J Braz Comput Soc (2012) 18:43–59
DOI 10.1007/s13173-012-0059-5

LADC 2011

An exception handling system for service component architectures

Fernando Castor · Douglas Siqueira Leite · Cecília Mary F. Rubira

Received: 20 December 2011 / Accepted: 16 January 2012 / Published online: 4 February 2012

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 service compositions are executed, one or more errors can occur, possibly at the same time, affecting the dependability of the composition. To guarantee that the composition succeeds or at least fails in a controlled manner, fault tolerance mechanisms must be employed. 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. The EH-SCA framework instantiates the proposed model 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.

F. Castor (✉)
Informatics Center, Federal University of Pernambuco, Recife-PE, Brazil
e-mail: castor@cin.ufpe.br

D.S. Leite · C.M.F. Rubira
Institute of Computing, University of Campinas, Campinas-SP, Brazil

D.S. Leite
e-mail: dougsleite@apache.org

C.M.F. Rubira
e-mail: cmrubira@ic.unicamp.br

Keywords Exception handling · Service-component architectures · Fault tolerance · Service-oriented computing · Coordinated exception handling

1 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 [23]. 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) [14], 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. There are two ways to recover a service composition from an error: backward and

forward error recovery. The former is based on rolling the system components back to a previous correct state, while the latter involves transforming the system into a new correct state using exception handling mechanisms. Considering that, in the SOA context, it is not always possible to roll-back 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) [26]. A CA action provides fault tolerance by integrating coordinated exception handling, cooperative multithreading, and atomic transactions.

WS-Business Process Execution Language (WS-BPEL) [25], 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. [21] 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 some of the particularities of SCA (Sect. 2.1) 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 [21]. We describe our solution using the primitives and abstractions of the Guardian model [15, 16] for exception handling (Sect. 2.3). 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 (Sect. 3), which extends the Apache Tuscany SCA platform [12] (Sect. 2.2). 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) [10] programming model that requires little effort to use (Sect. 4). We also provide an example of the usage of the EH-SCA framework to implement a primary-backup system (Sect. 5).

2 Background

2.1 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 [20]. 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 Fig. 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

Fig. 1 Example of a composition described through the SCDL

```
1  <composite name="MyComposition">
2
3    <component name="CalculatorComponent">
4      <implementation.java class="CalculatorImpl"/>
5      <reference name="AddService" >
6        <interface.java interface="AddService"/>
7        <binding.ws uri="..."/>
8      </reference>
9    </component>
10
11   <component name="AddServiceComponent">
12     <implementation.java class="AddServiceImpl"/>
13     <service name="AddService">
14       <interface.java interface="AddService"/>
15       <binding.ws uri="..."/>
16     </service>
17   </component>
18
19 </composite>
```

`AddService` reference (line 5) in `CalculatorComponent` matches the service provided by `AddServiceComponent` (line 13), since both have the same interface `AddService`.

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 [21], 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.

2.2 Apache Tuscany—an SCA platform

Apache Tuscany[1] is a platform for developing SCA-based applications. It aims to allow developers to focus 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 make the core capable of working with various technologies [12]. 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 Tuscany and described using the XML-based SCA assembly model (Sect. 2.1). Tuscany Extensions are implemented by using 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 for communication among services. 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 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 service components in memory. The next step is

[1] http://tuscany.apache.org.

