A CONTRACT BASED APPROACH FOR PROVIDING RELIABILITY TO SERVICES BASED APPLICATIONS

Alberto Portilla-Flores

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE
Spécialité **MSTII/INFORMATIQUE**

Arrêté ministériel : 7 août 2006

Présentée et soutenue publiquement par

PORTILLA-FLORES Alberto

Le **15 Octobre 2010**

« UNE APPROCHE A BASE DE CONTRATS POUR LA COORDINATION FIABLE DES SERVICES »

Thèse dirigée par **Christine COLLET** et codirigée par **José Luis ZECHINELLI-MARTINI**

JURY

Mr. Olivier PERRIN	MCF HDR Université Nancy 2,	Rapporteur
Mr. Samir TATA	Prof. Institut TELECOM Paris	Rapporteur
Mr. Omar BOUCELMA	Prof. Univ. Paul Cézanne Aix-Marseille 3	Examinateur
Mr. Luciano GARCIA-BAÑUELOS	PhD University of Tartu	Examinateur
Mme. Genoveva VARGAS-SOLAR	PhD CNRS, LIG	Examinateur

Thèse préparée au sein du **Laboratoire d'Informatique de Grenoble** dans **« l' Ecole Doctorale Mathématiques, Sciences et Technologie de l'Information, Informatique (27) »**

A CONTRACT BASED APPROACH FOR PROVIDING RELIABILITY TO SERVICES BASED APPLICATIONS

An Abstract of a Thesis

Presented to

Université de Grenoble

In Partial Fulfillment

of the Requirements for the Degree

Ph.D. in Computer Science

By

Alberto Portilla Flores

2010

Abstract

This research work addresses reliability of services coordination expressed as non-functional properties (e.g., performance, security, atomicity, persistency, etc) that must be ensured and enforced at execution time. Existing systems, models and languages provide ad-hoc solutions that weave the application logic, expressed as services coordination, with non functional properties, leading to applications difficult to evolve and maintain. In contrast, our approach promotes separation of concerns such that reliability can be personalized for a given services coordination where some services can run under persistent connection conditions, others participate in atomic executions, etc. Therefore, we propose a contract model for associating non-functional properties to a services coordination and associated contract evaluation strategies for verifying and enforcing them at run time. A proof of concept is presented, ROSE is a reliable services coordination execution engine able to add exception handling and atomicity properties to a given services coordination.

Résumé: Ce travail de recherche aborde la fiabilité de la coordination de services exprimée comme des propriétés non-fonctionnelles (e.g. la performance, la sécurité, l'atomicité, la persistance, etc.) qui doivent être assurées et renforcées en cours dexécution. Les systèmes existants, les modèles et les langages fournissent aujourd'hui des solutions ad hoc qui tissent la logique applicative avec les aspects non-fonctionnels, conduisant à des applications difficiles à faire évoluer et à maintenir. Notre approche favorise la séparation et la personnalisation d'aspects tels que la fiabilité. Nous proposons le modèle de contrats COBA pour décrire l'association des propriétés non-fonctionnelles à une coordination de services, et l'évaluation des contrats grâce aux

stratégies pour les vérifier et les renforcer en cours d'exécution. Une expérimentation et une validation du modèle ont été réalisées à travers la mise en œuvre de ROSE, un moteur d'exécution de coordinations de services fiables.

Acknowledgements

This thesis has been prepared during my PhD studies at the Grenoble Informatics Laboratory at the Université de Grenoble, France and the French Mexican Laboratory of Informatics at the Fundación Universidad de las Américas Puebla, Mexico. Thanks to the PROMEP-SEP Mexican program, the Universidad Autónoma de Tlaxcala, and the Jenkins Fellowship Excellence program at F-UDLA for funding my studies.

I would like to thank my advisors. Prof. Christine Collet, for sharing with me her deep insights of the research world and for helping me to face the challenges of this thesis. PhD Genoveva Vargas Solar, for encouraging my autonomy and giving me irreplaceable teachings, it is for me a great honor to be her first graduated Mexican PhD student. Prof. José Luis Zechinelli Martini, for many thoughtful conversations regarding the right way of doing research. Prof. Luciano García Bañuelos, for his support and for the extensive discussions which served to clarify several issues related to my research work.

I am grateful with, Prof. Omar Boucelma, PhD Olivier Perrin, Prof. Mauricio Osorio Galindo, and Prof. Samir Tata for their suggestions that enabled me to improve the quality of this document and for being part of the dissertation committee.

Many thanks to the members of the HADAS team at LIG: Christophe, Fabrice, Marie Christine and Alexandre, research scientists of the team; Trinidad, Genaro, Laurent, Remi, Ma. Del Pilar, Giang, Tan and Nagapraveen, ancient PhD Students; Victor, Javier, Carlos, Benjamin, Mohammed and Lourdes current PhD Students; thanks to all of them for sharing with me their thoughts, hints, culture and experience, while I was in Grenoble.

Last but not least, I would thank to my family in Mexico. Gracias a mis padres,

Alberta R. Flores Flores y Alfredo Portilla Mendieta, por su amor y su apoyo en todas las decisiones que he tomado en la vida. A mis dos hijos, Alberto André y Brenda Mayté, dos ángeles que el Señor me envió para acompañarme en este camino que aún debo terminar de recorrer; gracias por su paciencia y amor; a pesar de todo, ya pueden decir que papá es "Doctor". A mis hermanos, Alma Bertha, Edgar Alfredo y Víctor Hugo, porque gracias a sus ejemplos aprendí a superar los obstáculos de la vida. Prof. Jaime Flores Flores, gracias tío, porque de ti aprendí que la mejor herencia que podemos recibir es la educación. También gracias a mis familiares cercanos, Alma, Nico, Karla Carol, Karla, Tito y Ethian, gracias por estar al pendiente de mi trabajo. All of them were crucial supporters for finishing this thesis. Finally, I reserve a special thank to Carmen, my fiancee, for your endless love during this short time, I hope this will be forever.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Résumé: Les applications à base de services sont construites en coordonnant des composants logiciels existants que l'on appelle les services. La coordination de services définit l'interaction entre les services et elle est spécifiée par des propriétés (i) fonctionnelles qui définissent la logique applicative ; et (ii) des propriétés non fonctionnelles qui concernent des stratégies pour exécuter une coordination (e.g. des propriétés de fiabilité, de sécurité ou d'adaptabilité). Notre travail concerne la modélisation des aspects non fonctionnels des applications à base de services. Ce chapitre énonce le problème adressé dans cette thèse, il spécifie les objectifs et énumère les contributions de notre travail de thèse.

1.1 Context and motivation

Services based applications is an emerging paradigm for the construction of information systems. This kind of systems are not built from scratch, but using existing software components called services [ACKM04, SH05, Erl05]. A services based application is composed by coordinating several services providers where the application logic is abstracted by a coordination specification that usually captures functional and non functional aspects [PCVS+06, Por06b, PVSZM+06]. While functional aspects describe what the system does, the non functional aspects describe how the execution must be done with respect to some observable attributes like reliability, adaptability or security. Our work is related to model non functional aspects of services based applications.

Along this document we use a "purchase tickets application" example to illustrate our approach. Figure 1.1 introduces the application logic. It includes activities that

must be completed (rounded boxes) and the execution order in which they must be executed (arrows) for purchasing concert tickets on Internet. Given the concert information, for example concert name, seats, date, and time, the purchase is processed and payment is granted. Once the purchase has been authorized, the payment must be done, the tickets must be sent, and publicity for other events must be sent too. Besides, there are several business rules that must be considered, for example the following rules:

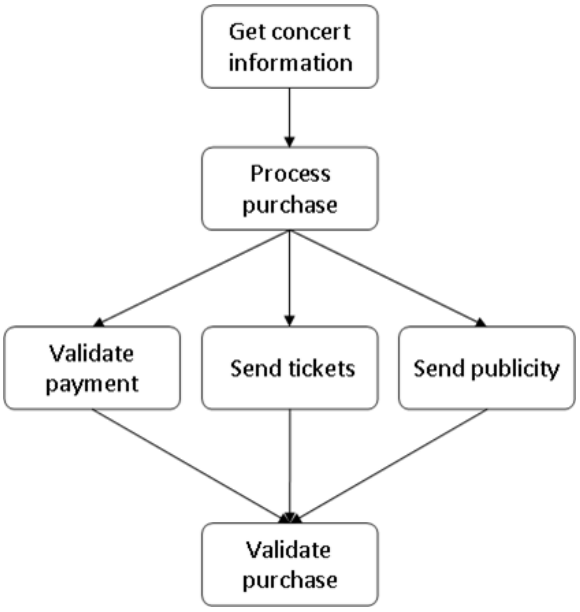

Figure 1.1: Purchase ticket application

- The activity *Get concert information* can be completed in several ways (e.g., by telephone or via Internet). It is a vital activity for the application because it implies not only information about the concert but also business policies for retaining customers. Thus it must retried in case of failure.

- The payment (i.e., activity *Validate payment*) can be granted in different ways (e.g., pre-paid, credit card payment, or personal check) and it can be done with different providers (i.e., Visa, Mastercard or American Express).

- Tickets can be sent (i.e., activity *Send tickets*) or picked up directly at the ticket office according to customer preferences.

- The activity *Send publicity* is not vital for the success of the application because it only exploits customers information for publicity purposes.
- A purchase order can be completed only if it was paid.

The above business rules impact the execution according to certain profiles (e.g., customer profiles) that reflect several execution cases. They do not modify the application logic but the execution, and they represent the non functional aspects of the application some of which are related to reliability of applications [Por06b, Por06a].

Nowadays there are academic and industrial efforts that have proposed standards, languages, tools and middlewares for building service oriented applications [ACKM04, Erl05, SH05]. Until now, a lot of effort has been devoted to services coordination and associated protocols that are used for adding non functional properties to coordinations (e.g., reliability aspects, security, adaptability, see [CCF+04, LW03, Fur04, BCR05, TMW+04, DFDB05, Bhi05, PBM02, NFG+05, VV04, HW06, SABS02, Lom05, ZHMS06]). Yet, these approaches do not considers the dynamic context that must face applications:

- When application logic is modeled, developers must reflect all execution cases: exceptional situations and application requirements [Por06a, Por06b, Por08a] (e.g., what to do if *Send tickets* fails). This situation makes that in the presence of new exceptions the application must be modified.
- A same application logic must be related to non functional aspects according to business rules (e.g., normal users and VIP users must be processed in a different way). This situation is very common nowadays because customer relationships are crucial in several applications.

Therefore, functional and non functional requirements are provided using *ad-hoc* strategies [PCZMHB08, PVSGB+08]. We noticed that hard coding such aspects make applications very complex in the sense that they are hard to maintain, not flexible, not adaptable and contradicts reusability philosophy of services oriented approach.

1.2 Problem statement and objective

1.2.1 Problem statement

The problem we address is that of providing non functional properties to services based applications. We consider a non functional property as the one that that describes

how the execution must be done with respect to some observable attributes (e.g., reliability, security, adaptability). Until now most of existing approaches provide ad-hoc solutions that weave the application logic, expressed a services coordination, with non-functional properties, leading to applications difficult to evolve and maintain [PCZMHB08, PVSGB+08].

1.2.2 Hypothesis

Given a services coordination describing the logic of a service based application, it is possible to add non functional properties by defining them independently from application logic, using contracts. This is possible because, there are non functional properties (e.g., atomic behavior, exception handling, and persistency) that can be ensured at execution time by the evaluation of its contracts. A contract must associate a property to an execution unit or to a set of contracts and to define the recovery operations to be considered for enforcing the property in the occurrence of specific situations.

1.2.3 Objective

To propose a model for associating non functional properties to a services coordination and associated contract evaluation strategies for verifying and enforcing them at run time. The model must provide concepts for representing:

- Application logic and non functional requirements in an orthogonal way.
- The specification of non functional properties must be done in a simple and expressively way (i.e., close to business rules that usually express such requirements).

Besides, a proof of concept must be provided to show the feasibility of the model:

- To show how to enact the services coordination with non functional properties. Details about the evaluation process of contracts at execution time must be presented.
- A general architecture must be proposed for implementing a contract evaluator that interacts with a coordination engine for triggering and executing a service coordination with non functional properties expressed as contracts.

1.3 Contributions

The contributions of this thesis can be summarized as follows:

- We propose COBA (i.e., COntracts BAsed model[1]), a model for representing the non functional aspects of a services coordination through the notion of contract [Por06a, PVSC+07a, PVSC+08a]. We assume that a services coordination is represented as:
  - A set of execution units.
  - A control flow represented by a set of execution dependencies among the execution units
  - An execution history represents ordering constraints over the state of the set of execution units.

  The concepts of COBA are:
  - Execution unit: it represents the execution of a process (e.g., an activity of a services coordination).
  - Property: It represents a non functional aspect.
  - Rule: it specifies the reactions to be executed for enforcing a property under a given situation.
  - Contract: it represents the association of a property to an execution unit or to a set of contracts and the rules to be considered for the property.
- We present a analysis of how reliability has been tackled for services coordination [PVSZM+06, PCVS+06, PHEO+08b, PHEO08a]. As a result of such an analysis, we use the concepts of COBA for providing reliability properties to given services coordination. We define a reliable services coordination as the one that tolerates failures at execution time. Therefore, we analyze how exception handling and state persistency can be specified by means of contracts:
  1. How to treat exceptions over the execution of execution units.
  2. How to provide atomic behavior to sets of execution units.
  3. How to treat the persistency guarantees of execution units.

[1] COBA is also the name of a Maya archeological site, located in the state of Quintana Roo, Mexico. In Maya it means "Abundant water".

4. How to handle the execution state of activities sets.

- A key element of our approach is the evaluation of contracts. Therefore, we propose strategies for evaluating the contracts associated to a services coordination [PVSC+08a, PHEO08a, PHEO+08b]. The strategies specify when to evaluate a contract with respect to the execution of an execution unit and when and how to execute the reactions of the contract.

- We introduce a general architecture with the basic requirements that a coordination engine must have to enact a coordination with reliable contracts. Using such architecture, we present ROSE, a services coordination engine which provides atomic behavior to web services [PVSC+07b, HBPZM07].

1.4 Document organization

The remaining of this document is organized as follows:

- Chapter 2 presents a state of the art of approaches providing reliability to services coordination.

- Chapter 3 describes COBA, the contract based model that we propose for adding non functional properties to services coordination.

- Chapter 4 presents the evaluation process of contracts at execution time.

- Chapter 5 introduces the how reliability can be addressed by means of contracts.

- Chapter 6 presents a proof of concept, ROSE a services coordination engine that enacts services coordinations with atomic requirements.

- Chapter 7 concludes this document by presenting the lessons learned and the ongoing work.

Chapter 2

Reliable services coordination

This chapter introduces the concepts of a reliable services coordination. A reliable services coordination tolerates failures at execution time. Reliability is very relevant due to current business oriented nature of services based applications. It provides QoS to applications by means of ensuring access to resources in a continuous way. To this respect, several works has been devoted to provide reliability to applications using several approaches (e.g., see [CCC+04, CCF+04, LW03, Fur04, BCR05, TMW+04, DFDB05, BGP05, PBM02, NFG+05, VV04, HW06, SABS02, Lom05, ZHMS06]). The way on which reliability is addressed, impacts the definition itself of the aspects (i.e., the level of abstraction used for its definition) and the type of developed application (i.e., the level of adaptability, maintenance and evolution that the application has). Such aspects are critical in distributed and heterogeneous environments (e.g., the Web), where the time to market and the constant evolution drives the software development.

This chapter is organized as follows. Section 2.1 defines the notion of reliability in the context of services coordination. Section 2.2 discusses about of existing approaches addressing atomic behavior to services coordination. Section 2.3 presents approaches providing persistency. Finally, Section 2.4 summarizes the present chapter.

Résumé: Ce chapitre définit la coordination fiable de services. La fiabilité assure des propriétés de qualité de service (QoS) des coordinations de services. Ainsi, une coordination fiable des services tolère des exceptions pendant son exécution. Plusieurs approches ont été proposées pour construire des applications fiables à base de services. Ce chapitre fait état des travaux qui ont adressé des aspects de fiabilité, en particulier, l'atomicité et la persistance dans les systèmes à base de services. Les approches sont décrites et analysées et leurs limitations sont mises en évidence.

2.1 Reliability

A reliable services coordination is able to maintain its functionality in unexpected situations (i.e., failures). Failures are of two kinds: semantic and system failures.

- A semantic failure is related to the application logic. Tolerating this kind of failures implies to handle states, not completely defined in the normal execution flow. The objective is to reach consistent execution states where the execution can be continued or finished. Semantic failures are tackled by providing atomic behavior (i.e., atomicity) to services coordination.

- A system failure is related to execution infrastructure failures. Tolerating this kind of failures ensures that given a coordination, its execution state survives in spite of system failures. System failures are tackled by providing persistency guarantees to services coordination.

Aspects related to reliability were first addressed in the database area by means of ACID properties (i.e., atomicity, consistency, isolation and durability) of transactions. In such a context, a transaction is an execution unit composed of several DB operations (i.e., queries and updates) that ensures the database consistency by controlling concurrent access to shared data according to ACID properties [DA82, Elm92, OV99, WV98, GMUW00]:

- Atomicity is based on the principle of "all-or-nothing" which means that either all operations of a transaction are completed or none of them. Therefore, an atomic transaction is treated as an execution unit which either is executed or not.

- Consistency refers to database integrity. A consistent database state is expected before, during and after transactions execution.

- Isolation is related to visibility degree of results within a transaction to other concurrent transactions. The current transaction behaves as the only transaction been executed.

- Durability concerns to data persistency. Once a transaction has been committed, its results survive system failures.

The need of revisiting such concepts to address new contexts (e.g., services coordination) has been pointed out by several authors.

Jim Gray [Gra81] proposes to integrate transaction constructors within programming languages. This strategy implies to use such constructors within the definition of the application logic which conducts to build ad-hoc applications.

Advanced transaction models [Elm92] introduce useful concepts such as control flow, compensation notion and complex committing protocols to distributed database environments. Such protocols usually are coded using programming languages resulting in applications hard to maintain.

Gustavo Alonso et al [AAA+98] extend two products of IBM with advanced transactional models as a proof that such models remain as theoretical efforts not being used in commercial products. This work represents an ad-hoc implementation of concepts related to database environments.

David Lomet and Gerhard Weikum [LW98] use logging and recovery techniques for supporting system failures for non database applications. The approach assumes the same requirements for all the components of the applications which makes the application hard to adapt to dynamic environments.

Finally, Frank Leymann and Dieter Roller [LR97] introduce the concepts of compensation and atomicity spheres (i.e., logical boundaries) into workflows. This approach proposes the separation of application logic and atomicity aspects in a homogeneous execution context (i.e., workflows).

In the context of services coordination reliability has taken a new dimension which demands adaptability and extensibility due to autonomy of services and the dynamic environment over which the applications are executed. Therefore, in the next sections we revise some approaches that tackle reliability (i.e., atomicity and persistency) according to the way they integrate reliability to services coordination.

2.2 Atomicity

Atomicity in the classic sense is ensured by means of transactions. In such a context, a transaction is a set of DB operations treated as an atomic execution unit which either is executed or not. In the case of services coordination, it associates a relaxed notion of atomicity that has been used to handle semantic failures by means of recovery strategies (e.g., forward execution, backward recovery, forward recovery) [HA00, CSDS03, BDO05]. Relaxed atomicity extends the classic concept of atomicity [PVSZM+06]:

- *Strict atomicity* respects the requirement of executing "all or nothing" [EGLT76].

For example, in the "purchase tickets application" this kind of atomic behavior can be associated to activities *Validate payment* and *Send tickets* to ensure that either a reservation which is paid is delivered, or a reservation which is not is not paid is not delivered. In this case strict atomicity is hard to provide because while the execution of both activities does not commit or fail, the resources used by them remain blocked.

- *Semantic atomicity* uses the notion of compensation for dealing with the impossibility of rolling back some operations [GM83]. A compensation amends the effects of a committed operation. Compensation is necessary because there are some operations that cannot be rolled back. For example in the "purchase tickets application", it is not possible to undone the activity *Validate payment*, therefore if it is necessary to cancel it, a compensation action is necessary (e.g., to apply an extra charge per cancelation).

- *Semi-atomicity* introduces the opportunity for deciding between two or more execution paths for committing [ZNBB94]. It assumes that there are several possible ways of committing a set of operations. For example, in the "purchase tickets application", tickets can be send to an address or they can be picked up at an office.

In the next sections we analyze existing approaches according to, i) the type of atomicity they provide, and ii) the way on which atomicity is integrated with the services coordination:

- Atomicity provided by means of protocols (i.e., protocol based) is when there are some transactional protocols that are used for ensuring a kind of atomicity.

- When atomicity is provided as a part of the coordination (i.e., coordination based), there are some atomicity constructors that complements the coordination operators.

- Atomicity is also provided by means of defining the required interactions among several coordinated activities (i.e., interaction based) to address a specific atomic behavior.

2.2.1 Protocol based

In this kind of approaches, the atomicity is implemented within the coordination by using pre-defined protocols (e.g., extensions of the two phase commit protocol[1] and advanced transactional models[2], see [Por06b]). When using a protocol it is assumed that operations within a transaction have a homogeneous behavior with respect to transactional requirements (e.g., all operations accept reservation for committing) and therefore it defines the way on which the processes for committing or undone a transaction must be done. It is also assumed that, services and execution infrastructure offer functions related to support atomicity (e.g., the compensation capability). Following approaches are representative examples of protocol based approaches:

- The Web services transaction (WS-Tx) [CCC+04] is a protocol to provide strict and semantic atomicity to services coordination. WS-Tx is layered over the Web services coordination specification (WS-C) [CCF+04, LW03]. Ws-Tx extends WS-C to create a transactional coordination context offering two types of atomicity:

  - Atomic transaction (AT). It is used to coordinate activities with strict atomicity-like behavior.
  - Business activity (BA). It is used to coordinate activities with a semantic atomicity-like behavior. In this protocol business activities can be sub divided into small ones called scopes. A scope is a collection of operations that can be nested to arbitrary degrees.

  WS-Tx has defined five protocols for committing based on classic 2PC: $Completion$, $CompletionWithAck$, $Phasezero$, $2PC$, and $Outcomenotification$.

- The business transaction protocol (BTP) [Fur04] is a protocol for coordinating processes with strict and semi-atomicity behavior. A BTP coordination protocol is a set of well-defined messages that are exchanged between participants to address a transactional behavior. BTP defines two coordination protocols to provide transactional behavior:

[1] The two phase commit protocol (2PC) consist of two phases : i) during the first phase (preparation) every participant in transaction extern its response to commit or abort, and ii) in the second phase (commitment), a coordinator makes a global decision which is communicated and executed by all participants of the transaction.

[2] Most representative examples of advanced transactional models are: saga [GMS87, GMGK+91, GMUW00], flexible transactions [ELLR90, ZNBB94], and contracts [WR92].

  - Atom protocol implements a strict atomicity behavior.
  - Cohesion protocol enables the definition of semi atomicity, where some participants commit and others cancel based on some pre-defined business rules. A cohesion commits using the rules that users define.

  Atom and cohesion protocols use a modified two phase commit protocol (2PC) which cannot be adapted or extended to new requirements. During the second phase of 2PC protocol, the set of rules is used by coordinator to make a decision about commit or abort.

- [BCR05] extends BTP by means of an ontology expressed in OWL-S. The ontology categorizes services according to three aspects:

  - The functionality of the service.
  - How it is accessible the service.
  - How the service works.

  Regarding to atomicity a service is classified as unprotected, semi-protected, protected, negotiable, and real. This classification is used for implementing the atom, cohesion, and atomic transactions of BTP on top of the transactional properties that a service can provide. In such a way, it is possible to provide strict and semi-atomicity behavior to a given services coordination.

- [TMW+04] proposes a policy based transactional model implemented on top of WS-Coordination, WS-Transaction, and WS-ReliableMessaging protocols. A policy is used to advertise and to match three types of atomicity:

  - Direct transaction processing provides atomicity based on the 2PC protocol.
  - Queued transaction processing provides atomicity using a non blocking 2PC protocol.
  - Compensation-based transaction processing provides atomicity using the notion of compensation which relaxes the notion of atomicity.

  Using such transaction models it is possible to provide strict and semi-atomicity behavior.

- [DFDB05] introduces a protocol for dynamic composition of services based on the tentative hold protocol that enables the definition of an atomic behavior

for activities. The tentative hold protocol adds a phase to 2PC protocol where participants can request tentative reservations on the resources that they want to use in following phases. This new phase can be seen as an exchanging of messages prior to the transaction for minimizing compensation actions. Using such a protocol, it is possible to associate an atomic behavior to a set of participants of a coordination as follows:

- It is defined which committing conditions must have each participant at execution time (e.g., a participant must fail).
- The minimum number of committed participants that must be for accepting the execution (e.g., strict atomicity behavior requires that all participants commit).

Using this appproach, strict atomicity and semi-atomicity can be provided to a services coordination.

2.2.2 Coordination based

A second strategy for addressing atomicity is to define it explicitly within the application logic. This is done by adding sequences of activities that implements transactional constructors such as Jim Gray proposes. In that way, it is possible to implement well known advanced transactional models within a given coordination language. Following works are good examples of coordination based approaches:

- [Bhi05] proposes an approach for Web services composition using a transactional approach based in patterns. This approach consist of a set of algorithms and rules for assisting to compose coordinations with transactional behavior. A transactional specification is composed as follows:
  - It includes the set of services to be considered within the desired atomic behavior.
  - An atomic behavior specified by means of a set of accepted termination states. An accepted termination state defines the state where a service can finishes its execution (e.g., some services can commits and others can fail). For example, in the "purchase tickets application" can be acceptable to finish the application with an order cancelation applying an extra charge.

This notion of accepted termination states relaxes atomicity. Using this approach it is possible to define strict atomicity, semi-atomicity or semantic atomicity behavior to a services coordination.

- WebTransact [PBM02] extends the Web Services Description Language (WSDL) with transactional behavior definitions. In this approach, a coordination is written using the so called Web Service Transaction Language (WSTL), which is built on top of WSDL [CMRW07]. WSTL offers a way to describe how to interact with services and its transactional support. A service can be associated with following behaviors:
  - The execution of a service cannot be aborted after being started or undone after it commits.
  - The execution of a service can be aborted after being started and can be compensated after it commits.
  - The execution of a service can be retried in case of failure.

  Exploiting the possible transactional behavior of each service it is possible to provide strict atomicity, semi-atomicity to a services coordination
- [NFG+05] introduces the so called GAT model (guard-activity-triggers). It offers a coordination language based on event-condition-action rules. Using such an approach, the execution is a coordinated execution of a number of actions where the control flow of the application is not defined explicitly. An action is an ECA-rule representing the execution of a set of activities that corresponds to normal execution and to exceptional situations. A rule is invoked when an event is detected and its actions are executed only if its guard conditions hold. A rule can be used as follows:
  - It can capture the normal control flow by defining ordering dependencies among activities within the guard conditions (e.g., a service A must be executed before the service B).
  - It can define how to treat the execution of activities to grant a given atomic behavior (e.g., what to do in case of failure).

  Using such an approach it is possible to implement the notion of accepted termination states to provide strict atomicity, semi-atomicity or semantic atomicity behavior to services coordination.

2.2.3 Interaction based

A third strategy for defining atomicity is to separate the specification of the transactional behavior and the services coordination. This captures the interactions that are necessary to provide an atomic behavior.

- [VV04] adapts the model presented in [SABS02] for supporting Web services coordination with transactional properties. Transactional properties are attached to coordination as follows:
  - Each activity is classified according to its atomic capabilities: i) whether an activity can be compensated or not, ii) whether an activity can be retried or not in case of failure, and iii) whether an activity is vital or not for an application.
  - Atomicity is specified by accepted termination states. Recall that, a termination state defines a possible state where is acceptable to finish an application.

  Using such an approach it is possible to provide strict atomicity, semi-atomicity or semantic atomicity behavior to services coordination.

