Exception Handling for Service Component Architectures

Douglas Siqueira Leite
Institute of Computing
University of Campinas
Campinas-SP, Brazil
dougsleite@apache.org

Cecília Mary F. Rubira
Institute of Computing
University of Campinas
Campinas-SP, Brazil
cmrubira@ic.unicamp.br

Fernando Castor
Informatics Center
Federal University of Pernambuco
Recife-PE, Brazil
castor@cin.ufpe.br

Abstract—The Service Component Architecture (SCA) makes it possible to combine existing and new services based on a variety of technologies with components built using a component-based development approach. However, when asynchronous services compositions are executed, one or more errors can occur, possibly at the same time, affecting the composition’s dependability. In many cases, parts of the composition are off-the-shelf components or Web services whose individual dependability cannot be reliably ascertained. In this paper, we propose a novel exception handling model that targets the needs of dependable SCA applications. The model is applicable to service-oriented systems and allows the creation of fault-tolerant asynchronous service compositions.

We also present the EH-SCA framework, an instantiation of the proposed model implemented as an extension of the Apache Tuscany SCA infrastructure. Developers can apply this instantiation of the model to both new and existing applications by using a simple and flexible aspect-oriented programming model. Finally, a case study of the EH-SCA framework shows how it can be used to build dependable distributed applications.

I. INTRODUCTION

Service-Oriented Architecture (SOA) is an architectural model that aims to enhance efficiency, agility and productivity of enterprise businesses by structuring services and service compositions [14]. A service is defined as a self-contained distributed unit, composed of two loosely coupled elements: a specification with a provided abstract interface and an implementation. Services can be grouped to be executed in a specific order, either synchronously or asynchronously, resulting in a service composition.

Different software technologies can be used to implement the SOA paradigm, such as Web Services technology, which is based on XML-based standards, like Simple Object Access Protocol (SOAP) and Web Services Description Language (WSDL). SOA can also be implemented using a Service Component Architecture (SCA) [6], which defines a component model for implementing services and service compositions. A service implemented within an SCA component model is called a service component. SCAs support interoperability among various SOA technologies, such as Web Services, WS-BPEL, Java Message Services (JMS), JSON-RPC, CORBA, and EJB.

In particular, when asynchronous services compositions are executed, one or more errors can occur concurrently in different services, possibly at the same time, affecting the composition’s dependability. In this way, fault tolerance mechanisms are necessary in order to prevent services compositions from reaching a failure state. Considering that, in the SOA context, it is not always possible to rollback services, since they are by definition autonomous and self-contained units, one must rely on exception handling mechanisms to bring the system to a new, correct state, for providing fault-tolerant asynchronous service compositions. However, exception handling mechanisms for asynchronous services compositions should consider the fact that different exceptions types can be raised concurrently by different services at the same time. This means that different combinations of concurrent exceptions might imply in the execution of different combinations of service handlers sets. This complex error handling scenario requires that exception handling mechanisms be flexible and dynamic during runtime. Moreover, some global coordination mechanism for error handling is required, such as Coordinated Atomic actions (CA actions) [15], which provide fault tolerance by integrating coordinated exception handling, cooperative multithreading, and atomic transactions.

WS-Business Process Execution Language (WS-BPEL) [1], one of the most popular languages to create Web services compositions, does not support fault-tolerant asynchronous service compositions. When a service signals an exception within a composition, all services invocations are terminated as soon as the exception is caught, and a single handler is executed. There is no support for error handling coordination, and concurrent exception handling. Tartanoglu et al. [13] proposes a solution in terms of a structuring unit called Web Service Composition Action (WSCA), based on the concept of CA action without transactional guarantees. However, this solution has some drawbacks. First, for each exception raised by a service, the same exception is delivered to all composition’s participants, decreasing the flexibility for the implementation of handling actions. Second, the recovery process is strongly based on compensation actions, which is not a mandatory feature present in the implementation of a service. Moreover, the proposed solution is based only on Web services technology.