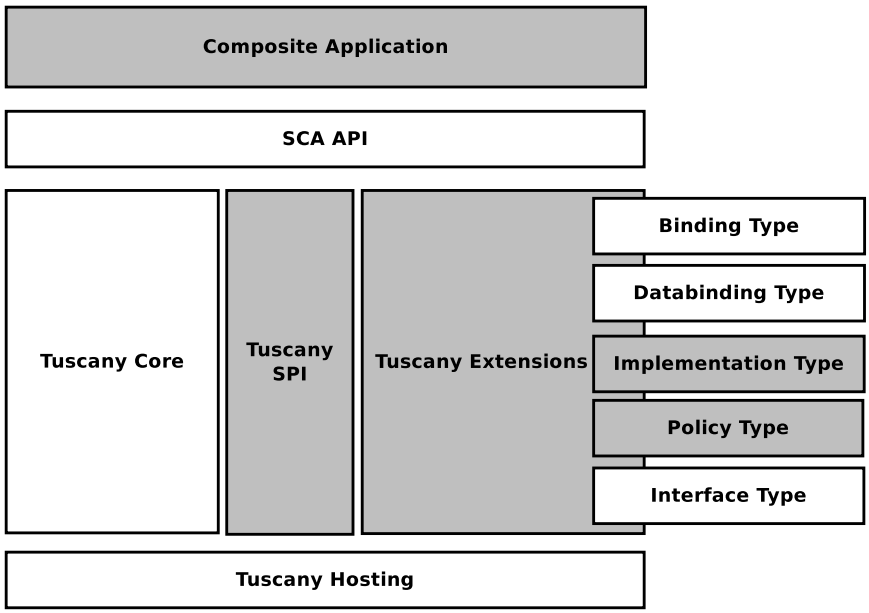

Fig. 2 Tuscany's runtime architecture

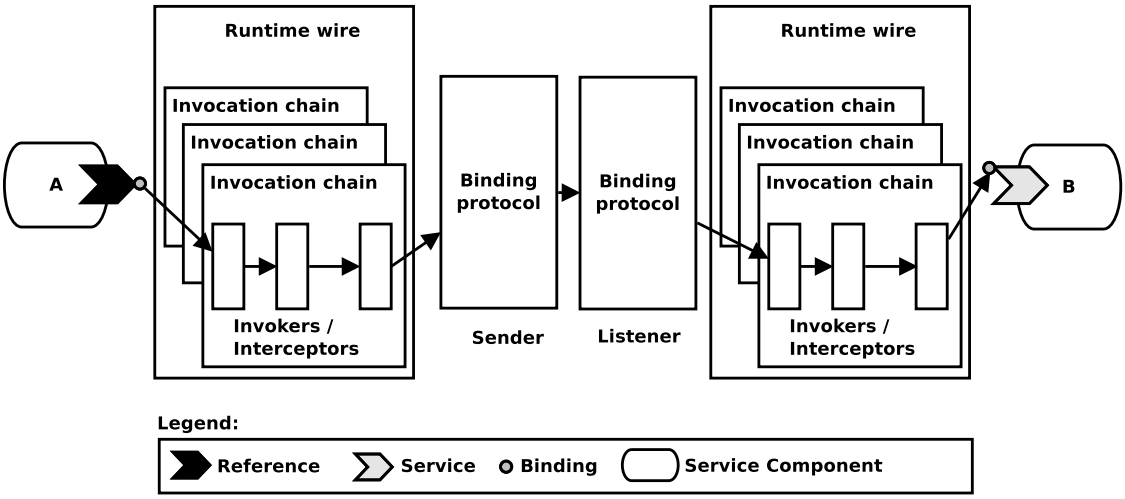

Fig. 3 Tuscany SCA invocation chain

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 environment 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 with the service, and another with the reference. An SCA handler determines which invocation chain within the reference's runtime wire is associated with 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 correspondent operation (implemented by a method or function)

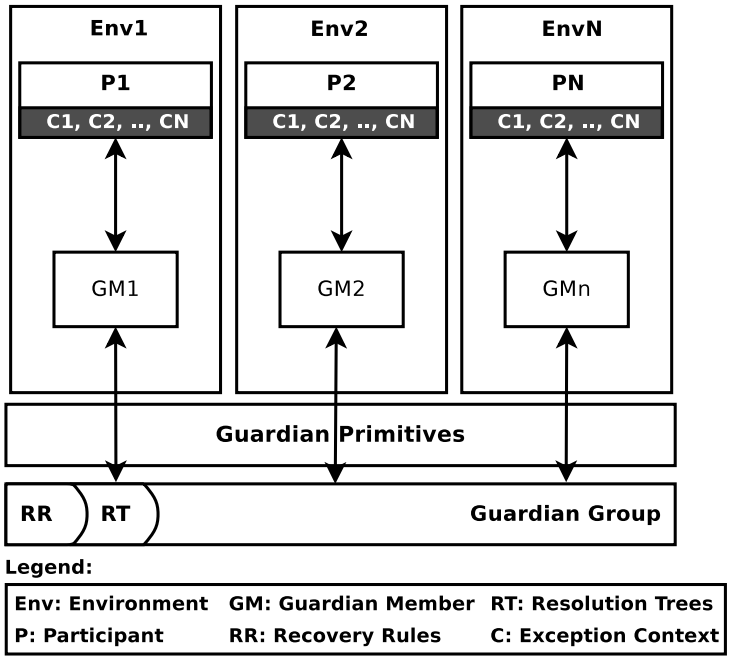

Fig. 4 Schematic view of the guardian model

is executed and, if there is a return value, the reverse path is taken back to the caller component.

2.3 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 [16]. 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 Fig. 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 [15].

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.

3 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.

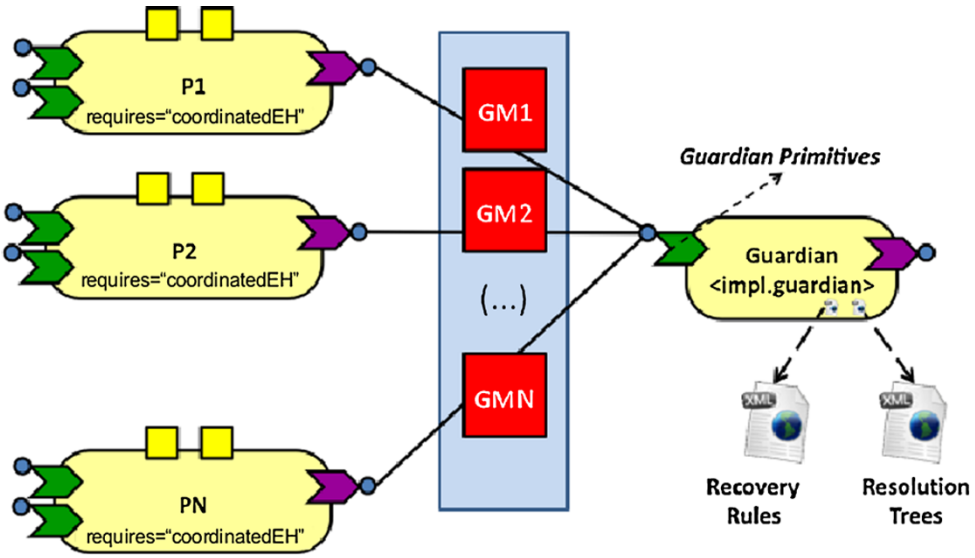

Fig. 5 Overview of the components of the EH-SCA framework

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 with specific points of an SCA system.

In the current version of EH-SCA, the components that signal, raise, and handle exceptions should be written in Java. To make EH-SCA language-independent would require more in-depth understanding of Tuscany's internals and a larger implementation effort. Since we organize EH-SCA in terms of the primitives of the guardian model, which is language-independent, we can say that it does not rely on the specifics of the Java language. At the same time, an SCA application whose components employ multiple technologies can still use EH-SCA, since Apache Tuscany can make these components communicate employing different technologies, such as WS-BPEL, JSON-RPC, and Web Services. In scenarios where third-party services must be integrated in a fault-tolerant manner by using EH-SCA, service components written in Java are responsible for raising and handling the exceptions. This does not limit the applicability of the proposed model and implementation. The composition of third-party services can be performed by using EH-SCA-based service components as proxies for the composed services, since it is not be possible to modify the composed services.

Figure 5 presents the overall structure of a fault-tolerant composition that uses EH-SCA. The Guardian is an instance of a new implementation type, impl.guardian, responsible for implementing the exception handling model. It manages exception handling contexts, exception propagation, and exception resolution. Each participant component (P1-N) is connected to the guardian by means of a guardian member (GM1-N). EH-SCA implements guardian members as policies (Sect. 2.2) and exceptions as regular Java types. In the remainder of this section, we present the elements of EH-SCA.