- [HW06] uses the meta-model ACTA concepts [CR90] for providing a transactional model for Web services coordination. The ACTA model provides a way of analyzing transactional protocols in terms of the execution effects of a transaction over other transactions (i.e., commit and abort dependencies), and over shared data (i.e., view set and access set). In the so called transactional Web service orchestrations model (TWSO) transactional requirements are expressed as dependencies among operations (e.g., abort, commit, and inter-operations dependencies) but it does not include view set, access set and delegation properties as is considered in the ACTA framework. In such a way, advanced transactional models are defined by using the TWSO model and attached to a given orchestration orthogonally.

2.2.4 Discussion

In this section we present several approaches to provide atomic behavior to services coordination (i.e., [CCC+04, Fur04, BCR05, TMW+04, DFDB05, Bhi05, PBM02, NFG+05, VV04, HW06]). With respect to existing approaches we have following aspects to discuss:

- We think that understanding atomicity concepts implies some complexity for application developers. Therefore, understanding transactional protocols (i.e., [CCC+04, Fur04, BCR05, TMW+04, DFDB05]) for defining the interactions related to an atomic behavior, conducts to a waste of time and application soundness is compromised.

- Approaches that propose new coordination languages and constructors (i.e., [Bhi05, PBM02, NFG+05]) for enacting atomic behavior weaves the definition of the application logic and the reliability aspects resulting in complex applications hard to maintain.

- Most of applications implemented using existing approaches are *ad-hoc* applications, which contradicts the current spirit of reusing practices existing in software engineering and services based applications.

Finally, a very important aspect that is poorly addressed is the necessity of adapting a given coordination to several atomicity requirements. This necessity is addressed, for example, in workflow technology by the "case" concept [vdAvH04]. It is assumed that a workflow has several use cases, and therefore each case has its own execution requirements. This situation is very common in today applications. For example, in the "purchase tickets application", a concert can accept only tickets payment with credit card (i.e., strict atomicity), but another concert can accept several payment types (i.e., semi atomicity). Because of existing approaches hard code the atomicity within the coordination, it is hard to provide several atomicity requirements for a given application.

2.3 Persistency

A first notion of persistency appears with the concept of durability in the ACID transactions. Durability is the guarantee that, when a transaction is committed its results persist [OV99, DA82, Elm92]. In the context of the services coordination, persistency ensures that given a coordination, its execution state survives in spite of system failures (i.e., execution state durability) [LW98, BLSW04, PCZMHB08]. A system failure is related to execution infrastructure failures. In case of a system failure, a recovery process returns the application execution state to a consistent state, because its execution state persists. For example, in the "purchase tickets application", if a

system failure occurs when *Bank authorization* activity is being executed there are two options for recovering the execution:

1. If the execution must be restarted, then the work done is lost. In this case it is said that coordination execution is not persistent.

2. If the execution state of the coordination (*Get payment information* state and if possible *Bank authorization* state) is recovered, then the coordination execution state survives. In this case it is said that coordination execution is persistent.

Persistency has been provided by means of logging techniques and using replication of processes. Due to autonomy characteristic of services, we focus on prior research work that uses logging techniques. In such a kind of approaches, the problems are related to i) how to handle in an efficient way the logging processes avoiding to overhead the normal execution, and ii) how to do an efficient recovery process after a failure.

In the next sections we present how persistency has been addressed by existing approaches based on the way it is provided to the services coordination:

- *Infrastructure based* persistency is a common strategy used by existing approaches (e.g., [AFH+99, LASS00, NFG+05, Con07a]). The management of the execution state is done by the execution infrastructure in a transparent way for the application developers which is not conscious of the problems related to provide persistency. Therefore, it can be said that execution state is a part of the facilities provided by the coordination engine.

- *Coordination based* persistency is an strategy where the application developers can use some mechanisms to define the persistency requirements of a coordination (e.g., [Fur04, BBC+05, Lom05, ZHMS06]). Therefore, it can be said that execution state concerns not only to the coordination engine but to the coordination.

2.3.1 Infrastructure based

Persistency has been addressed by most of existing approaches in a coarse manner. This means that persistency is provided explicitly by the underlying executing infrastructure. Therefore, it considers that all the services have the same persistency guarantees.

- [SABS02] uses the WISE (i.e., Workflow based Internet SErvices) platform [AFH+99, LASS00] to ensure persistency of coordination at execution time. WISE uses a database to make the state of each active process persistent in an automatic way so as to recover after system failures. For each active process there is a persistent copy of its execution state which guarantees that execution can be resumed after a failure. WISE platform has four modules: process definition, process enactment, monitoring and analysis, and coordination and communication. In particular, the module monitoring and analysis stores information about the entire execution history which is used to on-line and off-line analysis.

- [NFG+05] enacts coordinations where a persistent storage to made persistent data that is considered as critical for the execution (i.e., the execution state of the coordination) can be used. Because of the execution infrastructure is developed based on .NET technology, it is possible to use ADO.NET classes to connect, retrieve, and update any persistent data from repositories (e.g., SQL Databases). This approach captures the flow of the coordination by means of rules guard-activity-triggers. Therefore, the execution history contains information about the rules that had been executed and the date exchanged among them.

- [BGP05] uses Bonita engine [Con07a] for providing persistency to services coordinations. Each coordination is related to a project which includes all the information related to a given coordination (i.e., name and execution state). In fact, the executor of Bonita maintains information about all the executing projects into a local database. This information can be used for recovering a given execution point after a system failure.

2.3.2 Coordination based

Whereas the infrastructure based approaches focus on providing persistency in a general way, mainly supported by the underlying executing infrastructure, there are some approaches that provide persistency according to the requirements of each coordination that we called coordination based approaches.

- BTP [Fur04] addresses persistency by referring to recovery and failure handling in its definition. The objective is to ensure the delivery of a consistent decision for a transaction to the parties involved in such a transaction, even in the event of failures. However, the state persistency only concerns to participants of the

transaction. In case of a failure, it is ensured that the interaction among the participants of a transaction can continue to complete the transaction. The implementation of the persistent transactional protocol must be done by each particular implementation of the BTP protocol.

- WS family addresses communication failures by the WS-ReliableMessaging protocol [BBC+05] which defines a protocol for delivering messages in the presence of system failures. In fact, this protocol allows messages to be delivered between participants of a coordination in case of failures. Therefore, it can be assumed that using such information it is possible to built the execution state of parts of a coordination.

- [Lom05] proposes an approach for masking system failures through recovery guarantees which make persistent the execution state. In order to make persistent the interaction between components of an application they i) classify the components according to its persistency properties as persistent, transactional, and non persistent, and ii) use interactions contracts for defining the joint behavior of two interacting components. Based on this information they show how, using logging techniques, an execution infrastructure can provided data, messages, and state recovery.

- [ZHMS06] proposes an approach for addressing persistency in services based applications. They classify the participants of an application according to its persistency capabilities as database, replication, or memory based. Using such a classification, they proposes how it is possible to build and customize the persistent support of an application. This approach considers that state persistency is a service state attribute enabling to define persistency guarantees and requirements in a fine manner.

2.3.3 Discussion

Although approaches providing state persistency can be classified in terms of the overhead that causes its implementation (i.e., the use of software and hardware resources), we focus on the way it is related to resources (i.e., at service or coordination level):

- The first kind of approaches we analyze are those that consider that all services of a coordination have the same recovery properties and requirements, and therefore state persistency can be supported by the underlying executing infrastructure

(e.g., [AFH+99, LASS00, NFG+05, Con07a]). This kind of approach can be considered as a high level approach for the developer, but it does not capture the semantics associated to services based applications. This situation can leads to a waste of resources that compromises the applications performance. For example, in the "purchase tickets application", the activities *validate payment* and *send publicity* have different properties and requirements of persistency. While the monetary activity must be statefull, the other can be stateless. To this respect we think that persistency can be provided according to the capabilities of each coordinated service and the requirements of each application.

- A second way of addressing state persistency is to define them at coordination level (e.g., [Fur04, BBC+05, Lom05, ZHMS06]). Some approaches proposes the use of a protocol that ensures state persistency to parts of the coordination. However, using a protocol usually implies an ad-hoc solution linked to a implementation platform. Other approaches, propose mechanism to define the state persistency capabilities of services. In this way, semantics of the application is captured and overload is avoided.

Finally, although we agree with addressing persistency with adequate abstraction level for application developers (e.g., by the underlying execution infrastructure) we think that services coordination requires adaptable ways for providing persistency. State persistency must capture the heterogeneity of services and does not cause overhead at execution time.

2.4 Conclusion

Along this chapter we present how reliability has been provided to services coordination by means of atomic behavior and state persistency. First, we present approaches that provides atomicity according to the type of atomicity they provide, and to the way on which atomicity is integrated with the services coordination. Next we present several approaches that tackle persistency based on the way it is provided to the services coordination. From our point of view, we think that existing approaches do not address reliability in an optimal way:

- In the case of atomicity, using an approach implies to learn a coordination language and atomicity constructors or protocols that mixes the desired atomic behavior with the coordination. This situation conducts to built ad-hoc application not adaptable to existing dynamic environments.

- In the case of persistency, it is provided in a coarse manner and implicitly to the execution infrastructure of the services coordination which is ill suited to heterogeneity of services coordination.

We believe that it is necessary to provide an approach to define reliability and coordination in a separated way and considering the heterogeneous nature of services and the necessity of being adaptable to changes. To this respect, the following Chapters present our approach for defining reliability with contracts.

Chapter 3

The COBA model

This Chapter presents the COBA model (i.e., COntracts BAsed model) for representing the non functional aspects of a services coordination through the notion of contract. Such aspects specify the properties that the execution of a services coordination must ensure with respect to observable requirements like, the execution state (e.g., persistency), the conditions in which services calls are done (e.g., security), the way that exceptions are handled (e.g., atomic behavior), etc. In particular we are interested in reliability aspects of services coordination.

The main concepts of COBA are: execution unit, property, rule and contract. An execution unit represents the execution of a process (e.g., an activity of a services coordination) as a set of execution states through which it goes from the beginning of its execution to the end of its execution. A property represents a non functional aspect as a set of variables and its associated values as a constraint of the variables of the process. A rule specifies the reactions to be executed for enforcing a property under a given situation. A contract represents the association of a property to an execution unit (i.e., simple contract) or to a set of contracts (i.e., composite contract) and the rules (i.e., recovery operations) to be considered for the property. Besides, we assume that a services coordination is represented as a set of execution units and a control flow. A Control flow is represented by a set of execution dependencies among the execution units and an execution history. An execution history represents ordering constraints over the state of the set of execution units.

The chapter is organized as follows. The notation we used along this Chapter is described in Section 3.1. The rest of the Chapter is organized as follows. Sections 3.2 to 3.5 define the main concepts of the COBA model: execution unit, non functional property, rule and contract (i.e., simple and composite contract). Finally Section 3.6

concludes this Chapter.

Résumé: Ce chapitre décrit le modèle COBA (COntracts BAsed model) qui modélise les aspects non-fonctionnels d'une coordination de services à travers de la notion de contrat. Un contrat spécifie les propriétés que l'exécution d'une coordination des services doit assurer. Les propriétés sont observables dans l'état d'exécution de la coordination (e.g. la persistance), elles concernent les conditions dans lesquelles les appels aux services sont effectués (e.g. la sécurité), la faon dont les exceptions sont gérées et tolérées (e.g. le comportement atomique). Les concepts principaux de COBA sont: l'unité d'exécution, la propriété, la règle et le contrat. Une unité d'exécution représente l'exécution d'un processus (e.g. une activité d'une coordination de services) comme un ensemble d'états d'exécution. Une propriété représente un aspect non fonctionnel comme des "contraintes de type" associées aux variables d'exécution. Un contrat représente l'association entre une propriété et une unité d'exécution ou à un ensemble de contrats et les règles définissent les opérations de reprise à exécuter pour valider une propriété.

3.1 Preliminaries

This Section describes the notation used for defining the basic concepts of the COBA model.

3.1.1 Domain

A domain $\mathcal{D}$ is a set of values which also includes the "null" value.

- $\emptyset \in \mathcal{D}$, non-information value.
- $? \in \mathcal{D}$, unknown value.

$\mathcal{D}$:- v denotes the value v of $\mathcal{D}$.

Atomic domain

An atomic domain consists of indivisible values: `Boolean`, `Char`, `String`, `Integer`, `Float`, `Time`, `Date`, type `identifier`, and `void`. A type `identifier` is an unique value that identifies an instance.

For example, `String` :- "myname" is a value of the atomic domain `String`.

Union of domains

φ denotes the union of all domains. It includes the "null" value, denoted as δ, which is an information that does not exist and does not belong to any domain.

3.1.2 Type $\bigtriangledown$

The domain $\bigtriangledown$ of types is defined by the following rules:

- An atomic domain $\mathcal{AD}$ is a type, $\mathcal{AD}$:- $\mathcal{AD} \in \bigtriangledown$.
- A type denoted by $\mathcal{T} \in \texttt{String}$ is built in a recursive way as follows:
  $\mathcal{T}$:- $\texttt{tuple}(a_1\ T_1,\ a_2\ T_2,\ ..., a_n\ T_n) \in \bigtriangledown\ \forall\ i, j \in [1, ..., n],\ i \neq j,\ a_i \neq a_j$,
  $\mathcal{T}$:- $\texttt{list}(T_1) \in \bigtriangledown$,
  $\mathcal{T}$:- $\texttt{set}(T_1) \in \bigtriangledown$,
  where:
  - $a_1,\ a_2,\ ..., a_n\ \in \texttt{String}$ are identifiers and $T_1,\ T_2,\ ...,\ T_n \in \bigtriangledown$ are types.
  - If $\mathcal{T}$ is a tuple, each element of the type $\mathcal{T}$ can be referred using its identifier as follows $\mathcal{T}.a_i$, where $a_i \in \texttt{String}\ \forall\ i \in [1, ..., n]$.

Name and dom functions

The functions $\texttt{name} : \bigtriangledown \rightarrow \texttt{String}$ and $\texttt{dom} : \bigtriangledown \rightarrow \mathcal{D}$ enable to respectively access the name and the domain of a given type $\mathcal{T} \in \bigtriangledown$:

- If $\mathcal{T}$ is a type of the form $\mathcal{AD}$:- $\mathcal{AD} \in \bigtriangledown$, then:
  - $\texttt{name}(\mathcal{AD}$:- $\mathcal{AD}) = \mathcal{AD}$.
  - $\texttt{dom}(\mathcal{AD}$:- $\mathcal{AD}) = \mathcal{AD}$.

  $\mathcal{AD}$ denotes the domain ($\mathcal{AD} \subset \mathcal{D}$) and the type ($\mathcal{AD} \in \bigtriangledown$).
- If $\mathcal{T}$ is a tuple type as $\mathcal{T}$:- $\texttt{tuple}(a_1\ \mathcal{T}_1,\ a_2\ \mathcal{T}_2,\ ..., a_n\ \mathcal{T}_n) \in \bigtriangledown$, then:
  - $\texttt{name}(\mathcal{T}$:- $\texttt{tuple}(a_1\ \mathcal{T}_1,\ a_2\ \mathcal{T}_2,\ ..., a_n\ \mathcal{T}_n)) = \mathcal{T}$.
  - $\texttt{dom}(\mathcal{T}) = \{\mathcal{T}$:-$\texttt{tuple}(a_1\ v_1,\ a_2\ v_2,\ ...,\ a_n\ v_n) \mid n \geq 1, \forall i \in [1, ..., n], v_i \in \texttt{dom}(\mathcal{T}_i)\}$ is a set of tuples where each value
    $\mathcal{T}$:-$\texttt{tuple}(a_1\ v_1,\ a_2\ v_2,\ ...,\ a_n\ v_n)$ is an aggregation of values $v_1,\ v_2,\ ...,\ v_n$ each of them of type $\mathcal{T}_1, \mathcal{T}_2,\ ..., \mathcal{T}_n$ respectively.

To simplify we denote $\texttt{dom}(\mathcal{T})$= $\{\texttt{tuple}(a_1\ v_1,\ a_2\ v_2,\ ...,\ a_n\ v_n) \mid n \geq 1, \forall i \in [1,...,n], v_i \in \texttt{dom}(\mathcal{T}_i)\}$.

- If $\mathcal{T}$ is a list type as $\mathcal{T}$:- $\texttt{list}(\mathcal{T}_1) \in \bigtriangledown$, then:
  - $\texttt{name}(\mathcal{T}$:- $\texttt{list}(\mathcal{T}_1))= \mathcal{T}$.
  - $\texttt{dom}(\mathcal{T})$= $\{\mathcal{T}$:-$\texttt{list}(v_1,\ v_2,\ ...,\ v_n) \mid n \geq 1, \forall i \in [1,...,n], v_i \in \texttt{dom}(\mathcal{T}_1)\}$ is a set of lists where each value $\mathcal{T}$:-$\texttt{list}(v_1,\ v_2,\ ...,\ v_n)$ is a list of values $v_1,\ v_2,\ ...,\ v_n$ of type $\mathcal{T}_1$.

  To simplify we denote $\texttt{dom}(\mathcal{T})$= $\{\texttt{list}(v_1,\ v_2,\ ...,\ v_n) \mid n \geq 1, \forall i \in [1,...,n], v_i \in \texttt{dom}(\mathcal{T}_1)\}$
- If $\mathcal{T}$ is a set type as $\mathcal{T}$:- $\texttt{set}(\mathcal{T}_1) \in \bigtriangledown$, then:
  - $\texttt{name}(\mathcal{T}$:- $\texttt{set}(\mathcal{T}_1))= \mathcal{T}$.
  - $\texttt{dom}(\mathcal{T})$= $\{\mathcal{T}$:-$\texttt{set}(v_1,\ v_2,\ ...,\ v_n) \mid n \geq 1, \forall i \in [1,...,n], v_i \in \texttt{dom}(\mathcal{T}_1)\}$ is a domain of sets where each value $\mathcal{T}$:-$\texttt{set}(v_1,\ v_2,\ ...,\ v_n)$ is a set of different values $v_1,\ v_2,\ ...,\ v_n$ of type $\mathcal{T}_1$.

  To simplify we denote $\texttt{dom}(\mathcal{T})$= $\{\texttt{set}(v_1,\ v_2,\ ...,\ v_n) \mid n \geq 1, \forall i \in [1,...,n], v_i \in \texttt{dom}(\mathcal{T}_1)\}$.

Type instance creation

Every type $\mathcal{T}$ is a constructor of values.

$\mathcal{T}$ () creates a value of dom($\mathcal{T}$) initialized with default value. To overwrite the default value and initialize the created instance, it is possible to give values as parameters of the constructor.

For example, `String`() creates the value : `String` :- “”. `String`(“Alberto”) creates the instance `String` :- “Alberto”.

In a similar way, considering the following tuple:

city :- `tuple`(
name `String`,
country `String`
)

city(“DF”,“Mexico”) creates the instance:

city :- `tuple`(
name "DF",
country "Mexico"
)

Subtype

The type-subtype relationship is a *generalization-specialization* relationship. Where the supertype is the more general type, and the subtype is the more specialized type. We define the notion of subtype as follows:

- $\mathcal{T}'$:- `tuple`(a_1 $\mathcal{T}_1$, a_2 $\mathcal{T}_2$, ..., $a_n + k$ $\mathcal{T}_n + \mathcal{K}$) is a *subtype* of $\mathcal{T}$:- `tuple`(a_1 $\mathcal{T}_1$, a_2 $\mathcal{T}_2$, ..., a_n $\mathcal{T}_n$). The subtype $\mathcal{T}'$ has more attributes that the original tuple type but it has all the original attributes of $\mathcal{T}$ that can be redefined too.

- If $\mathcal{T}_1$ is a *subtype* of $\mathcal{T}_2$ then $\mathcal{T}'$: - `set`($\mathcal{T}_1$) is a *subtype* of $\mathcal{T}$: -`set`($\mathcal{T}_2$).

- If $\mathcal{T}_1$ is a *subtype* of T_2 then $\mathcal{T}'$: - `list`($\mathcal{T}_1$) is a *subtype* of $\mathcal{T}$: -`list`($\mathcal{T}_2$).

3.1.3 Meta-type

Variable The type *Variable* $\in \bigtriangledown$ is defined by:

Variable:-`tuple`(
name `String`,
type $\mathcal{T}$
)
where:

- *name* is a variable name.

- *type* represents the type of the variable ($\mathcal{T} \in \bigtriangledown$)

For example, the variable *Operator* of type string is defined, creating an instance of *Variable* as follows: *Variable* ("Operator", String).

It produces the instance

Variable:-`tuple`(
name Operator,
type `String`
)

This describes the *Operator* variable type that can have a string value. Actually *Variable* is a meta-type and operator can be used as a type to create instances of Operator. For example, *Variable* ("Operator", "XOR") or more simply *Operator* ("XOR"), assigns the "XOR" value to Operator.

Parameter The (meta-)type *Parameter* $\in \bigtriangledown$, which characterizes the input/output parameters of a method/operation, is defined by:

Parameter:-`tuple`(
$paramName$ `String`,
$paramType$ $\mathcal{T}$
)

where:

- $paramName$ is an identifier.
- $paramType$ represents the type $\mathcal{T} \in \bigtriangledown$ of the value of the parameter.

For example, *Parameter*("Age", Integer) defines the parameter *Age* of type integer:

Parameter:-`tuple`(
$paramName$ *Age*,
$paramType$ `Integer`
)

Such a definition will be used when using the Age parameter in an operation : *Parameter* {*Age* 12}.

Operation The (meta-)type operation is defined as:

Operation:-`tuple`(
$operationName$ `String`,
input `list`(`tuple`($paramName$ `String`, $paramType$ T_i)),
output T)

where:

- $T_i,\ T \in \bigtriangledown, \forall i \in [1...n]$.
- *operationName* is a unique identifier for the operation.
- *input* represents the input parameters of the operation.
- *output* of type $\mathcal{T}$, is the type of the output parameter.

In order to simplify we use the following notation:

$(\mathcal{T})\ \ operationName(paramName_1\ T_1, ..., paramName_n\ T_n)$

where:

- $paramName_i$ are the input parameters of type $T_i \in \bigtriangledown, \forall i \in [1...n]$.
- The output parameter is of type $T \in \bigtriangledown$.

We assume the existence of predefined operations:

- Operations of comparison defined over the domains of base and the domains of reference: $<, >, =, \neq, \leq, \geq$.
- Set operations defined over the domains tuples, list and set: $\in, \ni, \subset, \supset, \subseteq, \cup, \cap$.

3.1.4 Class

A class is a type that characterizes objects of the same type at meta level. The objects of a class have the same type and the same operations. The meta class *class* $\in \bigtriangledown$ is defined by:

class:- `tuple`(
    className `String`,
    attrSet `set`(`tuple`(*attrName* `String`, *attrType* $\mathcal{T}_i$)),
    operSet `set`(*Operation*)
)

where:

- $\mathcal{T}_i \in \bigtriangledown, \forall i \in [1...n]$

- *className* is the identifier of the class.
- *attrSet* is the set of attributes that characterizes the class. Each attribute of the class has a name and a type.
- *operSet* is the set of operations over the class.

In order to simplify we use the following notation:

class *className* {
$attrName_1$ $\mathcal{T}_1$,
...,
$attrName_n$ $\mathcal{T}_n$,
($\mathcal{T}_{n+1}$) $operationName_{n+1}(...)$,
...,
($\mathcal{T}_{n+m}$) $operationName_{n+m}(...)$,
}

where: $n \geq 1, m \geq 0, \mathcal{T}_i \in \bigtriangledown \ \forall i \in [1, ..., n+m]$.

For example, the class *Person* is defined as follows:

class *Person* {
name `String`,
birthDate `Date`,
`Integer` *age*(),
}

The class *Person* have two attributes (i.e., *name* and *birthDate*) and one operation (*age*).

Type and class

A class is a type. An instance of *class* defines a (tuple) type $className \in \bigtriangledown$ with the attributes set of the class. `dom`($className$)={ an instance of the class} $\subseteq \varphi$.

Creation and Initialization of an object

The new operation is used to create instance of a class with a unique identifier. For example, *new* *Person*() creates an instance of the class *Person* with identifier $idP1$:

($idP1$ $person\{$
$name$ “ ”,
$birthDate$ ’ ’
$\}$)

The attributes of an object can be initialized explicitly. For example:
$new\ Person(\text{“}Andre\text{”},'01\text{-}08\text{-}98')$
creates and initializes an instance of the class $Person$ with identifier $idP2$, the attribute $name$ with value “Andre”, and the attribute $birthDate$ with value ’01-08-98’.

($idP2$ $person\{$
$name$ “Andre”,
$birthDate$ ’01-08-98’
$\}$)

Object manipulation

Instances of classes are manipulated using methods or specific operations.

- $delete(Id)$: deletes the instance Id.
- $id.operationName(...)$: manipulates the instance id using the operation $operationName$.
- Id.getValue(attributeName) or more simply $Id.attributeName$: gets the value of the attribute $attributeName$ of the instance Id.
- $Id.setValue(attributeName, v\mathcal{T})$: sets a value of type $\mathcal{T}$ to the attribute $attributeName$ of the object Id
- An operation of an instance $idIstance$ is accessible using following notation: $Id.operName$.

Object naming

An instance Id can be named with $name$ using the bind operation: $bind(Id, name)$ Such a $name$ is a persistency root. It is used to access the associated instance and the underlaying graph of objects. Without a name, an instance cannot persist. For example:

bind(new Person ("Andre", '01-08-98'), "*Dd*")

associates the name "Dd" to the new instance of Person. Therefore, $Dd.birthDate$ returns '01-08-98'.

Inheritance

Inheritance relationship enables the specialization of classes by means of the subtype notion. Following notation indicates the specialization of the class *className* by the class *classChild*:

class classChild : *className* {
$attrName_1\ \mathcal{T}_1$,
...,
$attrName_n\ \mathcal{T}_n$,
$(\mathcal{T}_{n+1})\ operationName_{n+1}(...)$,
...,
$(\mathcal{T}_{n+m})\ operationName_{n+m}(...)$,
}

where:

- $n \geq 1, m \geq 0, \mathcal{T}_i \in \bigtriangledown\ \forall i \in [1, ..., n+m]$.
- *classChild* is a subtype of *className*.
- The child class has the attributes of the parent class *className* and the new defined attributes:
  - The attributes $attrName_1\ \mathcal{T}_1$ through $attrName_n\ \mathcal{T}_n$ are added to the original attributes of the class *className*.
  - If an attribute $attrName_i$ appears in the parent class $className$ and in the child class $childName$, then its type is redefined, but the type of $attrName_i$ in $childName$ has to be a subtype of $attrName_i$ in $className$.
- The child class has the operations of the class *className* and the new defined operations:
  - The operations $operationName_{n+1}(...)$ through $operationName_{n+m}(...)$ are add to the original operations of the parent class *className*.

- An operation defined within the parent class *className* can be applied to instances of *classChild*.
- If an operation of the parent class *className* has the same name of one of the added operations then it is redefined but it only applies to instances of the *classChild*.

We assume that a class can inherit of only one parent class.

For example, given the class *person*, we can define the subclass *employee* as follows:

class *employee* : *person* {
    position `String`,
}

Note that *employee* has the attributes *name* and *birthDate* inherited of *person*. Besides, the operation *age*() can be applied to instances of *employee*. Therefore, it can be said that employee is an special case of person.

3.2 Execution unit

An execution unit is described by its input data and output data (i.e., parameters), an execution state, and the software entity that will execute the process. For example, in the "purchase tickets application" there are six execution units: *Get concert information*, *Process purchase*, *Validate payment*, *Send tickets*, *Send publicity*, and *Validate purchase* (see Figure 1.1).

Definition 1. ***(Class Execution unit)***

class *EU* {
    provider `String`,
    input `set`(*Parameter*),
    output `set`(*Parameter*),
    states `list`(*State*)
}

where:

- *provider* represents the software entity that exports the functionality of the process through an API.

