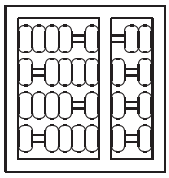

INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

An Architectural Approach for Fault-Tolerant Component Composition based on Exception Handling

Ricardo Silva *Fernando Castor*
Paulo Guerra *Cecília Rubira*

Technical Report - IC-04-02 - Relatório Técnico

March - 2004 - Março

Abstract

The development of modern component-based software systems usually needs to integrate autonomous component-systems which are developed independently from each other. Examples of such component-systems include COTS components, legacy software systems, and Web Services. For developing this new kind of software system, we need innovative software engineering approaches that rely on the system's software architecture to achieve the desired quality properties of the resulting system, such as fault tolerance. In this paper, we propose an architectural approach to the dependable composition of component-systems based on composition contracts and an exception handling scheme which considers the concurrent execution of architectural components.

1 Introduction

Modern component-based software systems, such as e-commerce and e-banking, usually are developed by integrating component-systems, which are autonomous and heterogeneous systems developed, maintained, and concurrently operated by independent organizations. In a rapidly changing world, these new software systems should be easily adaptable to changes in the business rules [3]. Moreover, many of these new software systems are safety-critical because financial loss and even life loss can result from their failure [12].

In general, no assumptions can be made about the internal design and implementation of a component-system. For instance, when integrating a web service [13] whose actual implementation is dynamically bound at run-time, the only information available to the system integrator is the specification of its public interface. So, there are no guarantees about the quality attributes of the actual implementation, such as its correctness, availability and reliability. Hence, in order to achieve these quality properties, we should focus on solutions mainly at the software architecture [18].

In this paper, we propose an architectural approach to fault-tolerant composition of concurrent component-systems based on composition contracts. A composition contract extends the concept of coordination contracts [3] to include an exception handling [8] scheme based on Coordinated Atomic Actions (CA Actions) [23]. A coordination contract is a connection that can be established between a group of objects through which rules and constraints are imposed on their collaboration, thus enforcing specific forms of interaction or adaptation to new requirements [3]. The concept of CA action has been introduced by Xu et al [23], as a mechanism for structuring fault-tolerant concurrent systems which unifies the notions of forward and backward error recovery. Forward error recovery in a CA action is supported by means of an exception handling and resolution scheme which takes into account the possibility of multiple exceptions being raised concurrently by the participant objects. CA actions integrate competitive and cooperative concurrency within a unified conceptual framework which facilitates the construction of complex dependable systems.

Our approach also employs the C2 architectural style [22], which is a component-based style directed at supporting large grain reuse and flexible component com-position, emphasizing weak bindings between them. By architectural style, we mean a set of design rules that identify the kinds of components and connectors that may be used to compose a

system or subsystem, together with local or global constraints on the way the composition is done [20].

The rest of this paper is organised as follows. Section 2 provides some background information. Section 3 describes how we have adapted the CA Action concept to obtain fault tolerance in the context of component-based composition. Section 4 presents a software architecture that leverages the task of building dependable systems out of heterogeneous and autonomous components. Section 5 briefly describes some tools which support the development of systems based on the proposed architecture. In Section 6, we present a case study in order to experiment our approach. Section 8 discusses some related work and, finally, Section 8, presents our conclusions and some directions for future works.

2 Background

2.1 Coordination Contracts

A coordination contract [4] is a connection that can be established between a group of objects through which rules and constraints are superposed on their joint behaviour, thus enforcing specific forms of interaction or adaptation to new requirements [2]. It prescribes a set of coordination effects that are superposed on the interacting parties when the occurrence of one of the contract triggers is detected in the system. A trigger can be a condition on the state of the participants of the interaction, a request for a particular service, or an event issued by one or more participants.

Coordination contracts may apply to single objects, e.g. to regulate or adapt the way the objects behave in order to fulfill new requirements, or involve many partners, in which case the contracts behave as synchronization agents that coordinate the way partners interact. In the description of a coordination contract, partners are specified as a number of interfaces that can be instatiated with concrete implementations when the coordination contract is activated on a particular configuration.

2.2 The C2 Architectural Style

The C2 architectural style, components of a system can be completely unaware of each other. The components communicate only through asynchronous messages mediated by connectors that are responsible for message routing, broadcasting and filtering. A simple C2 architecture is depicted in Figure 1. Both components and connectors have a top interface and a bottom interface. Systems are composed in a layered style. The top interface of a component can be connected to the bottom interface of a single connector. The bottom interface of a component can be connected to the top interface of another single connector. Each side of a connector may be connected to any number of components or connectors.

There are two types of messages in C2: requests and notifications. By convention, requests flow up through the system's layers and notifications flow down. In response to a request, a component may emit a notification back to the components below, through its bottom interface. Upon receiving a notification, a component may react, as if a service was requested, with the implicit invocation of one of its operations.

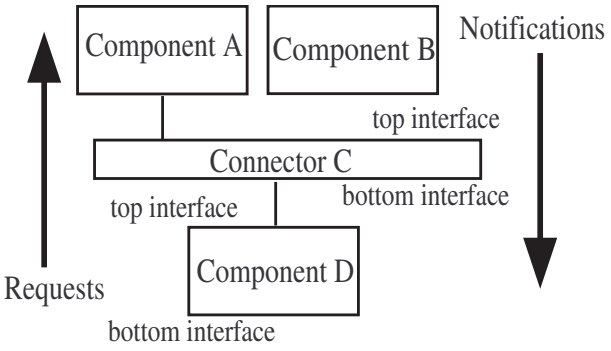

Figure 1: An example of C2 architecture.

