Towards Architecture-Level Middleware-Enabled Exception Handling of Component-based Systems

Gang Huang, Yihan Wu

Towards Architecture-Level Middleware-Enabled Exception Handling of Component-based Systems

Gang Huang, Yihan Wu
Key Laboratory of High Confidence Software Technologies, Ministry of Education
School of Electronics Engineering and Computer Science, Peking University,
Beijing, 100871, China
huanggang@sei.pku.edu.cn

Abstract

Exception handling is a practical and important way to improve the availability and reliability of a component-based system. The classical code-level exception handling approach is usually applied to the inside of a component, while some exceptions can only or properly be handled outside of the components. In this paper, we propose a middleware-enabled approach for exception handling at architecture level. Developers specify what exceptions should be handled and how to handle them with the support of middleware in an exception handling model, which is complementary to software architecture of the target system. This model will be interpreted at runtime by a middleware-enabled exception handling framework, which is responsible for catching and handling the specified exceptions mainly based on the common mechanisms provided by the middleware. Though the approach is general enough for almost all middleware-enabled systems, the framework is specific to the concrete middleware. Consequently, we demonstrate the approach in JEE (Java Platform Enterprise Edition) application servers and experiment on JEE benchmark system. We believe that this architecture-level exception handling approach, together with the classical code-level approach, can handle exceptions in component-based systems in a sufficient, efficient and flexible manner.

Keywords

Exception handling; component-based system; software architecture; middleware; JEE

1. Introduction

Exception handling is always a challenging task in software development. It is usually considered at the code level with the special programming language support, e.g. try and catch clauses. However, such code-level exception handling is not sufficient, efficient and flexible enough for today's component-based systems.

Firstly, in typical component-based software development [20], components are usually implemented by third parties who are independent with each other. Component providers cannot predict all possible usages and contexts of a component and then cannot handle all exceptions the component faces in different systems. On the other hand, the assemblers have the knowledge of the target system as well as the responsibility of exception handling of the whole system. However, it is very difficult for these assemblers to read and modify the source code of third-party components. Moreover, many components, like the Commercial-Off-The-Shelf components and web services, are black boxes and their source code can only be read and modified by their providers. Therefore, code-level exception handling is insufficient, inefficient and even infeasible for component-based systems.

Secondly, today's component-based systems usually run on middleware, which encapsulates plentiful solutions for common problems in software development and operation, such as component lifecycle management, concurrency control, interoperability and security. Since the middleware itself is very complex (having millions lines of code), it is likely to cause exceptions [11]. These middleware-caused exceptions are usually thrown as system errors and unable to be handled by application components. Meanwhile, being a runtime space for the components, middleware provides plentiful mechanisms for system management. These management mechanisms could be used to handle exceptions, e.g. resending a request and rebooting a component. But they are usually used by system administrators and ignored by component providers.

Last but not the least, changeability is a nature of component-based systems. Because code-level exception handling is hard-coded in the components, both trivial and significant changes of components and their interactions will cause a re-implementation of exception handling. On the other hand, exception handling itself has to be revised or enhanced time to time because it is impossible to handle all exceptions perfectly at one time. Changes of code-level exception handling will change the code of the involved components. Furthermore, such code changes require re-compile and re-deployment, which put negative impacts on the quality of service.

In our previous work [13][16], we proposed a systematic approach to handling exceptions raised from middleware services. The services, such as communication, transaction, security and persistence, integrated by the middleware are considered as black-box components of the whole system and their exceptions are handled by the middleware's adaptive mechanisms. This approach is demonstrated on open source JEE (Java Platform Enterprise Edition) [19] application servers, including JBoss, JOnAS and PKUAS. Almost all exceptions raised by JEE standard services can be efficiently handled. We find that this approach does not suffer the above limits of code-level exception handling. However, since the type and number of middleware services are relatively few and seldom changed, middleware providers (i.e. assemblers of middleware services) can handle all exceptions by themselves and the exception handling solutions can be reused in all systems running on the middleware. On the contrary, application components are diverse, numerous and ever-changing. Their exceptions are usually handled case by

case and the architects or assemblers of an application need a more user-friendly and application-oriented exception handling approach.

In this paper, we propose a middleware-enabled approach for exception handling at architecture level. First of all, it is complementary to code-level exception handling. If the assemblers or architects find some exceptions unable or unfit to be handled inside components, they can handle these exceptions by the model of software architecture, i.e. outside the components. In an exception-handling model, the assemblers or architects can specify what exceptions should be handled and how to handle them with the mechanisms provided by the middleware. In the middleware-enabled exception handling framework, the specified exceptions can be caught when the middleware delivers the messages. The corresponding handling plan will be interpreted by invoking the middleware mechanisms. Compared to that the code-level approach scatters and embeds exception handling into components, our approach separates exception handling from components, designs it at architecture level and implements it with the support of middleware. Since the exception handling is described in the architecture model and interpreted by the middleware, it is flexible enough for the changes of components and exceptions. For illustration, demonstration and evaluation, we implement this approach in JEE application servers, which are popular middleware for today's component-based systems.

The remainder of the paper is organized as follows: Section 2 gives the rationale and an overview of the proposed approach; Section 3 describes how to model exceptions and their handling strategies in the exception handling model; Section 4 shows how to interpret and execute the exception handling model in JEE. We give an experimental validation in Section 5 and an overview of related work in Section 6. At last, we give an insight into conclusions and future work in Section 7.

2. Approach Overview

2.1 Exceptions to be Handled

Just like the popular exception handling in modern programming languages, e.g. Java and C++, a component usually defines some exceptions in its interface definition. Our approach focuses on such explicitly defined exceptions in a component-based system. In these exceptions, which one will be handled is determined by the assemblers or architects.

An exception indicates a state transition from normal to abnormal. It is not easy to understand the exact meaning of an exception and it is hard and even impossible for developers to understand all exceptions of the target system. As a result, although developers try their best, usually they can only handle a part of all exceptions. If the target system contains more third-party or black-box components, it is harder to handle all possible exceptions. In this sense, the completeness of exception handling mainly relies on developer's experiences, skills and efforts. On one side, our approach does not require the assemblers or architects to handle all exceptions. We only throw warnings if some exceptions in a component interface definition have no corresponding handlers. On the other side, our approach does not restrict developer's freedom of what exceptions should be handled inside or outside a component. If developers want to handle an exception inside a component, they can use the classical code-level approach. If they cannot handle the exception inside a component or consider the code-level approach is unsuitable, they can use our architecture-level approach.