- *input/output*: represents a set of parameters. A parameter is an instance of the type *Parameter*. It is a tuple giving the name of the parameter and its type (see its definition in Section 3.1.3).

- *state* is an ordered list of states. A state within this list represents a given instant of the execution that it is interesting for ensuring a non functional aspect.

In the "purchase tickets application" all the execution units can be defined as instances of the class $\mathcal{EU}$. For example, for defining *Get concert information* as an execution unit, first is necessary to specialize the class $\mathcal{EU}$ to define its execution requirements:

SubClass (EU gCI)

class $EUgCI$: $\mathcal{EU}$ {
input {{*name* `String`}, {ccNumber `String`}},
output {{*result* `Boolean`}},
}

where:

- *name* is an input parameter representing a username.

- *ccNumber* is an input parameter representing a credit card number.

- *result* is an output value representing the execution result.

Next an instance of class $EUgCI$ with name gCI is defined as follows:

gCI $EUgCI$ {
provider "myProviderAddress",
input {{*name* "Brenda"}, {ccNumber "1707120601080512"}},
output {{*result* $False$}},
state {$idPS$}
}

Where, the provider of the execution unit gCI is the provider of name "myProvider-Address". gCI has two input parameters (i.e., an username and a credit card number), one output parameter (i.e., the result of executing the execution unit), and one state instance of name $idPS$.

3.2.1 State

A state is described by a set of values of the variables associated to an execution unit in a given instant of the execution. For example, it can represent the end of the execution of an execution unit (i.e., when all resources related to the execution of an execution unit are available to be used by another process).

Definition 2. ***(Class State)***. *The class state type of an execution unit is defined as follows:*

class State{
timeStamp `Time`,
values `set`(*Variable*)
}

where:

- *timeStamp* represents the point in time at which the state was reached.
- *values* represents the variables of the execution unit with its associated values at the time *timeStamp*.

3.2.2 Control flow

A control flow represents the execution history of a set of execution units and its execution dependencies. It is represented by a set of execution units, an execution order defined over the state of the execution units, and a set of execution dependencies among the execution units. In our example, the control flow is represented by means of arrows (see Figure 1.1).

Definition 3. ***(Class Control flow)***

$class\ \mathcal{CF}\ \{$
eus `set`$(\mathcal{EU})$,
$execOrder$ `list`(`tuple`$(eu_1 : \mathcal{EU}, eu_2 : \mathcal{EU})$),
$ctrlLogic$ `list`(`tuple`$(eu_1 : \mathcal{EU}, op :$ `Operators`$, eu_2 : \mathcal{EU})$)
$\}$

where:

- *eus* represents a set of execution units.

- Any tuple $\langle eu_i,\ eu_j \rangle$ of the list *execOrder* follows the rules:
  - $eu_i \in eus$,
  - $eu_j \in eus$,
  - $eu_i \neq eu_j$,
  - $lastEuState(eu_i.states).timestamp < firstEuState(eu_j.states).timestamp$

  This means that the last state of eu_i was reached before that the first state of eu_j. $firstEuState$ returns the first reached state of a set of execution states, evaluated over the attribute *timeStamp* of each state. In a similar way, $lastEuState$ returns the last reached state of a set of execution states.

- *ctrlLogic* is a list of tuples, each of them of the form $\langle eu_i,\ op_k,\ eu_j \rangle$ that represents the execution dependencies among pairs of execution units by means of ordering operators such as sequential routing, parallel routing and selective routing. The meaning of any tuple $\langle eu_i,\ op_k,\ eu_j \rangle$ of the *ctrlLogic*, where $eu_i \neq eu_j$, is: "eu_i has the order operator op_k with respect to eu_j".

For example, the control flow of the "purchase tickets application" is defined by the object pTA as follows:

$pTA\ \mathcal{CF}\{$
$eus\ \{gCI,\ pP,\ vPa,\ sT,\ sP,\ vPu\},$
$execOrder\ \{\},$
$ctrlLogic\{\{gCI,\ seq,\ pP\},\ \{pP,\ andSp,\ vPa\},\ \{pP,\ andSp,\ sT\},$
$\{pP,\ andSp,\ sP\},\ \{vPa,\ andJn,\ vPa\},\ \{sT,\ andJn,\ vPa\},$
$\{sP,\ andSp,\ vPa\}\}$
$\}$

We assume that gCI (related to *Get concert information*), pP (related to *Process purchase*), vPa (related to *Validate payment*), sT (related to *Send tickets*), sP (related to *Send publicity*), and vPu (related to *Validate purchase*) are instances of the class $\mathcal{EU}$, and that seq, $andSp$, and $andJn$ are operators. Note also that, *execOrder* is set to null value.

3.3 Property

A property is represented by a set of variables, where the valid combinations of values that the variables can have along the execution are used for constraining the execution.

For example, in the "purchase tickets application" the execution unit *Get concert information* can be retried in case of failure, because it can be completed in several ways.

Definition 4. ***(Class Property)***

class Property {
    values `set`(*Variable*)
}

where *values* is a set of *Variable* (see Section 3.1.3) values that must be verified over the execution state.

For example, in the "purchase tickets application" a property can specify whether an execution unit requires to be re-executed if it fails. An execution unit (e.g., *Get concert information*) having vital information for the whole application must be retried in case of failure, while an execution unit (e.g., *Send publicity*) used for transmitting publicity might not require to be retried in case of failure. The retry property is defined as a subclass of *PropertyT* as follows:

SubClass (Retry property)

class retPType : *Property* {
    values {{type {"yes", "no"}}}
}

Next, an instance of such a class can be defined as follows, where *rtP* is the name of the object:

rtP retPType {
    values {{type "yes"}}
}

3.4 Rule

A rule specifies the reactions to be executed for enforcing a property under a given situation. For example, the re-execution of the *Get concert information* execution unit in case of failure in the "purchase tickets application".

Definition 5. ***(Class Rule)***

class Rule {
on Event,
do Reaction
}

where:

- *on* represents the type of *event* that triggers the reaction of the rule.
- *do* represents the *Reaction* that must be executed.

For example, the following instance named $recRule1$ of the class *Rule* defines the re-execution of an execution unit in case of failure as follows:

$recRule1$ *Rule* {
on $ifEv$,
do iRt
}

The definition of the instances $ifEv$ and iRt are presented in the next subsections.

3.4.1 Event

An event represents a significant situation occurring at a point in time during the execution of an coordination. For example, the execution failure of an execution unit.

Definition 6. ***(Class Event)***

class Event {
timeStamp `Time`,
delta `set`*(Variable)*
}

where:

- *timeStamp* is the instant at which the event was produced.
- *delta* is a set of *Variable* (see Section 3.1.3) instances that represents the conditions under which the event was produced.

For example, an event representing the execution failure of an execution unit specializes the class *event* as follows:

SubClass (Failure event)

class $failEv$: *Event* {
$delta$ {{$euName$ `String`}}
}

where $euName$ is a string representing the name of an instance of the class $\mathcal{EU}$.

Next, an instance named $ifEv$ of the class $failEv$ is defined as follows:

$ifEv$ $failEv$ {
$timeStamp$ '2010-01-01 00:00:01',
$delta$ {{$euName$ "gCI"}}
}

3.4.2 Reaction

A reaction represents an action to be taken in order to enforce a property. For example, the re-execution of a execution unit.

Definition 7. ***(Class Reaction)***

class *Reaction* {
input `set`(*Parameter*),
output `set`(*Parameter*)
}

where:

- *input* represents the set of input parameters of the reaction (see Section 3.1.3).
- *output* represents the set of output parameters of the reaction (see Section 3.1.3).

For example, the class *Reaction* can be specialized by a subclass to indicate that the execution of a given execution unit must be retried:

SubClass (Retry reaction)

class $retry$: *Reaction* {
$input$ {{eu $\mathcal{EU}$}},
$output$ {{r `Boolean`}}
}

where:

- *eu* represents the identifier of an instance of the class $\mathcal{EU}$.
- *r* represents the execution result of the reaction.

Next, an instance of the class *retry* is defined as follows:

iRt *Reaction* {
input {{*euName gCI*}},
output {{*rResult False*}}
}

where *gCI* represents the identifier (interne) of an $\mathcal{EU}$.

3.5 Contract

A contract represents the relationship between a property and a scope. It specifies how to enforce the property once it is not verified within a given execution state of the scope. Enforcing is expressed by means of event action (E-A) rules. The scope of a contract can be an execution unit or a set of contracts. Indeed, a contract can be associated to several rules in order to specify the conditions under which the property must be evaluated and the reactions to be executed if the property is not "respected". For example, in the "purchase tickets application" a contract can be defined for ensuring the business rule that states that the payment can be granted in different ways assuming that the scope of the contract is the execution unit *validate payment*.

Definition 8. ***(Class Contract)***

class *Contract* {
property *Property*,
priority `Integer`,
rules `set`(*Rule*)
}

where:

- *property* represents the valid combinations of values that the property variables can have along the execution.

- *priority* is an integer representing the order of evaluation of the contract with respect to other contract types associated to the same *scope*.
- *rules* represents the reactions to be taken once the property is not verified.

3.5.1 Simple contract

A simple contract is related to an execution unit. It specializes the class *Contract* as follows:

Definition 9. ***(Subclass Simple contract)***

class SimpleContract : *Contract*{
scope EU
}

where *scope* represents the execution unit associated with the contract.

For example, we can specialize the class *SimpleContract* for associating the class *retPType* (i.e., retry property) with a contract as follows:

SubClass (Retry contract)

class retryC : *SimpleContract* {
property retPType,
}

Next, we can define an instance named $idC2$ for associating the instance rtP of class *retPType* to the identifier gCI (i.e., *Get concert information*) of an instance of the class $\mathcal{EU}$ as follows:

$idC2$ *retryC* {
$scope\ gCI$,
$property\ rtP$,
$priority\ 0$,
$rules\ \{recRule1\}$
}

In a similar way, we can define an instance of name $idC3$ for associating the instance rtP with the identifier pP (i.e., *Process purchase*) as follows:

$idC3$ *retryC* {
scope pP,
property rtP,
priority 0,
rules {$recRule1$}
}

3.5.2 Composite contract

There are properties that must be evaluated on the states of a set of execution units. For example, in the "purchase tickets application", the business rule that states that, "a purchase order can be completed only if it was paid", implies to associate an execution constraint to execution units *Validate payment* and *Send tickets*. This situation implies that such kind of properties must be expressed by combining other properties. The notion of composite contract models the composition of properties and their associated reactions. This composition is done by associating a property to a set of contracts.

In a composite contract, the conditions under which its property must be evaluated is related to the type of contracts that are associated:

- If the contracts are associated with execution units, the composite contract can be verified over the execution state of the execution units.
- If the contracts are composite contracts, the composite contract is verified over the properties of the contracts.

A composite contract specializes the class *Contract* as follows:

Definition 10. ***(Class Composite contract)***

class CompositeContract : *Contract* {
scope `set`(*Contract*),
}

where *scope* represents a set of contracts.

For example, given two contracts instances with identifiers $idC2$ and $idC3$ respectively (see Section 3.5), the contract instance *CompositeContract*({idC2, idC3},...) defines a new contract if following composition rules are verified:

- $\{idC2, idC3\}$ is the scope of the composite contract.
- $idC2$ and $idC3$ cannot participate in another composite contract (i.e., with the same property within the contract).
- $\texttt{dom}(idC2.property) = \texttt{dom}(idC3.property)$, i.e., $idC2$ and $idC3$ have the same type of property.

Such rules can be generalized to n contracts within the *scope* of a composite contract. Indeed, a composite contract can be represented by a tree:

- A leaf represents an execution unit.
- An intermediate node represents a contract whose scope is a set of contracts.
- Edges represent composition relationships between contracts. Contracts or execution units within the scope of a given contract (the parent) are called its children.
- The order priority is defined by a bottom-up left-right policy:
  - Leaf nodes have higher priority than intermediate nodes.
  - Root node has the lowest priority.

  Such an order is used when contracts must be evaluated at execution time. For example, it is used for defining which contract is evaluated first.

For example let us consider the following composite contract instance $idC1$ that associates a the property instance $myProperty$ and the rule instance $recRule2$ to contract instances $idC2$ and $idC3$:

$idC1$ *CompositeContract*{
$scope$ $\{idC2, idC3\}$,
$property$ $myProperty$,
$priority$ 1,
$rules$ $\{recRule2\}$
}

The Figure 3.1 shows the contract tree of the contract instance idC1.

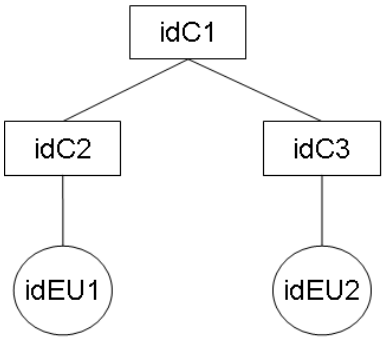

Figure 3.1: Contract tree example

3.6 Conclusion

This chapter presents COBA, a contract based model for adding reliability to services coordination. Contracts can be of two types: simple and composite contracts. A simple contract is used for associating reliability aspects to single execution units and its associated rules for reinforcing such properties. A composite contract is used for associating complex reliability properties to contracts enabling contract composition. In such a way, using COBA, reliability properties to be verified at execution time can be specified and associated to services coordinations without implying to modify the logic of the coordination. Indeed, a coordination can be associated with several contracts according to specific requirements (e.g., business rules, user requirements, execution context). Next Chapter presents how our model can be used for defining the classes required for reliability aspects of the services coordination.

Chapter 4

Reliability with contracts

This chapter shows how to use the COBA model introduced in the Chapter 3 for adding reliability properties to given services coordinations. A reliable services coordination is the one that tolerates failures at execution time. Therefore, the execution of the coordination is able to maintain its functionality in unexpected situations (i.e., failures). Two aspects must be considered:

- The definition of reliability properties related to individual execution units. Such properties are associated to the behavior of the execution units with respect to some attribute at execution time. For example, whether it is possible to know the execution state of an execution unit, whether it is possible to undone the actions of an execution unit once it has committed, etc.

- The definition of complex reliability properties related to execution units sets (i.e., composite contracts). For example, the atomic behavior that a set of execution units must have within an application.

Consequently, in this chapter we analyze key properties to address reliability: exception handling and state management. Next we show how they can be defined and related to a services coordination by means of contracts.

The chapter is organized as follows. Section 4.1 defines a reliable services coordination and shows how reliability properties can be provided by means of contracts. Consequently, Section 4.2 introduces the notion of exception contract. Section 4.3 provides the definition of atomicity contracts. Section 4.4 introduces the notion of state management contract. Section 4.5 provides the definition of persistency contracts. Finally Section 4.6 concludes this Chapter.

Résumé: Ce chapitre décrit comment utiliser le modèle COBA pour rendre fiable une coordination de services. Pour cela, le chapitre présente la définition de contrats pour associer des propriétés de fiabilité aux:

- Unités d'exécution d'une coordination de services et définir des stratégies de traitement d'exceptions.
- Ensembles des unités d'exécution et programmer des stratégies de tolérance d'exceptions comme l'atomicité.

Ce chapitre propose en particulier des contrats qui (i) permettent d'adresser le traitement d'exceptions et la gestion de l'état d'exécution d'une unité d'exécution ; et (ii) qui composent des contrats pour assurer les propriétés d'atomicité et de persistance d'un ensemble d'unités d'exécution d'une coordination.

4.1 Reliable services coordination

A reliable services coordination tolerates failures at execution time. Failures are of two kind: semantic and system failures. We address reliability to services coordination by providing exception handling (i.e., to handle semantic failures), and by providing state persistency (i.e., to handle system failures). The next sections explain how and why specifying reliability properties by contracts that are related to individual execution units (i.e., simple contracts) and to sets of execution units (i.e., composite contracts).

4.1.1 Exception handling

A way of addressing semantic failures is to provide exception handling through atomic behavior (i.e., atomicity). This strategy was first used in the database area, where the atomic behavior associates database operations with the principle of all or nothing at execution. The principle works because it is assumed that all operations commits or fails, and if committed they can be rolled back (undone). This notion is relaxed in the context of services coordination, where execution units are the operations that must be executed with an atomic behavior. Therefore, to address exception handling to services coordination we use the following approach:

1. We specify how to treat exceptions over the execution of execution units. For example, we define how to handle the failure of a vital execution unit for the

coordination (e.g., the execution unit *validate payment* of the "purchase tickets application" must be retried in case of failure).

2. We specify how to provide atomic behavior to sets of execution units. For example, we define how to handle the possible execution dependencies of several execution units (e.g., in the "purchase tickets application" the tickets cannot be sent without a payment).

4.1.2 State persistency

State persistency refers to make durable the execution state in case of a system failure. Tolerating this kind of failures ensures that given a coordination, its execution state survives in spite of system failures. In the context of a services coordination, this implies to make persistent the state of the coordination itself and the state of its execution units (i.e., its execution history). A key element to consider is where to store the execution history and how to do the recovery process using it. Therefore, we address state persistency as follows:

1. We specify how to treat the persistency guarantees of execution units, assuming that execution units offers several persistency guarantees according to how they are implemented by its providers. For example, the state of banks services is persistent because it usually can be queried.

2. We specify how to handle the execution state of activities sets. We assume that there are parts of the coordination where special attention must be taken. For example, in the "purchase tickets application" it is important to ensure that the state of execution units *Validate payment* and *Send tickets* survives system failures due to its importance for the application.

4.2 Exception contract

There are two situations that must be considered to handle exceptional situations over the execution of execution units, i) what do if the execution of an execution unit fails (i.e., the execution can continue or the execution unit must be retried), and ii) what to do if a committed execution unit must be compensated (i.e., whether is possible to undo the actions of its execution). Consequently, we consider following aspects:

- The fact that, some execution units may be compensated but some others not. A compensation action undoes semantically the actions of a committed execution unit, but it does not necessarily return the coordination to the previous state before the execution unit was committed. In our "purchase tickets application", *validate payment* cannot be compensated without an extra charge per cancelation. However, there are some activities that cannot be compensated, for example personalizing an item using laser engraving.

- The side effects that can be caused by compensating an execution unit. There are some execution units that can be compensated, but side effects must by taken into account. For example, undoing *sent tickets*, in the "purchase tickets application", implies a waste of material (envelopes, covers, etc.).

- The possibility of executing several times an execution unit (i.e., to retry its execution). In our example, *get concert information* can be executed several times if something is missing in data customer. However, it must be noted that retrying an execution unit can be constrained by other concerns, such as: time constraints, quality of service, or application semantics among others.

We address exceptional situations by means of the exception contract. It is a simple contract that associates exception handling property to an execution unit. The exception contract subclass specializes the class *SimpleContract* (see Definition 9) as follows:

SubClass (Exception contract)

class exceptionContract : *SimpleContract* {
property exceptionProperty,
}

4.2.1 Exception property

The exception property describes the possible values that must be considered to treat the execution of an execution unit in case of an exception. It specializes the class *Property* (see Definition 4) as follows:

SubClass (Exception property)

```
class exceptionProperty : Property {
                values { {"compensable", Boolean},
                         {"side-effects", Boolean},
                         {"retriable", Boolean},
                         {"max-retry-no", Integer}}
}
```

where the attribute *values* represents execution properties:

- "*compensable*" specifies whether an execution unit can be undone by compensating it with the execution of another execution unit or whether it cannot be compensated.
- "*side-effects*" specifies whether undoing an execution unit generates side-effects.
- "*retriable*" specifies whether an execution unit can be executed several times.
- "*max-retry-no*" specifies the maximum retry number.

According to the above values of the exception property, execution units can be associated with several combinations. The combinations are shown in Table 4.1 and we discuss about its validity in the next lines:

Case	Can be compensated	Causes side-effects	Can be retried
1	No	No	No
2	No	No	Yes
3	No	Yes	No
4	No	Yes	Yes
5	Yes	No	No
6	Yes	No	Yes
7	Yes	Yes	No
8	Yes	Yes	Yes

Table 4.1: Execution unit types according to exception property

- Case 1 characterizes a critical execution unit because in case of failure it cannot be compensated and it cannot be retried.

- Case 2 and 5 characterize a non vital execution unit because in case of failure the execution can continue. It is not necessary to compensate or retry the execution unit.
- Case 3 and 4 do not make sense because an execution unit which cannot be compensated cannot causes side effects.
- Case 6 characterizes an undoable execution unit which can be compensated without side effects and can be retried.
- Case 7 and 8 characterize a compensatable execution unit which can be compensated with side effects and can be retried a restricted number of times.

Therefore, we have identified four exception contract subtypes (See Table 4.2): critical contract, non vital contract, undoable contract, and compensatable contract.

Exception contract	Can be compensated	Causes side-effects	Can be retried
Critical	No	-	No
Non vital	-	No	-
Undoable	Yes	No	Yes
Compensatable	Yes	Yes	Maybe

Table 4.2: Exception contract subtypes according to exception property values

4.2.2 Recovery Rules

A recovery rule of an exception contract defines at which moment an action must be executed. A recovery rule specializes the class *Rule* (see Definition 5), for example following rules are related to the critical contract:

- $recRule1$ specifies that, if the execution of a given execution unit fails, then it is necessary to notify the failure:

$$\begin{aligned} &\textit{class}\ recRule1\text{: }\textit{Rule}\ \{ \\ &\qquad\qquad on\ failEv, \\ &\qquad\qquad do\ notifyFailure \\ &\qquad \} \end{aligned}$$

- *recRule*2 specifies that, if it is requested the compensation of a given execution unit, then an exception is launched:

$$\begin{array}{l} \textit{class } recRule2\text{: } \mathcal{Rule} \ \{ \\ \qquad\qquad\qquad\qquad\qquad\quad on\ compReqEv, \\ \qquad\qquad\qquad\qquad\qquad\quad do\ notifyExc \\ \qquad\qquad\qquad\qquad\quad \} \end{array}$$

The definitions of the class events $failEv$ and *compReqEv*, and the class reactions $notifyFailure$, and $notifyExc$ are presented in the next sections.

Events

Exception contract type reactions are triggered by two events:

- The execution failure of an execution unit. It specializes the class *Event* (see Definition 6):

$$\begin{array}{l} \textit{class } failEv\text{: } \mathcal{Event}\{ \\ \qquad\qquad\qquad\qquad\quad delta\ \{\{eu\ \mathcal{EU}\}\} \\ \qquad\qquad\qquad\quad \} \end{array}$$

- The necessity for compensating an execution unit. It specializes the class *Event* (see Definition 6):

$$\begin{array}{l} \textit{class } compReqEv\text{: } \mathcal{Event}\{ \\ \qquad\qquad\qquad\qquad\quad delta\ \{\{eu\ \mathcal{EU}\}\} \\ \qquad\qquad\qquad\quad \} \end{array}$$

Reactions

Reactions specify possible actions to take within the execution. A reaction specializes the class *Reaction* (see Definition 7), for example following reactions are related to critical contract:

- $notifyFailure$ indicates that the contract cannot be granted (i.e., it has failed):

class $notifyFailure$: *Reaction* {
input {{*eu* $\mathcal{EU}$}},
output {{$rResult$ `Boolean`}}
}

- $notifyExc$ indicates that an exception must be launched:

class $notifyExc$: *Reaction* {
input {{*eu* $\mathcal{EU}$}},
output {{$rResult$ `Boolean`}}
}

4.2.3 Critical contract

According to exception property there are four exception contracts that can be defined. For example, a critical contract can be associated to an execution unit that cannot be retried in case of failure and when it has committed the execution unit cannot be undone. A critical contract specializes the class *exceptionCcontract*:

class $crContract$: *exceptionContract*{
property {*values* {{*name* "compensable", *value* False},
{*name* "side-effects", *value* False},
{*name* "retriable", *value* False},
{*name* "max-retry-no", *value* 0}}},
rules {$r1$ $recRule1, r2$ $recRule2$}
}

For example, in the "purchase tickets application" the execution unit *validate payment* can be considered as an execution unit that cannot be retried in case of failure. This situation can be captured by means of a critical contract instance with name $crCvP$ as follows:

$crCvP$ $crContract$:{
scope vP,
priority 1
}

where:

- vP is the identifier of an instance of the class $\mathcal{EU}$ representing the execution unit *validate payment*.
- The priority of the contract is 1.

The complete definition of the exception contracts is presented in the Appendix A.

4.3 Atomicity contract

In the context of services coordination, atomicity is a relaxed notion of classic atomicity. For providing atomic behavior to services coordination, we assume that exception handling is provided to execution units (i.e., they commits, fails and possibly they can be compensated). Therefore, we compose atomicity behavior to execution units sets on top of exception handling. We cosider three atomic behavior types[Por06b]:

- The *strict atomicity* behavior conform the all or nothing execution requirement. For example, in the "purchase tickets application", the two execution units, *Validate payment* and *Send tickets*, or none of them are executed.
- The *alternative atomicity* behavior captures the possibility of using alternative execution paths for committing. For example, in the "purchase tickets application", the execution unit *Validate payment* can be completed by several providers.
- The *exception atomicity* behavior indicates that if something goes wrong, then an exception must be launched. For example, in the "purchase tickets application", if the execution unit *Get concert information* fails an exception must be launched because it obtains the purchase information.

We address atomic behavior by means of the atomicity contract. It is a composite contract that associates atomicity property to a set of contracts. The contracts within an atomicity contract must be of type exception or atomicity. The atomicity contract subclass specializes the class *CompositeContract* (see Definition 10) as follows:

SubClass (Atomicity contract)

class atomicityContract : *CompositeContract* {
scope `set`(*failureContract* ∪ *atomicityContract*),
property atomicityProperty,
}

4.3.1 Atomicity property

The atomicity property describes the type of atomicity associated to a set of contracts. It specializes the class class *Property* (see Definition 4) as follows:

SubClass (Atomicity property).

class atomicityProperty : *Property* {
values {{*name* "atomicityType", *value*
{"Strict", "Alternative", "Exception"}}}
}

where "atomicityType" specifies the type of atomic behavior: "*strict*", "*alternative*" or "*exception*" that must be ensured to the scope of the contract at execution time.

4.3.2 Atomicity rules

An atomicity rule defines at which moment it is necessary to take actions for ensuring a given atomic behavior. An atomicity rule specializes the class *Rule* (see Definition 5), for example the following rule is related to a strict atomicity contract:

- $acRule1$ specifies that, if a contract within the scope of a contract fails, then the contract fails and backward recovery is applied:

class $acRule1$: *Rule* {
on $contractFailure$,
do $backwardRecovery \wedge notifyContractFailure$
}

The definitions of the class event $contractFailure$ and the class reactions $backwardRecovery$ and $notifyContractFailure$ are presented in the next subsections.

Events

The reactions of the atomicity contracts are triggered by the fact that a contract is considered as failed. A contract has failed according to its type and the state of the execution units within its scope:

- An exception contract is considered as failed when its associated execution unit fails. For example, an execution unit associated with a non vital contract cannot be considered never as failed.

- An atomicity contract is considered as failed according to its atomicity property and the contracts within its scope:
  - A strict atomicity contract fails when one of the contracts within its scope fails. It must be noted that forward execution is implemented at failure contract level. Therefore, a failure at this level means that was not possible to continue the execution of one or more execution units.
  - An alternative atomicity contract fails when all its possible execution paths have failed.
  - An exception atomicity contract fails when one of the contracts within its scope fails.