The C2.FW framework [17] provides an infrastructure for building C2 applications. The C2.FW Java [10] framework comprises a set of classes and interfaces which implement the abstractions of the C2 style, such as components, connectors, messages, and connections. C2.FW has been also implemented in C++, Java, Python and Ada.

3 Exception Handling with CA Actions

A Coordinated Atomic Action (CA Action) is a multi-entry unit with roles that are bound to action participants which cooperate within the CA Action. The action starts when all roles have been activated and finishes when all of them reach the action end. If a participant raises an exception within a CA action, appropriate recovery measures should be invoked cooperatively, by all the participants, in order to reach some mutually consistent exception handling. If multiple exceptions are raised at the same time, a resolution scheme is used to combine these exceptions into a single one.

Externally, a CA Action behaves like an atomic transaction [14], in the sense that it should guarantee the ACID (atomicity, consistency, isolation and durability) properties. Hence, the effects of operations performed upon external objects are only visible to the rest of the system if the CA Action terminates successfully. When an exception is signalled, the participants initially apply a forward error recovery strategy in order to try to mask it and complete the CA Action successfully, either with a normal result or a degraded (exceptional) one. If this strategy fails, then the CA action should trigger backward error recovery in its participants and undo the undesired effects. A CA Action *aborts* if it cannot complete successfully but it is able to restore its initial state. Otherwise, it *fails*.

Composition contracts incorporate a relaxed CA Action mechanism slightly differnt from the original concept because they allow component-systems (CA Action's participants) to interact with external objects that are not transactional. When integrating autonomous components in a new system, the internal states of these components may not be accessible externally. Furthermore, it may not be possible to effectively bring a component to a consistent state exclusively by means of its external interface, since a component may be

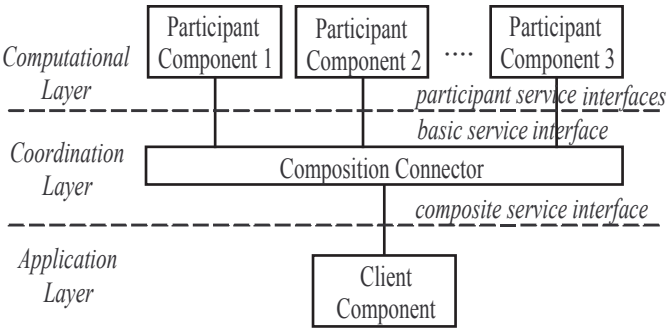

Figure 2: The proposed software architecture.

developed without taking error recovery into account. Due to these constraints on the use of autonomous systems for the construction of new systems, it may not always be possible to perform backward error recovery. However, our approach does provide support for the construction of evolvable dependable systems by means of a coordinated exception handling scheme and a software architecture which isolates new business rules from the autonomous parts of the system. It is important to notice that the aforementioned limitations are not inherent to our approach. They stem from the characteristics of the interacting parties (which are autonomous, heteogeneous, and independently developed components).

Component-systems can be designed recursively using action nesting. Fault tolerance features are always associated with such units, confining errors. When an action is not able to mask an error, an exception is propagated to the containing action. This exception may be an *abort exception*, when the participants are left in a state free of the effects of the action, or a *failure exception* otherwise, when the undo fails. In the latter case, the containing action is responsible for recovering the state of the system.

4 The Proposed Architecture

Our software architecture is organized in three layers (Figure 2). The *computational layer* encapsulates the service interfaces of the component-systems. We call the latter *participant components* because they play the roles of participants in a CA Action. Examples of participant components include COTS components, legacy systems, and Web services. These elements are adapted by means of wrappers[5], in order to interrelate to the rest of the system. The *coordination layer* consists of a *composition connector* that mediates the interactions between the computational layer and the application layer. The top interface of the composition connector, or its *basic service interface*, is the sum of all the participant service interfaces. The composition connector may impose business rules upon the basic service interface in order to provide new composite services. The bottom interface of the composition connector, or its *composite service interface*, is the sum of the basic service interface plus the new composite services provided by the composition connector. The composite service interface also adds fault tolerance to its new services by means of exception

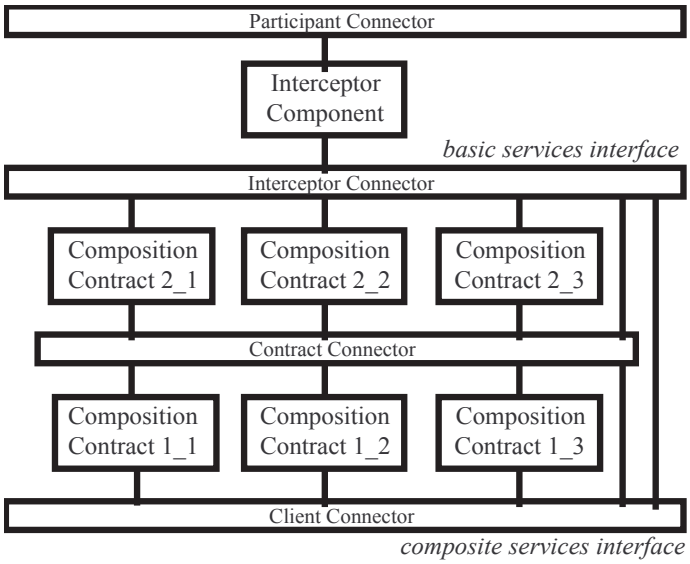

Figure 3: Basic structure of the composition connector.

handling. The *application layer* contains the components that implement the application logic (*client components*) and that may use the composite service interface.