2.2 Mechanisms for Handling Exceptions

Handling an exception means a state transition from the abnormal one to a normal one. For a component-based system, besides the code modification inside a component, two types of handling techniques can enable such a transition usually:

- Restore the component's abnormal states to normal ones. Because not all internal states of the components can be accessed, the technique often focuses on some particular states such as the initial states. Rebooting the failed component is such a technique.
- Do not try to restore the component's states but either retry the failed requests again or switch to an alternative that is correct. Such retrying and switching are always transparent to all involved components.

From the perspective of software-implemented fault tolerance, these techniques are typical error recovery or fault recovery mechanisms. In addition, exception catching is the process of error detection. A typical fault-tolerant mechanism takes two successive steps to tolerate faults: the error detection step aims to identify the presence of an error, while the recovery step aims to transit abnormal states into normal ones (some masking-based fault-tolerant mechanisms do not take the recovery step). Since today's middleware always provides plentiful fault-tolerant mechanisms, the exceptions in application components can be handled efficiently by the middleware. In our JEE demonstration, there is no architecture-level exceptions unable to be handled by JEE mechanisms.

2.3 Correlated Exceptions

There are different exception types in the scope of our study: individual exceptions, correlated exceptions, and concurrent exceptions. Each type corresponds to a specific recovery strategy.

- Individual exceptions are the simplest one. If a component throws an individual exception, the exception can be handled by recovering the component's states or just switching the requests to another component. No other component is involved in such a handling process.
- Correlated exceptions are raised sequentially in reverse order of the invocations. The first exception is the root cause of the fault and others are the consequence. If the former exception is not handled, it will propagate to the caller component and cause another exception. In a component-based system, an exception is propagated through the reply to the request. Since the middleware is responsible for delivery of all request and reply messages, it can catch all raised exceptions and stop their propagation by activating exception handling mechanisms. In that sense, if assemblers specify handling mechanisms for the root cause exception and can handle the exception successfully, the correlated exceptions will never occur. If no handling mechanisms are specified for the root cause exception or the handling fails, the correlated exceptions will occur. Such correlated exceptions have to be handled in a coordinated way, i.e. recovering all the components involved in the exceptions as a whole.
- Concurrent exceptions are those raised concurrently by different components [22]. They happened in distributed systems and due to resource competition or the conflicts in a design to achieve a global goal, instead of explicit invocations. Concurrent exceptions can be handled by a global recovery, i.e. performing a

universal activity instead of just handling the caught exceptions.

2.4 Architecture-Level Exception Handling

Based on the above observations and analysis, we propose a software architecture (SA) level exception handling approach for assemblers or architects (see Fig. 1). SA is widely used in component-based software development and is good at modeling interactions among components, in which exceptions may be raised. Since SA usually specifies the system at normal states but exception handling deals with abnormal states, we propose an exception handling model as a complementary part of the SA model of the target system.

An exception handling model describes when to handle the exceptions and which handling strategy should be taken. Specifically, an exception handler style, which is an architectural style for exception handling, describes a generic exception handling strategy corresponding to exceptions. To deal with the exceptions outside the components, assemblers or architects use the exception handling model to correlate an exception and an exception handler style. During the above configuration, an SA model, which has been constructed according to the functional requirements, helps assemblers or architects to grasp the relationships between provided services and required services of a component, and information about thrown exceptions, such as causes and induced failures.

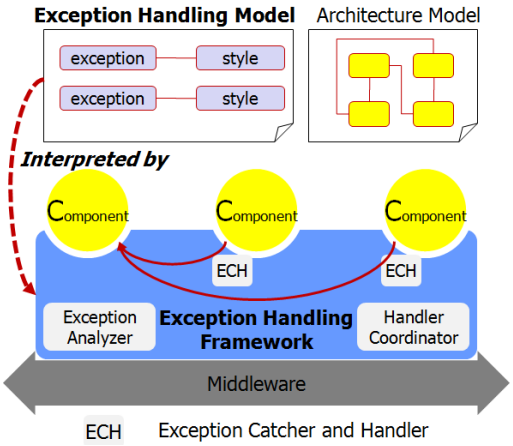


Fig. 1 The architecture-level exception handling approach

At the deploy time, the exception handling model is interpreted by the middleware, and the corresponding exception catching and handling mechanisms are configured in an exception handling framework. At runtime, if an exception is raised, it will be encoded into the invocation reply message and then caught by the framework. The framework will analyze the exception for identifying the root method, match it with the exceptions defined in the exception handling model, and activate the corresponding handler style if it is defined. If the handling succeeds, the abnormal states are transferred to normal ones and the raised exception becomes transparent to the client component. If it fails, the exception will be passed to the client in the usual way and the failed handling is also transparent to the client. The basic exception catcher and handler deal with the individual exceptions, while they can deal with the correlated and concurrent exceptions with the help of handler coordinators.

3. Exception Handling Model

The exception handling model is a complementary part of the software architecture model of the target system. There are two main parts in an exception handling model, i.e. an exception description and a set of exception handler styles. Because the exception handler styles describe how to deal with exceptions and they are the core of the exception handling model, we explain this part at first in the following sub-section.

3.1 Exception Handler Style

Once a raised exception is caught, it should be handled by a proper strategy. The typical handling strategies include rolling back to the initial states, retrying an operation using the original or alternate resources, and masking an exception by an alternative. We specify them as exception handler styles in this paper. An exception handler style is an architectural style. It follows the classical definition of architectural styles: a set of constraints on coordinated architectural elements and relationships among them; the constraints restrict the role and the feature of architectural elements and the allowed relationships among those elements in an SA that conforms to that style [17].