The event signaling the failure of a contract is defined as follows:

class $contractFailure$: *Event* {
    delta {{*contract* *Contract*}}
}

Reactions

Reactions of atomicity contracts specializes the class *Reaction*. For example, following subclasses defines the possibly reactions of the strict atomicity contract:

- $notifyContractFailure$ launches an event notifying the failure of a contract:

  class $notifyContractFailure$: *Reaction* {
      input {{*contract* *Contract*}},
      output {{$rResult$ `Boolean`}}
  }

- *backwardRecovery* indicates that committed contracts within an scope are undone, until the whole contract is compensated:

  class *backwardRecovery*: *Reaction* {
      input {{*contract* *Contract*}},
      output {{$rResult$ `Boolean`}}
  }

4.3.3 Strict atomicity contract

According to the possible values of the property *atomicityProperty*, an *atomicityContract* can be of three types: strict atomicity contract, alternative atomicity contract or exception atomicity contract. For example, the strict atomicity contract specializes the class *atomicityContract*. It specifies that all contracts or no contract at all within the scope of the contract are executed:

class $stAtC$: *atomicityContract* {
$property$ {$values$ {$name$ "atomicityType", $value$ "Strict"}},
$rules$ {$r1$ $acRule1$}
}

For example, in the "purchase tickets application" the execution units *validate payment* and *send tickets* must be executed with strict atomicity behavior (i.e., an order must be paid to be sent). This situation can be captured by means of a strict atomicity contract instance with name $stC1$ as follows:

$stC1$ $stAtC$:{
$scope$ {$crCvP$, $cpCsT$},
$priority$ 2
}

where:

- The *scope* of the contract contains only exception contracts which means that failures of the execution units are handled by the rules defined within each exception contract type.
  - $crCvP$ is the identifier of an instance of the contract class $crContract$. It represents that the execution unit *validate payment* has a critical contract.
  - $cpCsT$ is the identifier of an instance of the contract class $cpContract$. It represents that the execution unit *send tickets* has an undoable contract.
- The priority of the contract is 2.

The complete definition of the atomicity contracts is presented in the Appendix A.

4.4 State management contract

There are three aspects that must be considered to manage the state of an execution unit:

- The possibility of querying the execution state of an execution unit. For example, in the "purchase tickets application", the execution unit *Validate Payment* can be queried because the banks usually stores information about its transactions.
- The idempotent capability of an execution unit. For example, in the "purchase tickets application" for a given customer, the execution unit *Get concert information* can be executed several times resulting in the same result.
- The outcome assumption that can be done for an execution unit once it has been started. For example, in the "purchase tickets application" it can be presumed that execution unit *Validate purchase* commits if a failure occurs once its execution has been started.

We address the management of execution state of executions units by means of the state management contract. It is a simple contract that associates state management property to an execution unit. The state management contract subclass specializes the class *SimpleContract* (see Definition 9) as follows:

SubClass (State management contract)

class stateMContract : *SimpleContract* {
    property stateMProperty
}

4.4.1 State management property

The state management property describes the possible values that an execution unit can have for managing the persistency of its state. It specializes the class *Property* (see Definition 4) as follows:

SubClass (Property)

class stateMProperty : *Property* {
    values { {*name* "state-Verifiability", *value* `Boolean`},
               {*name* "idempotency", *value* `Boolean`},

{*name* "outcome-Assumption", *value* {"committed",
"failed",
"presumed-nothing"}}}
}

where the attribute values represents following properties:

- "state-Verifiability" specifies whether the execution state of the execution unit can be queried.
- "idempotency" specifies whether either the execution unit can be executed several times resulting in the same result or the multiple execution of the execution unit results in different results for each execution.
- "outcome-Asumption" specifies what can be presumed if a failure occurs once the execution of the execution has been started:
  - "*committed*": when a system failure occurs the execution of the execution unit is presumed to c1ommit.
  - "*failed*": when a system failure occurs the execution of the execution unit is presumed to fail.
  - "*presumed-nothing*": when a system failure occurs the execution of the execution unit cannot be presumed to some result.

According to the above values of the state management property, execution units can be associated with several combinations. The combinations are shown in Table 4.3 and we discuss about its validity in the next paragraph:

- Case 1 characterizes a non persistent execution unit because in case of failure its execution state neither can be queried nor supposed and it is non idempotent.
- Case 2 characterizes a presumable execution unit because in case of failure its result has an outcome assumption.
- Case 3 and 4 characterize an idempotent execution unit because it can be re-executed in case of failure with the same result.
- Case 5 to 8 characterize a verifiable execution unit because its execution state can be known in case of failure.

Case	Can be queried	Is idempotent	Has an outcome assumption
1	No	No	No
2	No	No	Yes
3	No	Yes	No
4	No	Yes	Yes
5	Yes	No	No
6	Yes	No	Yes
7	Yes	Yes	No
8	Yes	Yes	Yes

Table 4.3: Execution unit types according to state management property

Therefore, we have identified four state management contract subtypes (see Table 4.4): non persistent state management contract, presumable state management contract, idempotent state management contract, and verifiable state management contract.

Persistency properties contract	State verifiability	Idempotency	Outcome assumption
Non persistent	No	No	No
Presumable	No	No	Yes
Idempotent	No	Yes	-
Verifiable	Yes	-	-

Table 4.4: State management contract subtypes according to state management property values

4.4.2 State management rules

A rule of a state management contract determines how to make persistent the execution state of a given execution unit. A rule specializes the class *Rule* (see Definition 5), for example following rules are related to the presumable contract:

- *smcRule*1 specifies that, it is necessary to store in the log the state of an execution unit at the beginning of its execution:

class *smcRule*1: *Rule* {
on euStarted,

do writeEuState
}

- *smcRule*5 specifies that during recovery, if the beginning of the execution of the execution unit is stored in the log, then its state is the outcome assumption:

*class smcRule*5: *Rule* {
on recoverEuState where $begining(eu) \in log$,
do assumeEuState
}

Events

Events determines when to trigger reactions related to manage the state of execution units, for example following events are related to presumable contract:

- The necessity of recovering the execution state of an execution unit from the log:

class recoverEuState: *Event* {
delta {{*eu* $\mathcal{EU}$}}
}

- The beginning of the execution of an execution unit (i.e., the execution unit has been started):

class euStarted: *Event* {
delta {{*eu* $\mathcal{EU}$}}
}

Reactions

Reactions specifies where to store and how to recover the state of execution units. A reaction specializes the class *Reaction* (see Definition 7), for example following reactions are related to presumable contract:

- *writeEuState* reaction indicates that the execution state of the execution unit must be stored in the log:

class writeEuState: *Reaction* {
input {{*eu* $\mathcal{EU}$}},
output {{*rResult* `Boolean`}}
}

- *assumeEuState* indicates that the execution state of the execution unit must be assumed by using its outcome assumption:

class assumeEuState: *Reaction*{
input {{*eu* $\mathcal{EU}$}},
output {{*rResult* `list`(*State*)}}
}

4.4.3 Presumable contract

According to state management property there are four contracts subtypes that can be defined. For example, a presumable contract can be associated to an execution unit that has an outcome assumption. Therefore, it is only necessary to store in the log the beginning of its execution. During recovery an executed execution unit is signaled in the log by the beginning of its execution. Next, its execution state can be known by using its outcome assumption. A presumable contract specializes the class *stateMContract* as follows:

class presumContract: *stateMContract* {
property {*values* {{name "state-Verifiability", *value* False},
{*name* "idempotency", *value* False},
{*name* "outcome-Assumption",
value {"committed", "failed"}}}},
rules {r1 *smcRule*1, r2 *smcRule*5}
}

For example, in the "purchase tickets application" the execution unit *process purchase* can be considered as an execution unit that can be presumed to commit. This situation can be captured by means of a presumable contract instance with name *prCpP* as follows:

prCpP presumContract :{
scope pP,

property {*values* {{name "state-Verifiability", *value* False},
{*name* "idempotency", *value* False},
{*name* "outcome-Assumption",
value "committed"}}},
priority 2
}

where:

- pP is the identifier of an instance of the class $\mathcal{EU}$ representing the execution unit *process purchase*.
- The outcome-assumption of the execution unit is to commit.
- The priority of the contract is 2.

The complete definition of the state management contracts is presented in the Appendix A.

4.5 Persistency guarantees contract

We assume that the most important cost for doing log based recovery is the access to storage supports. Therefore, we have characterized the storage support as, cached which is volatile and with fast access rates; and stable which is permanent and with slow access rates. Based on these criteria, we have defined two persistency guarantees:

- *Best effort* guarantee specifies that the changes in the execution state are stored in a cached log. It implies that in case of system failures the log may not be available for the recovery process. Because part of the execution history is stored in a cached log until a forced write happens (e.g. by passivation or shutdown calls) the execution is allowed to continue.
- A *guaranteed* effort specifies that the changes in the execution state are stored in a stable log immediately at the contract end. It implies that in case of system failures the log survives and it will be available for doing the recovery process. Therefore, the execution is not allowed to continue until the writing operation is completed.

We address persistency guarantees by means of the persistency guarantees contract. It is a composite contract that associates persistency property to a set of contracts. The contracts within an persistency guarantees contract must be of type state management or persistency guarantees. The persistency guarantees contract subclass specializes the class *CompositeContract* (see Definition 10) as follows:

SubClass (Persistency guarantees contract)

class persistGContract : *CompositeContract* {
scope `set`(*stateMContract* ∪ *persistGContract*),
property persistencyProperty,
}

4.5.1 Property

The persistency property describes the type of persistency guarantees associated to a scope. It specializes the class *Property* (see Definition 4) as follows:

SubClass (Persistency property)

class persistencyProperty : *Property*{
values {{*name*"persistencyType",
value {"bestEffort", "guaranteed"}}}
}

where "persistencyType" specifies the type of persistency guarantee: "*best effort*" or "*guaranteed*".

4.5.2 Persistency guarantees rules

A persistency guarantees rule specifies where to store the execution state and how to do the recovery process using it. A persistency guarantees rule specializes the class *Rule* (see Definition 5), for example the following rules are related to a best effort contract:

- *pgcRule*1 specifies that, if there is a change in the execution state, then set writes to the cached log:

class $pgcRule1$: *Rule* {
on $exStateChg$,
do $setWrToCachedLog$
}

- $pgcRule2$ specifies that, a system failure begins a crash recovery process:

class $pgcRule2$: *Rule* {
on $syFailure$,
do $beginCoRecovery$
}

The definitions of the class events $exStateChg$ and $coFailure$, and the class reactions $setWrToCachedLog$ and $beginCoRecovery$ are presented in the next subsections.

Events

The reactions of a persistency guarantee contract are triggered by several events, for example the following events are related to a best effort contract:

- $exStateChg$ represents the fact that there is a change in the execution state of an execution unit. It specializes the class *Event* (see Definition 6):

class $exStateChg$: *Event* {
delta {{*eu* $\mathcal{EU}$}}
}

- $syFailure$ represents the fact that the coordination execution has failed by a system failure and a recovery process must be started. It specializes the class *Event* (see Definition 6):

class $syFailure$: *Event* {
delta {{$coName$ `String`}}
}

Reactions

Reactions of persistency guarantees contracts specializes the class *Reaction*. For example, following reaction types are related to best effort contract: Following action types are associated to persistency contracts:

- *setWrToCachedLog* represents the fact that all results of write operations must be stored in the cached log until a forced write happens:

  class $setWrToCachedLog$: *Reaction* {
  $input$ {},
  $output$ {{$rResult$ `Boolean`}}
  }

- *beginCoRecovery* represents the fact that the coordination execution has failed and a recovery process was started. Therefore, an event of type $recoverEuState$ is generated for all execution units stored in the log:

  class $beginCoRecovery$: *Reaction* {
  $input$ {{$coName$ `String`}},
  $output$ {{$rResult$ `Boolean`}}
  }

4.5.3 Best effort contract

According to the possible values of the property *persistencyProperty*, a *persistGContract* can be of two types: guaranteed or best effort. For example, the best effort contract specializes the class *persistGContract*. It specifies that, the changes in the execution state are stored in a cached log, and in case of a system failure a recovery process of the execution history must be started:

class $bePGC$: *persistGContract* {
$property$ {$values$ {$name$ "type", $value$ "bestEffort"}},
$rules$ {$pgcR1$ $pgcRule1$, $pgcR2$ $pgcRule2$ }
}

For example, in the "purchase tickets application" the execution of the execution unit *process purchase* can be handled with a best effort contract, assuming that it has an outcome assumption. This situation can be captured by means of a best effort contract instance with name $beC1$ as follows:

*beC*1 *bePGC* :{
scope {*prCpP*},
priority 2
}

where:

- The *scope* of the contract contains a state management contract. *prCpPC* is the identifier of an instance of the contract class *presumContract*. It represents that the execution unit *process purchase* has a presumable contract.
- The priority of the contract is 2.

The complete definition of the persistency guarantees contracts is presented in the Appendix A.

4.6 Conclusion

This chapter proposes an approach for adding reliability to coordination using our contract model. Such a model enables to address both atomic behavior and persistency guarantees in an orthogonal way:

- The treatment of semantics failures enables to provide recovery. It is provided to a given coordination by means of two contract types:
  - Failure contract defines how an execution unit can be treated in case of failure.
  - Atomicity contract ensures one of following behavior for a set of contracts: *Strict atomicity*, *Alternative atomicity* or *Exception atomicity*.
- Persistency requirement is related to how system failures can be treated in order to provide recovery. It can be defined to a given coordination by means of two contract types:
  - State management contract enables to know the execution state of an execution unit in case of a system failure.
  - Persistency contract determines some writing guarantees associated to execution history used in a recovery.

Next chapter presents the evaluation of contracts and discusses the orthogonality of reliability contracts.

Chapter 5

Contracts' evaluation

This chapter describes the strategies we propose for evaluating the contracts associated to a services coordination. Given a services coordination the strategies specify when to evaluate a contract with respect to the execution of an execution unit and when and how to execute the reaction of the contract. It must be noted that, contracts evaluation hides a high degree of complexity. Several questions about the evaluation process raise at execution time, for example, how to evaluate several contracts triggered simultaneously? At which moment is it useful to evaluate a contract? What kind of execution model must be used for evaluating the rules? What type of synchronization model must be used for executing the reactions within the coordination execution? What to do with incompatible reactions? Our work addresses the contracts' evaluation with the following hypothesis:

- The contracts are evaluated within the execution of execution units at two given points. Recall that such points are represented in COBA by the notion of execution unit state (see Section 3.2.1).
- The notion of contract tree (i.e., composite contract, see Section 3.5.2) and the execution order (i.e., control flow, see 3.2.2) are used to establish an evaluation order of contracts triggered simultaneously.
- The rules triggered at the same time are evaluated according to the evaluation order of its contracts.
- The synchronization of reactions and execution is done in a preemptive way.

Besides, according to the COBA model, there are two contract types to be evaluated. Therefore, we present the strategies associated to evaluating one simple contract and

one composite contract. We use as an example the evaluation of reliability contracts.

This chapter is organized as follows. Sections 5.1 and 5.2 describe respectively how to evaluate a simple and a composite contract. Section 5.3 presents the evaluation process of several contracts. In the Section 5.4 we discuss about the evaluation of contracts used for adding reliability to coordination. Finally, Section 5.5 concludes the chapter.

Résumé: Ce chapitre décrit les stratégies que nous proposons pour l'évaluation de contrats associés à une coordination de services. Les stratégies spécifient à quel moment il faut évaluer un contrat par rapport à l'exécution d'une unité d'exécution ; quand et comment faut il exécuter la réaction d'un contrat par rapport à la notification d'une exception ; comment évaluer plusieurs contrats déclenchés simultanément? Notre travail porte sur l'évaluation des contrats avec les hypothèses suivantes :

- Les contrats sont évalués dans l'exécution d'unités d'exécution à deux moments de l'exécution : au début et à la fin.
- La notion d'ordre d'exécution est utilisée pour établir un ordre d'évaluation des contrats déclenchés simultanément.
- Les règles déclenchées en même temps, sont évaluées en fonction de l'ordre d'évaluation des contrats.
- La synchronisation des réactions et l'exécution d'une unité d'exécution se fait de manière préemptive.

Le modèle COBA définit deux types de contrat : simple et composite. Par conséquent, le chapitre présente les stratégies associées à l'évaluation d'un contrat simple et d'un contrat composite. Nous utilisons comme exemple l'évaluation des contrats de fiabilité pour illustrer les stratégies d'exécution proposées.

5.1 Evaluation of one simple contract

The evaluation of one simple contract is done, within the execution of the execution unit defined within its scope. Recall that according to the COBA model, a contract represents the relationship between a property and a scope:

- A simple contract is associated to an execution unit which is defined in the scope of the contract (see Section 3.5.1).

- An execution unit can be associated with several states linked with its execution (see Section 3.2.1).
- A contract has associated a set of rules where a rule specifies the reactions to be executed for enforcing a property in the occurrence of an event (see Section 3.4).

Figure 5.1 shows the four states and the four steps of the evaluation process of one simple contract[1]:

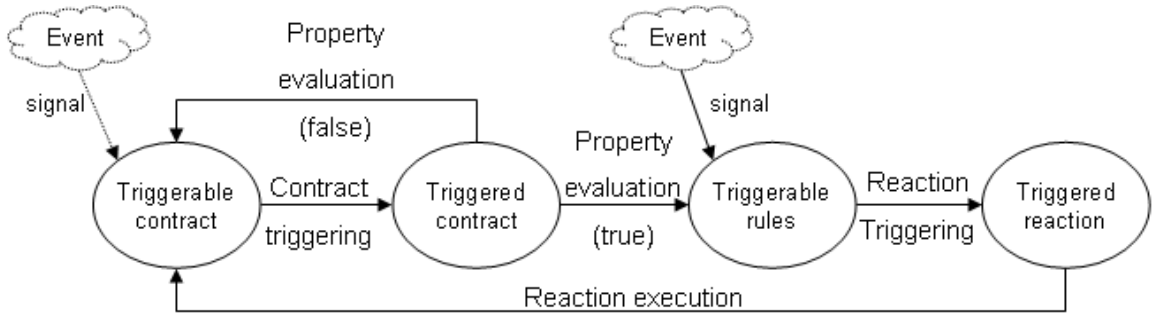

Figure 5.1: Evaluation of a simple contract

- In the state *triggerable*, a contract is waiting for triggering events.
- The state *triggered* is the state where a contract has been triggered by an event.
- In the state *triggerable rules*, the rules of a contract are activated and they are waiting for triggering events.
- The state *triggered reaction* is the state when a reaction has been activated for being executed.

Next Sections present details about each step involved in the evaluation process. To illustrate such a process, let us consider the following contract instance associated to the execution unit instance sT (i.e., *Send tickets*) of the "purchase tickets application".

$unCsT$ *unContract* :{
scope sT,

[1]The diagram can be seen as a kind of finite-state machine composed of a finite set of states (i.e., the circles that we called states) and transitions between those states (i.e., the arrows that we called steps).

property {*values* {{*name* "compensable", *value* True},
{*name* "side-effects", *value* False},
{*name* "retriable", *value* True},
{*name* "max-retry-no", *value* 5}}},
priority 1
}

The contract *unCsT* defines that the execution unit *Send tickets* has an undoable contract which is a simple contract (i.e., *unContract*, see Section A.1).

5.1.1 Contract triggering

Contract triggering is the process by which a simple contract goes from the state *triggerable* to the state *triggered*, after a triggering event has been notified (see Figure 5.1). A triggering event for a simple contract is detected within the execution of its associated execution unit. Therefore, we propose to associate an execution unit with four states related to three instants: i) the instant before its execution, ii) its execution itself, and iii) the instant after its execution (see Figure 5.2). The transitions among the states are defined as follows:

- *Activate* is the process by which an execution unit that is ready for being executed (*prepared* state) goes to being executed (*started* state).
- *Executing* corresponds to the execution of the execution unit. Once the execution of the execution unit has been completed, the execution unit goes to *terminated* state.
- *Commit* is the process in which the execution results of the execution are committed. At the end of this process the execution unit reaches the *validated* state.

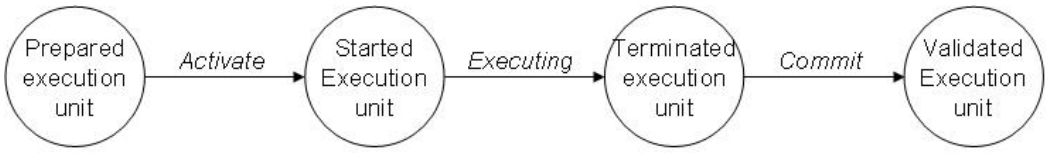

Figure 5.2: Execution of an execution unit

Taking into account the above states, we consider that a simple contract is triggered during the steps *Activate* and *Commit* (see Figure 5.2):

- A contract triggered during the step *Activate* (i.e., before the execution of the execution unit) can control the context under which the execution happens (e.g., to modify the execution requirements or to store the execution state).
- A contract triggered during the step *Commit* (i.e., after the execution of the execution unit) can take actions over the results of the execution before committing (e.g., to re-execute an failed execution unit).

In our example, *unCsT* is triggered at two instants within the execution of *sT*, as is shown in Figure 5.3, which correspond to the steps *activate* and *commit* of the execution of an execution unit.

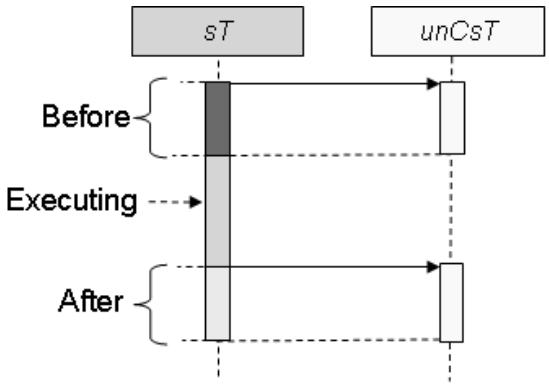

Figure 5.3: Contract triggering of *unCsT*

5.1.2 Property evaluation

Recall that, in the COBA model, a property is a the set of variables that represents a combination of values that constraint the execution. During this process the property is evaluated with respect to the execution state of the execution unit. If the property has been evaluated to true, then the rules of the contract are activated and they wait for triggering events, otherwise the contract goes back to the *triggerable* state (see Figure 5.1). At this point the rules must be evaluated. The evaluation of rules follows the same reasoning as contracts, if a triggering event is notified, a reaction is triggered. Therefore, the contract goes from the state *triggerable rules* to the state

triggered reaction. The evaluation of rules is inspired in the execution model framework proposed by Coupaye and Collet (see [CC98]). In our approach a rule can be evaluated immediately after the notification of an event. It is executed every time an event is notified. Finally, rules cannot be executed in cascade.

In our example, the values of the instance of *exceptionProperty* are analyzed for activating the rules of the contract (i.e., rule instances $r1$ and $r2$ of class $recRule5$ and $recrule6$ respectively, see Section A.1). The rules are activated only after the execution of the execution unit because it is when an execution unit can be retried in case of failure (i.e., rule class $recRule5$) or it can be compensated if it was committed (i.e., rule class $recRule6$). The Figure 5.4 shows the moments at which the rules are activated.

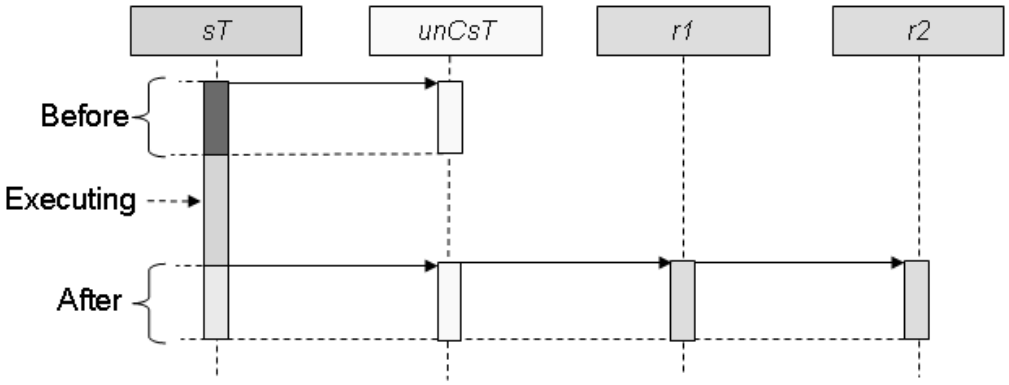

Figure 5.4: Property evaluation of $unCsT$

5.1.3 Reaction triggering

Is the process in which, a given triggering event was notified and it triggers one or more reactions. Reactions are triggered according to the order of rules within the definition of the contract. The execution of a reaction must be synchronized with the execution of the execution unit by means of an execution plan that orders its execution. In our approach, the execution plan is built considering, i) the instant at which the contract evaluation was triggered, ii) the number of triggered reactions, and iii) a preemptive criteria for executing reactions. For example, there are two cases when one reaction is triggered:

- Case 1: the contract was triggered before the execution of the execution unit. Therefore, the execution of the execution unit cannot start until the execution of

its reaction finishes. For example, the Figure 5.5 presents the interactions among, the execution of an execution unit, the evaluation of its associated contract and the execution of its reactions.

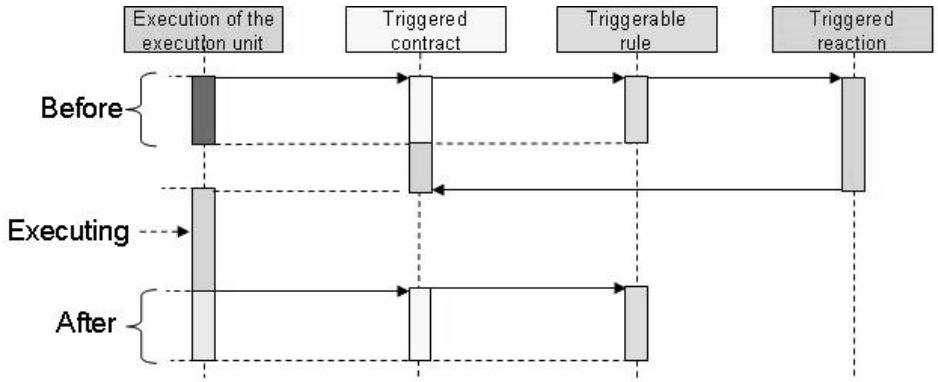

Figure 5.5: Execution example of one reaction before the execution of an execution unit

- Case 2: the contract was triggered after the execution of the execution unit. Therefore, the execution of the execution unit is synchronized with the execution of the reaction. For example, the Figure 5.6 shows a case where the evaluation of a contract and the execution of the execution unit does not finish until the reaction finishes.

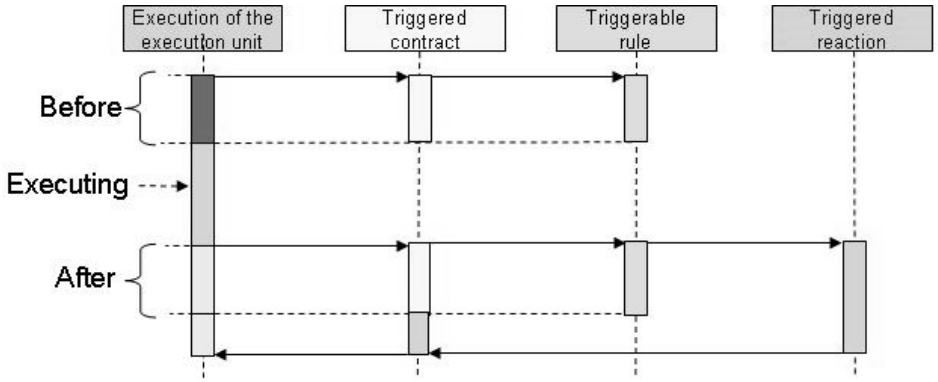

Figure 5.6: Execution example of one reaction after the execution of an execution unit

In our example, once the rules $r1$ and $r2$ are activated they wait for the triggering events $compReqEv$ and $failEv$. Let us consider that the execution of sT fails and

therefore event $failEv$ happens, then an execution plan is built for re-executing sT. Next, the reaction of $r1$ is triggered (i.e, reaction *retry*, see Figure 5.7).