The composition connector is a C2 connector built from an *interceptor* component and a set of *composition contract components* (Figure 3). A *composition contract*, or simply a contract, defines a set of related composite services. In this context, a composite service specifies an action that may be required to impose new business rules upon a service. The composite service may extend a single basic service or compose a more sophisticated service which defines a coordinated action of two or more participant components. The composition contract components can be organized in various *contract layers* that are connected by specialized C2 connectors. Figure 3 shows a composition connector with two contract layers. The composition contract components are responsible for: (i) providing new composite services; (ii) imposing the business rules upon basic and composite services; and (iii) implementing fault tolerance for basic and composite services. This is done by means of the fault-tolerant composition of one or more basic services. The composition contract components of a contract layer can use composite services provided by contract components located at upper contract layers, allowing action nesting. The interceptor component is responsible for monitoring the flow of events at the basic service interface and activating the composition contract components when needed.

4.1 The Composition Contract Component

A composition contract component implements a composition contract as a CA Action relaxed in its transactional requirements over the participant components, as discussed in Section 3. A composition contract is activated by a notification of an event associated with

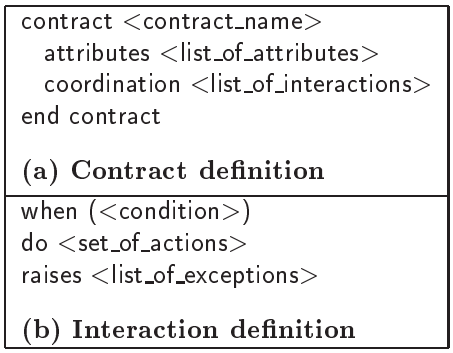

Figure 4: Composition contract definition.

a *contract trigger*. This notification is sent by the *interceptor component*. The activation of a composition contract implies in the implicit invocation of an associated composite service. The invocation of a composite service normally results in one or more service requests issued according to the coordination rules defined by the composition contract. The service requests can be concurrently executed. Furthermore, they may activate composition contract components included in upper contract layers, creating nested CA Actions. It is important to note, however, that a request sent by a composition contract is never delivered to another composition contract in a lower layer.

When a composite service completes successfully, it ends either with a normal notification or, if its result is degraded, an exceptional notification. If exceptions are raised by a requested service, the composition contract component collects the responses from the services and activates an exception handler. The exception handler resolves the raised exceptions and coordinates the appropriate recovery actions that should be taken by the participant components. If the exceptional condition cannot be masked, the composition contract component reacts with: (i) an *abort notification*, if the participants are left in a consistent state, or (ii) a *failure notification*, if one or more participants are left in an inconsistent state.

The composition contract definition is shown in Figure 3a. The attributes clause specifies the configuration parameters of an instance of a composition contract. The coordination clause specifies a list of interactions which defines how the participant components cooperate to perform a particular composite service (Figure 3b). The condition after the when clause specifies the *contract trigger* for this interaction. The do clause specifies a set of actions to be concurrently executed by the participant components. The raises clause specifies the types of exceptions that may be raised during the execution of the interaction.

Composition contract components should organized in layers which allow the nesting of actions within a system and facilitate the progressive extension of the business rules implemented by the contracts.

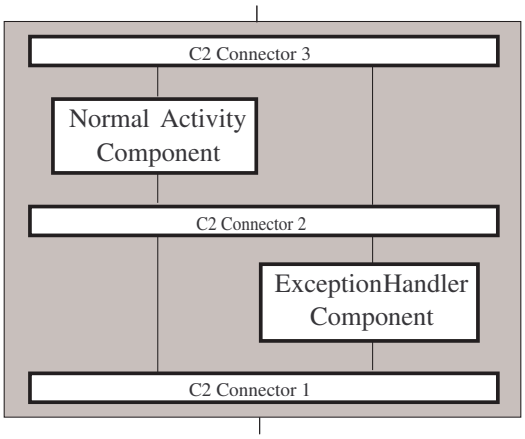

Figure 5: Internal structure of an architectural component.

4.2 The Interceptor Component

The *interceptor component* is responsible for monitoring events that enable contract triggers, determining when a composition contract should be activated. The events monitored by the interceptor component are service request messages addressed to the participant components. In order to decide whether a service request will or will not be intercepted, it inspects the parameters of the messages and the contract triggers definitions, which are part of its configuration. Services requests that do not trigger any contract are sent directly to the computational layer.

At first glance, it would seem obvious to place the interceptor component under the coordination layer, and not above it (Figure 3). In this manner, all request messages arriving at the composition connector would be promptly intercepted and sent to their intended destinations (composition contract or participant components). We have decided not to, however, because the interceptor should also able to intercept messages sent by the composition contracts to the participants, with possible nesting of different composition contracts.

The organization of composition contracts in sub-layers and the contract triggers defined within the interceptor component constitute all the knowledge necessary to define the way composition contracts will be composed, that is, the composite services provided to the application layer.

4.3 The Exception Handler Component

In our model, an architectural component can be a participant component, an interceptor component, or a composition contract component. Each of these components is composed by two subcomponents: a NormalActivity component and an ExceptionHandler component (Figure 5). The NormalActivity component implements the normal behaviour of an architectural component, when no exceptions occur. The ExceptionHandler component is

```
exceptional contract <contract_name>
  attributes <list_of_attributes>
  coordination <list_of_interactions>
end contract
```

Figure 6: Exception handler component definition.

responsible for: (i) handling exceptions raised by its associated NormalActivity component, and (ii) providing *handler services*.