3.1 Exception representation

We define exceptions as classes that extend the `GlobalException` class. These exceptions are global, i.e., they 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 membership global exceptions, such as `JoinException` and `LeaveException`. A `JoinException` is raised when a new participant joins a group (Sect. 3.2). 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.

A program raises a global exception by calling the `gthrow()` method (Sect. 3.2). 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 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. A number of modern programming

Fig. 6 Pseudo-schema for the "implementation.guardian" type

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

Fig. 7 Interface of a guardian group component

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

languages, such as C#, Scala, and Go, do not use checked exceptions because of their well-known maintainability issues [24]. Nonetheless, we consider that explicitly indicated exceptions provide useful documentation. In addition, they represent part of the contract that users of a service must be aware of. Furthermore, due to compiler support, they can improve system reliability by emphasizing the need for errors to be handled. Finally, if checked exceptions are employed only at the component interface level, changes to such interfaces do not necessarily imply global changes as would be the case for finer-grained checked exceptions.

3.2 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 can enable and disable contexts in a nonintrusive way, without the need to modify the source code of preexisting service components. When one or 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, intracomponent exception handling contexts since components might be implemented using radically different technologies and languages.

The guardian group (or simply guardian) element was implemented as an implementation type called "implementation.guardian" using the Tuscany SPI (Sect. 2.2). Figure 6 depicts the pseudoschema of this new implementation type. Note that the configuration is done through the "guardianProperties" element (line 2). The `recovery_rules` (line 3) and `resolution_trees` (line 4) attributes allow the definition of the recovery rules and resolution trees, respectively.

Figure 7 shows the `GuardianPrimitives` interface implemented by all guardian service components. This interface defines operations that are used to establish communication between participants and guardian members, and guardian members and their respective guardians, 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.

The first two methods control exception contexts. 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 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 pendent

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.

3.3 Handler attachment

In the proposed model, exception handlers can be attached to local or global contexts. Local contexts are the contexts that the underlying programming language implements. In EH-SCA, a local context corresponds to a block of statements, the only kind of exception handling context that Java supports, by means of `try-catch` blocks, where the `catch` blocks are local handlers. Local handlers address internal exceptions. Handling these exceptions does not require a coordinated approach.

In broad terms, we consider that global handlers can be attached to sets of service components taking part in a composition. At the same time, participants of a composition can have multiple global exception contexts associated with them. For each context, it is possible to attach a number of exception handlers. In fact, there are no bounds on the number of contexts per service component, nor on the number of exception handlers per global context. When a global exception is signaled by a method, it is passed on to the guardian. The latter, based on its recovery rules (Sect. 3.4), decides which exception will be raised in each participant of the composition and the context where this will happen. An exception handler is triggered in a service component if it has an exception handler attached to the selected context and targeting the raised exception.

In EH-SCA, a handler is any subclass of the `AbstractHandler` class. The latter defines the `execute()` method, which receives a single argument of type `GlobalException` and implements the handler logic. Components in an application that uses EH-SCA should extend the `HandlerContainer` class. This class implements methods for managing the handlers attached to the contexts enabled in a service component. We more carefully describe the implementation of exception handlers in EH-SCA in Sect. 4.

3.4 Exception propagation

Exception propagation is a difficult issue in service component architectures. As pointed out in Sect. 2.1, exceptions must be propagated in nonstandard 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.

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

Fig. 8 Pseudo-schema for the definition of the recovery rules

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 pseudoschema is shown in Fig. 8. The `rule` element (line 2) defines a named (via the `name` attribute) rule and the exception to 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 5).

Each participant has a dot-separated identifier defined by the elements 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 selected participants is determined in the `throw_exception` element (line 4), where the exception class is specified via `class` attribute and the context where the exception will be handled via the `target_context` attribute. The `min_participants_joined` and `max_participant_joined` optional attributes represent, respectively, the minimum and maximum number of participants that must join the guardian group for the exception to be

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

Fig. 9 Pseudo-schema for the definition of the resolution trees

delivered to the selected participants. Finally, the `affected_participants` optional element (line 9) yields a subset of the selected participants, for example, the first (`FIRST` keyword) or the last (`LAST` keyword) in the selected participant list. The order of participants in the list is determined by the order in which the guardian receives the requests for association.

3.5 Exception resolution

The proposed model, analogously to previous approaches [26], 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 in involved 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 Fig. 8.

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 Fig. 9. Currently, our implementation supports only one exception level. The `resolution_tree` element (line 2) defines a resolution tree for a given level, defined by the `exception_level` attribute. The tree itself consists of a hierarchy of exception types built using the `exception` element (line 3).

The guardian is responsible for finding a resolved exception when two or more exceptions are concurrently raised in a composition. Exception resolution uses the types of the raised exceptions, organized in a type hierarchy (using the `exception_resolution` element). Considering the

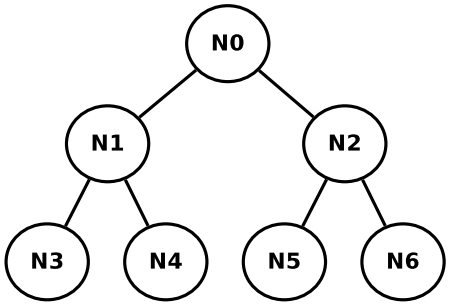

Fig. 10 An example of exception type hierarchy

types of the concurrently raised exceptions and the exception type hierarchy, exception resolution finds the lowest common ancestor of the types of all the exceptions. EH-SCA employs the algorithm of Bender and Farach [3] to find the common lowest ancestor. Considering the exception type hierarchy of Fig. 10, at initialization time, EH-SCA conducts three steps. Initially, the tree is traversed depth-first. Each node is included in a vector, E, each time it is visited. Therefore, for the hierarchy of Fig. 10, we would obtain the following:

$$E = \{\mathsf{N0}, \mathsf{N1}, \mathsf{N3}, \mathsf{N1}, \mathsf{N4}, \mathsf{N1}, \mathsf{N0}, \mathsf{N2}, \mathsf{N5}, \mathsf{N2}, \mathsf{N6}, \mathsf{N2}, \mathsf{N0}\}$$