In this paper, we present the design and implementation of a coordinated exception handling model targeting the par-
ticularities of SCA systems. It allows the creation of fault-tolerant asynchronous service compositions in a flexible way. Also, considering that SCA systems may be highly dynamic, it supports a flexible notion of exception propagation where recovery rules that dictate how exceptions are propagated can be defined on a per-application basis. The definition of application-specific recovery rules, which are not necessarily based on compensation actions, makes our solution more general and flexible than WSCA [13]. We describe our solution using the primitives and abstractions of the Guardian model [7], [8] for exception handling. Guardian is a general conceptual framework for describing coordinated exception handling models and mechanisms. We have implemented the proposed exception handling model as a framework, named EH-SCA, which extends the Apache Tuscany SCA platform [5]. The latter is an SCA infrastructure capable of integrating various SOA technologies. To use EH-SCA in their applications, developers can leverage a simple aspect-oriented programming (AOP) [4] programming model that requires little effort to use. We also provide an example of the usage of the EH-SCA framework to implement a primary-backup system.

II. BACKGROUND

A. Service Component Architecture

A software component is a unit of modularity composed of two parts: the specification part, with explicit provided and required interfaces, and the implementation part [12]. A service component provides support for implementing services using components. Each component can implement one or more services, where the service’s implementation is part of the component’s implementation, and the service’s specification is mapped to a component’s provided interface. An implementation defines the materialization of the business logic into a specific technology, including programming languages, like Java, C++, and Ruby, and frameworks and environments, such as Spring and BPEL. A provided interface, referred to as “service” in the SCA context, defines the operations provided by the service component. Moreover, a service component can use required interfaces provided by other service components, known as “references”.

SCA service components can be connected either manually (by a programmer) or automatically (by the SCA runtime environment), using services and references. The communication protocol used between two service components is specified over the SCA “binding” element, which implies that the service component’s communication infrastructure is separated from the business logic implementation, enhancing the service component’s reusability.

The specification of compositions and service components is based on a XML-based language called Service Component Definition Language (SCDL). In Figure 1, a composition named “MyComposition” connects the “CalculatorComponent” and “AddServiceComponent” service components, both implemented with Java (implementation.java, lines 4 and 12), over a Web services-based communication (binding.ws, lines 7 and 15). Note that the “AddService” reference (line 5) in “CalculatorComponent” matches the service provided by “AddServiceComponent” (line 13), since both have the same interface “AddService”.

```
<composite name="MyComposition">
  <component name="CalculatorComponent">
    <implementation java class="CalculatorImpl"/>
  </component>
  <reference name="AddService">
    <interface java interface="AddService"/>
    <binding ws uri="..."/>
  </reference>
  <component name="AddServiceComponent">
    <implementation java class="AddServiceImpl"/>
    <service name="AddService"/>
    <interface java interface="AddService"/>
    <binding ws uri="..."/>
  </service>
</composite>
```

Figure 1. Example of a composition described through the SCDL.

In terms of error handling, SCAs pose a number of challenges. First of all, SCA systems are essentially service-oriented. They may span different administrative domains and comprise services whose implementations are not available or that are hosted by different organizations. As pointed out previously [13], error recovery mechanisms cannot make assumptions about the recovery capabilities of these services. In this scenario, it is not possible to employ a rollback-based approach. Second, services can be invoked asynchronously and errors may be signaled concurrently. Since it is usually not safe to assume that concurrently signaled errors are independent, some means for coordinating error recovery are necessary. A third challenge is that SCA systems can integrate a number of different technologies with different features and based on different standards. In this sense, SCA differs from Web services, since the latter are only one of the technologies that can be used in applications based on the former. An exception handling model for SCAs must be generic and flexible enough to support service compositions involving diverse technologies. At the same time, it has to be well-integrated with the underlying SCA platform, the infrastructure that mediates the interactions amongst the parts of the composition. Finally, the exception handling model must work in a dynamic setting, because components may fail or become temporarily unavailable due to failures, upgrades, reboots, etc., services may be taken out since they can be hosted by different organizations, and application needs may trigger reconfigurations, which imply different interactions and, hence, different sets of parts to be involved in coordinated error handling.

B. Apache Tuscany – An SCA Platform

Apache Tuscany ¹ is a platform for developing SCA-based applications. It aims to allow developers to focus

¹http://tuscany.apache.org
on the business logic of the applications, without having to worry about communication and interoperation issues. At a very high level, the Tuscany SCA can be divided into a core infrastructure and a set of extensions which extend the core to work with various technologies [5]. The Tuscany SCA runtime environment was designed to encompass a large range of existing technologies, as well as new, emergent ones. Figure 2 provides an overview of the runtime architecture of Tuscany.