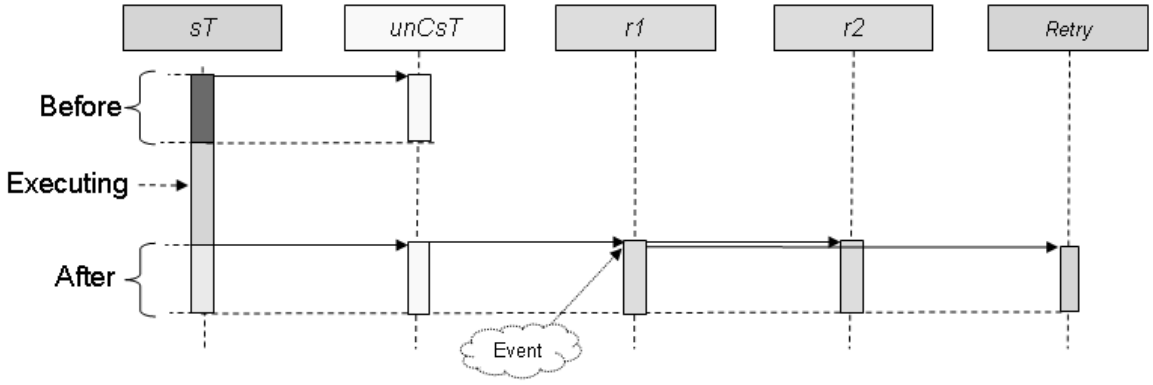

Figure 5.7: Reaction triggering of $unCsT$

5.1.4 Reaction execution

The execution of reactions must be synchronized with the coordination execution. It must take into consideration, on the one hand, that execution units are provided by autonomous services, and on the other, that the execution of some execution units can take a long time (i.e., hour or days). Therefore, synchronizing the execution of reactions with the coordination execution can have an impact over the execution of the application in terms of execution time. In our approach, once the execution plan is built, the execution is synchronized. Recall that, execution of reactions is preemptive, therefore the execution of the coordination must be interrupted for executing the reactions according to a specific order (e.g., when several reactions are triggered). The reactions execution is done in three phases:

1. The execution of the execution unit is interrupted at the step *Activate* or *Commit* (see Figure 5.2).

2. The reactions are executed according to the order defined in the execution plan built in the *Reaction triggered* step (see Figure 5.1).

3. The results of the reactions execution are committed and therefore, the execution continues (i.e., the contract was triggered before the execution of the execution unit) or finishes (i.e., the contract was triggered after the execution of the execution unit).

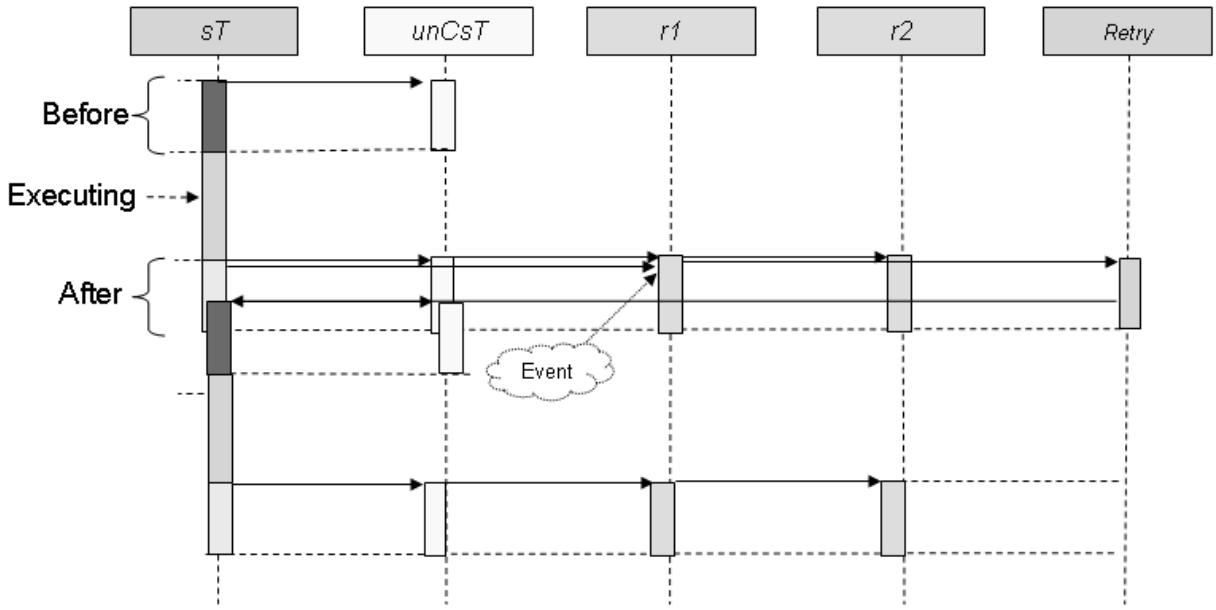

Figure 5.8: Reaction execution of $unCsT$

Finally, at the end of the execution of the reactions the contract comes back to the *triggerable* state (see Figure 5.1).

In our example, the reaction *retry* re-executes the execution unit if possible, according to the value of "max-retry-no" in *exceptionProperty* and the number of times that sT appears in the execution history as committed (see Figure 5.8). Note also that, the re-execution of sT triggers again the evaluation of $unCsT$.

5.2 Evaluation of one composite contract

The second case to consider in the evaluation of contracts is the evaluation of one composite contract. Recall that according to the COBA model, a composite contract is associated with a set of contracts (see Section 3.5.2). Therefore, its evaluation is done within the evaluation of the contracts defined in its scope. The contract tree related to the composite contract is useful within the evaluation process:

- The tree is used for determining when a composite contract must be evaluated.
- The states of the contracts belonging to the tree are used when evaluating the property of the contract.

- The state of the execution units related to the contract tree is also considered.

The evaluation process of a composite contract has four states and four steps as is shown in Figure 5.9. However, the event that triggers the evaluation of a composite contract is the evaluation process of one of the contracts within its scope. Next Sections present details about the evaluation process.

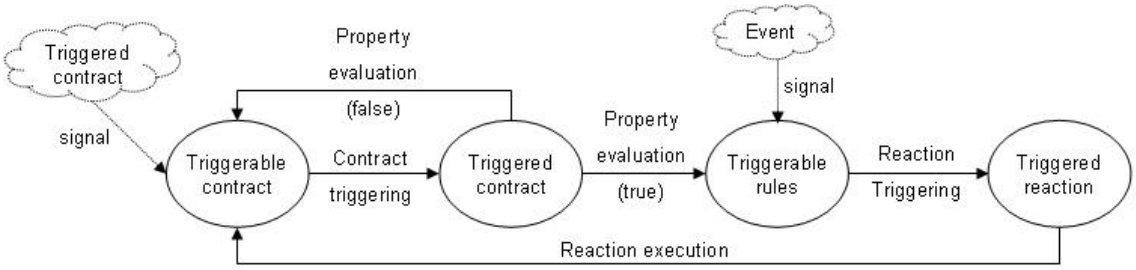

Figure 5.9: Evaluation of a composite contract

In order to illustrate the evaluation of a composite contract, let us consider the following contract instance related to the "purchase tickets application". The composite contract $c1$ (see Figure 5.10) associates three contract instances with strict atomicity behavior (i.e., contract type $\mathcal{stAtC}$, see Section A.2), according to the business rules of the application (i.e., "a purchase order can be completed only if it was paid"):

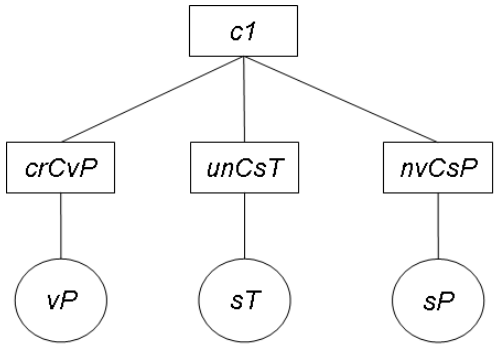

Figure 5.10: Composite contract $c1$

$c1\ stAtC\ \{$
$\quad scope\ \{crCvP,\ unCsT,\ nvCsP\},$
$\quad priority\ 2$
$\}$

where:

- $crCvP$ is an instance of a critical contract related to execution unit vP (i.e., *Validate payment*).
- $unCsT$ is an instance of an undoable contract related to execution unit sT (i.e., *Send tickets*).
- $nvCsP$ is an instance of a non vital contract related to execution unit sP (i.e., *Send publicity*).

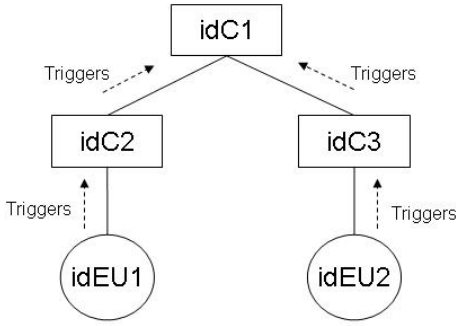

Figure 5.11: Triggering a composite contract

5.2.1 Contract triggering

Contract triggering is the process in which a composite contract goes from a *triggerable* state to a *triggered* state, because of a triggering event has been notified. Because of the evaluation of a composite contract is done within the evaluation of the contracts defined in its scope, a composite contract is triggered by the evaluation of one or more of the contracts within its scope. Note that, all ancestors of a simple contract within its contract tree are triggered when the simple contract is being evaluated.

For example, let us consider the contract tree of Figure 5.11, the evaluation of the simple contract instance $idC2$ triggers the evaluation of the composite contract instance $idC1$ (see Figure 5.12).

In our contract example (see Figure 5.10), the contract $c1$ is triggered when the contracts within its scope are triggered (see Figure 5.13):

- The contract $crCvP$ is triggered within the execution of vP.

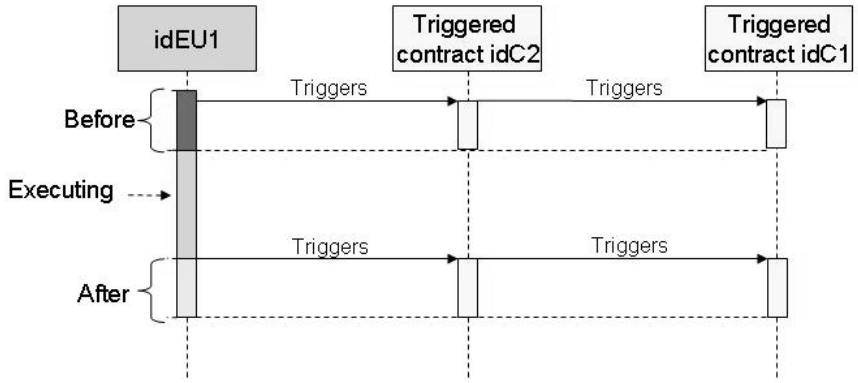

Figure 5.12: Sequence diagram for triggering a composite contract

- The contract $unCsT$ is triggered within the execution of sT.
- The contract $nvCsP$) is triggered within the execution of sP.

5.2.2 Property evaluation

The evaluation of a property of a composite contract is done over the values of the contracts within its scope. Therefore, the variables that represent the property are evaluated over the execution state and over the the values of the contracts within its scope. If the property is evaluated to true, then the rules of the contract are activated (i.e., they wait for triggering events), otherwise the contract goes back to the *triggerable* state (see Figure 5.9). When a rule is activated and a triggering event is detected, a reaction is triggered. Recall that, a rule can be evaluated immediately after the notification of an event, it is evaluated every time an event is notified and rules cannot be executed in cascade.

As an example, let us consider the execution of execution unit vP. The rule is activated only after the execution of the execution unit because it is when an execution unit can fail. The Figure 5.15 shows the moments at which the rule is activated.

In our example, let us consider the execution of execution unit vP. The rule $r1$ of the contract $c1$ is activated only after the execution of the execution unit because it is when an execution unit can fail. The Figure 5.15 shows the moments at which the rule is activated.

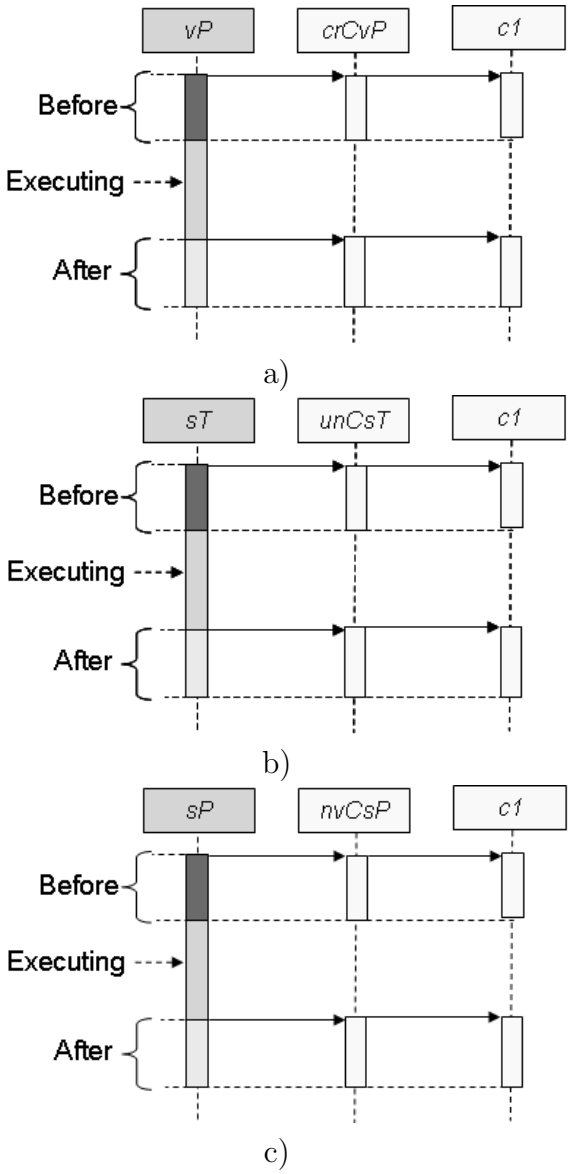

Figure 5.13: Contract triggering of *unCsT*

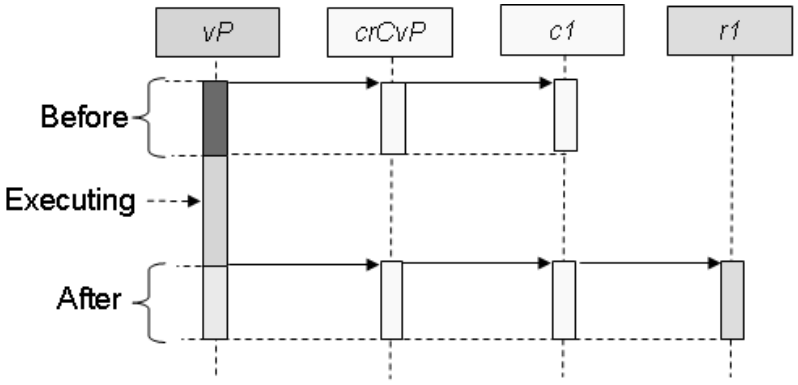

Figure 5.14: Property evaluation of $c1$

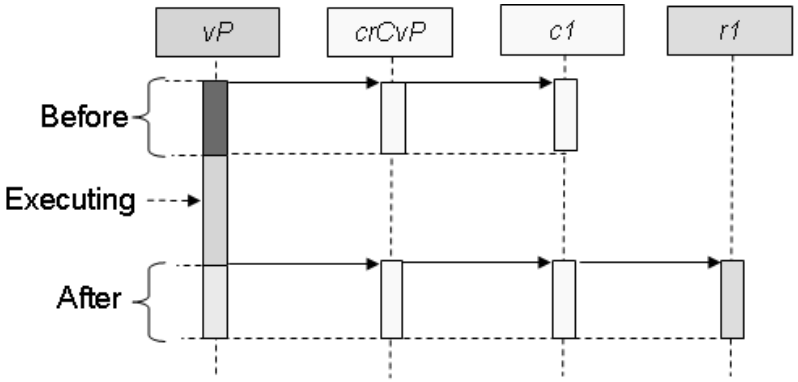

Figure 5.15: Property evaluation of $c1$

5.2.3 Reaction triggering

At this phase, an execution plan is built for executing the triggered reactions. To synchronize the execution, the building process of the plan takes into account: i) the instant at which the contract evaluation was triggered, ii) the number of triggered reactions within the contract tree, and iii) the contract tree itself. Our approach uses the following rules for determining the execution order of reactions:

1. Simple contracts: a reaction triggered by a simple contract has priority over the triggered reactions of its ancestors. Consequently, reactions triggered in composite contracts must wait its finalization to be executed.

2. Composite contracts: a reaction initiated by a composite contract is executed first that reactions triggered by its ancestors (bottom-up order). We assume that, a node close to the root has less priority that a node close to the leafs.

3. FIFO order: if a composite contract is triggered by two or more contracts, then FIFO order is used for executing its reactions. This means that, evaluation is done according to the execution order of the execution units.

For example, let us consider that the contract instance $idC1$ of the contract tree of the Figure 5.11 is being evaluated under the following scenario:

- $idC1$ was triggered by $idC2$.
- $idC2$ was triggered before of the execution of $idEu1$.
- An event is notified to rule $idRc1$ of contract $idC1$ triggering the reaction $idR1$.

Therefore, the execution of $idEu1$ waits the finalization of reaction $idR1$ for beginning its execution (see Figure 5.16), because the execution of reactions is done in a preemptive way.

In our example, once the rule $r1$ is activated it waits for the triggering event $contractFailure$. Let us consider that the execution of vP fails, therefore contract $crCvP$ signals a failure and the event $contractFailure$ is notified. At this moment an execution plan is built for applying backward recovery (see Figure 5.17).

5.2.4 Reaction execution

The execution of reactions in a composite contract follows the same approach that simple contracts. Once the execution plan is built, the execution is synchronized. The reactions execution is done in three phases:

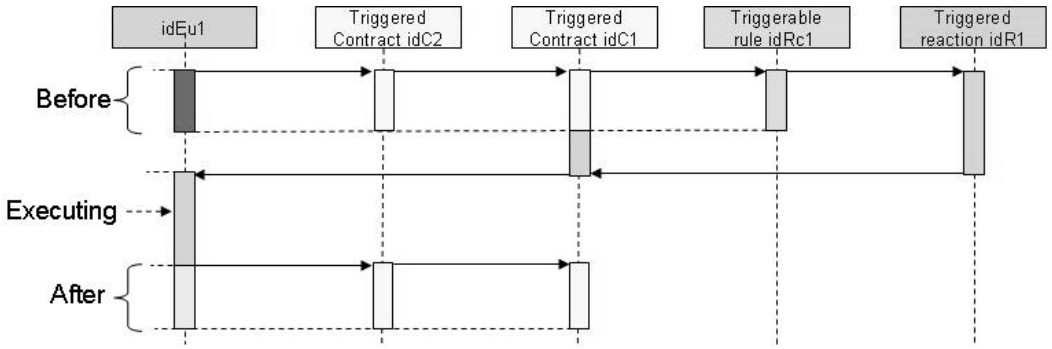

Figure 5.16: Evaluation example of a composite contract

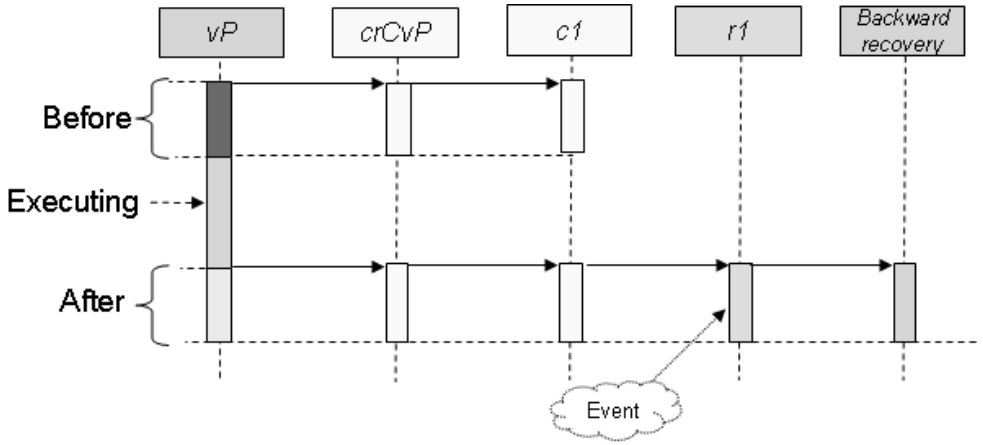

Figure 5.17: Reaction triggering of $c1$

1. The execution of the execution unit, the one that is descendant of the composite contract, is interrupted at the step *Activate* or *Commit* (see Figure 5.2).

2. The reactions are executed according to the order defined in the execution plan built in the *Reaction triggered* step (see Figure 5.1). Such reactions can be triggered by several contracts within the contract tree.

3. The results of the reactions execution are committed and therefore, the execution continues (i.e., the contract was triggered before the execution of the execution unit) or finishes (i.e., the contract was triggered after the execution of the execution unit).

Finally, at the end of the execution of the reactions the contract comes back to the *triggerable* state (see Figure 5.1).

In our example (see Figure 5.10), the reaction *backward recovery* undoes the previous committed execution units within the scope of the contract $c1$. In this case, the only execution unit that must be compensated is sT because of:

- sT is associated with an undoable contract (i.e., $unCsT$), therefore it must compensated to undone its actions.

- sP is associated with a non vital contract (i.e., $nvCsP$), therefore it does not requires to be compensated.

- vP is the execution unit that has failed and initiates the backward recovery reaction because it cannot be retried (i.e., it has associated a critical contract).

5.3 Evaluation of several contracts

When a coordination is related with several contract types, the evaluation of the different contracts must be ordered. The evaluation process implies, i) to order the evaluation of several contracts, and ii) to synchronize the execution of several reactions and the execution of the coordination. There are three scenarios to be considered:

1. Two or more non related contracts must be evaluated. In this case, the evaluation order of the contracts is done according to the execution order using a FIFO policy. We assume that, the contracts that are not related can be evaluated and its reactions can be executed in an isolated way. For example, in the Figure 5.18, the evaluation process of contract tree 1 (i.e., contracts $idC1$ and $idC2$) does

not cause conflicts with the evaluation process of contract tree 3 (i.e., contracts $idC6$ and $idC7$) because they are not related in any way (i.e., by a contract or by an execution unit). In a similar way, contract tree 1 and 2 can be evaluated without conflicts.

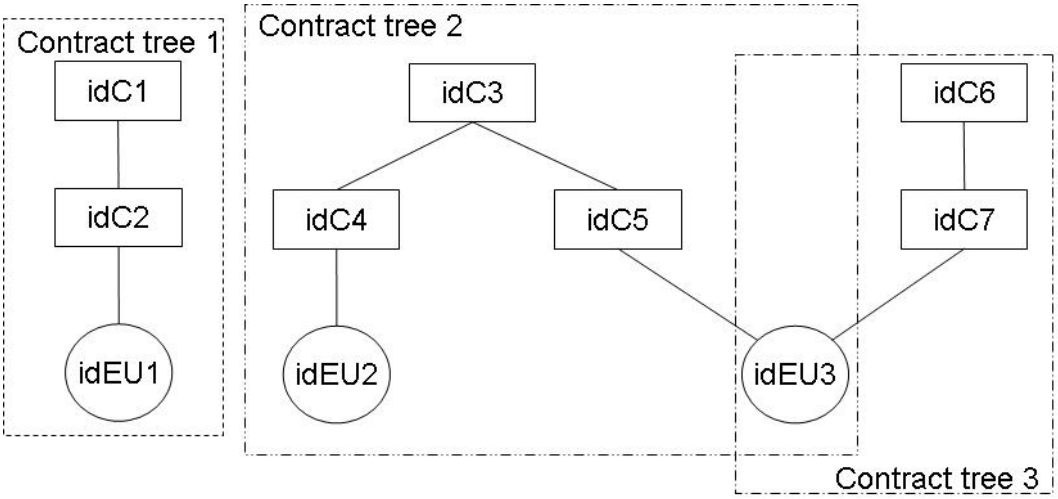

Figure 5.18: Evaluation of several contracts within a same coordination

2. Two or more contracts related by a contract must be evaluated. This is the case of a composite contract and the rules for evaluating such a kind of contracts are applied (see Section 5.2). For example, in the Figure 5.18 the contracts instances $idC4$ and $idC5$ are related by the contract instance $idC3$. Because of $idC3$ is a composite contract it must be evaluated applying the rules for evaluating one composite contract.

3. Two or more contracts related by an execution unit must be evaluated. In this case, the priority of each contract is used (see Definition 3.5). For example, in the Figure 5.18 the contracts instances $idC5$ and $idC7$ are related by the execution unit $idEU3$. When the execution of $idEU3$ triggers $idC5$ and $idC7$ the priority of each contract type is used for determining which contract has higher priority over the other.

5.3.1 Evaluation example

The Figure 5.19 shows an example of how the evaluation order is determined for several contracts. In the top of the Figure, the contracts to be evaluated are shown with its

corresponding contract trees. In the middle, the execution order and the triggering of the contracts is shown. In the bottom, the evaluation order is presented (dashed lines), it is determined as follows:

- First, execution unit $EU1$ is executed and therefore its associated contract and its ancestor is triggered for being evaluated (i.e., contracts $C1$ and $C3$).
- Next, execution unit $EU3$ is executed which triggers the evaluation of its associated contract and its ancestors (i.e., contracts $C4$, $C5$ and $C6$).
- Finally, execution unit $EU2$ is executed and its associated contract and is ancestors are triggered for being evaluated (i.e. contracts $C2$ and $C3$).

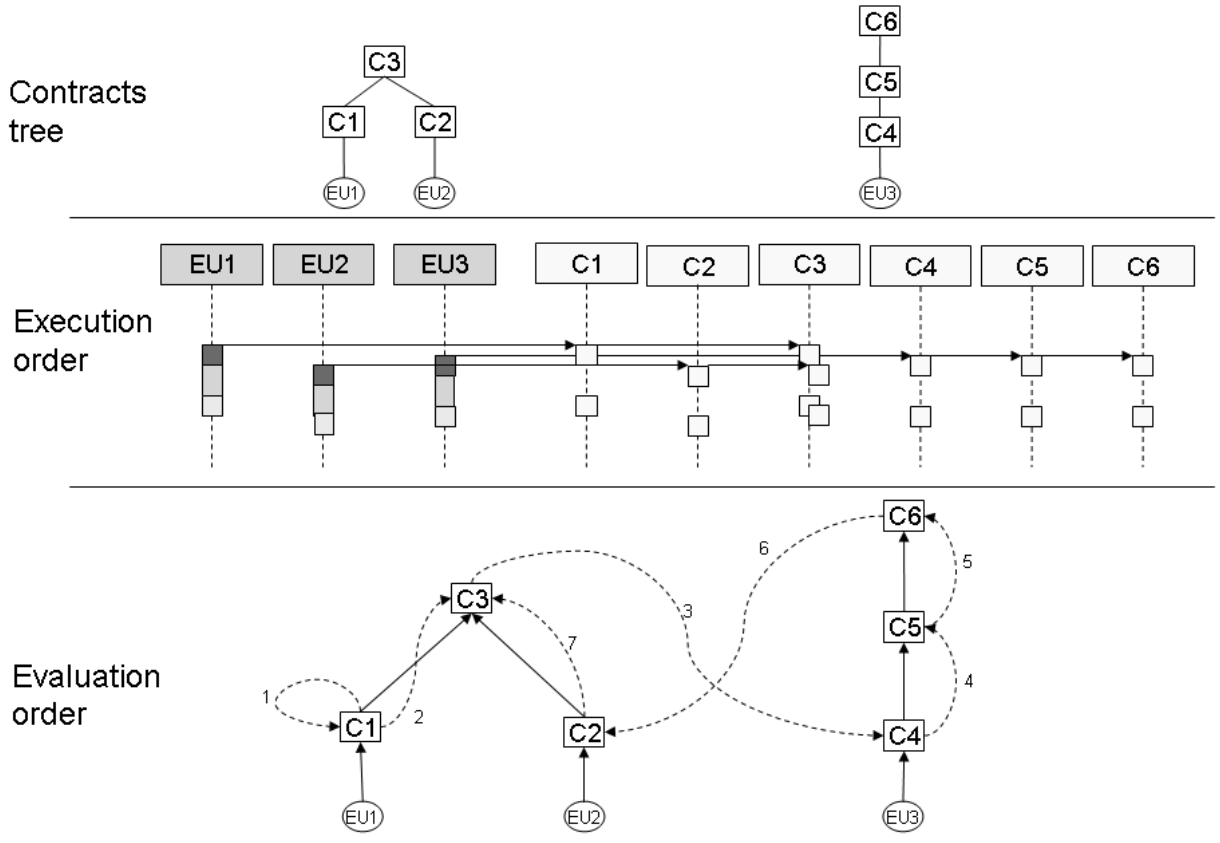

Figure 5.19: Evaluation order for several contracts belonging to a coordination

5.4 Orthogonality of reliability contracts

In this section we discuss about the orthogonality of the contracts' types because although we argued that contracts are orthogonal, there are properties that can cause conflicts. This issue has high relevance when evaluating several contracts. An analysis of this kind must be done considering the type of contract (i.e., simple or composite) and the properties of the contracts (i.e., when the properties are or not related). As an example, we conduct an analysis of four contracts that are related to reliability: i) contracts whose scope is a execution unit (i.e., exception contract and state management contract, see Sections 4.2 and 4.4) and ii) contracts whose scope is a set of contracts (i.e., atomicity contract and persistency guarantees contract, see Sections 4.3 and 4.5).