Afterward, we compute the depth of each node in the tree. For each position that a node occupies in E, we record its depth in the corresponding position of the L vector:

$$L = \{0, 1, 2, 1, 2, 1, 0, 1, 2, 1, 2, 1, 0\}$$

For example, $E[2] = \mathsf{N3}$ and $L[2] = 2$, the depth of node N3. Finally, we build the R vector, which contains the position of the first occurrence of each node in E. Each position in R corresponds to a node in the exception type hierarchy. The order of the nodes in R is the order in which the nodes would be traversed in a breadth-first search.

$$R = \{0, 1, 7, 2, 4, 8, 10\}$$

In the event that two exceptions are concurrently raised, EH-SCA first obtains the values stored at the corresponding positions of R. It then uses these values as indexes for vector L. The lowest depth appearing between these values in L is the depth of the resolved exception. EH-SCA obtains it by inspecting the corresponding position in vector E. Both the time required to construct the E, L, and R vectors and the time to perform exception resolution are linear with the number of nodes in the tree.

3.6 Guardian member

In EH-SCA, the guardian member is an infrastructure element responsible for mediating communication between participants of a composition and the corresponding guardian. It is not part of the exception handling model, strictly

Fig. 11 Definition of the `guardianExceptionHandling` intent to abstract guardian members

```
1  <definitions xmlns="http://www.osoa.org/xmlns/sca/1.0"
2    targetNamespace="http://tuscany.apache.org/xmlns/sca/1.0"
         xmlns:sca="http://www.osoa.org/xmlns/sca/1.0">
3  <intent name="guardianExceptionHandling"  constrains="
       sca:implementation.java/sca:reference">
4    <description> All messages to the guardian group implementation
         will be intercepted
5    by the guardian member. </description>
6  </intent>
7  </definitions>
```

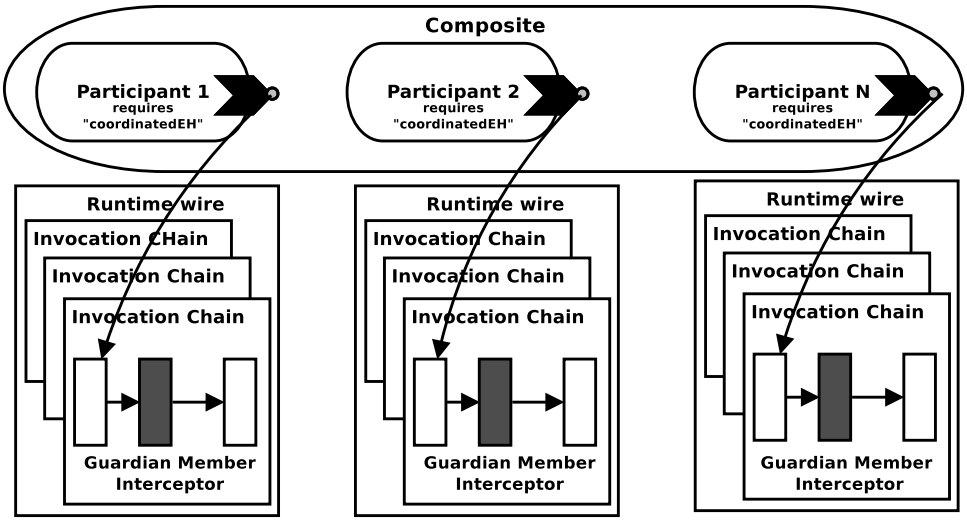

Fig. 12 Guardian members represented as an interceptor within the invocation chains

speaking, but it is necessary to ease the burden on software developers. We implemented guardian members in Apache Tuscany SCA as a new policy type (Sect. 2.2), more specifically, a new intent: `guardianExceptionHandling`. An intent is an abstract policy that can affect component interactions but does not include any deployment information. Figure 11 presents the definition of the `guardianExceptionHandling` intent. The `constraints` attribute (line 3) specifies that the intent applies to the references of a Java component. When `guardianExceptionHandling` is enabled for a participant, an interceptor is created in the invocation chains associated with the methods of the references to which the intent is associated. This interceptor is responsible for invoking the guardian member corresponding to the participant.

Figure 12 shows how guardian members fit within the structure of an EH-SCA service composition. To obtain the instance of the guardian member associated with each participant (an instance of type `GuardianMemberImpl`), each interceptor employs the `GuardianMemberFactoryImpl` factory. The internal usage of factories ensures that a single guardian member is created for each service component participant of a fault-tolerant composition.

The usage of a policy avoids the need to explicitly declare the guardian members in the SCDL file. At the developer side, a service component participant communicates directly with the guardian (another service component). However, this communication is mediated by an interceptor that hides the guardian member logic, and the defined communication model is held: participants ↔ guardian members, guardian members ↔ guardian group.

As an alternative to using a policy to implement guardian members, we could have implemented guardian members as service components. In this scenario, `GuardianMemberImpl` would need to implement a hypothetical `GuardianMember` interface which extends `GuardianPrimitives` (Sect. 3.2). Moreover, it would have a reference to the guardian service component. This approach has two drawbacks. The first one is that there is a runtime overhead due to the need to manage extra service components. A policy is much cheaper than a component. The second one is that developers of service-oriented applications would then need to be aware of guardian member components, declaring them in the SCDL file. By using policies to represent the latter, developers only indicate the participants of a fault-tolerant composition, associating them with the `guardianExceptionHandling` intent.