Tuscany’s runtime environment has a modular and pluggable architecture so users can choose the functionality that they need. The Composite Application block represents the business application built with the Tuscany and described using the XML-based SCA assembly model (Section II-A). Tuscany Extensions are implemented through the use of the Tuscany SPI (Service Provider Interface), which offers a modularized way to define bindings, databindings, implementation types, policies, and interface types. Bindings provide support for different kinds of communication protocols. Databindings provide support for different data formats used among the services communication. Implementation types provide support for different programming languages and container models. Policies provide flexibility to adjust architecture concerns, such as security and transactions, without impacting the business logic code.

To run an SCA business application, the first new step the Tuscany runtime takes is to load and configure the SCA SCDL file. The various SCDL file artifacts are inspected, and factory methods help instantiate the various objects, which represent the services components in memory. The next step is to instantiate the runtime wires that connect the components. In this phase, runtime wires are created for component references and component services over the mentioned bindings in the SCDL file.

A runtime wire is a collection of invocation chains. Each invocation chain consists of a set of invokers and interceptors. Invokers provide the invocation logic to binding protocols and implementation technologies, while interceptors are a special kind of invoker that provides additional functionality, such as data transformation, security, and transaction control. The runtime creates an invocation chain for each operation in a service/reference interface [2].

Figure 3 shows a schematic view of the Tuscany SCA invocation architecture, where service component A invokes, via a reference, an operation provided by a service implemented by service component B, using a specific binding. When the service component A invokes an operation of B, the Tuscany runtime creates and associates a runtime wire to the service, and another to the reference. An SCA handler determines which invocation chain within the reference’s runtime wire is associated to the invoked operation. Parameters passed to the method that represents the invoked operation are used by invokers and interceptors, creating a proper message to be sent over the chosen communication protocol. On the receiving component, a listener takes the message from the underlying protocol and routes it to the correct receiving component’s invocation chains. The corresponding operation (implemented by a method or function) is executed and, if there is a return value, the reverse path is taken back to the caller component.

C. The Guardian Model of Exception Handling

The Guardian Model is a set of primitives, responsibilities, and mechanisms aimed at supporting the definition of exception handling models, in particular ones that require coordination. It is based on the notion of global exception handling, in which a distributed global entity called “guardian” orchestrates the exception handling actions for each involved participant. This is achieved by raising an appropriate exception in each participant. The guardian model uses specific application-defined recovery rules to determine which exception should be raised in each participant. The raised exception in turn causes the exception handler to be invoked [8]. Figure 4 shows a conceptual vision of the Guardian model, as well as its main elements: exception context, guardian members, guardian group, guardian primitives, recovery rules, and resolution trees.

An exception context corresponds to an application specific execution phase or region of a program, which has a symbolic name and a handler associated with it (e.g., in Java, a try-catch block). There are three kinds of exception contexts: signaling context, raising context, and
target context. The first one is the context a participant is in when an exception is signaled within it. The second is the context a participant is in when an exception is raised in it. Finally, the target context is the context in which an exception is handled. Furthermore, as depicted in Figure 4, the communication occurs among participants and guardian members, and guardian members and the guardian group, where a guardian member is a logical replication of the guardian associated with each participant. A set of operations, called guardian primitives, are used as the communication channel. The guardian primitives are responsible for controlling contexts (enabling and removing a context) and exception propagation (throwing global exceptions) at runtime and checking whether there are any pendent global exceptions to be delivered to a specific participant. In a coordinated exception handling mechanism, a global exception is an exception that needs to be handled cooperatively by a set of participants. On the other hand, an exception that can be handled locally within a participant is called local exception [7].

When one or more different exceptions types are raised concurrently, the guardian uses a resolution tree mechanism to find a resolved exception. Finding a resolved exception consists of searching for the lowest common ancestor in a resolution tree, which establishes a hierarchy among a set of exceptions that can be raised concurrently. Since the resolved exception is found, the guardian relies on the recovery rules to determine which exception should be raised in each involved participant, as well as the proper target context.