More specifically, the ExceptionHandler component of a participant component implements *handler services* in order to undo operations that affect the participant's state. On the other hand, the ExceptionHandler component of the interceptor component handles system configuration errors.

The ExceptionHandler component of a composition contract component implements the exceptional contract defined for the composition contract (Figure 6) and coordinates the activation of *handler services* on the participant components. It tries to mask an exception and return the control flow to the NormalActivity component using a forward error recovery strategy. If this strategy fails, the ExceptionHandler component should perform backward error recovery by executing compensation actions in order to undo undesirable effects over the participant components, when possible

5 Tool Support

In order to apply a software development technique to the solution of real-world problems, tool support is an important feature. With this in mind, some tools have been built which support software development based on our architecture.

First, we have implemented an object-oriented framework which leverages development of fault-tolerant systems based on the proposed approach. This framework is an extension of the Java [10] version of the C2.FW framework which supports the architecture proposed in Section 4. Moreover, we have added a concurrent exception handling and resolution scheme to C2.FW, an important addition, since the latter does not provide adequate support for the construction of fault-tolerant systems. The resulting framework has been used in the implementation of the case study described in Section 6.

Furthermore, we have built a simple modeling environment based on the Generic Modeling Environment (GME)[15]. The GME is used primarily for model-building. It supports the description of a model's meta-model and, based on this information, is capable of functioning as a modeling environment for models which are instances of the specified meta-model.

Our environment allows developers to describe a software architecture based on our approach by means of box-and-line diagrams. A models may be checked for consistency and exported to an XML file. The latter is used as input for code generation and the model itself is a useful source of architectural documentation. Since this is the first version of our environment, many useful features have been left out, such as the specification of contract

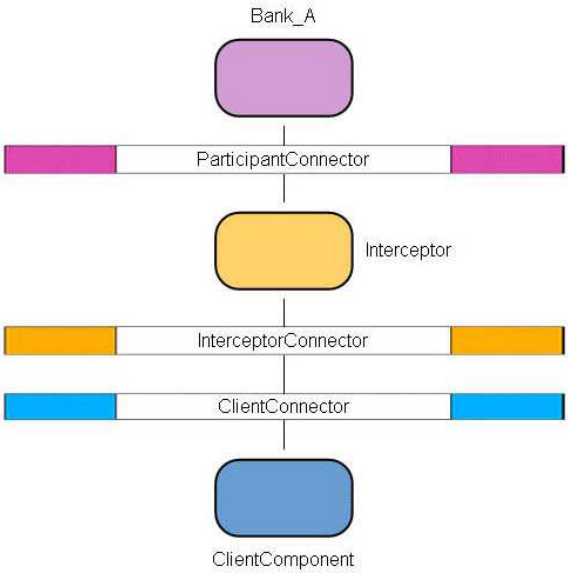

Figure 7: Banking system with one participant and no contracts.

triggers and the definition of exception handling components (Figure 5). We intend to incorporate these features in future versions.

We have also built a simple code generation utility which takes the exported XML description of an architectural model as input and generates code skeletons for the classes which implement the architectural elements and the connections among them. Furthermore, the generated classes are tailored specifically to the aforementioned framework, and the generated implementation can be executed "as-is", with no further configuration.

6 Case Study

In order to evaluate our approach, we have built a case study based on a banking system. In this case study, client components request services from the participants which correspond to the banking systems of two different banks, A and B. The coordination layer uses the participants in order to provide services involving more than one participant (banking system). Our case study consists of three increasingly complex scenarios. Subsections 6.1, 6.2, and 6.3 describe each of these scenarios.

6.1 Scenario 1: One Participant and No Contracts

Scenario 1 consists of a very simple system in which clients are able to deposit funds in a bank account at Bank A. Figure 7 presents the architectural configuration for this system, composed by three components: a participant component corresponding to the banking system of Bank A (Bank_A), the interceptor (Interceptor), and a client component (ClientComponent). In this scenario, ClientComponent issues a deposit request which is received

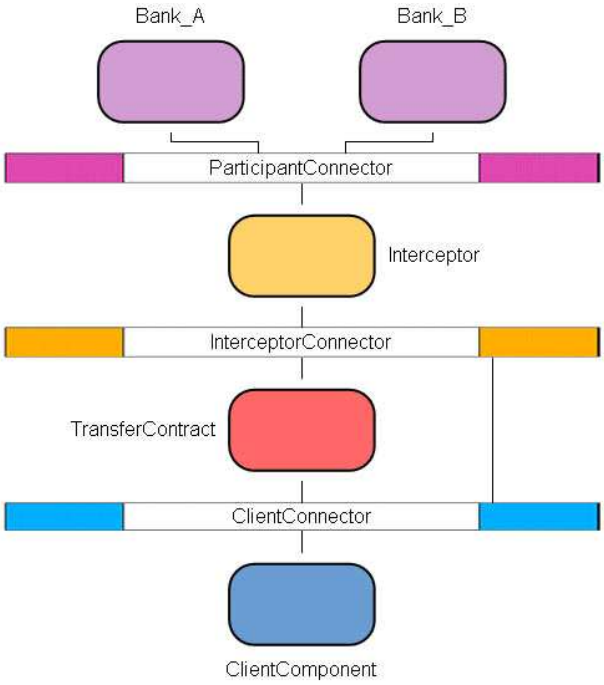

Figure 8: Banking system with two participants and a single contract.

by the Interceptor component and forwarded by the latter to the Bank_A participant component. When Bank_A receives the request, it performs a deposit in the chosen account and issues a notification message which is delivered to the ClientComponent.