4 AOP programming model for EH-SCA

The EH-SCA programming model, as the guardian programming model defined originally by Miller [15, 16], consists of the invocation of the guardian primitives by a participant using a predefined programming pattern. The way in which participants invoke the guardian primitives depends on recovery actions implemented in the application logic. However, a general structure based on the conversation concept [17], that is suitable to a large range of applications, is depicted in Fig. 13.

```
1  public void method() { //Scope
2  //Enables a context
3    guardian.enableContext(c);
4
5    try{
6      //Check for any pendent global exceptions
7      guardian.checkExceptionStatus();
8
9      //Application specific code
10     . . .
11
12   }catch (Exception e) { . . . }
13   finally {
14     //Removes the context
15     guardian.removeContext();
16   }
17 }
```

Fig. 13 Conversation-based structuring pattern using the guardian primitives

The code snippet enables a context within a scope, checks for any pendent global exceptions, executes the application-specific code, and then removes the enabled context. There is also a handler for each global and local exception that can be raised, respectively, by the `checkExceptionStatus()` guardian primitive and by the application-specific code. The problem with this approach is that application logic is tangled with error recovery code. Ideally, the parts of the code that enable and disable contexts, and handle exceptions should be separated from application code [1]. In this manner, it becomes possible to change exception handlers without affecting application-specific parts of the code. At the same time, one can reuse exception handlers across components within the same application, in order to avoid duplication [22].

Using AOP, 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 and reduces the syntactic overhead that EH-SCA imposes on applications. Application developers only need to directly interact with EH-SCA in two ways: (i) by throwing or propagating exceptions with `gthrow` and `propagate`; and (ii) by annotating the relevant methods (see below). The main elements of the EH-SCA programming model with aspects are depicted in Fig. 14.

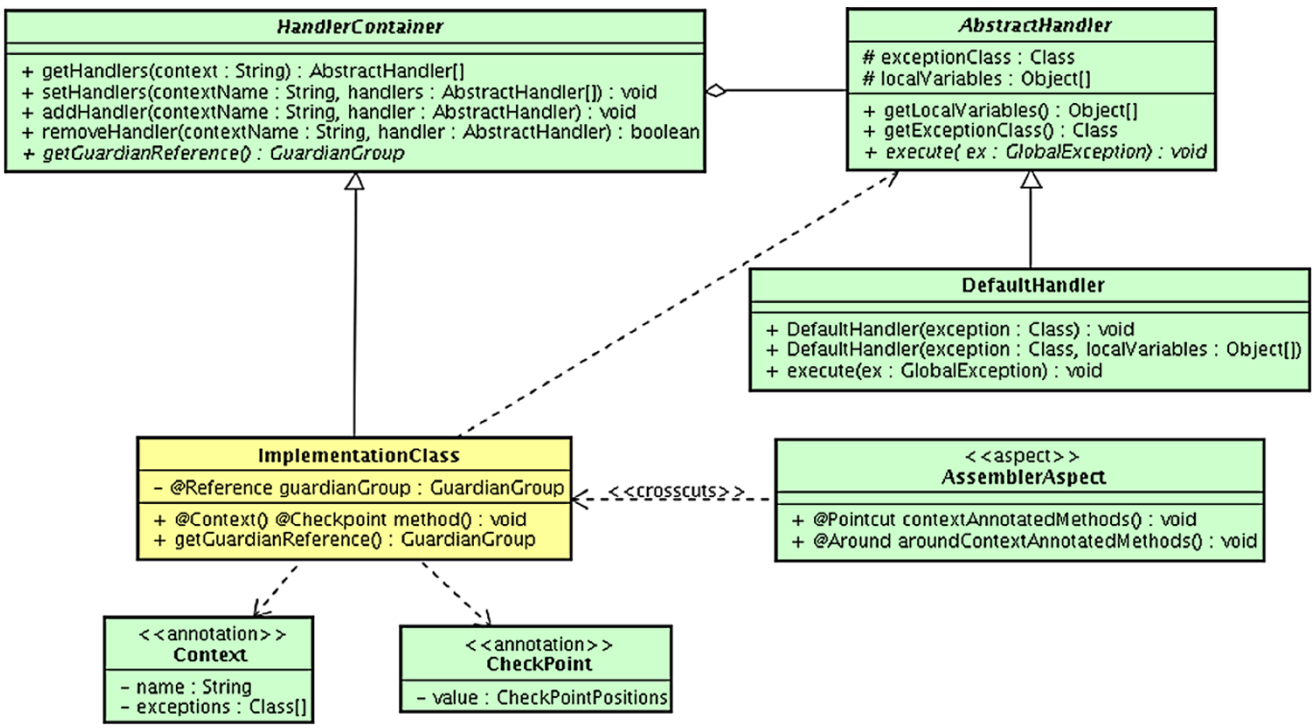

Fig. 14 The main elements necessary to use EH-SCA following a conversation-based structure with separation between the normal and exceptional code

```
1  // Enables context, defines handler
2  @Context(name = "METHOD", exceptions = {Exception.class})
3  // Checks for pendent global exceptions
4  @Checkpoint(CheckpointPositions.BEFORE)
5  public void method {//Scope
6
7      //Application specific code
8      . . .
9  }
```

Fig. 15 Using the EH-SCA aspect-based programming model

Assuming that the scope of the structure of Fig. 13 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 name of the context is specified by means of the `name` attribute, whereas the `exceptions` attribute lists the exceptions that are handled at that context. On the other hand, the `Checkpoint` annotation specifies when EH-SCA should invoke the `checkExceptionStatus()` primitive (Sect. 3.2), before or after executing the method. Two values are possible for this annotation: `CheckpointPositions.BEFORE` or `CheckpointPositions.AFTER`.

The `HandlerContainer` abstract class implements a number of operations to manipulate global exception handlers. These operations support the retrieval, attachment, and deletion of handlers (Fig. 14) associated with contexts of a service component. The main class implementing each service component must be a subclass of `HandlerContainer` in an application that uses EH-SCA. In addition, the class must provide an implementation for the `getGuardianReference()` abstract method to make the guardian accessible.