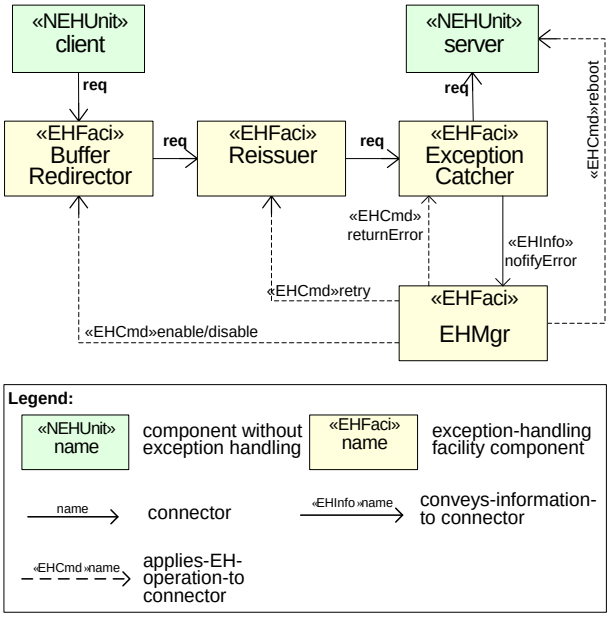


Fig. 2 The Micro-reboot style.

Exception handler styles specify the structural and behavioral characters of the available handling strategies for components. Although different handling strategies have different structure and interactions with application components, their structural unit and primary activities are similar. Entities involved in a strategy can be specified as components, interactions among the entities can be specified as connectors. The activities in a handling strategy include controlling the messages passed in or received from a component, and monitoring or modifying a component's states. The constraints of the organization of the entities and the temporal order of the activities can be specified as an exception handler style. By analyzing the style, we can ensure the desired properties such as correctness, quality of service contracts, transactional integrity, or safety. It should be noted that the idea of exception handler style is derived from fault tolerant styles [16][21][23], which abstract the usual fault tolerant solutions and enable them reusable in many distributed systems. As mentioned above, exception handling and fault tolerance are similar from the perspective of middleware. We believe that abstracting exception handling solutions as architectural styles makes them reusable in component-based systems. In particular, we abstract the usual mechanisms provided by the middleware as exception handler styles and then the architecture-level exceptions can be handled by the middleware inherently.

Fig. 2 shows the exception handler style of the micro-reboot strategy. We use UML 2.0 component diagram to specify the structure of an exception handler style since UML is widely

used and easy for communication. It should be noted that such styles could be defined in other model languages, e.g. those architecture description languages. There are two kinds of components in the style: «NEHUnit» components are the original application components without configuring architectural-level exception handling capabilities; «EHFaci» components are pre-fabricated facility components that provide exception handling capabilities for «NEHUnit» components. There are also two kinds of connectors: «EHInfo» connectors are responsible for conveying a component's states to another; «EHCmd» connectors are responsible for changing an «NEHUnit» component's states. An exception handler style specifies how a set of «EHFaci» components, «EHInfo» connectors and «EHCmd» connectors are interacted with a given «NEHUnit» component to handle its exceptions in the latter.

The micro-reboot style consists of several «EHFaci» components (*ExceptionCatcher*, *Reissuer*, and *BufferRedirector*) and an «EHCmd» Reboot connector for a «NEHUnit» component. The *ExceptionCatcher* catches all exceptions specified by the architects or assemblers in the «NEHUnit» components. After the caught exceptions are analyzed and the failed component is identified, the failed component will be rebooted. Meanwhile, the *BufferRedirector* blocks incoming requests for the component during recovery. When the failed component is recovered successfully, the *BufferRedirector* re-issues all blocked requests and the normal process is resumed.

The other popular exception handling solutions can be specified in the same notation. The Simple Retry style is similar to the Micro-reboot style except it only re-invokes a failed component again, without rebooting it. The Recovery Blocks style is similar to the Simple Retry style except it uses another alternative component to process the inputs again. The N-Version Programming style uses several «NEHUnit» components simultaneously that implement the same functions, in order to mask exceptions in one component. The details of these exception handler styles are omitted for the space limit.

3.2 Coordinated Exception Handler Style

The above exception handler styles deal with exceptions that affect only one component. To handle the exceptions affecting more than one component, i.e. correlated and concurrent exceptions mentioned in Section 2.3, these styles should be enhanced with coordination mechanisms. The coordination mechanisms are specific to the cause of the exceptions. In general, the cause of correlated exceptions is the reference-based dependencies among components, and the cause of concurrent exceptions is the temporal-related dependencies among components.

Reference-based dependency exists when a component invokes another component's services by a reference. We say that the former depends on the latter, and call the former as *depending* component and the latter as *depend-on* component. Usually a depending component C_1 cannot invoke the depend-on component C_2 in component-based systems, until after C_1 gets a valid reference to C_2 via looking up. Since the looking up operation is time-consuming, developers usually write the code for looking up C_2 in C_1's initializing function and keeping C_2's reference if the invocation is frequent. Then C_1 invokes C_2 directly by the reference. However, such a manner incurs exceptions that affect both C_1 and C_2: if C_2 failed, the reference stored in C_1 becomes invalid immediately. Although exceptions in C_2 would be caught by the middleware, as we explained in Section 2.3, there is a latent fault in C_1 because any new invocation from C_1 to C_2 by the reference would cause an exception. Heavy-weight mechanisms, such as recursive reboot [3], recover both C_2 and C_1 sequentially. Light-weight mechanisms only update the reference in C_1, without recovering the component. Both of the mechanisms ensure the consistency of system states before and after handling. Fig. 3 shows the micro-reboot style with the lightweight mechanism for coordinated exception handling.

Temporal-related dependency exists when several exceptions are raised concurrently. It should be noted that there are no invocations between these components because a raised exception will be caught immediately by the middleware and then it does not cause another exception via invocations. However, Xu et al. explain the situation that although the handling strategy for each exception is defined, the strategy for the concurrently raised exception is a universal one, which covers all these exceptions [22]. Such a universal exception can be identified manually by the architects or automatically by the exception-resolving algorithm proposed by Xu et al. Regardless of how to identify a universal exception, its handling belongs to coordinated ones and needs a handler style with coordination mechanisms.

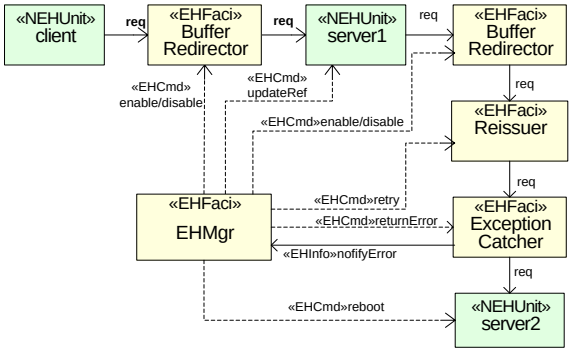


Fig. 3 The Micro-reboot style with coordination mechanisms

3.3 Details of Exception Handling Model

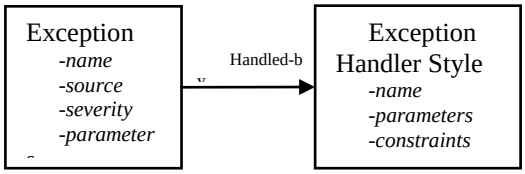


Fig. 4 Exception handling model

The exception handling model provides information including the source and the severity of an exception, and the exception handler style to deal with the exception. As shown in Fig. 4, we divide the model into two parts: an exception description part describes the name, source and parameters, and an exception handler part describes the name of a style used in face of some kinds of exceptions, the parameters, and the constraints.