6.2 Scenario 2: Two Participants and a Single Contract

In scenario 2, we introduced a new participant and a new operation. Figure 8 presents the new architectural configuration of the system. The new participant corresponds to the banking system of Bank B (Bank_B). The new operation is implemented by the contract component TransferContract, and it is a transference of funds between accounts belonging to different banks.

The client wishes to make a transfer of $ 100.00 from an account at Bank_A to an account at Bank_B. At first, ClientComponent issues a funds transfer request message which is received by the Interceptor. The Interceptor has been configured to redirect all funds transfer requests to the TransferContract component. As soon as the latter receives the request, it issues simultaneously a withdraw request and a deposit request. Since no contract triggers have been configured, these messages are delivered to the two participant components. If both participants perform as expected, the TransferContract receives two notifications informing that the operations were successful, and issues another one, informing this fact to the ClientComponent. This scenario is described in the sequence diagram in Figure 9.

The TransferContract is also responsible for performing coordinated exception handling,

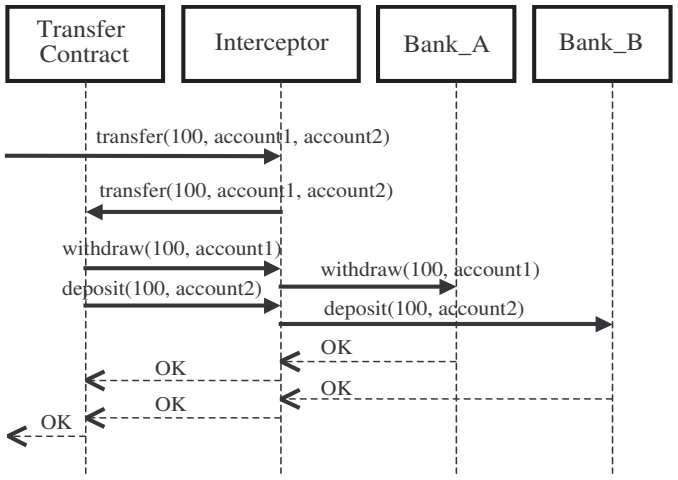

Figure 9: UML sequence diagram for Scenario 2.

if the transference cannot be successfully performed. When one of the participant raises an exception, TransferContract collects all the notifications issued by the participants, resolves exceptions in case more than one has been raised, and initiates the handling of the resolved exception within the participants. Figure 10 shows a scenario where the participant Bank_B could not perform the deposit and aborts the operation. TransferContract handles this exception by depositing $100.00 on the account at Bank_A, from which the same amount had been withdrawn. Upon successful recovery of the state of the system, TransferContract issues an *abort exception* which is received by the ClientComponent. This exception indicates that, although the requested operation could not be performed, the systems state remains consistent.

6.3 Scenario 3: Three Participants and Two Contracts

In 1993, the Brazilian Government created a new tribute with the goal of increasing the budget of the Brazilian public health-care system. This new tribute, called CPMF (an acronym for Provisory Contribution over Financial Maneuvering, in portuguese), is collected whenever an individual performs a financial maneuver to or from a bank account, and consists of a small percentage of the maneuvered value. The CMPF is only collected when the financial maneuver is performed between different individuals. When funds are transferred between two accounts belonging to the same individual, no tribute is collected.

We have adapted the system described in Section 6.2 in order for CPMF to be collected whenever a transference is performed between accounts belonging to different individuals. Figure 11 presents the architecture for this scenario. A new contract, TaxableTransferContract, and a new participant, DutyControl, have been added. The DutyControl participant component automates the task of logging all the requested financial operations. TaxableTransferContract uses TransferContract and DutyControl in order to perform transferences which trigger the collection of CPMF.

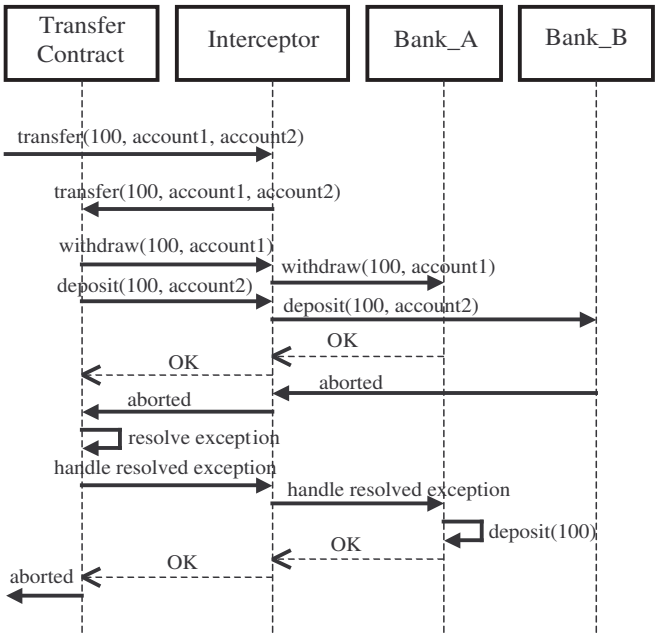

Figure 10: UML sequence diagram for an exception handling action in Scenario 2.

In this scenario, a new configuration is added to the Interceptor component, so that every funds transfer message is intercepted by TaxableTransferContract. The sequence diagram presented in Figure 12 illustrates the behavior of the system, upon receipt of a funds transfer request.