5.4.1 Simple contracts

The compatibility of exception contract and state management contract must be analyzed when an execution unit is within the scope of two simple contracts. Recall that the evaluation of such contracts is done within the execution of the execution unit. Therefore, the reactions associated to the contracts can define not compatible actions (e.g., in case of failure, a contract can specify a retry reaction and the other an exception reaction).

The Table 5.1 introduces the compatibility matrix for exception and state management contracts. A "N" indicates conflict and a "Y" indicates compatibility. It must be noted that in the case of failure contract the typification includes only the four contract subtypes (i.e., columns, see Section 4.2), and in the case of state management contract it is necessary to consider all the cases (i.e., rows, see Section 4.4). Let us analyze each case:

- The worst case of an non persistent contract (see line 1 in Table 5.1) is when its execution was started and its result is unknown because a failure arises and therefore an an exception is launched signaling that the execution state cannot be recovered. Let us analyze how this situation is compatible with failure contracts:
  - Critical contract. Although a failure in a critical contract can be treated at atomicity level, it is acceptable to launch an exception in case of unknown execution state.
  - Non vital contract. No matter which was the execution result, execution can continue. Therefore, non vital and non persistent contract are incompatible.

– Undoable/compensatable contract. No matter which was the execution result, an undoable/compensatable contract can be retried in case of failure. Therefore, undoable/compensatable and non persistent contracts are incompatible.

- Presumable contract (see line 2 in Table 5.1) is compatible with all failure contracts. The worst case is when after recovery the execution unit fails. In such a case, the execution unit failure can be treated at semantic level by the corresponding failure contract.

- Idempotent contract (see lines 3 and 4 in Table 5.1) is incompatible with critical and non vital contracts because a critical contract cannot be retried and non vital contract does not require its re-execution in case of failure.

- Verifiable contract (see lines 5 to 8 in Table 5.1) has a compatibility problem when the contract is also idempotent (lines 7 and 8). In that case, similar to idempotent contract, verifiable contract is incompatible with critical and non vital contracts.

		Exception contract			Critical	Non vital	Undoable	Comp.
State manag.		Verifiable	Idempotent	Presumable				
1	Non persistent	No	No	No	Y	N	N	N
2	Presumable	No	No	Yes	Y	Y	Y	Y
3	Idempotent	No	Yes	No	N	N	Y	Y
4		No	Yes	Yes	N	N	Y	Y
5	Verifiable	Yes	No	No	Y	Y	Y	Y
6		Yes	No	Yes	Y	Y	Y	Y
7		Yes	Yes	No	N	N	Y	Y
8		Yes	Yes	Yes	N	N	Y	Y

Table 5.1: Matrix of compatibility for exception and state management contracts

5.4.2 Composite contracts

To provide reliability to services coordination we propose two kind of composite contracts: atomicity contracts and persistency guarantees contracts. While the former associates recovery strategies during execution in case of failures (semantic failures), the second associates recovery strategies when coordination execution crashes (system

failures). Although such contracts are associated to different kind of failures, it must be noted that there are some failures that can be treated by both contracts (e.g., the failure of an execution unit). Therefore, an atomicity contract can be associated with the same scope that a persistency contract, but a priority order for its evaluation must be used.

We assume that persistency guarantees contracts are processed first than atomicity contracts. The reason for this decision is because, before treating a failure according to the application semantics, it is necessary to recover if possible the execution state of the execution.

5.5 Conclusion

This chapter describes the strategies for evaluating contracts. The approach presented in this chapter follows several hypothesis with respect to such a process. The contracts are evaluated within the execution of execution units at two given points. The notion of contract tree and the execution order are used to establish an evaluation order of contracts triggered simultaneously. The rules triggered at the same time are evaluated according to the evaluation order of its contracts. The synchronization of reactions and execution is done in a preemptive way. Using such assumptions, we first present the evaluation phases of one simple contract: contract triggering, property evaluation, reaction triggering and reaction execution. Next we present the evaluation of one composite contract, where the key element is how the contract tree is used for applying the rules that we propose for its evaluation. Finally, we present how to evaluate a set of contracts associated to a coordination and we discusses about the orthogonality of contracts for providing reliability. Next chapter presents an proof on concept of the COBA model and the evaluation process.

Chapter 6

Validation and proof of concept

This chapter describes a proof of concept of the approach that we propose for providing reliability properties to a given services coordination. We present ROSE, a services coordination engine which provides atomic behavior to web services.

The chapter is organized as follows. Section 6.1 introduces the general architecture of a coordination engine for executing reliable coordinations by means of a contracts evaluator. Section 6.2 presents how to extend a coordination engine with a contract evaluator to be able to add exception handling and atomicity properties to a given services coordination. Section 6.3 shows an experimental validation of enacting a services coordination with support to semantic failures (e.g., atomic and exception handling requirements). Finally, Section 6.4 concludes this chapter.

Résumé: Ce chapitre présente les résultats de la validation expérimentale de l'approche que nous proposons dans cette thèse. Le chapitre décrit ROSE, un moteur d'exécution des coordinations de services à base de contrats. Il présente l'architecture générale d'un moteur de coordination et d'un évaluateur de contrats pour l'exécution atomique des coordinations. L'architecture consiste en trois composants principaux :

- Le moteur de coordination qui interagit avec l'évaluateur de contrats à travers une interface qui exporte des méthodes spécifiques pour arrêter, annuler et relancer les unités d'exécution d'une coordination de services.

- L'évaluateur des contrats évalue des contrats en se synchronisant avec l'exécution de la coordination de services.

- Le stockage gère l'histoire d'exécution de la coordination, en particulier, les

modifications de l'état d'exécution. Il est utilisé par l'évaluateur de contrats pour construire le plan d'exécution des contrats.

Le chapitre décrit également les détails techniques de l'évaluation de ROSE avec le moteur de coordination Bonita.

6.1 Coordination engine architecture

In this section we present the basic architecture that a coordination engine must have to evaluate contracts based on the COBA model. The Figure 6.1 presents the components of the architecture we proposed for enacting a services coordination in a reliable way. There are three main elements in the architecture:

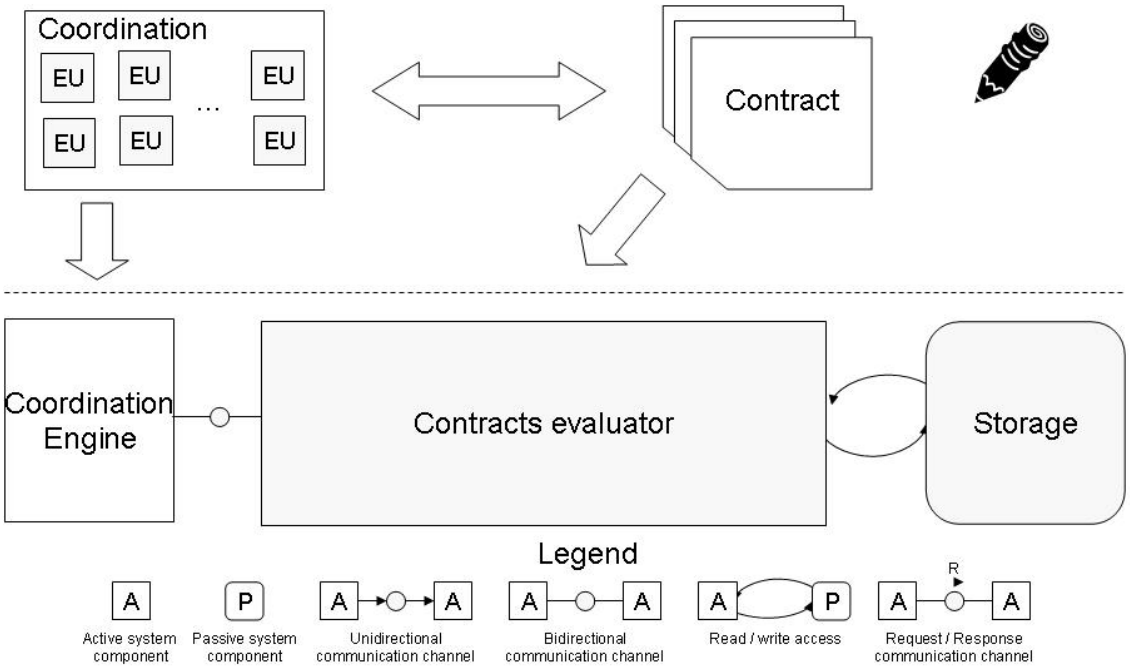

Figure 6.1: General architecture

- The *Coordination engine* is the core of the coordinator because it enacts the coordination. Recall that we assume that such component already exists. Therefore, it must comply with the following characteristics:

  - Preemption right for enabling to interrupt the execution of the coordination. In such a way, it is possible to interrupt the execution of an activity at given points of its execution (e.g., see Figure 5.2). This is necessary because in such points the evaluation of the contracts is done. Besides, it is possible to synchronize the execution of the reactions with the coordination execution.
  - The execution of the units execution must be atomic. This characteristic is hard to provide in the case of services (e.g., long running activities). However, we assume that the coordination engine is able to take a decision to fail the execution of activities in specific situations (e.g., by using a timeout). In such a way, the execution of an execution unit always commits or fails which enables the evaluation of the contracts.
  - The execution state of the coordination can be modified arbitrarily. This is necessary because after the execution of some reactions the execution state may change (e.g., when an execution unit is successfully retried).

- The *Contracts evaluator* is on charge of evaluating the contracts at execution time. The functions of the evaluator are the following:
  - To detect the events that triggers a contract (e.g., an execution unit is going to be executed).
  - To evaluate the property associated to the contract (e.g., the exception handling property). The evaluation of the values representing the property are related to, the current state of an execution unit, a possible contract tree, and the execution history of the coordination.
  - To build if it is necessary an execution plan for synchronizing the reactions with the execution (e.g., to retry a failed execution unit).

- The *storage* is a place where the execution history is stored. The execution history contains the execution state changes and it is used when an execution plan must be built by the contracts evaluator.

The next section details how this architecture is used for extending an existing coordination engine. In particular we analyze the case of enacting coordinations with atomicity requirements.

6.1.1 Bonita

In this section we present a coordination engine called ROSE that enacts coordinations with atomicity requirements. ROSE is not build from scratch, it extends Bonita [Con07a], a coordination engine for enacting web-services based coordinations. Bonita is an engine based on the J2EE platform specification for developing services based applications that runs on JoNAS application server [Con07b]. The engine is able to execute coordinations in a flexible way. It allows activities to share intermediate results when executing. In Bonita, a coordination is expressed by means of workflows. A workflow represents by means of activities and control flow operators the control flow of a coordination where the data flow is encompassed within the control flow. Execution is based on the principle of anticipation, which allows an activity to escape to the start-end synchronization model. Besides, it is possible to cancel the execution of an activity or to change the execution mode of an activity at runtime.

The architecture of Bonita includes following modules (see Figure 6.2):

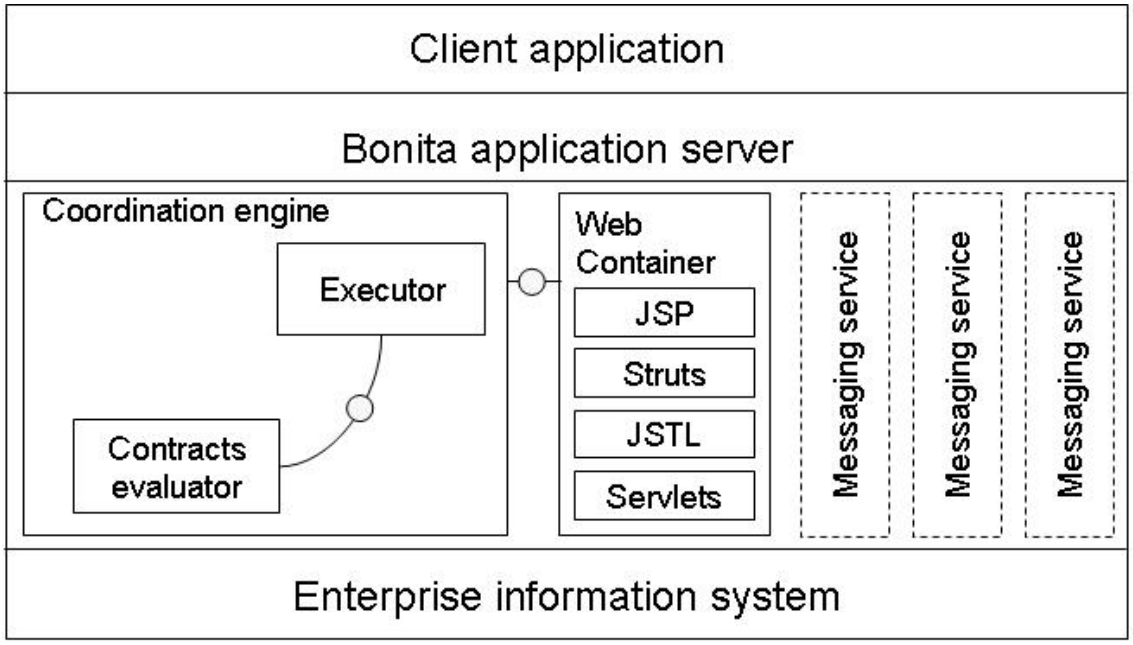

Figure 6.2: General architecture of Bonita

- The *client application* is used for defining projects. A project represents coordinations as workflows. Each project contains the information for being executed: activities names (i.e., method services calls), connectors among activities

(i.e., control flow operators),users, participants, roles and hook components. In Bonita the notion of execution unit of the COBA model is represented by activities.

- The *Bonita application server* layer is composed by the EngineSession bean, the execution modes, and the Worklist application. The Engine StatefulSession bean defines all process execution operations: start activity, terminate activity, cancel activity and terminate process execution. It is based in a recursive implementation that manages the previous execution operations and propagates the activity state changes to the activities that are connected to this one. The coordination engine is responsible of managing different execution modes. This layer integrates services that control and simplify many cooperative aspects:
  - JMS message service implementation notifies the definition and execution changes within a workflow process. Every user interaction is notified to the executor and it throws a JMS event.
  - Activity deadline service that uses the Java Management Extensions (JMX) to advice the user if the execution of an activity does not terminate at the expected date.

  Within the coordination engine it is implemented the contract evaluator as an activity hook. Such an integration is explained in the next Section.

- Finally, the *entreprise information system* layer provides persistency. The execution state persists to system failures thanks to a local database which is used to store information about the projects being executed (i.e., coordinations).

6.1.2 Activity hook

The Bonita application server provides the notion of hook as a way of enhancing its capabilities. A hook performs user-defined operations. It can be coded in an scripting language (i.e., XPDL) or in java (i.e., as java libraries). Bonita considers two types of hooks:

- Process hooks operate at process level, at the very beginning and the very end of a process lifetime.
- Activity hooks operate at activity level, at different activity moments.

We use the notion of activity hooks because an activity hook can be called at different states during activity lifetime. Lifetime of an activity includes not only its execution but the time before and after the execution. This mode of execution is straightforward with the evaluation process defined in the Chapter 5.

An activity hook can be executed in a transactional or a non transactional context, depending upon the detection of certain events that define following points:

- **Before start** hook is called just before the activity starts. It is not considered to be in the same transaction as the activity.
- **After start** hook is called just after the activity has started. It is considered to be in the same transaction as the activity.
- **Cancel** hook is called before canceling an activity and it is considered to be in the same transaction as the activity.
- **Before terminate** hook is called just before the activity terminates. It is considered to be in the same transaction as the activity.
- **After terminate** hook is called just after the activity has terminated. It is not considered to be in the same transaction as the activity.
- **On ready** hook is called when an activity becomes ready. This hook could be used to notify the user responsible for executing the activity with information. It is not considered to be in the same transaction as the activity.
- **On deadline** hook is called when an activity deadline expires. It is not considered to be in the same transaction as the activity.

The above points can be mapped to the execution units states that we propose for the evaluation of reliability contracts (see Figure 6.3).

Using the hook mechanism of Bonita we implement the contracts evaluation (see Chapter 5). Therefore, the contract evaluator is a plug-in component for the engine implemented using hooks.

6.2 Architecture of the contract evaluator

The contract evaluator is on charge of the evaluation process of the atomicity contracts. Figure 6.4 presents how the contract evaluator is implemented within Bonita (we use the notation proposed by [FC08]). The contract evaluator includes five modules:

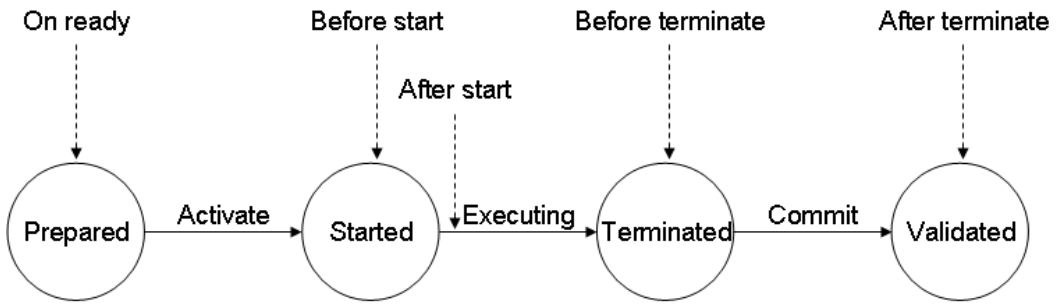

Figure 6.3: Hook events related to execution states of execution units

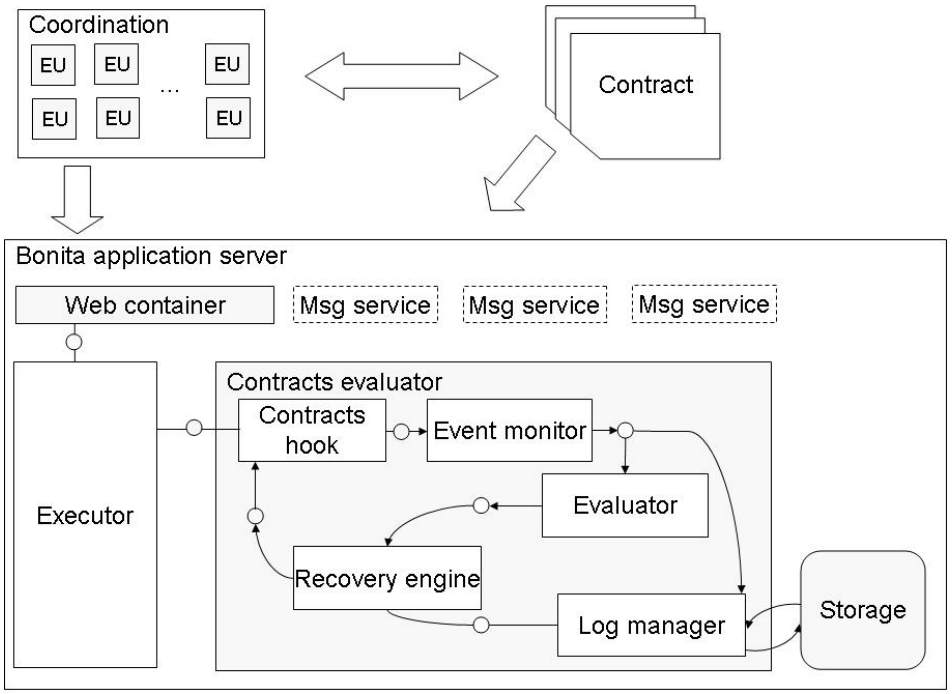

Figure 6.4: Contract evaluator architecture in Bonita engine

- **Contracts hook** is an activity hook. It is on charge of the communication among the executor and the other modules (see Figure 6.4). According to the evaluation model of contracts, this hook is executed at two moments, before and after the execution of an activity (i.e., "before start" and "after terminate, see Figure 5.2). It is executed in a preemptive way while the contract evaluation is done. At the beginning of its execution it provides information about the activity that triggers the hook (i.e., activity name, execution state of the activity, and project name). As a result of its execution a reaction can be executed to ensure the property of a given contract.

- **Event monitor** manages the event occurrences happened at execution time. When an interesting event happens the monitor notifies it to evaluator and to the log manager.

- **Contract evaluator** evaluates the events from event monitor. If necessary it triggers the rules of contracts and the reactions.

- **Recovery engine** generates a recovery plan when a reaction was triggered. Recall that recovery plans are defined given an atomicity contract and the types of execution units within the atomicity contract. After this module sends instructions to the executor through the contracts hook in order to continue the execution.

- **Log manager** manages the execution history of coordination and saves it into a database (i.e., storage component implemented using a MySQL database). Data is related with the event occurrence. When an event is detected, this module updates the execution history. It also retrieves execution the history when it is required.

6.3 Experimental validation

We conducted an experimental validation of the COBA model and the contract evaluator developing the "purchase tickets application"that has atomicity requirements which can be expressed as atomicity contracts. The objective of this experiment is to show how business rules defining the semantics of an application can be ensured by contracts. In our experiment, the following assumptions are used:

- The application logic (i.e., the coordination) is defined using a language based on a coordination formalism (i.e., XPDL). The Listing 6.1 shows a fragment of the XPDL file defining the application logic of the "purchase tickets application". It includes, the Datafields section where data of the application is defined (i.e., lines 15 to 32), the Activities section where activities of the application are defined (i.e., lines 33 to 38), and the Transitions section where the execution dependencies are defined (i.e., lines 39 to 45).

Listing 6.1: XPDL coordination for the "purchase tickets application"

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <Package xmlns="http://www.wfmc.org/2002/XPDL1.0"
3   xmlns:xpdl="http://www.wfmc.org/2002/XPDL1.0"
4   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5   xsi:schemaLocation="http://www.wfmc.org/2002/XPDL1.0
6   http://wfmc.org/standards/docs/TC-1025_schema_10_xpdl.xsd"
7   Id="Concerts_Project" Name="Concerts">
8   ...
9    <WorkflowProcesses>
10     <WorkflowProcess AccessLevel="PUBLIC" Name="Concerts" Id="Concerts">
11       <ProcessHeader />
12       <RedefinableHeader>
13         <Version>1.0</Version>
14       </RedefinableHeader>
15       <DataFields>
16         <DataField Id="decision" Name="decision">
17           <DataType>
18             <EnumerationType>
19               <EnumerationValue Name="grant" />
20               <EnumerationValue Name="reject" />
21             </EnumerationType>
22           </DataType>
23           <InitialValue>grant</InitialValue>
24           <ExtendedAttributes>
25             <ExtendedAttribute Name="PropertyActivity" />
26           </ExtendedAttributes>
27         </DataField>
28         <DataField Id="Acount_number" Name="Acount_number">
29           <DataType>
30             <BasicType Type="STRING" />
31           ...
32       </DataFields>
33       <Activities>
34         <Activity Id="Get_concert_information" Name="Get concert information">
35           ...
36         </Activity>
37         ...
38       </Activities>
39       <Transitions>
40         <Transition Id="Send_publicity_Validate_purchase"
41          Name="Send publicity_Validate purchase" From="Send_publicity"
42          To="Validate_purchase" />
43         <Transition Id="Process_purcahse_Send_publicity"
44          Name="Process purcahse_Send publicity" From="Process_purcahse"
45          To="Send_publicity" />
46         ...
47       </Transitions>
48     </WorkflowProcess>
49   </WorkflowProcesses>
50   <ExtendedAttributes>
51     <ExtendedAttribute Name="MadeBy" Value="ProEd" />
52     <ExtendedAttribute Name="View" Value="Activity" />
53   </ExtendedAttributes>
54 </Package>
```

- The definition of the atomicity contracts is sound for the target coordination.

It must be noted that, due to orthogonality of contracts and coordination, it is possible to have inconsistent requirements of atomicity (e.g., two critical contracts within a strict atomicity contract). We assume that the evaluation does not makes a verification process.

- It is possible to extend the coordination engine with the contrac evaluator. For example, using a plug-in mechanism or modifying the original code of the engine.

6.3.1 Purchase ticket application

The application logic is as follows (see Figure 1.1): given the concert information, the purchase is processed and payment is granted. Once the purchase has been authorized, the payment must be done, the tickets must be sent, and publicity for other events must be sent too.

Besides, let us consider the following business rules (BR) for the application:

- BR-1: when a purchase is done, it is desirable to send publicity about other related events to the customer.
- BR-2: once a purchase has been paid, it cannot be canceled.
- BR-3: a processed purchase that must be canceled generates a process for canceling the reservation and it generates a bad record for the customer.
- BR-4: there are some activities within the application that are internal and canceling them if necessary does not generates any problem.
- BR-5: an order can be validated only if it was paid and sent.

The above business rules are fulfilled by defining exception handling and atomicity contracts as is presented in the next section.

6.3.2 Atomicity contracts

In this section we present how atomicity and exception contracts are used to ensure the business rules requirements at execution time.

1. Execution units are associated with exception handling contracts as follows.
    - Derived from the BR-1 the execution unit *Send publicity* has a non vital contract:

$nvCsP$ *nvContract* :{
scope sP,
priority 1
}

where sP is an instance of the class $\mathcal{EU}$ representing the *Send publicity* activity.

- It is inferred from the BR-2 that the execution unit *Validate payment* has a critical contract:

$crCvP$ *crContract* :{
scope vP,
priority 1
}

where vP is an instance of the class $\mathcal{EU}$ representing the *Validate payment* activity.

- According to the BR-3, the execution unit *Process purchase* has a compensatable contract:

$cpCpP$ *cpContract* :{
scope pP,
property {*values* {{*name* "compensable", *value* True},
{*name* "side-effects", *value* False},
{*name* "retriable", *value* True},
{*name* "max-retry-no", *value* 5}}},
priority 1
}

where pP is an instance of the class $\mathcal{EU}$ representing the *Process purchase* activity.

- Derived from the BR-4 the execution units *Get concert information*, *Send tickets* and *Validate purchase* have undoable contracts:

$unCgCI$ *unContract* :{
scope gCI,

property {*values* {{*name* “compensable”, *value* True},
{*name* “side-effects”, *value* False},
{*name* “retriable”, *value* True},
{*name* “max-retry-no”, *value* 5}}},
priority 1
}

where gCI is an instance of the class $\mathcal{EU}$ representing the *Get concert information* activity.

$unCsT$ *unContract* :{
scope sT,
property {*values* {{*name* “compensable”, *value* True},
{*name* “side-effects”, *value* False},
{*name* “retriable”, *value* True},
{*name* “max-retry-no”, *value* 5}}},
priority 1
}

where sT is an instance of the class $\mathcal{EU}$ representing the *Send tickets* activity.

$unCvPu$ *unContract* :{
scope vPu,
property {*values* {{*name* “compensable”, *value* True},
{*name* “side-effects”, *value* False},
{*name* “retriable”, *value* True},
{*name* “max-retry-no”, *value* 5}}},
priority 1
}

where vPu is an instance of the class $\mathcal{EU}$ representing the *Validate purchase* activity.

2. Atomicity contracts are defined as follows:

- As it is specified in the BR-5 and BR-1, the execution units *Send publicity*, *Validate payment* and *Send tickets* execution units are associated by a strict atomicity contract. Recall that an atomicity contract is a composite contract that must have within its scope exception handling or atomicity contracts. Therefore, the scope of this contract contains the exception handling contracts of each execution unit specified in the business rules.

  $c1\ stAtC\ \{$
  $scope\ \{nvCsP,\ crCvP,\ unCsT\},$
  $priority\ 2$
  $\}$

  where $nvCsP$, $crCvP$, and $unCsT$ are exception handling contracts associated to execution units *Send publicity*, *Validate payment* and *Send tickets* respectively.

6.3.3 Implementing atomicity contracts in ROSE

Atomicity contracts are implemented as follows in Rose:

- The process for evaluating atomicity contracts is coded in the contract evaluator.
- There are an editor that reads XPDL coordinations and enables to developers to defines atomicity contracts using a graphical user interface (see Figure 6.5). In the editor, exception handling contracts are associated to execution units directly by means of a color codification, and atomicity contracts are associated to other contracts by means of a dialog box. As a result of associating atomicity contracts to an XPDL file, there is a XML file containing the definitions of the contracts (see Listing 6.2).

Listing 6.2: XML file for the contracts of the "purchase tickets application"