The elements in the exception description part are as follows:

- *Name*. Each exception has a name. The name may not be unique, and it combines with the name of the component raised the exceptions to identify an exception uniquely.
- *Source*. Each exception has a source component, which raises the exception. The source is designated by the component name, interface name and method name together to specify in which method of the component interface the exception is raised.

- *Severity*. Intuitively, not all exceptions have the same severity. Some exceptions cause immediate system crashes whereas others only cause less catastrophic consequences. In addition, the more important a component in an application is, the more severe the exception in the component is. If multiple exceptions are raised simultaneously at some time or their handler styles compete or conflict with each other, the exception with higher severity will be handled earlier.

- *Parameter*. Additional information of the exception is embodied as a set of parameters and corresponding values, such as the causes of the exception, details about the states of the component when the exception occurs, etc. Some parameters may only be descriptive, and others may be valuable for selecting an appropriate handler style. For example, if a parameter *frequency* states that the exception is often raised, Simple Retry style is not suitable for this exception because it is likely to meet the exception when repeat the request.

The exception handler part describes which style will be used to handle a raised exception. The elements in this part are as follows:

- *Name*. Each style has a unique name to identify itself and it will be used in the binding with exceptions.

- *Parameter*. Because an exception handler style is defined as a reusable utility, there are some parameters to identify the target of operations and designate additional information about the exception.

- *Constraint*. A set of constraints will be defined in this section to constrain the usage of the exception handler style in terms of *PreCondition*, *PostCondition* and *Invariant*. In addition, a timing constraint plays a large role in determining what types of handling strategy are feasible. For example, a timing constraint may demand the frequency at which checkpoints must be taken, or whether a system can reboot itself quickly enough to prevent an overall system outage. The satisfaction of these constraints ensures the correctness and consistence of handling actions and system states.

We separate the description of exception and style and correlate them via binding, mainly because one style can handle more than one exception while one exception can be handled by more than one style. If more than one style is attached to an exception, there must be some conditions to decide the selection of different styles. In the description of a condition, parameters and constraints can be used to distinguish different styles.

3.4 Case Study on ECperf

ECperf is an EJB (Enterprise JavaBean) benchmark, which runs upon a JEE application server. It simulates a typical process of manufacturing, supply chain management, and order-inventory management. We consider a typical business scenario in ECperf – creating a new order. In this scenario, a customer issues a *newOrder* request, which may contains several product items, to an *OrderSes* session bean. Then an *OrderEnt* component creates a new *OrderLineEnt* entity bean, accesses an *ItemEJB*, and checks the customer's discount and credit using *CustomerEnt* in *Corp* domain. The architecture of EJBs in the scenario is shown in Fig.5.

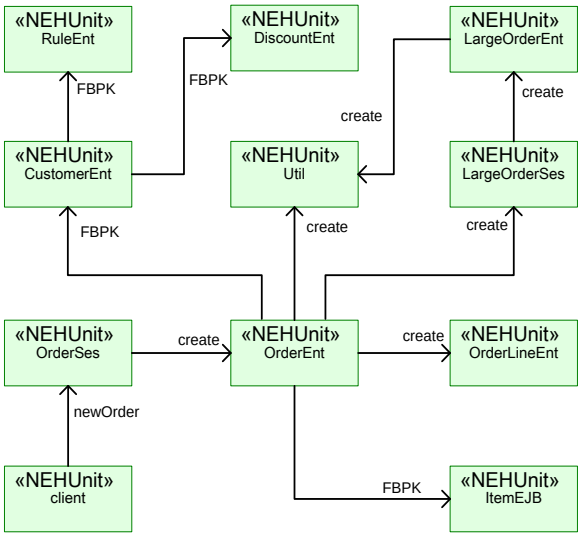


Fig. 5 The architecture of ECperf

The ItemEJB component is frequently used in the scenario. There are more than 400 invocations for the ItemEJB when creating a new order. The availability of the ItemEJB depends on both its implementation and the data service in JEE application server, which would be imperiled by database failures or unreliable database connections. The data service provides a standard way to use the database and to manage database connection pools to gain better performance. Connection-related operations, such as getting or closing a connection, are executed by pools, and concrete database operations, such as querying and updating, are performed by specific JDBC (Java Database Connectivity) driver classes provided by database venders. The above situation requires assemblers to create an exception handling model for ItemEJB. We assume assemblers have enough knowledge of the middleware or the middleware can represent itself as some abstractions at architectural level (e.g., we abstract the middleware adaptive mechanisms as manipulation primitives of software architecture model in [6]). In addition, we have proposed a model checking-based approach to selecting the suitable style for third-party components [12]. As a result, assemblers will find out that the Micro-reboot style is a potential resolution for such case. Moreover, existing architecture model shows that OrderEnt preserves a reference to ItemEJB and OrderSes preserves a reference to OrderEnt. Fig. 6 shows the component dependency graph that depicts the above relationships. So, the ItemEJB, OrderEnt and OrderSes should be handled in a coordinated manner.

Fig. 7 illustrates the exception handling model for the ItemEJB. *DataServiceUnavailableException* is the exception name and it is raised by the ItemEJB. The cause of the exception is unavailability of the data service in JEE application server. In the exception description part, the components and the interfaces throw these exceptions are given. It should be noted that if we do not designate the interface and method in the description of the exceptions, it means all public methods in the component will raise the exception. All these exceptions are serious because the exception causes failures when creating a new order. The micro-reboot style deals with the exceptions raised by ItemEJB, which require a coordinated recovery because there are reference-based dependencies from OrderSes to OrderEnt,

and from OrderEnt to ItemEJB. The constraints for the above handler style require that the enforcement time of the micro-reboot style is both deploy time and runtime.