First, the Interceptor receives the request message asking for a funds transfer to be performed from an account at Bank A to an account at Bank B. The Interceptor checks if this request activates the trigger of some contract. In this example, the triggers of both TransferContract and TaxableTransferContract are activated, but the request is only intercepted by the latter, since it is in a lower layer of the architecture.

After receiving the intercepted request, TaxableTransferContract concurrently issues three request messages. Two of these messages request some information from Bank_A and Bank_B about the parties involved in the transference. The other request message initiates a simple funds transfer, as described in Section 6.2. In this case, TransferContract performs a subaction for TaxableTransferContract and any results of this subaction are delivered to the latter. If the funds transfer is successful, TaxableTransferContract checks if the two accounts belong to the same individual, using the information obtained from Bank_A and Bank_B about the parties involved in the operation. In case the accounts belong to different individuals, TaxableTransferContract issues a request to the DutyControl component, in order for the latter to log the operation, so that CPMF can be later collected. If everything runs smoothly, TaxableTransferContract issues a notification to the ClientComponent informing that the transfer has been successfully performed.

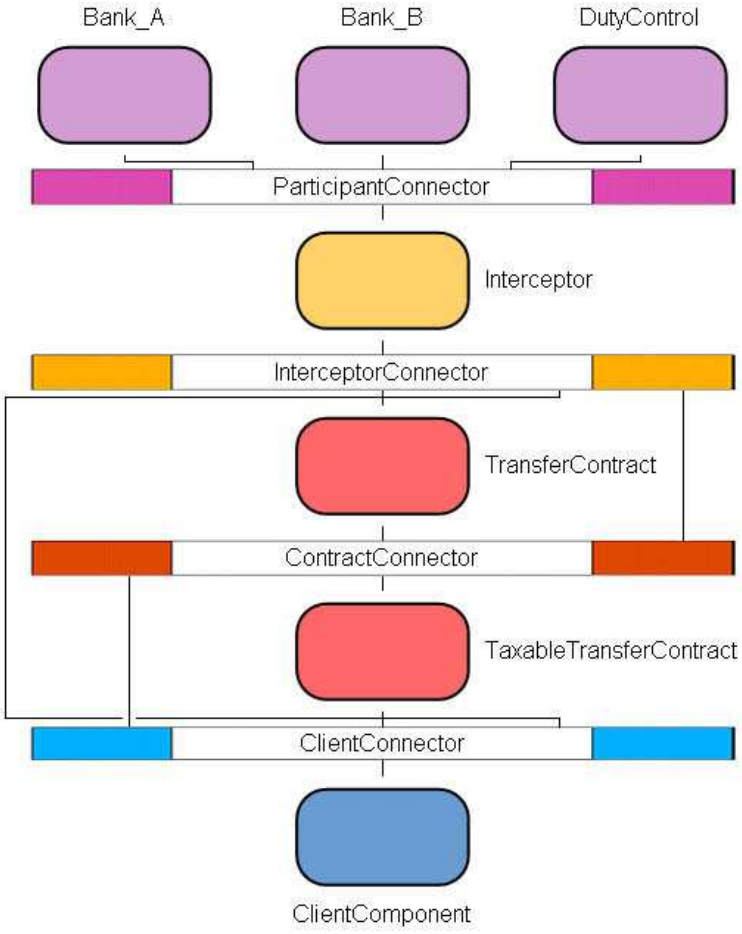

Figure 11: Banking system with three participants and two contracts.

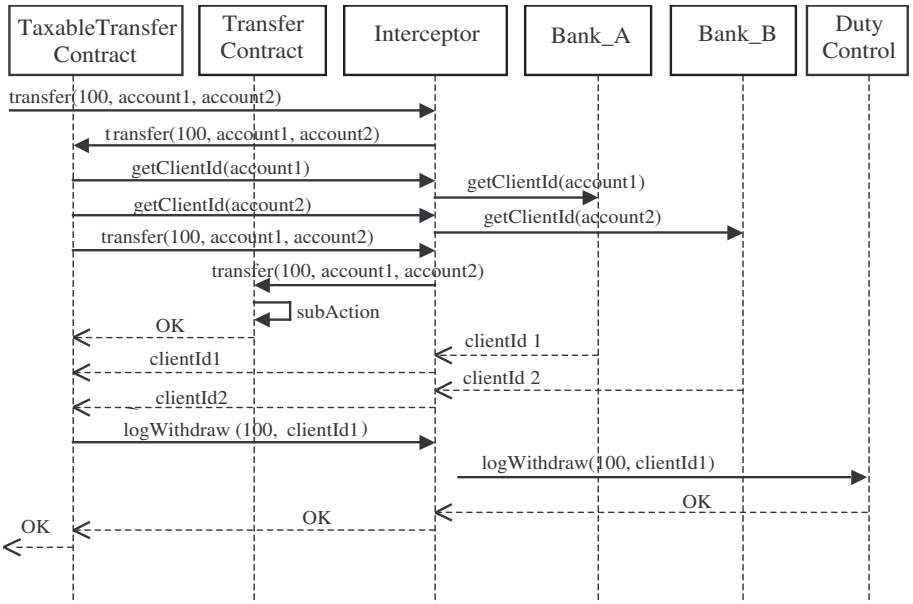

Figure 12: UML sequence diagram for Scenario 3.

If TransferContract aborts the funds transfer, TaxableTransferContract receives the *abort exception* raised by the former (Section 6.2) and resolves it. The recovery action performed by TaxableTransferContract upon receipt of this exception consists simply of not requesting the DutyControl component to log the operation and issuing an *abort exception* which is delivered to the ClientComponent.