As briefly mentioned in Sect. 3.3, the `AbstractHandler` abstract class represents a generic handler for `GlobalException`. Application-specific exception handling strategies for a certain context are realized by implementations of the `execute()` abstract method. The remaining methods of `AbstractHandler` provide: (i) the `Class` object corresponding to the type of the handled exception; and (ii) the actual parameters of the method to which the `Context` annotation is linked. EH-SCA provides an implementation for `AbstractHandler`, the `DefaultHandler` class. The latter provides constructors that developers can use to pass information to the handler, and an empty `execute()` method.

The `AssemblerAspect` aspect combines all the aforementioned elements. It intercepts the execution of application-specific methods annotated with `Context` and `Checkpoint` in subclasses of `HandlerContainer` and uses the handler management methods to integrate normal and exceptional behavior. Using the aspect-oriented programming model involves the following steps: (1) to make the main class of a participant service component a subclass of `HandlerContainer`; (2) to implement the `getGuardianReference()` method so as to return the guardian; (3) to define a set of exception handlers as subclasses of `AbstractHandler`, attaching them to service components by means of the `addHandler()` and `setHandlers()` methods (Fig. 14); and (4) to annotate each method that takes part in a context where exceptions are handled with the `Context` and `Checkpoint` annotations.

Figure 15 shows what a method looks like as a consequence of the AOP programming model. The method implementation itself includes the code implementing the application logic. It may also invoke `gthrow` and `propagate`. The annotations provide all the information about contexts, handlers, and the moment when the runtime verifies whether an exception was received. The error handling concern in this case is textually separated from the application logic. This separation can be further improved if the developer decides to use AOP to introduce the annotations [11]. In this case, exception handling code is completely separated from the normal application code, improving maintainability and reusability.

5 Case study: primary-backup system

This section describes how to use EH-SCA, including the AOP programming model, 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 [16] has employed a primary-backup system as a case study.

5.1 System description

Figure 16 presents an overview of a client-server system that employs the primary-backup technique with three backup replicas. 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, maintaining replica consistency.

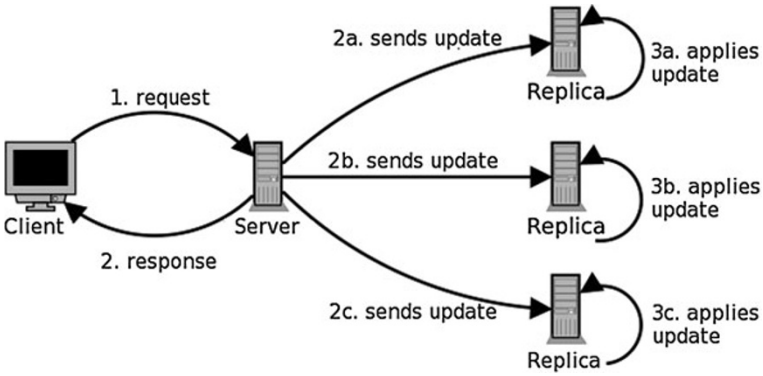

Fig. 16 Overview of a client-server system that employs the primary-backup technique

Fig. 17 Application server side description over SCDL

```
1  <composite name="ServerBackupComposition">
2    <component name="ServerNode1">
3      <implementation.java class="ServerNodeImpl"/>
4      <service name="ServerNodeInterface"/>
5      <reference name="nodes"
6        target="ServerNode2␣ServerNode3␣..."/>
7      <reference name="guardian"
8        target="GuardianComponent"
9        requires="coordinatedExceptionHandling"/>
10   </component>
11   . . .
12   <component name="GuardianComponent">
13     <implementation.guardian>
14       <guardianProperties
15         recovery_rules="recoveryRules.xml"
16         resolution_trees="resolutionTrees.xml"/>
17     </implementation.guardian>
18   </component>
19 </composite>
```

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 17 shows part of the SCDL file for the server application side using the primary-backup technique. The component `ServerNode1` (lines 2–10) describes a server node that can be used either as a primary or backup. The component, implemented in Java, exposes the `ServerNodeInterface` service interface 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 `coordinatedExceptionHandling`, ensuring that the guardian members will be created and intercept the communication among a participant and the guardian.

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

```
1  @Context(name = "PRIMARY",
2    exceptions = {BackupFailedException.class,
3      BackupJoinedException.class,
4      PrimaryServiceFailureException.class})
5  @Checkpoint(CheckpointPositions.BEFORE)
6  public void primaryService() {
7    //Process the request then...
8    . . .
9    //Send the updates to the backups
10   . . .
11   //Send the responde back to the client
12   . . .
13 }
```

Fig. 18 Description of the primaryService method

In a similar way, the `backupService` method, shown in Fig. 19, carries the business logic of a backup server. The method simply applies the updates received from the primary server.

When the operations `primaryService()` and `backupService()` are invoked, the assembler aspect intercepts their execution and combines the normal code with the

Table 1 Global exceptions and the contexts in which they need to be handled

Exception	Context	Meaning
PrimaryFailedException	MAIN, PRIMARY	Internal primary server error
PrimaryExistsException	MAIN	Indicates the existence of a primary server
BackupFailedException	MAIN, PRIMARY, BACKUP	Internal backup server error
BackupJoinedException	PRIMARY	Indicates that a new backup has joined in the group

```
1  @Context(name = "BACKUP",
2    exceptions = {BackupFailedException.class})
3  @Checkpoint(CheckpointPositions.BEFORE)
4  public void backupService() {
5    //Apply the updates receive from the primary
6     . . .
7  }
```

Fig. 19 Description of the backupService method

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 1 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.

Also it is necessary to specify how the handlers are activated. The recovery rules in Fig. 20 are responsible for this.

5.2 Execution scenarios

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 the `ServerNode2` component starts executing, the raising of the `JoinException` causes the delivery of a `BackupJoinedException` to be handled in the `PRIMARY` context of `ServerNode1`, and a `PrimaryExistsException` to be handled in the `MAIN` context of the `ServerNode2`. When an invocation of `checkExceptionStatus()` triggers the raising of exceptions in the service components, the proper handlers are activated and the `ServerNode1` updates its backup list, while the `ServerNode2` reaches the `MAIN.BACKUP` context.