```
1  <?xml version="1.0" standalone="yes"?>
2  <ROSE>
3  <EXCEPTION-CONTRACTS>
4          <CONTRACT type="UNDOABLE"        name="unCgCI">Get_concert_information</CONTRACT>
5          <CONTRACT type="UNDOABLE"        name="unCvP">Validate_purchase</CONTRACT>
6          <CONTRACT type="NONVITAL"        name="nvCsP">Send_publicity</CONTRACT>
7          <CONTRACT type="UNDOABLE"        name="unCsT">Send_tickets</CONTRACT>
8          <CONTRACT type="COMPENSATABLE"   name="cpCpP">Process_purcahse</CONTRACT>
9          <CONTRACT type="CRITICAL"        name="crCvP">Validate_payment</CONTRACT>
10 </EXCEPTION-CONTRACTS>
11 <ATOMCITY-CONTRACTS>
12         <CONTRACT type="Strict" name="c1">
13                 <CN>crCvP</EU>
14                 <CN>nvCsP</EU>
15                 <CN>unCsT</EU>
```

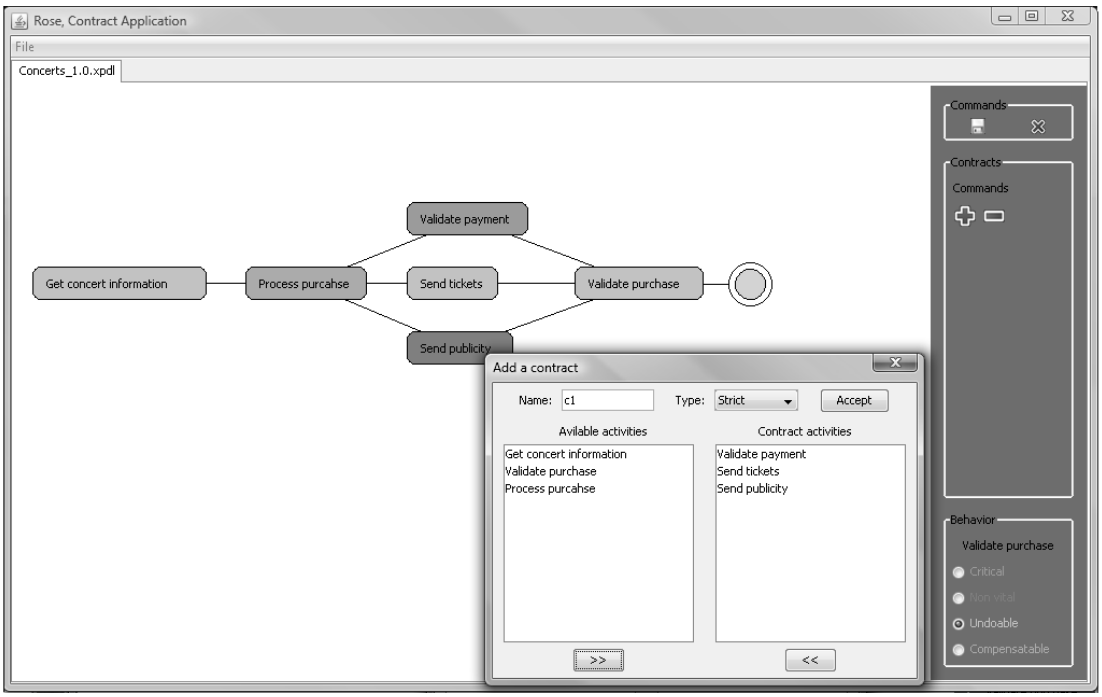

Figure 6.5: Contracts editor of ROSE

```
16              </CONTRACT>
17              ...
18 </ATOMICITY-CONTRACTS>
19 </ROSE>
```

6.3.4 Example of execution

In this section we show how the coordination engine enacts a coordination with atomicity requirements (i.e., failure and atomicity contracts). We use UML sequences diagrams to show the interactions between the components of ROSE for enacting this application and its contracts.

- First according to the control flow of the "purchase tickets application", the engine executes *Get concert information* (see Figure 1.1). Recall that within the execution of an execution unit the **Contracts hook** is executed twice: before and after the execution. The Figure 6.6 shows the messages exchanged among the modules of the engine. Assuming that the execution unit is executed successfully the order is a follows:

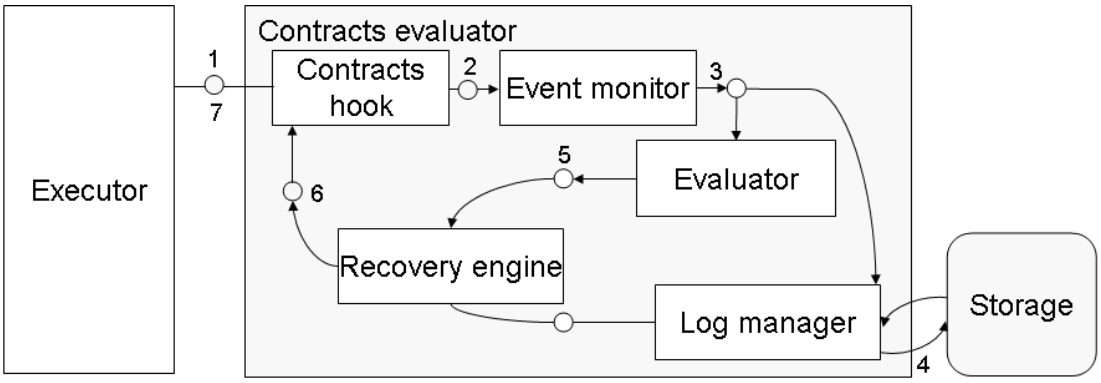

Figure 6.6: Interactions among components of ROSE for executing *Get concert information*

1. The **Executor** executes the **Contacts hook** (before and after the execution of the activity) sending all the information about the execution context (i.e., activity name, execution state of the activity, and project name).
2. The hook notifies the event to the **Event monitor**.
3. The **Event monitor** notifies an state change in the execution to the **Evaluator** and to the **Log manager**.
4. The **Log manager** stores in the **Storage** the state change.
5. The **Evaluator** evaluates the property of the exception contract with respect to the execution state and notifies an action to the **Recovery engine**. In this case there is no action to be taken.
6. The **Recovery engine** notifies an execution plan (e.g., no action) to the **Contracts hook**.
7. Finally, the **Contracts hook** communicates that there is no action to be executed to the **Executor**.

- Next, *Process purchase* is executed. We assume that there is no problem with its execution, therefore the interactions among the ROSE components is similar to the interactions when executing *Get concert information* (see Figure 6.6).
- After it is necessary to execute three activities for completing the purchase: *Validate payment*, *Send tickets* and *Send publicity* (see Figure 1.1). Note that these activities are executed in parallel by the executor, a service is contacted for validating the payment, other is in charge of sending the tickets and another is

contacted for sending the publicity. With respect to contracts, each activity has associated an exception management contract and such contracts are included within the scope of an atomicity contract. Let us analyze the interactions when executing the activity *Validate payment* (see Figure 6.7):

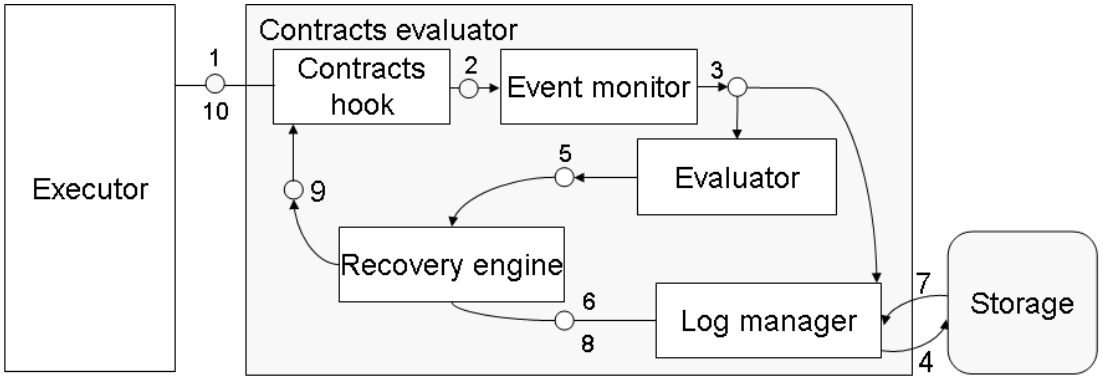

Figure 6.7: Interactions among components of ROSE for executing *Validate payment*

1. The **Executor** calls the hook (in this case we assume that it is executed after the execution of the activity) sending all the information about the execution context (i.e., activity name, execution state of the activity, and project name). Let us consider that the execution of the activity fails.
2. The hook notifies the event to the **Event monitor**.
3. The **Event monitor** notifies an state change in the execution to the **Evaluator** and to the **Log manager**.
4. The **Log manager** stores the state change within the log representing the execution history.
5. The **Evaluator** evaluates the properties with respect to the execution state and notifies possible actions to the **Recovery engine**. Note that there are two contracts that are being evaluated: the exception management contract associated to the activity and the atomicity contract associated to the exception management contract:
   - First, when the property of the exception management contract (i.e., the contract associated to *Validate payment*) is evaluated, an activity failure is signaled according to the rules associated to the critical contract. This state is propagated into the contract tree.

  - Next, the property linked to atomicity contract is evaluated. According to the rules of the strict atomicity contract backward recovery must be applied.

6. The **Recovery engine** requires the execution history to the **Log manager**.
7. The **Log manager** retrieves the execution history from the **Storage**.
8. The **Log manager** returns the execution history to the **Recovery engine**.
9. The **Recovery engine** notifies an execution plan (e.g., backward recovery) to the **Contracts hook**. Note that execution history is necessary to determine which activities must be compensated. For example, if the activity **Send tickets** has been committed it is necessary to undone its actions by executing a compensation.
10. Finally, the **Contracts hook** communicates the execution plan to the **Executor**.

- At this point execution finishes because a contract establish a recovery strategy to grant an atomic behavior. With the failure of *Validate payment*, the other activities are compensated if necessary and therefore no activity within the contract is executed.

6.4 Conclusion

This chapter introduces a proof of concept of how the COBA model can be implemented in an coordination engine for enacting reliable coordinations.

First, we present a general architecture of a coordination engine for enacting services coordinations with reliable requirements. In such an architecture, we introduce the requirements that a existing coordination engine must have to be enhanced with a contracts evaluator.

Next, we present details about ROSE, a coordination engine that extends Bonita. ROSE is an engine that enacts coordinations with atomicity requirements. The key element of ROSE is the contracts evaluator. Therefore we present details about the modules of the evaluator.

Finally, a running example is used for analyzing the evaluation process and the interactions among the components of the evaluator.

In such a way, we have shown in this chapter the feasibility of our contract model. It is show how it is possible to make the contracts evaluation at execution time without overcharging the execution.

Chapter 7

Conclusions

In this thesis we present COBA, a model for associating non-functional properties to a services coordination by means of contracts. In particular we focus on providing reliability properties to services coordination. Therefore, we show the contract evaluation strategies for verifying reliability properties at run time. As an experimental validation we implement a reliable services coordination execution engine called ROSE which is able to add exception handling and atomicity properties to a given services coordination.

Résumé: Ce chapitre conclue cette thèse, il énumère les contributions et les perspectives de ce travail. Les contributions de notre travail sont:

- Le modèle COBA pour représenter les aspects non fonctionnels d'une coordination des services à travers la notion de contrat.
- La validation de ce modèle en l'utilisant pour représenter les propriétés de fiabilité de la coordination des services. Dans cette validation nous avons défini des contrats de traitement d'exceptions de l'exécution d'unités d'exécution; des contrats d'atomicité associés à des ensembles d'unités d'exécution; des contrats de persistance de l'état d'exécution d'une unité d'exécution; et enfin, des contrats de gestion de l'état d'exécution des ensembles des unités d'exécution.
- Des stratégies d'évaluation des contrats.
- ROSE, un évaluateur de contrats COBA d'atomicité de coordination de services.

Les perspectives de notre travail portent sur :

- La spécification formelle du modèle qui permettra d'assurer des propriétés de terminaison et d'absence d'embrassement mortel des coordinations lorsque des contrats sont associés. Une première approche a été réalisée pour les contrats d'atomicité [PVSGB+08].
- La modélisation d'autres propriétés non fonctionnelles comme la sécurité [Vu08] et l'auto-adaptation [Tan09].
- La programmation des coordinations de services à base de contrats et la généralisation des stratégies d'évaluation de contrats [CIC09].
- La construction de mashups pour aider à l'intégration fiable de données produites par des services de données[1].

7.1 Contributions and main results

The contributions and results of our research work were published in different research forums (see [Por06a, PCVS+06, Por06b, PVSZM+06, PVSC+07a, HBPZM07, PVSC+07b, PVSC+08b, Por08a, PCZMHB08, PVSGB+08, PHEO08a, PHEO+08b, PHBVS+08, Por08b]). They can be summarized as follows:

- The COBA model for representing the non functional aspects of a services coordination through the notion of contract was proposed [Por06a, PVSC+07a, PVSC+08a]. The concepts of such a model are:
  - Execution unit: it represents the execution of a process (e.g., an activity of a services coordination).
  - Property: It represents a non functional aspect.
  - Rule: it specifies the reactions to be executed for enforcing a property under a given situation.
  - Contract: it represents the association of a property to an execution unit or to a set of contracts and the rules to be considered for the property.
- We present a proof of concept of the COBA model which addresses reliability for services coordination [PVSZM+06, PCVS+06, PHEO+08b, PHEO08a]. In particular we use the contract notion to define:

[1] Ce thème est abordé dans le cadre de la thèse de M. Othman-Abdallah, HADAS-LIG.

1. How to treat exceptions over the execution of execution units.
2. How to provide atomic behavior to sets of execution units.
3. How to treat the persistency guarantees of execution units.
4. How to handle the execution state of activities sets.

- We show the contract evaluation strategies for verifying and enforcing reliability properties [PVSC+08a, PHEO08a, PHEO+08b].
- We conduct an experiment for implementing a contract evaluator within a coordination engine. We show a general architecture with the basic requirements that a coordination engine must have in order to enact a coordination with reliable contracts [PVSC+07b, HBPZM07].

7.2 Perspectives

There are several aspects that we addressed in our research work that remain as opportunities areas:

- In [PCZMHB08] we sketched a proof of concept for providing persistency guarantees to services coordination. We propose to extend the coordination engine Xflow [pro08] with a contract evaluator. Xflow enacts coordination expressed as workflows. It is a JBoos based engine running using a Tomcat sever that enables to coordinate Web services. The contract evaluator was coded directly into the Xflow engine. We are currently testing our approach with several scenarios related to e-commerce context.
- Along with the separation of application logic and reliability aspects a verification problem arises (i.e., to validate that such requirements are sound for the target application). In order to conduct the verification of reliable coordinations we present in [PVSGB+08] an approach for verifying atomicity requirements in services coordination. A services coordination represents an execution order where execution is supposed to be free of problems (e.g., deadlocks and race condition). Such an execution order, is extended by atomicity requirements that deal with exceptional situations. Yet, contracts can introduce new states that should be verified in order prevent deadlocks and race conditions at execution time, and to determine possible termination states. The verification process cannot be done manually due to the number of combination of states. We propose

an approach for statically verifying contract based atomic services coordinations using a model checker. The contracts defining the atomicity properties are expressed using the B method [Abr96]. The application logic is expressed using CSP processes [LB03]. Therefore, the verification process is made by guiding the resulting B machine by the CSP process. We are extending our approach to persistency properties and automatizing the validation process.

- The COBA model can be used for representing the non functional aspects of a services coordination through the notion of contract. However, it was validated only with reliability properties. We are planning to analyze other properties that the execution of a services coordination must ensure with respect to observable requirements. In particular we are interested in aspects that where analyzed in our research team from different point of view to the one that we follows (i.e., separation of concerns) to adapt them to our contract based approach:
  - [Vu08] proposes a model for executing a services coordination with security properties. It proposes a model and associated strategies for ensuring security requirements at running time.
  - [Tan09] addresses the problem of providing adaptability to services coordination. It proposes an ontology based approach in order to change dynamically the execution of a given services coordination.
- Finally, we think that our approach can be used in different execution contexts, where non functional requirements are associated to restrictions of the environment:
  - In [PHEO08a] and [PHEO$^+$08b] we present an approach for building reliable mobile applications based on services oriented paradigm and the use of the COBA model. Contracts ensure, for example, transactional properties at execution time in the presence of exceptions and make applications aware of execution context (QoS).
  - In [PHBVS$^+$08] we present an approach for building secure and adaptable services based mashups using the COBA model. A mashup is an application that presents content available from different sources by reusing the contents provided by third parties (e.g Web pages, Web services). Although a mashup is usually built on the fly by users, we argue that there is a necessity for building reliable mashups by providing QoS properties to such

kind of applications in an easy and intuitive way. Furthermore, we propose an architecture to ensure QoS properties during the mashup execution.

Bibliography

[AAA+98] Gustavo Alonso, D. Agraval, A. El Abbadi, M. Kamath, R. Guntor, and C. Mohan. Advanced tansactions models in workflow contexts. Technical report, IBM, Research Division, 1998.

[Abr96] J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.

[ACKM04] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services - Concepts, Architectures and Applications*. Springer Verlag, first edition, 2004.

[AFH+99] Gustavo Alonso, Ulrich Fiedler, Claus Hagen, Amaia Lazcano, Heiko Schuldt, and N. Weiler. Wise: Business to business e-commerce. In *Research Issues on Data Engineering: Information Technology for Virtual Enterprises*, pages 132–139, 1999.

[BBC+05] Ruslan Bilorusets, Don Box, Luis Felipe Cabrera, Doug Davis, Donald Ferguson, Christopher Ferris, Tom Freund, Mary Ann Hondo, John Ibbotson, Lei Jin, Chris Kaler, David Langworthy, Amelia Lewis, Rodney Limprecht, Steve Lucco, Don Mullen, Anthony Nadalin, Mark Nottingham, David Orchard, Jamie Roots, Shivajee Samdarshi, John Shewchuk, and Tony Storey. Web services reliable messaging protocol (ws-reliablemessaging). Technical specification, BEA Systems, International Business Machines Corporation, Microsoft Corporation Inc., TIBCO Software Inc., February 2005.

[BCR05] Laura Bocchi, Paolo Ciancarini, and Davide Rossi. Transactional aspects in semantic based discovery of services. In Jean-Marie Jacquet and Gian Pietro Picco, editors, *COORDINATION*, volume 3454 of *Lecture Notes in Computer Science*, pages 283–297. Springer, 2005.

[BDO05] Alistair Barros, Marlon Dumas, and Phillipa Oaks. A critical overview of the web services choreography description language (ws-cdl). *BP-Trends*, March 2005.

[BGP05] Sami Bhiri, Claude Godart, and Olivier Perrin. Reliable web services composition using a transactional approach. In IEEE International, editor, *O. e-Technology, e-Commerce and e-Service*, volume 1 of *eee*, pages 15–21, March 2005.

[Bhi05] Sami Bhiri. *Approche transactionelle pour assurer des compositions fiables de services web*. PhD thesis, Université Henri Poincaré - Nancy 1, LORIA, Octobre 2005.

[BLSW04] Roger Barga, David Lomet, German Shegalov, and Gerhard Weikum. Recovery guarantees for internet applications. volume 4, pages 289–328, New York, NY, USA, 2004. ACM.

[CC98] Thierry Coupaye and Christine Collet. Semantics based implementation of flexible execution models for active database systems. In Mokrane Bouzeghoub, editor, *BDA*, pages 0–, 1998.

[CCC+04] William Cox, Felipe Cabrera, George Copeland, Tom Freund, Johannes Klein, Tony Storey, and Satish Thatte. Web services transaction (ws-transaction). Technical specification, BEA Systems, International Business Machines Corporation, Microsoft Corporation, Inc, November 2004.

[CCF+04] Felipe Cabrera, George Copeland, Tom Freund, Johannes Klein, David Langworthy, David Orchard, John Shewchuk, and Tony Storey. Web services coordination (ws-coordination). Technical specification, BEA Systems, International Business Machines Corporation, Microsoft Corporation, Inc, November 2004.

[CIC09] Securely coordinating services using contracts. In *Proceedings of the 8th Mexican International Conference on Computer Science (ENC 2009)*, Mexico City, Mexico, 2009. IEEE.

[CMRW07] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. Web services description language (wsdl) version 2.0 part 1: Core language. Report, World Wide Web Consortium, W3C, June 2007.

[Con07a] OW2 Consortium. Bonita, April 2007. http://forge.objectweb.org/projects/bonita.

[Con07b] OW2 Consortium. Jonas, April 2007. http://forge.objectweb.org/projects/jonas.

[CR90] Panayiotis K. Chrysanthis and Krithi Ramamritham. Acta: a framework for specifying and reasoning about transaction structure and behavior. In *SIGMOD '90: Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 194–203, New York, NY, USA, 1990. ACM Press.

[CSDS03] Fabio Casati, Eric Shan, Umeshwar Dayal, and Ming-Chien Shan. Business-oriented management of web services. *Commun. ACM*, 46(10):55–60, 2003.

[DA82] Claude Delobel and Michel Adiba. *Bases de données et systèmes relationnels*. Dunod, Informatique, 1982.

[DFDB05] Helga Duarte, Marie-Christine Fauvet, Marlon Dumas, and Boualem Benatallah. Vers un modèle de composition de services web avec propiétés transactionelles. *Ingénierie des systèmes d'information*, 10(3):9–28, 2005.

[EGLT76] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.

[ELLR90] Ahmed K. Elmagarmid, Y. Leu, W. Litwin, and Marek Rusinkiewicz. A multidatabase transaction model for interbase. In *Proceedings of the sixteenth international conference on Very large databases*, pages 507–518, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[Elm92] A. K. Elmagarmid. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, 1992.

[Erl05] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall, October 2005.

[FC08] FMC-Consortium. Fundamental modeling concepts, August 2008. http://www.fmc-modeling.org/.

[Fur04] Peter Furniss. Business transaction protocol. Technical specification, OASIS, November 2004.

[GA08] Alexander Gelbukh and Michel Adiba, editors. *Ninth Mexican International Conference on Computer Science*, IEEE Computer Society. IEEE, 2008.

[GM83] Héctor García-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Trans. Database Syst.*, 8(2):186–213, 1983.

[GMGK+91] Héctor García-Molina, Dieter Gawlick, Johannes Klein, Karl Kleissner, and Kenneth Salem. Modeling long-running activities as nested sagas. *Data Engineering*, 14(1):14–18, 1991.

[GMS87] Héctor García-Molina and Kenneth Salem. Sagas. In ACM, editor, *9th Int. Conf. on Management of Data, San Francisco, California, USA*, pages 249–259, 1987.

[GMUW00] Héctor García-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database system implementation.* Prentice Hall, 2000.

[Gra81] Jim Gray. The transaction concept: Virtues and limitations (invited paper). In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 144–154. IEEE Computer Society, 1981.

[HA00] Claus Hagen and Gustavo Alonso. Exception handling in workflow management systems. *IEEE Transactions on Software Engineering*, 26(10):943–958, October 2000.

[HBPZM07] Víctor Hernández-Baruch, Alberto Portilla, , and José-Luis Zechinelli-Martini. Rose: A transactional services coordination engine. In *8th Mexican International Conference on Computer Science (ENC 2007)*, pages 122–130. ENC-SMCC, IEEE, sep 2007.

[HW06] Peter Hrastnik and Werner Winiwarter. Twso transactional web service orchestrations. *Journal of Digital Information Management*, 4(1):–, 2006.

[LASS00] Amaia Lazcano, Gustavo Alonso, Heiko Schuldt, and Christoph Schuler. The WISE approach to electronic commerce. *International Journal of Computer Systems Science and Engineering*, 15(5), 2000.

[LB03] Michael Leuschel and Michael Butler. ProB: A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.

[Lom05] David Lomet. Robust web services via interaction contracts. In *Technologies for E-Services*, pages 1–14. LNCS, 2005.

[LR97] Frank Leymann and Dieter Roller. Workflow-based applications. *IBM Systems Journal*, 36(1), 1997.

[LW98] David Lomet and Gerhard Weikum. Efficient transparent application recovery in client-server information systems. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 460–471. ACM, 1998.

[LW03] Mark Little and Jim Webber. Introducing ws-coordination. *Web Services Journal*, 3(5), April 2003.

[NFG+05] Surya Nepal, Alan Fekete, Paul Greenfield, Julian Jang, Dean Kuo, and Tony Shi. A service-oriented workflow language for robust interacting applications. In Robert Meersman, Zahir Tari, Mohand-Said Hacid, John Mylopoulos, Barbara Pernici, Özalp Babaoglu, Hans-Arno Jacobsen, Joseph P. Loyall, Michael Kifer, and Stefano Spaccapietra, editors, *OTM Conferences (1)*, volume 3760 of *Lecture Notes in Computer Science*, pages 40–58. Springer, 2005.

[OV99] M. Tamer Ozsu and Patrick Valduriez. *Principles of distributed database systems*. Prentice Hall, second edition, 1999.

[PBM02] Paulo F. Pires, Mario R. F. Benevides, and Marta Mattoso. Building reliable web services compositions. In Akmal B. Chaudhri, Mario Jeckle, Erhard Rahm, and Rainer Unland, editors, *Web, Web-Services, and Database Systems*, volume 2593 of *Lecture Notes in Computer Science*, pages 59–72. Springer, 2002.

[PCVS+06] Alberto Portilla, Christine Collet, Genoveva Vargas-Solar, José-Luis Zechinelli-Martini, and Luciano García-Bañuelos. Towards a transactional services coordination model. In Bipin C. Desai and S.K. Gupta, editors, *IDEAS*, pages 319–320. IEEE Computer Society, 2006.

[PCZMHB08] Alberto Portilla, Christine Collet, José-Luis Zechinelli-Martini, and Víctor Hernández-Baruch. Providing persistency guarantees to services coordination. In Gelbukh and Adiba [GA08], pages 169–178.

[PHBVS+08] Alberto Portilla, Víctor Hernández-Baruch, Genoveva Vargas-Solar, José-Luis Zechinelli-Martini, and Christine Collet. Building reliable services based mashups. In José-Manuel López-Cobo, Antonio Vallecillo, and Antonio Ruiz-Cortés, editors, *JSWEB 2008*, volume 1 of *JSWEB*, pages 151–163. Jornadas Científico-Técnicas en Servicios Web y SOA, 2008.

[PHEO08a] Alberto Portilla, Tan Hanh, and Javier-Alfonso Espinosa-Oviedo. Building reliable mobile services based applications. In *ICDE Workshops*, pages 121–128. IEEE Computer Society, 2008.

[PHEO+08b] Alberto Portilla, Tan Hanh, Javier-Alfonso Espinosa-Oviedo, Christine Collet, and Genoveva Vargas-Solar. Construire des applications fiables a base de services mobiles. In Emmanuel Dubois and Jean-Marc Pierson, editors, *UbiMob*, volume 277 of *ACM International Conference Proceeding Series*, pages 57–64. ACM, may 2008.

[Por06a] Alberto Portilla. Providing transactional behavior to services coordination. In Junho Shim and Fabio Casati, editors, *VLDB 2006 PhD. Workshop*, volume 170 of *CEUR Workshop proceedings*, pages –. CEUR, 2006.

[Por06b] Alberto Portilla. Services coordination with transactional properties. Master's thesis, UDLA-INPG, México, Grenoble, June 2006.

[Por08a] Alberto Portilla. Providing reliability to services coordination. In Gabriel Lopéz-Moreno and J. Antonio García-Macías, editors, *Avances en las ciencias de la computación, ENC'08*, ENC, pages 136–137. SMCC, 2008.

[Por08b] Alberto Portilla. Reliable services coordination for mashing-up systems. In *Memorias del Primer Encuentro de Estudiantes de Doctorado en Ciencias de la Computacion en México*, pages 136–137. CINVESTAV-IPN, september 2008.

[pro08] OpenSource project. Xflow, June 2008. http://xflow.sourceforge.net/.

[PVSC+07a] Alberto Portilla, Genoveva Vargas-Solar, Christine Collet, José-Luis Zechinelli-Martini, and Luciano García-Bañuelos. A flexible model for providing transactional behavior to service cordination in an orthogonal way. In Joaquim Filipe and José A. Moinhos Cordeiro, editors, *WEBIST 2007*, pages 104–111. INSTICC Press, 2007.

[PVSC+07b] Alberto Portilla, Genoveva Vargas-Solar, Christine Collet, José-Luis Zechinelli-Martini, Luciano García-Bañuelos, and Víctor Hernández-Baruch. Rose: a transactional services coordination engine. In *Proceedings of the 23emes Journees Bases de Données Avancees (BDA 2007), Marseille,France*. BDA, October 2007.

[PVSC+08a] Alberto Portilla, Genoveva Vargas-Solar, Christine Collet, José-Luis Zechinelli-Martini, and Luciano García-Bañuelos. Contract based behavior model for services coordination. In Joaquim Filipe and José A. Moinhos Cordeiro, editors, *WEBIST (Selected Papers)*, volume 8 of *Lecture Notes in Business Information Processing*, pages 109–123. Springer, 2008.

[PVSC+08b] Alberto Portilla, Genoveva Vargas-Solar, Christine Collet, José-Luis Zechinelli-Martini, and Luciano García-Bañuelos. Contract based behavior model for services coordination. In Joaquim Filipe and J. Cordeiro, editors, *Web Information Systems and Technologies Third International Conference, Revised Selected Papers WEBIST 2007*, volume 8 of *Lecture Notes in Business Information Processing*, pages –. Springer, 2008.

[PVSGB+08] Alberto Portilla, Genoveva Vargas-Solar, Luciano García-Bañuelos, Christine Collet, and José-Luis Zechinelli-Martini. Verifying atomicty requirements of services coordination using b. In Gelbukh and Adiba [GA08], pages 238–248.

[PVSZM+06] Alberto Portilla, Genoveva Vargas-Solar, José-Luis Zechinelli-Martini, Christine Collet, and Luciano García-Bañuelos. A survey for analyzing transactional behavior in service based applications. In *7th Mexican International Conference on Computer Science (ENC 2006)*, pages 116–126. ENC-SMCC, IEEE, sep 2006.

[SABS02] H. Schuldt, G. Alonso, C. Beeri, and H.-J. Schek. Atomicity and Isolation for Transactional Processes. *ACM Transactions on Database Systems (TODS)*, 27(1):63–116, March 2002.

[SH05] Munindar P. Singh and Michael N. Huhns. *Service-Oriented Computing*. John Wiley and Sons, first edition, 2005.

[Tan09] Hanh Tan. *Coordination adaptative de services à base de contrats*. PhD thesis, Grenoble Institut of Technology, LIG, Juin 2009.

[TMW+04] Stefan Tai, Thomas Mikalsen, Eric Wohlstadter, Nirmit Desai, and Isabelle Rouvellou. Transaction policies for service-oriented computing. *Data Knowl. Eng.*, 51(1):59–79, 2004.

[vdAvH04] Wil M. P. van der Aalst and Kees van Hee. *Workflow Management, Models, Methods, and Systems*. The MIT Press, first edition, 2004.

[Vu08] Thi-Huong-Giang Vu. *Composition sécurisé de services*. PhD thesis, Grenoble INP, nov 2008.

[VV04] K. Vidyasankar and Gottfried Vossen. A multi-level model for web service composition. In *ICWS*, pages 462–. IEEE Computer Society, 2004.

[WR92] Helmut Wachter and Andreas Reuter. The contract model. In Ahmed K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, chapter 7, pages 219–263. Morgan Kaufmann Publishers, 1992.

[WV98] Gerhard Weikum and Gottfried Vossen. *Transactional information systems, theory, algorithms, and the practice of concurrency control and recovery*. Morgan Kaufmann Publishers, 1998.

[ZHMS06] Xianan Zhang, Matti A. Hiltunen, Keith Marzullo, and Richard D. Schlichting. Customizable service state durability for service oriented architectures. *edcc*, 0:119–128, 2006.

[ZNBB94] Aidong Zhang, Marian Nodine, Bharat Bhargava, and Omran Bukhres. Ensuring relaxed atomicity for flexible transactions in multidatabase systems. In *SIGMOD International Conference on Management of Data*, pages 67–78, 1994.

Appendix A

Reliability contracts

This Section presents the reliability contract types definitions using the COBA model.

A.1 Exception contract

The exception contract type specializes the class *SimpleContract* (see Definition 9) as follows:

SubClass (Exception contract)

class exceptionContract : *SimpleContract* {
property exceptionProperty,
}

A.1.1 Exception property

The exception property specializes the class *Property* (see Definition 4):

SubClass (Exception property)

class exceptionProperty : *Property* {
values { {"compensable", `Boolean`},
{"side-effects", `Boolean`},
{"retriable", `Boolean`},
{"max-retry-no", `Integer`}}
}

A.1.2 Recovery Rules

A recovery rule of an exception contract type defines at which moment an action must be executed. A recovery rule specializes the class *Rule* (see Definition 5):

- *recRule*1 specifies that, if the execution of a given execution unit fails, then it is necessary to notify the failure:

  class $recRule1$: *Rule* {
  $on\ failEv$,
  $do\ notifyFailure$
  }

- *recRule*2 specifies that, if it is requested the compensation of a given execution unit, then an exception is launched:

  class $recRule2$: *Rule* {
  $on\ compReqEv$,
  $do\ notifyExc$
  }

- *recRule*3 specifies that, if the execution of a given execution unit fails, then the execution can proceed anyway:

  class $recRule3$: *Rule* {
  $on\ failEv$,
  $do\ continue$
  }

- *recRule*4 specifies that, if it is required the compensation of a given execution unit, then the execution unit does not requires compensation:

  class $recRule4$: *Rule* {
  $on\ compReqEv$,
  $do\ continue$
  }

- *recRule*5 specifies that, if the execution of a given execution unit fails, then the execution unit is retried:

class $recRule5$: *Rule* {
on $failEv$,
do $retry$
}

- $recRule6$ specifies that, if it is requested the compensation of a given execution unit, then its compensation is executed:

class $recRule6$: *Rule* {
on $compReqEv$,
do $compensate$
}

- $recRule7$ specifies that, if the execution of a given execution unit fails, then the execution unit is retried if possible:

class $recRule7$: *Rule* {
on $failEv$,
do $retry$
}

The definitions of the class events $failEv$ and $compReqEv$, and the class reactions $notifyFailure$, $notifyExc$, $continue$, $retry$ and $compensate$ are presented in the next sections.

Events

Exception contract type reactions are triggered by two events:

- The execution failure of an execution unit. It specializes the class *Event* (see Definition 6):

class $failEv$: *Event* {
delta {{$euName$ `String`}}
}

- The necessity for compensating an execution unit. It specializes the class *Event* (see Definition 6):

class compReqEv: *Event*{
delta {{*euName* `String`}}
}

Reactions

Reactions of exception contract type specializes the class *Reaction* (see Definition 7):

- *continue* reaction indicates that execution can proceed:

class continue: *Reaction*{
input {{*euName* `String`}},
output {{*rResult* `Boolean`}}
}

- *retry* reaction indicates that the execution of a given execution unit must be retried:

class retry: *Reaction*{
input {{*euName* `String`}},
output {{*rResult* `Boolean`}}
}

- *compensate* indicates that the compensation action of a given execution unit must be done:

class compensate: *Reaction* {
input {{*euName* `String`}},
output {{*rResult* `Boolean`}}
}

- $notifyExc$ indicates that an exception must be launched:

class $notifyExc$: *Reaction* {
input {{*euName* `String`}},
output {{*rResult* `Boolean`}}
}

- $notifyFailure$ indicates that the contract cannot be granted (i.e., it has failed):

$$\begin{array}{l} \textit{class } notifyFailure\text{: } \textit{Reaction}\ \{ \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad input\ \{\{euName\ \texttt{String}\}\}, \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad output\ \{\{rResult\ \texttt{Boolean}\}\} \\ \qquad\qquad\qquad\qquad\qquad\qquad \} \end{array}$$

A.1.3 Exception contracts

There are four exception contracts that specializes the class *exceptionContrat*.

- A critical contract can be associated to an execution unit that cannot be retried in case of failure. When committed the execution unit cannot be undone.

  SubClass (Critical contract)

$$\begin{array}{l} \textit{class } crContract\text{: } \textit{exceptionContract}\{ \\ \qquad\qquad property\ \{values\ \{\{name\ \text{“compensable”}, value\ \text{False}\}, \\ \qquad\qquad\qquad\qquad\qquad\qquad \{name\ \text{“side-effects”}, value\ \text{False}\}, \\ \qquad\qquad\qquad\qquad\qquad\qquad \{name\ \text{“retriable”}, value\ \text{False}\}, \\ \qquad\qquad\qquad\qquad\qquad\qquad \{name\ \text{“max-retry-no”}, value\ 0\}\}\}, \\ \qquad\qquad rules\ \{r1\ recRule1, r2\ recRule2\} \\ \qquad\qquad\qquad\qquad \} \end{array}$$

- A non vital contract can be associated to an execution unit that does not need to be compensated if it has to be undone after having committed.

  SubClass (Non vital contract)

$$\begin{array}{l} \textit{class } nvContract : \textit{exceptionContract}\{ \\ \qquad\qquad property\ \{values\ \{\{name\ \text{“compensable”}, value\ \text{Null}\}, \\ \qquad\qquad\qquad\qquad\qquad\qquad \{name\ \text{“side-effects”}, value\ \text{False}\}, \\ \qquad\qquad\qquad\qquad\qquad\qquad \{name\ \text{“retriable”}, value\ \text{Null}\}, \\ \qquad\qquad\qquad\qquad\qquad\qquad \{name\ \text{“max-retry-no”}, value\ 0\}\}\}, \\ \qquad\qquad rules\ \{r1\ recRule3, r2\ recRule4\} \\ \qquad\qquad\qquad\qquad \} \end{array}$$

- An undoable contract can be associated to an execution unit that can be undone by a compensating execution unit without causing side-effects once it has committed. An undoable execution unit eventually commits after retrying it several times.

  SubClass (Undoable contract)

  class $unContract$: *exceptionContract*{
  $property$ {$values$ {{$name$ "compensable", $value$ True},
  {$name$ "side-effects", $value$ False},
  {$name$ "retriable", $value$ True},
  {$name$ "max-retry-no", $value$ `Integer`}}},
  $rules$ {$r1$ $recRule5, r2$ $recRule6$}
  }

- A compensatable contract can be associated to an execution unit that can be undone by a compensating execution unit with associated side-effects once it has committed. A compensatable execution unit can be retried a restricted number of times. For retrying a compensatable execution unit other aspects must be considered such as application restrictions, monetary costs, or temporal constraints.

  SubClass (Compensatable contract)

  class $cpContract$: *exceptionContract*{
  $property$ {$values$ {{$name$ "compensable", $value$ True},
  {$name$ "side-effects", $value$ True},
  {$name$ "retriable", $value$ True},
  {$name$ "max-retry-no", $value$ `Integer`}}},
  $rules$ {$r1$ $recRule6, r2$ $recRule7$}
  }

A.2 Atomicity contract

The state management contract type specializes the class *SimpleContract* (see Definition 9) as follows:

SubClass (State management contract)

class stateMContract : *SimpleContract* {
property stateMProperty
}

A.2.1 State management property

The state management property describes the possible values that an execution unit can have for managing the persistency of its state. Is specializes the class *Property* (see Definition 4) as follows:

SubClass (Property)

class stateMProperty : *Property* {
values { { "state-Verifiability", `Boolean`},
{ "idempotency", `Boolean`},
{ "outcome-Assumption", {"committed",
"failed",
"presumed-nothing"}}}
}

A.2.2 Atomicity rules

An atomicity rule defines at which moment it is necessary to take actions for ensuring a given atomic behavior. An atomicity rule specializes the class *Rule* (see Definition 5):

- $acRule1$ specifies that, if a contract within the scope of a contract fails, then the contract fails and backward recovery is applied:

  class $acRule1$: *Rule* {
  $on\ contractFailure$,
  $do\ backwardRecovery \wedge notifyContractFailure$
  }

- $acRule2$ specifies that, if a contract within the scope of a contract fails and there is another alternative execution path, then forward recovery is applied:

class $acRule2$: *Rule* {
on $contractFailure$ where $existPath(this.$*atomicityContract*$)$,
do $forwardRecovery$
}

- $acRule3$ specifies that, if a contract within the scope of a contract fails and there is not another alternative execution path, then the contract fails and backward recovery is applied:

class $acRule3$: *Rule* {
on $contractFailure$ where $\neg existPath(this.$*atomicityContract*$)$,
do $backwardRecovery$
}

- $acRule4$ specifies that, if a contract fails, then the contract fails and an exception is launched:

class $acRule4$: *Rule* {
on $contractFailure$,
do $notifyExc$
}

The function $existPath$ returns true if the contract is an atomicity contract and there are at least an alternative contract not tried:

(`Boolean`) $existPath(input$: *atomicityContract*$)$
where:
`dom`$(existPath)$=$\{output$:`Boolean` $\mid \exists\ c \in input.scope, \neg failedContract(c)\}$

Events

The reactions of the atomicity contracts are triggered by the fact that a contract is considered as failed:

class $contractFailure$: *Event* {
$delta$ {{$cnName$ `String`}}
}

Reactions

Reactions of atomicity contracts specializes the class *Reaction*.

- $notifyContractFailure$ launches an event notifying the failure of a contract:

  class $notifyContractFailure$: *Reaction* {
  $input$ {{$cnName$ `String`}},
  $output$ {{$rResult$ `Boolean`}}
  }

- $notifyExc$ indicates that an exception must be launched because an exceptional situation happened during the execution of a contract:

  class $notifyExc$: *Reaction* {
  $input$ {{$cnName$ `String`}},
  $output$ {{$rResult$ `Boolean`}}
  }

- *backwardRecovery* indicates that committed contracts within an scope are undone, until the whole contract is compensated:

  class *backwardRecovery*: *Reaction* {
  $input$ {{$cnName$ `String`}},
  $output$ {{$rResult$ `Boolean`}}
  }

- $forwardRecovery$ indicates that a forward recovery strategy must be followed for recovering the execution. It combines backward recovery and forward execution. First, backward recovery is applied until compensate the failed contract. Next, forward execution with a different contract is applied. It is defined as follows:

  class $forwardRecovery$: *Reaction* {
  $input$ {{$cnName$ `String`}},
  $output$ {{$rResult$ `Boolean`}}
  }

A.2.3 Atomicity contracts

An atomicity contract class is specialized by three contracts subtypes: strict atomicity contract, alternative atomicity contract and exception atomicity contract.

- A strict atomicity contract specifies that the contracts within the scope of a contract conform the all or nothing execution requirement. All contracts or no contract at all within the scope of the contract are executed:

  SubClass (Strict contract)

  class $stAtC$: *atomicityContract*{
  $property$ {$values$ {$name$ "atomicityType", $value$ "Strict"}},
  $rules$ {$r1$ $acRule1$}
  }

- An alternative atomicity contract specifies that the contract commits if one of the contracts within its scope commits. Therefore, the contracts within its scope represent alternative execution paths. An alternative atomicity contract is defined by an instance of *atomicityContract* as follows:

  SubClass (Alternative contract)

  class $alAtC$: *atomicityContract* {
  $property$ {$values$ {$name$ "atomicityType", $value$ "Alternative"}},
  $rules$ {$r1$ $acRule2, r2$ $acRule3$}
  }

- An exception atomicity contract specifies that not all the contracts within the scope of the contract have to execute but when one of them does not execute an exception is launched. An exception atomicity contract is defined by an instance of *atomicityContract* as follows:

  SubClass (Exception contract)

  class $exAtC$: *atomicityContract* {
  $property$ {$values$ {$name$ "atomicityType", $value$ "Exception"}},
  $rules$ {$r1$ $acRule4$}
  }

A.3 State management contract

The state management contract type specializes the class *SimpleContract* (see Definition 9) as follows:

SubClass (State management contract)

class stateMContract : *SimpleContract* {
property stateMProperty
}

A.3.1 State management property

The state management property specializes the class *Property* (see Definition 4) as follows:

SubClass (Property)

class stateMProperty : *Property* {
values { {*name* "state-Verifiability", *value* `Boolean`},
{*name* "idempotency", *value* `Boolean`},
{*name* "outcome-Assumption", *value* {"committed",
"failed",
"presumed-nothing"}}}
}

A.3.2 State management rules

A rule of a state management contract determines how to make persistent the execution state of a given execution unit. A rule specializes the class *Rule*:

- *smcRule*1 specifies that, it is necessary to store in the log the state of an execution unit at the beginning of its execution:

*class smcRule*1: *Rule* {
on euStarted,
do writeEuState
}

- *smcRule*2 specifies that, it is necessary to store in the log the state of the execution unit at the end of its execution:

  class *smcRule*2: *Rule* {
  on $euTerminated$,
  do $writeEuState$
  }

- *smcRule*3 specifies that during recovery, if the beginning and the end of the execution of the execution unit are stored in the log, then the execution unit was committed:

  class *smcRule*3: *Rule* {
  on $recoverEuState$ where $begining(this.eu) \in log \wedge$
  $end(this.eu) \in log$,
  do $commitEu$
  }

- *smcRule*4 specifies that during recovery, if the beginning of the execution of the execution unit is stored in the log, then an exception must be launched because it was executed and its result is unknown:

  class *smcRule*4: *Rule* {
  on $recoverEuState$ where $begining(this.eu) \in log$,
  do $notifyExc$
  }

- *smcRule*5 specifies that during recovery, if the beginning of the execution of the execution unit is stored in the log, then its state is the outcome assumption:

  class *smcRule*5: *Rule* {
  on $recoverEuState$ where $begining(this.eu) \in log$,
  do $assumeEuState$
  }

- *smcRule*6 specifies that during recovery, if the end of the execution of the execution unit is stored in the log, then it was committed:

class $smcRule6$: *Rule* {
on $recoverEuState$ where $end(this.eu) \in log$,
do $commitEu$
}

- $smcRule7$ specifies that during recovery, if the end of the execution of the execution unit is not stored in the log, then it can be re-executed:

class $smcRule7$: *Rule* {
on $recoverEuState$ where $begining(this.eu) \in log \wedge$
$end(this.eu) \notin log$,
do $retry$
}

- $smcRule_8$ specifies that during recovery, if the beginning of the execution of the execution unit is stored in the log, then the state of the execution unit must be queried:

class $smcRule8$: *Rule* {
on $recoverEuState$ where $begining(this.eu) \in log$,
do $queryEuState$
}

Events

State management contract reactions are triggered by four events:

- The execution failure of an execution unit (see event *failEv* in Section A.1.2).
- The necessity of recovering the execution state of an execution unit from the log:

class $recoverEuState$: *Event* {
delta {{$euName$ `String`}}
}

- The beginning of the execution of an execution unit (i.e., the execution unit has been started):