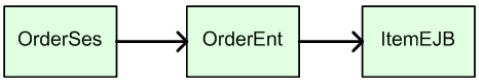


Fig. 6 The component dependency graph for ECperf in the scenario of creating a new order

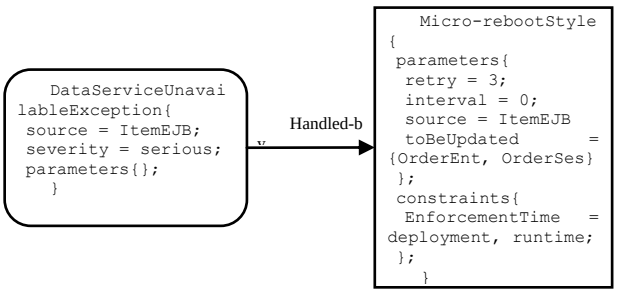


Fig. 7 The exception handling model for ItemEJB

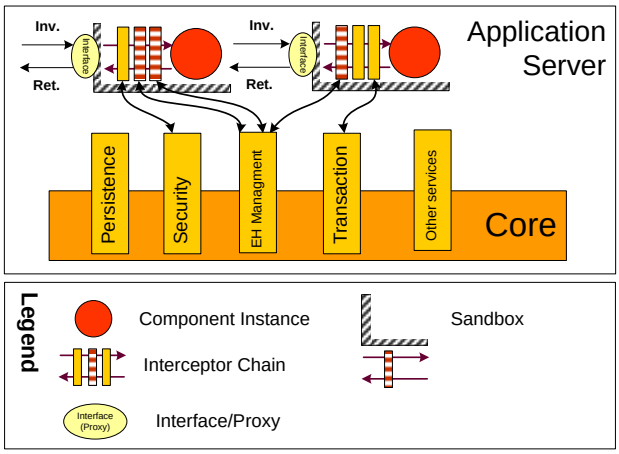
4. Middleware-enabled Framework

To support the interpretation and the execution of exception handling models, we implement a framework in JEE application servers. This middleware-enabled framework reads the configuration file (corresponding to the above exception handling models), configures the exception-handling facilities, and performs handling operations when the exceptions are raised. It should be noted that the framework is based on the de facto standard elements of JEE application servers, including component containers, container interceptors and common services. Since we always experiment on JBoss, JOnAS and PKUAS, we argue that this framework is general enough for almost all JEE application servers and can be applied to other types of middleware.

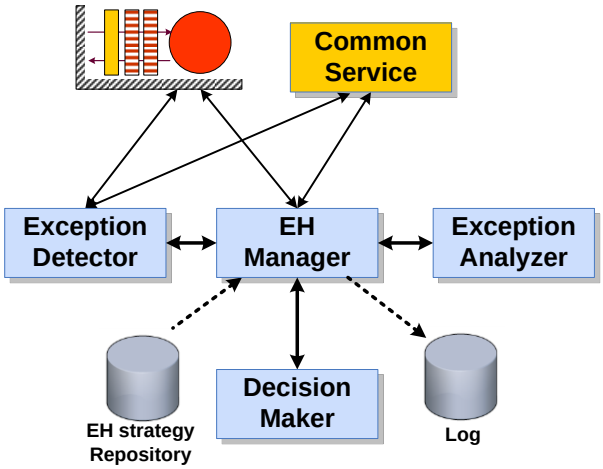
4.1 Framework Overview

The design and implementation of the framework are built atop the component container and interceptor design, which is prevalent in modern middlewares, such as EJB container, CORBA container, and Spring container for Plain Old Java Objects. A container provides an execution environment for components and ensures them to use middleware services at runtime. Containers mediate all client/component invocations, and interceptors in a container work as filters for the invocations. Multiple interceptors in a container form an interceptor chain, where each does some specific processing on an incoming invocation before passing it to the next interceptor. The last interceptor in the chain invokes the component's business method. The ordinary processing in an interceptor includes checking authorization, modifying invocation's parameters, starting/committing transactions, etc. Common services in application servers provide both applications and application developers with good support in common functions such as communication, transaction and security. The coordination of exception catching and handling, and the coordination of multiple handling activities in multiple application components are to be controlled by an exception handling management service, which is also a middleware service.

The framework consists of three parts: sandboxes for application components to catch and handle exceptions, built-in mechanisms for middleware services to catch and handle exceptions, and an exception handling management service. The concept of sandbox comes from Java programming language's sandbox model for security [4], in which untrusted codes are only allowed to run in a restricted environment to isolate potential security threats. Similarly, a sandbox (see Fig. 8 (a)) is a component's execution environment, which is extended from application server's containers. They manage invocations to/from the hosted components, catch and confine exceptions thrown by the components, and handle these exceptions in a transparent way.



(a) Sandboxes in application servers.



(b) The structure of exception handling management service.

Fig. 8 The framework for architecture-based exception handling.

By extending the component container and interceptors, the sandboxes are the units of exception catching and handling. Setting down these functions at sandboxes level, instead of at application level, decreases the Mean-Time-To-Recover, and hence obtains a better availability. The sandboxes support multiple exception handling styles. Its contribution is not the creation of new styles but the uniform integration of well-known styles. Different exception handler styles, such as Micro-reboot, Simple Retry, and Recovery Blocks, are implemented as a set of well-organized interceptors and operations in sandboxes.

The exception handling management service (see Fig. 8 (b)) plays a central role in coordinating exception catching and handling in sandboxes and services, and in coordinating handling activities in multiple sandboxes. When a sandbox

throws an exception, the exception handling management service will be notified and then activates an exception analysis process and handling activities according to the given exception handling model. The process of exceptions in services is the similar. The exception handling management service is also responsible for coordinating handling activities of multiple sandboxes if necessary. It performs a sequence of handling operations one by one and each operation recovers a component. The correspondences of the entities in exception handler style and mechanisms in application servers are shown in Table. 1.

As a supporting part, we also provide similar service-level exception handling mechanisms to improve the availability and reliability of services in application servers. Because the invocations between component containers and services, as well as between services are direct calls, it is hard to catch the exceptions outside of the service. As a solution, we use dynamic aspect programming to weave a try-catch block around the service methods to catch the exceptions. This aspect is called the *exception catcher aspect* in the paper. The try clause works as a wrapper for service methods, and the catch clause works as an exception notification for the exception handling management service. Because of the complex semantic dependency between different services in different application servers, we modified the standard services to make them recoverable. The services include Java authentication and authorization service (JAAS), Java Naming and Directory Interface (JNDI), Java Message Service (JMS), data service and transaction service. The details of the modification are given in our previous work [16].