### III. THE EH-SCA FRAMEWORK

In this section, we describe the proposed exception handling model in terms of the abstractions and primitives of the Guardian model. Concomitantly, we present our implementation of these abstractions and primitives, the EH-SCA framework, as an extension to the Apache Tuscany platform. Then, in the next section, we present the EH-SCA programming model, which leverages aspect-oriented programming techniques to simplify the definition of handlers and their association to specific points of an SCA system.

#### A. Exception Representation

We define exceptions as classes that extend the GlobalException class. These exceptions flow between different service components. Instances of GlobalException carry the information of which participant has raised the exception, the context in which this exception was raised (the signaling context), as well as the context in which the exception should be handled (target context). The model also predefines some memberships global exceptions, such as JoinException and LeaveException. A JoinException is raised when a new participant joins a group (Section III-B). In a similar way, a LeaveException is raised when a participant leaves the group. These exceptions are useful to maintain compatibility with the Guardian model.

Global exceptions exceptions are raised by calling the gthrow() method (Section III-B). This method is homonym to the primitive of the Guardian model responsible for signaling a global exception. In our Java implementation, local exceptions are raised by using the throw statement. We do not impose any constraints on how exceptions are represented within a component, though, since programming languages employ different approaches for exceptions. Some of them, such as C, do not have the concept of exception nor anything similar to it.

The signatures of the services in component interfaces should explicitly indicate the exceptions they raise. During composition, the compiler should check whether clients of these services handle these exceptions and, if they do not, produce an error message. In other words, global exceptions are checked exceptions.

#### B. Exception Handling Contexts and Coordination

The guardian group is the central entity responsible for mediating the interaction between participants of a composition when errors occur during its execution. The guardian group provides an interface to enable and disable raising, signaling, and handling exception contexts. In EH-SCA, the guardian group is itself a service component.

Each participant is associated with a set of exception handling contexts, enabled dynamically, at runtime, and explicitly, by the application. Nonetheless, applications enable and disable contexts in a non-intrusive way, without the need to modify the source code of preexisting service components. When one of more global exceptions are raised within a context, all the participants where that context is enabled are involved in coordinated error recovery. In the proposed model, contexts are always associated with sets of participants and may be nested, similarly to Java’s try blocks (although the latter define static scopes). As discussed in the previous section, for SCA systems, it does not make
sense to define finer-grained, intra-component exception handling contexts since components might be implemented using radically different technologies and languages.

The guardian group element was implemented as an implementation type called “implementation.guardian” using the Tuscany SPI (Section II-B). Figure 5 depicts the pseudo-schema of the new defined implementation type. Note that the configuration is done through the “guardianProperties” element (line 2). The attributes “@recovery_rules” (line 3) and “@resolution_trees” (line 4) allow the definition of the recovery rules and resolution trees, respectively.

```
1 <implementation guardian>
2  <guardianProperties
3   recovery_rules="pathXMLRecoveryRulesFile"
4   resolution_trees="pathXMLResolutionTreesFile"
5 />
6 </implementation guardian>
```

Figure 5. Pseudo-schema for the “implementation.guardian” type.

Figure 6 shows the GuardianGroup interface implemented by all guardian group service components. This interface defines operations that are used to establish communication between participants and guardian members, and guardian members and their respective guardian groups, where a participant is implemented as a service component. It is important to stress that the implementation of GuardianGroup is internal to EH-SCA; applications to not need to implement this interface.

```
1 public interface GuardianPrimitives {
2   public void enableContext(Context c);
3   public Context removeContext();
4   public void gthrow(GlobalException ex,
5      String[] participantList);
6   public boolean propagate(GlobalException ex);
7   public void checkExceptionStatus() throws
8      GlobalException;
9   }
```

Figure 6. Interface of a guardian group component.

The first two methods are used for the context controlling. The enableContext() method adds and enables a context c in a LAST-IN-FIRST-OUT (LIFO) context list associated with a participant. The removeContext() method removes the last added context in same list. The Context class implements the concept of exception context, aggregating a name and a list of exceptions that can be handled in the context.

The gthrow() and propagate() methods control the flow of global exceptions. The first one was explained previously and is used by a participant to throw a global exception ex to a set of participants specified in participantList. The invocation of gthrow() causes the suspension of all the participants listed in participantList, as well as the interruption of the invoker participant. The propagate() method determines whether the global exception ex should be handled in the current context or propagated to an upper level context. In other words, the method compares the current context with the target context specified in ex. Since the guardian element does not have control over the exception flow inside the participant, the existence of the propagate() method is necessary.