If an internal error occurs in the primary server, an exception of type `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 `ServerNode1`. At the same time, `ServerNode2` becomes the new primary server by executing the proper handler for `PrimaryFailedException` 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 invoked explicitly. The same applies to the `propagate()` primitive. The latter is necessary to make exceptions reach their intended contexts. 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 relies on the resolution tree to find a common exception to be handled. Figure 21 shows a possible resolution tree to the primary-backup application. Assume that the primary server and a backup server fail simultaneously. In this case, a `PrimaryFailedException` and a `BackupFailedException` are signaled to the guardian. Since they arrive within the same time frame, the guardian considers them as concurrent exceptions and checks the resolution tree to obtain `PrimaryBackupFailedTogetherException`. This exception is delivered to the recovery rules, and `Rule4` is executed, interrupting the failed primary and backup servers and setting up a new backup to assume the primary server role.

It is important to stress that, without the resolution tree, the guardian would process the exceptions sequentially. If `PrimaryFailedException` is processed first, the primary server will be interrupted and the first backup in the list will became the new primary. However, the backup is down, and will not be able to assume the primary role. If `BackupFailedException` is be processed first, a similar problem occurs: the primary receives a notification indicating the failure of the backup but the primary itself has failed as well. As a consequence, the backup list is not updated, and the new primary sends messages to a failed backup.

Fig. 20 Recovery rules associated to the primary-backup application

```
<recovery_rules>
  <!-- A new participant joins in the group -->
  <rule name="Rule1" signaled_exception="JoinException">
    <participant match="*.PRIMARY">
      <throw_exception class="BackupJoinedException"
          target_context="PRIMARY"/>
    </participant>
    <participant match="SIGNALER">
      <throw_exception class="PrimaryExistsException"
          target_context="MAIN"
        min_participant_joined="2"/>
    </participant>
  </rule>
  <!-- The Primary fails -->
  <rule name="Rule2" signaled_exception="PrimaryFailedException">
    <participant match="*.PRIMARY">
      <throw_exception class="PrimaryFailedException"
          target_context="INIT_CONTEXT"/>
    </participant>
    <participant match="*.BACKUP">
      <throw_exception class="PrimaryFailedException"
        target_context="MAIN">
        <affected_participants> FIRST </affected_participants>
      </throw_exception>
    </participant>
  </rule>
  <!-- The Backup fails -->
  <rule name="Rule3" signaled_exception="BackupFailedException">
    <participant match="*.PRIMARY">
      <throw_exception class="BackupFailedException"
          target_context="PRIMARY"/>
    </participant>
    <participant match="SIGNALER">
      <throw_exception class="BackupFailedException"
          target_context="INIT_CONTEXT"/>
    </participant>
  </rule>
  <!-- The Primary and Backup fail together -->
  <rule  name="Rule4" signaled_exception="
      PrimaryBackupFailedTogetherException">
    <participant match="*.PRIMARY">
      <throw_exception class="PrimaryFailedException"
          target_context="INIT_CONTEXT"/>
    </participant>
     <!-- Backup signaler -->
    <participant match="*.BACKUP,SIGNALER">
      <throw_exception class="BackupFailedException"
          target_context="INIT_CONTEXT"/>
    </participant>
    <!-- Excluding the backup signaler -->
    <participant match="*.BACKUP,!SIGNALER">
      <throw_exception class="PrimaryFailedException"
          target_context="MAIN">
        <affected_participants> FIRST </affected_participants>
      </throw_exception>
    </participant>
  </rule>
</recovery_rules>
```

6 Related work

Garcia and Toledo [8] present an architecture for the construction of fault-tolerant Web Services. It extends the Universal Description, Discovery, and Integration (UDDI—the standard protocol for publishing and discovering services in Web Service-based systems) protocol to enable quality of service monitoring. This architecture introduces two new elements in service-oriented applications: the monitor and the mediator. The first one detects, notifies, and confines er-

Fig. 21 Resolution tree associated to the primary-backup application

```
<resolution_trees>
  <resolution_tree exception_level="1">
    <exception class=
      "PrimaryBackupFailedTogetherException">
      <exception class="PrimaryFailedException"/>
      <exception class="BackupFailedException"/>
    </exception>
  </resolution_tree>
</resolution_trees>
```

rors, intercepting messages with the goal of analyzing and testing services. It is also responsible for forwarding service invocations to replicas, when necessary. On the other hand, the mediator creates and manages these replicas. This approach leverages replication as a means for fault tolerance. In a similar vein, Chen and Romanovsky [6] introduce a mechanism named WS-Mediator to improve the reliability of Web service integration. The WS-Mediator is responsible for monitoring Web Services, collecting information such as response time and failure rate. Moreover, it applies user-defined policies depending on the information it obtains. Both of these approaches improve the dependability of service-oriented applications. However, none of them uses coordinated exception handling. Also, they are only applicable to Web Services and not to SCA.

In the literature some solutions provided fault tolerance over distributed systems based on the CA action concept. The work by Tartanoglu et al. [21] 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).

The work of Gorbenko et al. [9] proposes an approach to model reliable Web Services that comprise unreliable components. It uses WSCA as a means to structure Web Services using CA actions. The authors apply the proposed approach to a travel agency case study and claim that this significantly increases the number of successful Web Service requests. Since this work leverages WSCA, it exhibits the same limitations, when compared to EH-SCA.

Silva et al. [18] propose the use of composition contracts, an adaptation of coordination contracts [7] for the definition of concurrent fault-tolerant compositions. A coordination contract is a connection among a set of objects that is governed by rules and restrictions that are necessary for collaboration to ensue. Composition contracts extend coordination contracts by using CA actions without atomicity guarantees to implement fault tolerance. Exception propagation is static in this approach. Moreover, it does not define some aspects of the underlying exception handling model, such as what the granularity of exception handling contexts.