4.2 Exception Catcher

The prerequisite of exception handling is to catch the exceptions and prevent them from propagation. Two kinds of facilities in the framework are provided to catch the exceptions in application components and middleware services.

The ExceptionCatcher interceptor in sandboxes is responsible for catching all unexpected exceptions in application components. It is a build-in interceptor also for all sandboxes to confine raised exceptions. Because an exception raised by a component can be handled only after being caught, the ExceptionCatcher interceptor is placed in a container interceptor chain, very near to the component. If the ExceptionCatcher interceptor is placed ahead of other exception handling interceptors, these interceptors would fail before the ExceptionCatcher catches the exception. The ExceptionCatcher interceptor would notify the exception handling management service.

The EH manager in the exception handling management service is responsible for receiving the caught exceptions. It calls the Exception Analyzer to identify the type and the source of the exceptions. Exception tracing provides valuable information to analyze the cause because exceptions are thrown in an inverse order of invocations. The top exception in an exception trace stack is the root cause. For exceptions that are caught individually, the Exception Analyzer compares the exception information with the name and source in the exception handling models, in order to identify the exception and the corresponding handler strategies. For exceptions that are caught in a very small interval, the Exception Analyzer invokes an exception tree analysis to identify whether they are concurrent exceptions.

4.3 Exception Handling

An exception handler style is responsible for handling a kind of caught exceptions. This architectural abstraction for handling strategies is implemented in interceptors and operations in application servers. There is a one-to-one mapping relationship between the entities in exception handler styles and the modules in application servers. The «EHFaci» components and «EHCmd» or «EHInfo» connectors in the styles are mapped to a set of well-organized interceptors and operations, and a set of exception handling management operations.

The «EHFaci» BufferRedirector component in exception handler styles helps to minimize the side effect of exceptions and provides transparent handling. It buffers the requests for a component after catching an exception, and redirects them after the component is recovered. Because a client's request obeys request-reply pattern in application servers, the order of requests issued by a client is not disturbed when using BufferRedirector. This means a client would not send the next request unless the former response is received. In application servers, a thread pool is used usually to minimize the performance overhead of creating and releasing threads. A thread is put back to the pool, instead of being released after finishing its execution. The requests for the components are placed in a queue of the pool, waiting free threads. The BufferRedirector in exception handler styles is implemented in the framework by stopping the assignment of free threads to the waiting requests so that the requests will wait until the recovery finishes or the request is timeout. This

Table. 1 The correspondences of entities in exception handler styles and mechanisms in application servers.

Entities/Activities in exception handling	*Entities in exception handler styles*	*Mechanisms in AS*
The buffer/redirector facilities	«EHFaci»BufferRedirector	Thread pool management operations
The components to backup or propagate the requests	«EHFaci» component	Exception handling interceptors in the sandbox.
The operations to read/write component states	The state-access interface of the application components	The state-access interface provided by the component, or the life cycle management interface provided by the AS
The decision-making procedure	The interfaces of the «EHFaci»EHMgr	The functions provided by EH Manager

implementation has a benefit that any request can be cached and the caching mechanism need not be implemented for every application component or in every service separately.

The «EHFaci» Reissuer component in exception handler styles helps to minimize the failed requests and hence improves user availability except a little delay. If an exception is caught, the requests under execution can be canceled at that time and re-executed after the exception is handled. This function does not apply for all requests but only for idempotent requests, which allow to re-execute the same request more than once. The Reissuer for application components is implemented as a container interceptor for backing up the invocation and resending the invocation by the instruction of the exception handling management service. The Reissuer for services is specific to service type. For example, the Reissuer for data service is implemented by re-do capability, which tries to re-do the failed request for data service again.

There are two typical operations to deal with the root cause of the exceptions: recovering components' or services' states, or masking the failed components or services. The Micro-reboot style uses the first kind of operations. It tries to restore a component's or a service's abnormal states to the initial states. The component' states are restored by un-deploying and re-deploying the component. This is a generic operation for all application components. However, the service's states cannot be restored by a generic operation because of the difference in different implementations. Although they are modified case by case, we present some common ideas to modify services in spite of some specific code modifications in our previous work [16].

The exception handler styles using the second kind of operations, such as the Recovery Block style, are supported by the component array [13]. A component array is a group of well-organized components that implement the same functions but provide different qualities due to different implementations, different efforts in development, different investments, and so on. The component array can mask a failed component by switching to another implementation in the array. It is also implemented in sandboxes as a generic mechanism for application components. The support for multiple versions of the same service is similar, except there is not a generic mechanism for all services.

The exception handling management service is responsible for linking the exception catching and handling activities. When the ExceptionCatcher interceptor in a sandbox or the entities in the exception catcher aspect notifies the EH manager of an exception, the EH manager activates the Exception Analyzer to identify the corresponding handler style. Then the EH manager triggers handling activities according to the style. The EH manager is also responsible for coordinated handling of multiple sandboxes or multiple services if necessary. It performs a sequence of handling activities one by one and each activity handles an exception. The dependencies among components or services (explained in Section 3.2) decide the order of the activities.

The above framework is general enough for JEE application servers. We implemented it in PKUAS [15] at first, and then some parts of the framework have already migrated to JBoss and JOnAS, the top two open source JEE application servers, without any fundamental modifications.

5. Experimental Validation

To validate the effectiveness of the proposed approach, we perform a set of controlled experiments on ECperf. We inject Java exceptions to simulate faults in the ItemEJB and faults in the data service in PKUAS. An automatic testing tool generates multiple virtual clients to trigger the Create-A-New-Order transaction concurrently. The number of successful transactions is collected as availability metric.

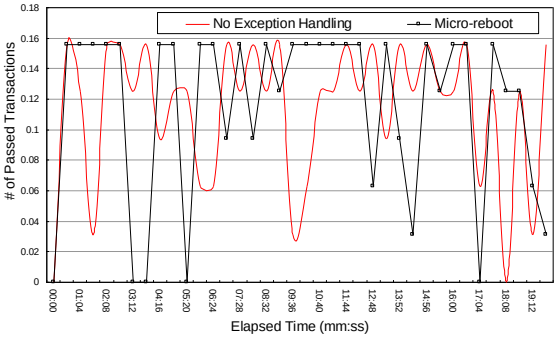


Fig. 9 The Micro-reboot strategy handles exceptions in ItemEJB.