At last, the checkExceptionStatus() method allows a participant to check the existence of any pending global exception that needs to be handled. It is executed periodically by the participants. If there are any global exceptions to be delivered, they are raised within the participant. Otherwise, the method simply returns.

C. Exception Propagation

Exception propagation is a difficult issue in service component architectures. As pointed out in Section II-A, exceptions must be propagated in non-standard ways because SCA systems are intrinsically dynamic due to user needs, heterogeneous technologies, administrative issues, and the distributed setting in which they run. As a consequence, SCA systems require more flexible policies for exception propagation, to make it possible to deal with situations such as the enabling and disabling of exception handling contexts at runtime or simply the unavailability of certain service components.

In the proposed model, recovery rules determine the exception propagation paths in an application. They establish the destination of an exception raised in a set of contexts associated with a set of participants. These rules also determine the exception that will be handled by each participant of a guardian group involved in coordinated exception handling. To better cope with the dynamism of SCA systems, recovery rules can be enabled and disabled at runtime. Hence, the propagation paths in an application can be modified dynamically, orthogonally to its structure, as required during its execution. To the best of our knowledge, this is the first exception handling model to provide this kind of flexibility to software developers. In the EH-SCA framework, recovery rules are defined by an XML-based language whose pseudo-schema is shown in Figure 7.

```
1 <recovery_rules>
2   <rule name="name">
3     signaled_exception="className">
4       <participant match="regularExpression"></participant>
5       <throw_exception class="className">
6         target_exception="contextName">
7         min_participant_joined="number"?
8         max_participant_joined="number"?
9         <affected_participants>
10         </affected_participants>?
11         </throw_exception>
12   </rule>
13 </rules>
```

Figure 7. Pseudo-schema for the definition of the recovery rules.

The rule element (line 2) defines a new rule, with its name (via @name attribute) and the exception for which that rule is applied (via @signaled_exception attribute). Each rule can select one or more sets of participants in order to signal a new exception. This is accomplished via the
participant element with a regular expression assigned to the @match attribute (line 4).

Each participant has a dot-separated identifier defined by the elements present in its context list. The same structure is applied for building the regular expression associated with the @match attribute, where the character "*" can be used as a wildcard, meaning that it does not matter the context the participant is. Also, the keyword SIGNALER can be used to retrieve the participant that has raised the referred exception.

The exception that should be raised in the set of participants selected by the participant element, is determined in the throw_exception element (line 4), where the exception class is specified via @class attribute, and the context in which the exception needs to be handled via the @target_context attribute. The optional min_participants_joined and max_participants_joined attributes represent, respectively, the minimum and maximum number of participants in the guardian group required for the delivery of the exception to the participants selected via the participant element.

Finally, the optional element affected_participants (line 8) is used to obtain a subset of the selected set of participants. A way to do this is through the predefined keywords FIRST and LAST, where the former selects the first element in the selected participant list, and the latter the last element. The participant order is determined by the order in which the guardian receives the requests for association.

D. Exception Resolution

The proposed model, analogously to previous approaches [15], uses exception resolution trees to determine which exception represents a set of exceptions raised concurrently in a certain context. In summary, the exceptions that can be raised in a system are organized as the nodes of a tree. When two or more exceptions are raised in a given context, the tree is looked up to find a node E that has all of the raised exceptions as its children. E is then called a “resolved exception” and it is the exception sent to all the participants involved in coordinated error recovery. On the other hand, the resolved exception may undergo a transformation before it is delivered to each participant of the composition. This transformation is useful to allow independently-developed exception handlers to work as a unit because they may have been defined in terms of different exception types. As a consequence, after resolution, a number of different exceptions can be delivered to the various handlers. The transformation of resolved exceptions is defined by means of recovery rules, more specifically, the throw_exception element of Figure 7.

The model supports the definition of various resolution trees, associated with different levels of an application. The usage of levels allows the establishment of semantic relationships among different sets of exceptions. The set of resolution trees in an application is defined using a XML-based language, as shown in Figure 8.

```
1 <resolution_trees>
2   <resolution_tree exception_level="level">
3     <exception className="className" />
4   </resolution_tree>
5 </resolution_trees>
```

Figure 8. Pseudo-schema for the definition of the resolution trees.