Souchon and colleagues [19] 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 abstract. On the one hand, this means that it may be applicable to systems that use different technologies. On the other hand, it does not consider the peculiarities of real approaches for the construction of distributed systems, in general, or SOA technology, in particular.

Capozucca et al. [4] describe a framework to develop software systems based on CA actions. Their framework, CAA-DRIP, is an evolution of the DRIP framework proposed by Zorzo and Stroud [27] 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 preexisting components and employ more constrained exception propagation mechanisms. Later on, Capozucca et al. [5] conducted a survey of existing frameworks implementing the concept of CA actions. Among the frameworks that they examine, only one tackles the problems of SOA technology, the work by Tartanoglu et al. [21], described above.

A preliminary version of this paper appeared elsewhere [13]. It did not discuss exception resolution and handler attachment in detail, nor the AOP programming model. Furthermore, it presented the case study only superficially.

7 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.[2]

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.

Acknowledgements We would like to thank the anonymous referees, who helped to improve this paper. Cecília 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 573964/2008-4 and FACEPE APQ-1037-1.03/08).

References

1. Anderson T, Lee PA (1990) Fault tolerance: principles and practice, 2nd edn. Springer, Berlin
2. Becker D (2009) Service component architecture (SCA) lets you invoke components from different technologies. http://www.ibm.com/developerworks/opensource/library/os-apache-tuscany-sca/index.html
3. Bender MA, Farach-colton M (2000) The lca problem revisited. In: 4th Latin-American symposium on theoretical informatics, Punta del Este, Uruguay, April 2000, pp 88–94
4. Capozucca A, Guelfi N, Pelliccione P, Romanovsky A, Zorzo AF (2006) CAA-DRIP: a framework for implementing coordinated atomic actions. In: Proc of IEEE international symposium on software reliability engineering, Raleigh, USA, November 2006, pp 385–394
5. Capozucca A, Guelfi N, Pelliccione P, Romanovsky A, Zorzo AF (2009) Frameworks for designing and implementing dependable systems using coordinated atomic actions: a comparative study. J Syst Softw 82(2):207–228
6. Chen Y, Romanovsky A (2008) Improving the dependability of web services integration. IT Prof 10(3):29–35
7. Fiadeiro JL, Andrade LF (2001) Interconnecting objects via contracts. In: Proc 38th international conference on technology of object-oriented languages and systems, Zurich, Switzerland, March 2001, pp 182–183
8. Garcia DZG, de Toledo MBF (2007) A fault tolerant web service architecture. In: 5th Latin-American Web congress, Santiago, Chile, October/November 2007, pp 42–49
9. Gorbenko A, Kharchenko V, Romanovsky A (2007) On composing dependable web services using undependable web components. Int J Simul Process Model 3(1/2)
10. Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes C, Loingtier J-M, Irwin J (1997) Aspect-oriented programming. In: Proceedings of the 11th ECOOP, June 1997
11. Laddad R (2009) AspectJ in action: enterprise AOP with spring applications. Manning Publications, Cambridge
12. Laws S, Combellack M, Feng R, Mahbod H, Nash S (2010) Tuscany SCA in action, 1st edn. Manning Publications, Cambridge
13. Leite DS, Rubira CMF, Castor F (2011) Exception handling for service component architectures. In: 5th Latin-American symposium on dependable computing, São José dos Campos, Brazil, pp 84–93
14. Margolis B, Sharpe J (2007) SOA for the business developer—concepts, BPEL, and SCA, 1st edn. MC Press, Paris
15. Miller R, Tripathi A (2002) The guardian model for exception handling in distributed systems. In: SRDS'02: proceedings of the 21st IEEE symposium on reliable distributed systems, Washington, DC, USA. IEEE Computer Society, Los Alamitos, p 304
16. Miller R, Tripathi A (2004) The guardian model and primitives for exception handling in distributed systems. IEEE Trans Softw Eng 30(12):1008–1022
17. Randell B (1975) System structure for software fault tolerance. In: Proceedings of the international conference on reliable software, New York, NY, USA. ACM, New York, pp 437–449
18. Silva R, Guerra P, Rubira C (2003) Component integration using composition contracts with exception handling. In: 3rd workshop on exception handling in object-oriented systems, Darmstadt, pp 1–20
19. Souchon F, Dony C, Urtado C, Vauttier S (2004) Improving exception handling in multi-agent systems. In: Software engineering for multi-agent systems II. Springer, Berlin, pp 333–337
20. Szyperski C (2002) Component software: beyond object-oriented programming, 2nd edn. Addison-Wesley, Reading
21. Tartanoglu F, Issarny V, Romanovsky A, Levy N (2003) Coordinated forward error recovery for composite web services. In: 22nd symposium on reliable distributed systems (SRDS), pp 167–176
22. Taveira JC, Queiroz C, Lima R, Saraiva J, Castor F, Soares S, Oliveira H, Temudo N, Barreiros E, Araujo A, Amorim J (2009) Assessing intra-application exception handling reuse with aspects. In: Proceedings of the 23rd Brazilian symposium on software engineering, Fortaleza, Brazil, October 2009. IEEE Computer Society, Los Alamitos
23. Thomas E (2007) SOA principles of service design, 1st edn. Prentice Hall, New York
24. van Dooren M, Steegmans E (2005) Combining the robustness of checked exceptions with the flexibility of unchecked exceptions using anchored exception declarations. In: OOPSLA'05, pp 455–471
25. Web services business process execution language version 2.0 (2007) http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf
26. Xu J et al (1995) Fault tolerance in concurrent object-oriented software through coordinated error recovery. In: Proceedings of FTCS '95, Washington, DC, USA. IEEE Computer Society, Los Alamitos, p 499
27. Zorzo A, Stroud RJ (1999) A distributed object-oriented framework for dependable multiparty interactions. In: ACM conference on object-oriented programming, systems, languages, and applications, Denver, USA, November 1999, pp 435–446

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