According to the exception handling model for ECperf presented in Section 3.4, the micro-reboot style is configured for ItemEJB, OrderEnt and OrderSes components. We inject environment-independent and transient faults into ItemEJB to simulate the failure of the data service. For the sake of comparison, the original ECperf and PKUAS run as a baseline case, in which we take an optimistic assumption that the ItemEJB would resume execution immediately after the faults disappear. The numbers of passed transactions of two cases under the faults are show in Fig. 9. Every trough in the figure stands for a pause of processing or a failure, due to the injection of the fault. The rate of successful transactions for the micro-reboot style case is 85.3%, compared with 76.5% in the baseline case.

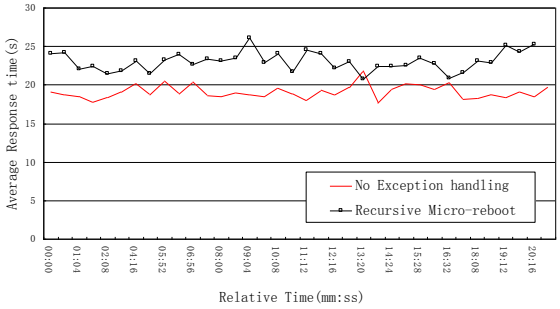


Fig. 10 The comparison of response time.

It should also be minded that the exception handling framework induces performance penalty. When no exceptions are injected in the experiments, the implementation in PKUAS increases 2.21% of response time on average. We believe the performance penalty is acceptable. When exceptions are injected in the experiments and the micro-reboot style is applied, the implementation increases 20.45%. This cost correlates with the frequency of invocations to the components. As we assume users would bear a longer response time than a failed request, the performance impact induced by the framework is acceptable, considering about 18.8% of improved availability.

The above improvement is the case under the worst condition: we impose a heavy workload on ECperf (there are about 5%

unsatisfied requests due to the timeout when no faults are injected), and the ItemEJB is the most frequently invoked component. If the workload is lower, the success rate will increase. Furthermore, we do another experiment on the same exception but with another exception handler style, the N-Version Programming style. For simplicity, we simulate two versions of ItemEJB with the same implementation. In this style, two ItemEJB components are running simultaneously. We inject faults into one ItemEJB and the requests to this failed ItemEJB will be redirected to the other healthy ItemEJB by the handler. In this case, the exceptions do not cause any failed transaction. It should be noted that the N-Version Programming style also has its own limit: it is not easy to get multiple versions of the same component, the multiple versions have a non-trivial runtime cost if no exception is raised. In fact, if the workload is low enough, the micro-reboot style will handle all exceptions in time. Whatever, these two experiments indicate one aspect of our future work: since an exception may be handled successfully by different styles in different situations, it may be necessary to define situational correlation between the exception and the styles in the exception handling model, and the middleware-enabled framework can dynamically reconfigure the style for a given exception.

6. Related Work

There are some similar definitions of exceptions. Goodenough defines exception conditions as those brought to the attention of the operation's invoker, which becomes part of normal exit or return [8]. Cui defines exceptions as implementation insufficiencies, which exclude software errors, unanticipated program conditions, domain failures, and range failures [5]. The most commonly accepted definition of exception is the union of "error," "exceptional case," "rare situation," and "unusual event." The entity that is raising an exception stops and waits for the completion of the exception processing [18].

Exception handling usually is only considered at the implementation level, such as try-catch mechanism provided by programming languages. In component-based software engineering, code-level exception handling is insufficient, inefficient and even infeasible.

6.1 Architecture-Based Exception Handling

Exception handling at software architecture level is drawing attention in recent years [10][23][8] because it may increase the quality of software. Some existing studies take the software architecture-level reconfiguration as a strategy to handling exceptions. V. Issarny et al. consider exceptions raised owing to violations of architectural invariant, and provide an Aster environment to implement the mapping from SA-level reconfiguration to middleware configuration [10]. The interceptor facility implements the reconfiguration of component instances at middleware. Some others focus on the component models that deal with exceptions gracefully. R. de Lemos et al propose a component model [9] based on the idealized fault-tolerant model and the C2 architectural style. The model partitions components and connectors into normal and abnormal parts, and the abnormal are manifested as exceptions.

As explained by V. Issarny et al. [10], the architectural level solution complements but does not substitute to the exception handling implemented within architectural elements since they serve distinct purposes: architectural exception handling consists of changing the system's configuration for preventing further occurrence of raised exceptions; exception handling within architectural elements implements the exceptional specifications of the elements. We adopt a similar way to handle exception at the architectural level, but focus on the supporting mechanisms of middlewares. The approach proposed in our work is not specific to architectural level or component/connector level handling strategies. All depends on the exception handler style. If an exception handler style defines an architectural level handling strategy, the framework deals with the exceptions at architectural level. If it defines a handling strategy at component/connector level, the framework deals with the exceptions at component/connector level. According to the classification of fault-tolerant mechanisms [2], each handling strategy is good at dealing with one kind of exceptions and the plenty of the strategy is more likely to obtain a better fault coverage.

There are also some studies with focus on the reasoning and validation of the software with exception handling support at software architecture level. Yuan et al. use Object-Z formal language to specify an application's architecture with coordinated exception handling support as conforming to a hybrid architectural style [23]. Then, they manually prove a set of fault-tolerant properties on both the style and the resulting FTSA. The formal specification for the architecture and the exception handling mechanism, as well as the verification of the properties can give developers confidence in the fault-tolerant SA design. The exception handler style proposed in our study supports SA-level analysis either. We have already presented some aspect of the analysis in previous work [12], except we consider the style from the fault tolerance perspective in the study.

6.2 Architecting Fault-Tolerant Software Systems

The study of Architecting Fault-Tolerant Systems [1] aims to achieve better fault-tolerant software by including FT in earlier development phase to bridge the gap between the requirement to build dependable software systems and the implementation to deal with failures in the software. As one of the important fault-tolerance mechanisms, exception handling is widely used in the study of architecting fault-tolerant software systems. A notable study is the CORRECT project [4] in Luxembourg, which introduces the Coordinated Atomic Actions (CAAs) mechanism in SA specification phase. The resulting SA specification with fault-tolerance notations is transformed into CAAs model automatically and further, transformed to an implementation framework. The output of such approach is a skeleton code that satisfies the functional and fault-tolerant requirements.