EH-SCA supports the definition of N resolution trees applicable to different exception levels, using the element “resolution_tree” and its “@exception_level” attribute. However, our current implementation supports only one exception level. The definition of a resolution tree for a specific level is done through an exception hierarchy, built using the “exception” element (line 3).

E. Guardian Member

The guardian member concept was implemented in Apache Tuscany SCA as a new policy type (Section II-B). In this way, a service component participant that needs to use coordinated exception handling has to include this facility in its SCDL configuration file. As shown in Figure 9, the choice of this policy creates an interceptor, which provides the guardian member logic, in the invocation chains. The internal usage of factories ensures that a single guardian member is created for each service component participant of a fault-tolerant composition.

![Diagram showing the concept of guardian members as interceptors within the invocation chains.](image)

Figure 9. Guardian members represented as an interceptor within the invocation chains.

The usage of policy avoids an explicit declaration of the guardian members in the SCDL file. At the developer side, a service component participant communicates directly with service component implemented as the guardian group. However, this communication is intercepted by an interceptor that hides the guardian member logic, and the defined communication model is held: participants ↔ guardian members, guardian members ↔ guardian group.

IV. AOP PROGRAMMING MODEL FOR EH-SCA

The EH-SCA programming model, as the guardian programming model defined originally by Miller [7], [8], consists of the invocation of the guardian primitives by a participant using a predefined programming pattern. The way
that the guardian primitives are invoked depends on recovery actions implemented in the application logic. However, a general structure based on the conversation concept [9], that is suitable to a large range of applications, is depicted in Figure 10.

```java
// Scope
{
  // Enables a context
  guardian.enableContext(c);
  try{
    // Check for any pending global exceptions
    guardian.checkExceptionStatus();
    // Application specific code
    ...
  } catch (Exception e) {
    finally {
      // Removes the context
      guardian.removeContext();
    }
  }
}
```

Figure 10. Conversation-based structuring pattern using the guardian primitives.

The application enables a context within a scope, checks for any pending global exceptions, executes the application specific code, and then removes the enabled context. There is, also, a handler to each global and local exception that can be raised, respectively, by the checkExceptionStatus() guardian primitive and by the application specific code.

Using aspect-oriented programming, Java annotations, and a predefined class hierarchy, it is possible to separate the normal application specific code from the exception handling code, i.e., the invocation of the guardian primitives and Java handlers. Moreover, this solution enables services to be easily reused. The AOP mechanism intercepts exceptions which belong to the compositions themselves and not the service implementations. The aspects do not need to intercept the OTS components, as they might not be interceptable (e.g., they might be implemented in a different language, such as C or Ruby). They intercept the service interface (more specifically, the proxies that implement it, provided by Tuscany), which is always accessible in EH-SCA because it is implemented in Java, Tuscany's implementation language. The main elements of the EH-SCA programming model with aspects are depicted in Figure 11.

Assuming that the scope of the structure of Figure 10 is the scope of a method, the Context annotation is used to specify the context that will be enabled, at the beginning of the method, and disabled at the end. The Checkpoint annotation specifies where the guardian primitive checkExceptionStatus() should be invoked within the method: before or after the application-specific code.

The HandlerContainer abstract class offers Create, Retrieve, Update, and Delete operations to manage the handlers. A service component that uses the aspect programming model needs to make its implementation class as a subclass of HandlerContainer. The abstract class AbstractHandler defines a generic global exception handler, where DefaultHandler is a concrete implementation with empty method's body.

The conversation-based structure is built over the annotated methods by the AssemblerAspect. The aspect intercepts the execution of the methods and, by retrieving the information passed through the annotations together with the handler management operations, combines the parts of the system implementing the normal and exceptional behavior.

V. CASE STUDY: PRIMARY-BACKUP SYSTEM

This section describes how to use the EH-SCA AOP to implement a primary-backup structuring technique in a client-server application. The primary-backup approach is well-known in the fault tolerance literature and, as a consequence, a good showcase for the capabilities of EH-SCA. Moreover, previous work on the Guardian model [8] has employed a primary-backup system as a case study.

In this application, a server receives requests from a client, processes them, and sends a response to the client. The fault tolerance mechanism consists of replicating the server so that, when the primary server fails, a backup assumes the primary role. For each request, the backups receive the updated primary server state in order to update its own state, keeping the consistency. Using the EH-SCA framework, each primary/backup is implemented as a service component, and its role behavior is controlled by the guardian coordinator. Figure 12 shows part of the SCDL file for the server application side using the primary-backup technique.

