot implemented in that way.

Baggiolini and Harms [BH97] limit their component–oriented system to components that are stateless. These components, similar to UNIX-Filters, are connected with pipes. If a fault occurs the system tries to locate the faulty component and considers possible measures. A measures, for example, is to replace automatically the component.

The aim of the Microsoft product *Transaction Server* [Mic97] is to facilitates the development of fault tolerant server applications using ActiveX components [Cha95]. The product provides a framework for developing transactional applications with ActiveX components. The developer is working in a virtual single user environment and does not have to consider problems like concurrent data accesses. The framework is doing the synchronisation. In this approach all components must be implemented as resource manager, that means, they must be transactional. The functionality of the workflow system must be programmed in components and cannot be modelled. The application developer can only create component–based applications, that means, the finished applications cannot be adapted by the end user.

9 Conclusion

Workflow systems are a large and rewarding application domain for component–oriented systems. The approach to use applications built with components in workflow activities solves several problems current workflow systems are faced. To make that possible cooperation between these two systems is needed.

An important requirement in such a coupled system is fault tolerance, which can be achieved with flexible fault handling concepts.

Fault handling can take place on all layers of the coupled system. Some fault handling concepts can be used on more than one layer. If a layer can't cope with an fault, it passes the fault to the next layer.

The concept of catching a fault and doing an alternative action requires no precondition to the elements of the inner layer. The elements does not need to have certain properties.

The concept of transaction, compensation and manual intervention cannot be implemented independently, that means without cooperation, on the inner elements. These concepts requires certain properties.

This means that the property of fault tolerance generally cannot be added to a component application by adding fault tolerant components. The components often need some minimal properties. Separating completely functionality from fault tolerance is not possible. This expose a violation of the model of independently extendable systems.

References

[BH97] Baggiolini, Vito ; Harms, Jürgen: Toward Automatic, Run-time Fault Management for Component-Based Applications. **In:** Weck (Hrsg.) et al. (siehe [WBS97])

[Cha95] Chappel, David: *Understanding ActiveX and OLE*. Microsoft Press, 1995

[GR93] Gray, Jim ; Reuter, Andreas: *Transaction Processing*. Morgan Kaufmann, 1993

[Jab95] Jablonski, Stefan: *Workflow-Management-Systeme: Modellierung und Architektur*. erste. Bonn : Thomson Publ., 1995 (Thomson's aktuelle Tutorien 9)

[Ley95] Leymann, Frank: Workflows Make Objects Really Useful. **In:** *Proceedings of the 6th International Workshop on High Performance Transaction Systems (HPTS)*, 1995. – http://www3.hursley.ibm.com/hpts95/proc95.htm

[LR94] Leymann, Frank ; Roller, Dieter: Business Process Management With FlowMark. **In:** *Proc. 39th IEEE Computer Society Int. Conference (CompCon)*. San Francisco, Cal. : IEEE, Februar 1994, S. 230–234

[Mic97] Microsoft Corporation: Microsoft Transaction Server — White Paper. 1997. – Forschungsbericht http://www.microsoft.com/transaction

[Obj97] Object Management Group (OMG): *The Common Object Request Broker Architecture and CORBA 2.0/IIOP Specification (Revision: 2.0)*. Februar 1997. – http://www.omg.org/corba/corbaiiop.htm

[SB96a] Schreyjak, Stefan ; Bildstein, Hubert: Beschreibung des prototypisch implementieren Workflowsystems Surro / Universität Stuttgart, Software–Labor, Fakultätsbericht Nr. 1996/19

[SB96b] Schreyjak, Stefan ; Bildstein, Hubert: Fehlerbehandlung in Workflow–Management–Systemen / Universität Stuttgart, Software–Labor, Fakultätsbericht Nr. 1996/17

[Sch97] Schreyjak, Stefan: Coupling of Workflow and Component–Oriented Systems. **In:** Weck (Hrsg.) et al. (siehe [WBS97]), S. 77–86

[TV97] Troya, J. M. ; Vallecillo, A.: On the Addition of Properties to Components. **In:** Weck (Hrsg.) et al. (siehe [WBS97])

[WBS97] Weck, Wolfgang (Hrsg.) ; Bosch, Jan (Hrsg.) ; Szyperski, Clemens (Hrsg.): *Proceedings of the* 2^{nd} *International Workshop on Component–Oriented Programming (WCOP'97)*. Turku Centre for Computer Science, September 1997 (TUCS General Publications No 5)

[Wor94] Workflow Management Coalition: The Workflow Referenz Model. 1994. – Forschungsbericht Document Number WFMC-TC00-1003, Issue 1.1, http://www.aiai.ed.ac.uk/project/wfmc/DOCS/refmodel/rmv1-16.html