7. Discussion and Conclusion

This paper proposes a middleware-enabled approach to handling exceptions in a component-based system at the architecture level. It does three contributions to exception handling: Firstly, it provides a complementary approach to the classical code-level exception handling. The exceptions that cannot be handled inside a component at all or inefficient can be handled outside a component flexibly. We believe this architecture-level approach and the code-level approach form a necessary and sufficient solution for exception handling in component-based systems. Secondly, it utilizes middleware capabilities rather than inventing new things. Since the

middleware plays a central role in today's component-based systems, this middleware-enabled approach gains better feasibility and applicability. Furthermore, since almost all non-functional issues are handled by the middleware, it becomes possible to handle these issues in a comprehensive manner. Thirdly, it abstracts the middleware-enabled exception handling solutions as architectural styles. Such styles not only make the exception handling design much easier, but also improve the reusability of exception handling solutions.

Still, the approach needs to be polished in the future. At one side, the approach just provides an enablement to handle exceptions outside a component. What exceptions should be handled and how to handle are given by developers. Therefore, how to facilitate the use of the enablement is critical for practice. For example, a powerful exception analysis support can alleviate the burden of developers in identifying which type the exceptions belong to. The consistence checking and conflict detection mechanisms help to find mistakes in the configuration created by developers. Automated synthesis of multiple styles can generate richer styles besides abstracting more styles from middleware capabilities. At the other side, the middleware-enabled framework may be more general and efficient. However, since the number of popular middleware in a period is relatively few, we argue that making a concrete middleware more powerful on exception handling is more important. For example, runtime information can be a feedback to improve the exception-handling model. This study is being carried out now, with the help of our Runtime Software Architecture [6].

8. Reference

[1] Workshop on Architecting Dependable Systems. http://www.cs.kent.ac.uk/wads

[2] A. Avizienis, J-C. Laprie, B. Randell, C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing", IEEE Trans. on Dependable and Secure Computing, 1(1), pp. 11-33, 2004.

[3] G. Candea, et al. "JAGR: an autonomous self-recovering application server", Proc. of the 5th Int'l Workshop on Active Middleware Services, Seattle, USA, pp. 168-177, 2003.

[4] A. Capozucca, N. Guelfi, P. Pelliccione, H. Muccini, "An Archi-tecture-driven Methodology for Developing Fault-Tolerant Sys-tems," Software Engineering Competence Center Technical Re-port nr. TR-SE2C-05-10, SE2C, Luxembourg, 2005

[5] Q. Cui, Data-oriented exception handling. Ph.D. thesis, Univ. of Maryland, 1989.

[6] G. Huang, H. Mei, F-Q Yang. "Runtime Recovery and Manipulation of Software Architecture of Component-based Systems." International Journal of Automated Software Engineering, Springer, 2006, 13(2): 251-278.

[7] L. Gong, M. Mueller, et al., "Going beyond the sandbox: an overview of the new security architecture in the javaTM development Kit 1.2". In Proc. of the USENIX USITS, 1997.

[8] J. B. Goodenough, "Exception handling: Issues and a proposed notation". ACM Commun. 18(12), 1975, pp. 683–696.

[9] R. de Lemos, P. Guerra, and C. Rubira, "A fault-tolerant architectural approach for dependable systems", IEEE Software, 23(2) 2006 pp.80-87.

[10] V. Issarny, and J. Banatre, "Architecture-Based Exception Handling", In Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34), Vol. 9. Washington, DC, USA, pp. 9058, 2001.

[11] J. Li, G. Huang, J. Zou and H. Mei. "Failure Analysis of Open Source J2EE Application Servers." Proc. of the International Conference on Quality Software (QSIC), USA, 2007, pp 198-208.

[12] J. Li, X. Chen, G. Huang, H. Mei, and F. Chauvel, "Selecting Fault Tolerant Styles for Third-Party Components with Model Checking Support," accepted by 12th International Symposium on Component Based Software Engineering (CBSE 2009), June 24-26, 2009.

[13] T. Liu, G. Huang, G. Fan, H. Mei, "The Coordinated Recovery of Data Service and Transaction Service in J2EE," In Proceedings of 29th Annual International Computer Software and Applications Conference (COMPSAC05), Edinburgh, Scotland, July 2005, pp. 485-490.

[14] Z. Liu, G. Huang, H. Mei, "The Model and Implementation of Component Array Container", In Proc. of 30th Annual Int'l Computer Software and Applications Conference. 2006.

[15] H. Mei, G. Huang. "PKUAS: An architecture-based reflective component operating platform." invited paper, Proc. of 10th IEEE International Workshop on Future Trends of Distributed Computing Systems, 2004, pp. 163-169.

[16] H. Mei, G. Huang, T. Liu, J. Li, "Coordinated Recovery of Middleware Services: A Framework and Experiments," Int J Software Informatics, Vol.1, No.1, December 2007, pp. 101-128

[17] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture", SIGSOFT Software Engineering Notes, 17(4), 1992, pp. 40-52.

[18] J. Rebecca, "Toward Exception-Handling Best Practices and Patterns," IEEE Software, 23(5), 2006, pp.11-13

[19] SUN Microsystems, Java 2 Platform Enterprise Edition Specification, Version 1.3, SUN Microsystems, 2001.

[20] C. Szyperski, D. Gruntz and S. Murer. "Component Software – Beyond Object-Oriented Programming", Second Edition. Addison-Wesley and ACM Press, 2002.

[21] H.Sözer, and B.Tekinerdogan, "Introducing Recovery Style for Modeling and Analyzing System Recovery", Proc. 7th IEEE/IFIP Working Conference on Software Architecture, Vancouver, Canada, pp. 167-176, 2008.

[22] J. Xu, A. Romanovsky, and B. Randell, "Concurrent Exception Handling and Resolution in Distributed Object Systems". IEEE Transactions on Parallel and Distributed Systems, Vol.11, No.10 1019-1032, 2000.

[23] L. Yuan, J.S. Dong, J. Sun, and H.A. Basit, "Generic Fault Tolerant Software Architecture Reasoning and Customization", IEEE Trans. on Reliability. 55(3) 2006, pp.421- 435.