The component ServerNode (lines 2-10) describes a server node that can be used either as a primary or backup. The component, implemented in Java, exposes a service interface ServerNodeInterface used for establishing the communication with different server nodes by the matching with its nodes reference. Further, each server node communicates with the GuardianComponent (lines 12-18) through the guardian reference. This reference requires coordinatedExceptionHandler, ensuring that the guardian members will be created and intercept the communication among a participant and the guardian group.

Since a server node can assume the role of a primary or backup, three different contexts were defined: MAIN,
PRIMARY, and BACKUP. The MAIN context represents a configuration context where the server node is set up to operate as a primary (the PRIMARY context is activated) or backup (the BACKUP context is activated). The contexts are associated to methods through the Context annotation.

Figure 13 shows the primaryService method, which provides the business logic of a primary server. The method implements only the normal behavior, which consists in: 1) processing the client request; 2) sending the new state to the backups node and, 3) sending the reply to the client.

```java
@Context(name = "PRIMARY",
exceptions = {PrimaryFailedException.class, BackupJoinedException.class})
@Checkpoint(CheckpointPositions.BEFORE)
public void primaryService() {
  //Process the request then...
  //Send the updates to the backups
  //Send the response back to the client
}
```

Figure 13. Description of the primaryService method.

In a similar way, the backupService method, shown in Figure 14, carries the business logic of a backup server. The method simply applies the updates received from the primary server.

```java
@Context(name = "BACKUP",
exceptions = {BackupFailedException.class})
@Checkpoint(CheckpointPositions.BEFORE)
public void backupService() {
  //Apply the updates receive from the primary
}
```

Figure 14. Description of the backupService method.

When the operations primaryService() and backupService() are invoked, the assembler aspect intercepts their execution and combines the normal code with the exceptional code. The exceptional code is provided by the definition of handlers, subtypes of AbstractHandler, that are stored in the component through the use of the inherited HandlerContainer operations. Table I shows the global exceptions that can be raised in each participant, as well as, their meanings. Note that the same exception can be raised in one or more different contexts.

<table>
<thead>
<tr>
<th>Exception</th>
<th>Context</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>PrimaryFailedException</td>
<td>MAIN, PRIMARY</td>
<td>Internal primary server error</td>
</tr>
<tr>
<td>PrimaryExistsException</td>
<td>MAIN</td>
<td>Indicates the existence of a primary server</td>
</tr>
<tr>
<td>BackupFailedException</td>
<td>MAIN, PRIMARY, BACKUP</td>
<td>Internal backup server error</td>
</tr>
<tr>
<td>BackupJoinedException</td>
<td>PRIMARY</td>
<td>Indicates that a new backup backup has joined in the group</td>
</tr>
</tbody>
</table>

Also it is necessary to specify how the handlers are activated. The recovery rules in Figure 15 are responsible for this.

A normal execution of the application with no internal errors in the server nodes follows these steps: the ServerNode1 component is started in the MAIN context, causing the raising of a JoinException by the guardian. Rule1 is executed, but no exception is delivered since there is not a minimum of participants joined in the group. Thus, the component reaches the MAIN.PRIMARY context. When a second node ServerNode2 is executed, the raising of the JoinException causes the delivery of a BackupJoinedException to be handled in the PRIMARY context of the ServerNode1, and a PrimaryExistsException to be handled in the MAIN context of the ServerNode2. When the exceptions are raised in the services components due to invocations of checkExceptionStatus(), the proper handlers are activated and the ServerNode1 updates its backup list, while the ServerNode2 reaches the MAIN.BACKUP context.

In the case of an internal error in the primary server, a PrimaryFailedException is signaled to the guardian, where the Rule2 is executed. The guardian delivers a PrimaryFailedException to the ServerNode1 and ServerNode2, but with different target contexts: INIT and MAIN respectively. When the checkExceptionStatus() is invoked within both service components, an interruption is caused in the ServerNode1, and ServerNode2 becomes the new primary server, by invoking the proper PrimaryFailedException handler associated to the MAIN context. In case of a backup failure, a similar path is executed over the recovery rule Rule3. At the end, the primary removes a backup from its backup list, and the failed backup node is interrupted.