```
class euStarted: Event {
                    delta {{euName String}}
                }
```

- The end of the execution of an execution unit (i.e., the execution unit has been terminated):

```
class euTerminated: Event{
                    delta {{euName String}}
                }
```

Reactions

Reactions of state management contracts specializes the class *Reaction* (see Definition 7):

- *retry* reaction indicates that the execution of the execution unit must be retried (see Section A.1.2).
- *notifyExc* indicates that an exception must be launched (see Section A.1.2).
- *writeEuState* reaction indicates that the execution state of the execution unit must be stored in the log:

```
class writeEuState: Reaction {
                    input {{euName String}},
                    output {{rResult Boolean}}
                }
```

- *restoreEuState* reaction indicates that the execution state of the execution unit must be restored from the log:

```
class restoreEuState: Reaction{
                    input {{euName String}},
                    output {{rResult Boolean}}
                }
```

- *queryEuState* reaction indicates that the execution state of the execution unit must be queried:

class queryEuState: *Reaction* {
input {{*euName* `String`}},
output {{*rResult* *State*}}
}

- *assumeEuState* indicates that the execution state of the execution unit must be assumed by using its outcome assumption:

class assumeEuState: *Reaction*{
input {{*euName* `String`}},
output {{*rResult* `list`(*State*)}}
}

- *committEu* annotates in the execution state of the execution unit that it was committed:

class commitEu: *Reaction* {
input {{*euName* `String`}},
output {{*rResult* `Boolean`}}
}

A.3.3 State management contract subtypes

There are four state management contract subtypes that specializes the class *stateMContrat*:

- A non persistent state management contract can be associated to an execution unit that does not have any persistency property. Therefore, it is necessary to store in the log the beginning and the end of its execution. During recovery a committed execution unit in the log is signaled by both the beginning and the end of its execution.

SubClass (Non persistent contract)

class nonPContract: *stateMContract* {
property {*values* {{name "state-Verifiability", *value* False},
{*name* "idempotency", *value* False},
{*name* "outcome-Assumption",

value “presumed-nothing”}}},
rules {*r*1 *smcRule*1, *r*2 *smcRule*2,
*r*3 *smcRule*3, *r*4 $smcRule_4$}
}

- A presumable contract can be associated to an execution unit that has an outcome assumption. Therefore, it is only necessary to store in the log the beginning of its execution. During recovery an executed execution unit is signaled in the log by the beginning of its execution. Next, its execution state can be known by using its outcome assumption.

  SubClass (Presumable contract)

  class presumContract: *stateMContract* {
  property {*values* {{name “state-Verifiability”, *value* False},
  {*name* “idempotency”, *value* False},
  {*name* “outcome-Assumption”,
  value {“committed”, “failed”}}}},
  rules {*r*1 *smcRule*1, *r*2 *smcRule*5}
  }

- An idempotent state management contract can be associated to an execution unit that is idempotent. Therefore it is only necessary to store in the log the end of its execution. During recovery a committed execution unit in the log is signaled by the end of its execution other wise it must be re-executed.

  SubClass (Idempotent contract)

  class idempContract: *stateMContract* {
  property {*values* {{name “state-Verifiability”, *value* False},
  {*name* “idempotency”, *value* True},
  {*name* “outcome-Assumption”,
  value {“committed”, “failed”,
  “presumed-nothing”}}}
  rules {*r*1 *smcRule*2, *r*2 *smcRule*6,
  *r*3 *smcRule*7}
  }

- A verifiable state management contract can be associated to an execution unit that can be queried about its execution state. Therefore it is only necessary to store in the log the beginning of its execution. During recovery an executed execution unit in the log is signaled by the beginning of its execution. Next, its execution state can be queried.

  SubClass (Verifiable contract)

  class $verifContract$: *stateMContract* {
  $property$ {$values$ {{name "state-Verifiability", $value$ True},
  {$name$ "idempotency", $value$ `Boolean`},
  {$name$ "outcome-Assumption",
  $value$ {"committed", "failed",
  "presumed-nothing"}}}
  $rules$ {$r1$ $smcRule1, r2$ $smcRule8$}
  }

A.4 Persistency guarantees contract

The persistency guarantees contract type specializes the class *CompositeContract* (see Definition 10) as follows:

SubClass (Persistency guarantees contract)

class persistGContract : *CompositeContract* {
scope `set`(*stateMContract* ∪ *persistGContract*),
$property$ *persistencyProperty*,
}

A.4.1 Persistency guarantees property

The persistency property specializes the class *Property* (see Definition 4) as follows:

SubClass (Persistency property)

class persistencyProperty : *Property*{
$values$ {{$name$"persistencyType",
$value$ {"bestEffort", "guaranteed"}}}
}

A.4.2 Persistency guarantees rules

A persistency guarantees rule specifies where to store the execution state and how to do the recovery process using it. A persistency guarantees rule specializes the class *Rule* (see Definition 5):

- *pgcRule*1 specifies that, if there is a change in the execution state, then set writes to the cached log:

  class *pgcRule*1: *Rule* {
  on *exStateChg*,
  do *setWrToCachedLog*
  }

- *pgcRule*2 specifies that, a coordination failure begins a crash recovery process:

  class *pgcRule*2: *Rule* {
  on *syFailure*,
  do *beginCoRecovery*
  }

- *pgcRule*3 specifies that, if there is a change in the execution state, then set writes to the stable log:

  class *pgcRule*2: *Rule* {
  on *exStateChg*,
  do *setWrToStableLog*
  }

Events

The reactions of a persistency guarantee contract are triggered by two events:

- *exStateChg* represents the fact that there is a change in the execution state of an execution unit. It specializes the class *Event* (see Definition 6):

  class *exStateChg*: *Event* {
  delta {{*euName* `String`}}
  }

- *syFailure* represents the fact that the coordination execution has failed by a system failure and a recovery process must be started. It specializes the class *Event* (see Definition 6):

```
class syFailure: Event {
                    delta {{coName String}}
                }
```

Reactions

Reactions of persistency guarantees contracts specializes the class *Reaction*:

- *setWrToCachedLog* represents the fact that all results of write operations must be stored in the cached log until a forced write happens:

```
class setWrToCachedLog: Reaction {
                            input {},
                            output {{rResult Boolean}}
                        }
```

- *setWrToStableLog* represents the fact that all results of write operations must be stored in the stable log:

```
class setWrToStableLog: Reaction {
                            input {},
                            output {{rResult Boolean}}
                        }
```

- *beginCoRecovery* represents the fact that the coordination execution has failed and a recovery process was started. Therefore, an event of type *recoverEuState* is generated for all execution units stored in the log:

```
class beginCoRecovery: Reaction {
                            input {{coName String}},
                            output {{rResult Boolean}}
                        }
```

A.4.3 Persistency guarantees contract subtypes

There are two persistency guarantees contract subtypes that specializes the class *persistGContract*:

- A best effort contract specifies that, the changes in the execution state are stored in a cached log, and in case of a system failure a recovery process of the execution history must be started::

  SubClass (Best effort contract)

  class $bePGC$: *persistGContract* {
  $property$ {$values$ {$name$ "type", $value$ "bestEffort"}},
  $rules$ {$r1$ $pgcRule1, r2$ $pgcRule2$ }
  }

- A guaranteed persistency contract specifies that, the changes in the execution state are stored in a stable log, and in case of a system failure a recovery process of the execution history must be started:

  SubClass (Guaranteed contract)

  class $gPGC$: *persistGContract* {
  $property$ {$values$ {$name$ "type", $value$ "guaranteed"}},
  $rules$ {$r1$ $pgcRule2, r2$ $pgcRule3$ }
  }