7 Discussion

The case study presented in Section 6 helped us to assess some advantages and disadvantages of the proposed architecture. In this section, we summarize some of the lessons learned. First of all, the case study has shown that independence between client components (which *request* services) and participant components (which *provide* services) is an achievable goal. For instance, in Scenario 3, the participant components Bank_A, Bank_B, and DutyControl are completely oblivious to the fact that they are part of a system composed by many interacting components. Moreover, ClientComponent does not know anything about how the services provided by the system are implemented. It only knows about the roles that must be played in order for the intended services to be provided.

The participant components are unaware of the system-specific business rules. These rules are implemented by the composition contracts, which intercept the requests issued by the client components and compose the services provided by the participants in order to provide new services. New business rules are easily introduced by adding new composition contracts to the coordination layer (Section 4). Our experiments have shown that adding new composition contracts to an existing system incurs in almost no implementation over-

head to the system. In Scenario 3, neither contracts nor participants needed to be modified in order for the TaxableTransferContract component to be added to the architecture. It is important to notice, however, that the composite services provided by the coordination layer are still limited by the services provided by the interfaces of the participant components. Hence, when employing third-party components as participants, it may be more economic to build a brand new component than trying to employ one which does not provide enough functionality.

We have used our GME-based modeling environment for describing the architectural configurations of our case study. This approach has simplified the task of building the configurations, since we could use a friendly point-and-click interface, instead of coding directly, produced some architectural documentation, and, most importantly, provided the input for our code generation module, which was capable of generating approximately 20% of the implementation of the case study. Moreover, the use of these tools in the construction of the case study has helped us in perceiving some desirable features which have not been implemented yet. Section 9 describes some of these features.

In Scenario 2, when an exception was raised, TransferContract was capable of performing recovery actions which brought the system back to its original state, before raising an exception to the ClientComponent. The same held for Scenario 3. There may be cases, however, in which it is not possible to bring the system back to its original state, or even bring it to a consistent state at all. This limitation is inherent to the computational model assumed by our approach, since the participant components may encapsulate whole systems which may have been built without any regard for fault tolerance. A possible approach for alleviating this limitation is currently under study and is briefly discussed in Section 9.

8 Related Work

Our work has been inspired mainly by the ideas presented by Beder, et al [6], for the construction of dependable systems of systems [16] using an architectural approach. The authors present a software architecture based on the concept of idealised fault-tolerant component [1] for the construction complex software systems integrated with coordinated atomic actions.

The WSCA (Web Service Composition Action) concept also exploits the concept of CA Actions to enable the dependable composition of Web Services [21]. The primary difference between our work and the WCSA concept is that our approach also includes concerns about coordination contracts, which improve adaptability, and an architectural design to be applied to a more general class of service-based systems, not restricted to Web Services.

Similarly to our approach, Pires [19] proposes an architectural solution for providing reliable Web services compositions using a layered architectural style. However, this work only provides backward error recovery, not considering concurrent exception handling.

Zorzo and Stroud [24] describe an object-oriented framework for implementing dependable multiparty interactions (DMI's). The authors use this framework to implement the concept of CA actions. This work differs from ours in the fact that it does not employ contracts to assist in the integration of component-systems. Furthermore, the design of the

framework assumes that objects participating in a DMI implement their own error recovery measures. This assumption is reasonable for object-oriented systems, but may be too strong for component-based systems, as discussed in Section 3.

The work of De Lemos [9] describes an architectural style in which connectors are considered first-class entities which embody the description of collaborative behaviour between components. These special connectors are called *cooperative connectors* by the author. This work is complementary to ours in the sense that the composition connector (Section 4) may be seen as a cooperative connector described in a greater level of detail.

The concept of idealised C2 component (iC2C)[11] defines a structure for incorporating fault tolerance into component-based systems at the architectural level. This work has been later refined by Castor, et al [7], who defined an architectural-level exception handling system based on the concept of iC2C. Both of these approaches address the problem of building dependable component-based systems by employing architectural-level strategies. However, none of them deals with concurrent exception handling. Furthermore, they do not focus on the issues related to the composition of heterogeneous components.

9 Conclusions and Future Work

In this work, we proposed an architectural solution for the development of dependable software systems out of concurrent autonomous component-systems. This solution favours the adaptability, extensibility, and reliability attributes of the resulting system. The concepts of coordination contracts and CA Actions were adapted to a service-oriented approach applied to the system's software architecture.

Our main motivation for devising the architecture presented in this paper was the construction of dependable software systems using autonomous, heterogeneous, unreliable component-systems (systems of systems [16]). Initially, we intended to devise a conceptual infrastructure which allowed component-systems to be plugged and to interact with minimum effort. We believe this goal has been reached, provided the results presented in Section 6. However, after evaluating our empirical results, we have concluded that our architecture also allows contracts to be easily composed, promoting the reuse of existing business rules, and simplifying the task of introducing new ones.

The separation of concerns promoted by placing computation and coordination in distinct architectural layers makes it possible for participants to be unaware of the system-specific business rules. Participant components can be easily replaced by new versions and architectural mismatches can be dealt with be means of component wrappers [5]. Furthermore, new contracts can be added almost seamlessly to the coordination layer as the system evolves.

One of the key aspects for decreasing the complexity of communication in our architecture is the hierarchical organization of composition contracts. This hierarchical structure helps the interceptor to decide which contract will receive a given message when multiple triggers are simultaneously activated. However, there is a tradeoff associated with the hierarchical organization of contracts. The contract hierarchy imposes a partial order on the execution of contracts that may restrict the order the contract can be activated. We are

currently studying some possible solutions to this problem.