It is important to emphasize that exceptions are signaled to the guardian through the use of the gthrow() guardian primitive. In fact, the EH-SCA programming model with aspects hides most of the details related to the guardian primitives programming. However, some of them still need to be
relies on the resolution tree to find a common exception to be handled. Figure 16 shows a possible resolution tree to the primary-backup application.

Suppose that an error occurs simultaneously in the primary and in a backups server. In this case, a PrimaryFailedException and a BackupFailedException are signaled to the guardian. Since they arrive in the same time window, the guardian considers them as concurrent exceptions, and checks in the resolution tree for the lowest common ancestor, getting the PrimaryBackupFailedTogetherException.

This exception is delivered to the recovery rules, and the Rule4 is executed, resulting in the interruption of the failed primary and backup servers, and setting up a new backup to assume the primary server role.

VI. RELATED WORK

In the literature some solutions provided fault tolerance over distributed systems based on the CA actions concept. The work by Tartanoglu et al. [13] is based on CA Action concepts without transactional guarantees, resulting in a structure called Web Service Composition Action (WSCA). The WSCA description uses a XML-based language called Web Service Composition Action Language (WSCAL). The solution is implemented using the Web services technology, and is strongly based on the use of compensation actions. In our solution, a wide range of SOA technologies can be used (it is not restricted to Web services), and the recovery rules allow different ways to provide the recovery actions (including compensation actions).

Silva et al. [10] propose an approach based on contract composition for the definition of concurrent fault-tolerant compositions. A composition’s contract is extended to include the notion of coordination using CA actions. The composition’s contract is governed by rules and restrictions.

Souchon et al. [11] present a model for exception handling in multi-agents system. The model focuses on two problems: 1) preserving the agent programming paradigm, and 2) providing support to cooperative concurrency among the agents. The model implements coordinated exception handling, where resolution is based on resolution functions. However, the recovery rules are fixed, making the model less flexible when compared to the EH-SCA framework. Further, the proposed mechanism is not based on SOA technology. This factor hinders its interoperability.

Capozucca and colleagues [3] describe a framework to develop software systems based on CA actions. Their frame-

invoked explicitly. The same applies to the propagate(), that is used to reach the correct target context of an exception. Further, the INIT context is a top-level context defined by the guardian in an application.

The most interesting scenario occurs when two or more components fail at the same time. In this case, a set of concurrent exceptions is signaled to the guardian, which
work, CAA-DRIP, is an evolution of the DRIP framework proposed by Zorzo and Stroud [16] to structure software systems in terms of dependable multiparty interactions. None of them targets the construction of distributed systems. Furthermore, they are not targeted at the integration of pre-existing components and employ more constrained exception propagation mechanisms.

VII. Conclusions

In this paper, we have presented the design and implementation of a new exception handling model that targets the so-called Service Component Architectures. We are not aware of any middleware platform, service-oriented or otherwise, that provides support for coordinated exception handling in the way that EH-SCA does. Previous work that described actual middleware platforms did not focus on coordinated exception handling and previous work targeting coordinated exception handling did not cater for middleware platforms.

The first contribution of this paper is to allow the creation of fault-tolerant asynchronous service compositions in a SCA architecture. The second contribution is the development of a framework implementing the proposed model named EH-SCA. The EH-SCA framework defines a way to build applications in a conversation-based structuring unit, hiding the explicit usage of most of the guardian primitives. Our implementation is open-source and distributed under Apache2 license.

Future work includes extending the resolution trees model to allow the usage of different exception levels and implementing some fault tolerance mechanism in the guardian group element because it currently is a single point of failure. In addition, we would like to conduct an experimental validation of the ability of the framework to structure exception handling in a practical application and how that affects the application’s reliability and performance.

Acknowledgments

We would like to thank the anonymous referees, who helped to improve this paper. Cecilia is supported by CNPq (305331/2009-4) and FAPESP (2010/00628-1). Fernando is supported by CNPq (308383/2008-7 and 475157/2010-9), FACEPE (APQ-0395-1.03/10), and by INES (CNPq 573964/2008-7 and 475157/2010-9), FACEPE (APQ-1037-1.03/08).

References


http://www.apache.org/licenses/LICENSE-2.0.html