In the architecture employed in the case study (Section 6), only one interceptor component is used for the whole architecture. This may create a bottleneck for systems with high availability requirements subject to heavy loads. A possible solution for this problem consists of creating multiple distributed instances of the interceptor component and distributing the load between these instances.

The framework described in Section 5, at its current version, only supports the static configuration of the interceptor component. Since the architectural configurations of the contracts and participants depend on the configuration of the interceptor, these elements can not be added dynamically to a system. Extending our framework in order to support dynamic configuration would promote availability, because the system execution would not need to be interrupted for new participants and contracts to be introduced.

In the architectural components described in Section 4.3, there is an explicit separation between normal and abnormal behaviour. However, until the present moment, this separation is purely structural and no constraints on the interaction between the normal activity and exception handler components have been defined. Hence, another future work consists of introducing the concept of idealised C2 component in our architecture as a means for defining exception handler components and damage confinement regions [1]. In fact, we intend to merge the exception handling system proposed by Castor et al[7], with our architecture in order to guarantee that both the interactions and the interacting parties in a system are dependable.

References

[1] T. Anderson and P. A. Lee. *Fault Tolerance: Principles and Practice.* Prentice-Hall, 2nd edition, 1990.

[2] L. Andrade and et al J. Fiadeiro. Coordination for orchestration. In *Proceedings of the 5th International. Conference on Coordination Languages and Models*, Lecture Notes in Computer Science 2315, pages 5–13. Springer-Verlag, 2002.

[3] L. F. Andrade and J. L. Fiadeiro. Coordination patterns for component-based systems. In *Proceedings of the V Brazilian Symposium on Programming Languages (SBLP'2001)*, pages B29–B39, Curibita, PR, Brazil, 2001.

[4] L. F. Andrade and J. L. Fiadeiro. Feature modeling and composition with coordination contracts. In *Proceedings of Feature Interaction in Composed System (ECOOP 2001)*, pages 49–54, Universitat Karlsruhe, 2001.

[5] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice.* SEI Series in Software Engineering. Addison-Wesley, 2nd edition, 2003.

[6] D. M. Beder, B. Randell, A. Romanovsky, and C. M. F. Rubira. On applying coordinated atomic actions and dependable software architectures for developing complex systems. In *Proceedings of the 4th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'2001)*, Magdeburg, Germany, May 2001.

[7] Fernando Castor Filho, Paulo Asterio de C. Guerra, and Cecília M. F. Rubira. An architectural-level exception-handling system for component-based applications. In R. de Lemos, T. Weber, and J. Camargo Jr., editors, *Proceedings of the First Latin-American Symposium on Dependable Computing*, LNCS 2847, pages 321–340. Springer-Verlag, 2003.

[8] Flaviu Cristian. *Dependability of Resilient Computers*, chapter Exception Handling. BSP Professional Books, 1989.

[9] Rogério de Lemos. Describing evolving dependable systems using co-operative software architectures. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, pages 320–329, 2001.

[10] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[11] Paulo Guerra, Cecília Rubira, and Rogério de Lemos. An idealized fault-tolerant architectural component. In *Proceedings of the 24th International Conference on Software Engineering - Workshop on Architecting Dependable Systems*, May 2002.

[12] J. C. Knight. SafetyCritical systems: Challenges and directions (summary of state-of-the-art presentation). In *Proceedings of the 24th International Conference on Software Engineering*, Orlando, FL, USA, 2002. ACM Press.

[13] H. Kreger. Web services conceptual architecture, May 2001. IBM Software Group.

[14] B. W. Lampson. Atomic transactions. In *Distributed Systems: Architecture and Implementation*, Lecture Notes in Computer Science, pages 254–259. Springer-Verlag, Berlin, 1981.

[15] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason IV, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The generic modeling environment. In *Workshop on Intelligent Signal Processing*, Budapest, Hungary, May 2001.

[16] M. Maier. Architecting principles for systems-of-systems. *System Engineering*, 1(4):267–284, 1998.

[17] N. Medvidovic, P. Oreizy, and R. N. Taylor. Reuse of off-the-shelf components in c2-style architectures. In *Proceedings of the 1997 Symposium on Software Reusability*, 1997.

[18] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[19] P. F. Pires. Building reliable web services compositions. In *Proceedings of the NET.Object Days Conference (WS-RDS'02)*, pages 551–562, Erfurt, Germany, October 2002.

[20] M. Shaw and P. Clements. A filed guide to boxology: Preliminary classification of architectural styles for software systems. In *Proceedings of COMPSAC'96*, Washington, DC, USA, August 1996.

[21] F. Tartanoglu, V. Issarny, A. Romanovsky, and N. Levy. Dependability in the web services architecture. In R. de Lemos, C. Gracek, and A. Romanosvsky, editors, *Architecting Dependable Systems*, Lecture Notes in Computer Science, LNCS 2677. Springer-Verlag, June 2003.

[22] R. N. Taylor, N. Medvidovic, K.M. Anderson, Jr. E. J. Whitehead, and J.E. Robbins. A component- and message- based architectural style for GUI software. In *Proceedings of the 17th International Conference on Software Engineering*, pages 295–304, April 1995.

[23] J. Xu, B. Randell, A. Romanovsky, C. Rubira, R. Stroud, and Z. Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, pages 499–508, Pasadena, USA, 1995.

[24] A. Zorzo and R. J. Stroud. A distributed object-oriented framework for dependable multiparty interactions. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, pages 435–446, Denver, CO, USA, November 1999.