Dissertation

Integration of Transaction Management in Web Service Orchestrations

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der Sozial- und Wirtschaftswissenschaften

unter der Leitung von

Univ. Prof. Dr. Jürgen Dorn
E184-2 Institut für Informationssysteme

eingereicht an der

Technischen Universität Wien
Fakultät für Informatik

von

Mag. Peter Hrastnik
9401792
Aegidigasse 7-11/2/23
A-1060 Wien
Österreich
peter@hrastnik.at

Wien, im April 2006

Unterschrift

Abstract

This thesis tries to spur on transaction–oriented processing in distributed Web service systems, particularly in Web service orchestrations that implement virtual enterprises.

For this purpose, existing scientific work in the area of advanced transaction semantics is analyzed in terms of its applicability in Web service systems. Moreover, the thesis examines prominent existing proposals for transactional Web service systems. These proposals are quite alike and certain shared deficits, which may be the reason for their minor success, are pointed out.

The main part of the thesis specifies a new approach for distributed transactional Web service systems: TWSO – Transactional Web Service Orchestrations. This approach is based on a programmatic paradigm, as also normally used in traditional transaction systems. Thus, a clear and well–known usage pattern is presented. Complex and arbitrary transactions semantics can be implemented by using concepts as follows. A set of transaction primitives, that is appropriate for Web service needs, is provided and it is possible to coordinate multiple concurrent transactions. TWSO concepts can be integrated with arbitrary Web service orchestration technologies. The thesis contains integration proposals for XPDL and Java Web service orchestrations.

To evaluate the TWSO approach on the basis of certain predefined criteria, a scenario, which stems from a real–world project in the tourism domain, is used. It is implemented using a prototypical TWSO system. Evaluation results show that TWSO satisfies the postulated criteria.

Acknowledgments

First and foremost I want to thank my mother and my father. They laid the foundation and made this all possible.

I am grateful to Jürgen Dorn, who supervised my doctoral studies at the Vienna University of Technology. His excellent foresight and support shaped this work considerably. I also want to thank Werner Winiwarter, who supervised my work at the University of Vienna for giving indispensable support for publishing key results of my work in journals and conference proceedings. Further, I deeply appreciate his huge efforts in putting the finishing touches to my thesis.

I also want to mention the E–Commerce Competence Center (EC3). It is a pleasure to work in such a fruitful environment with so many smart and nice colleagues.

Last but not least, I thank my friends for encouraging my ambitions, and for making my spare times so pleasant and recreational.

Contents

I Preliminaries 1

1 Introduction 2
- 1.1 Transactions and Web Services . . . 2
- 1.2 Goals . . . 11
- 1.3 Structure . . . 13

2 Scenario 15
- 2.1 City Trip Scenario Case . . . 16
- 2.2 Car Rental Tour Scenario Case . . . 17

3 State–of–the–Art 21
- 3.1 Transaction Basics . . . 21
- 3.2 ACID Transactions . . . 21
  - 3.2.1 Usage Pattern of ACID Transactions . . . 23
  - 3.2.2 Preserving Isolation in ACID Transactions with Locks . 24
  - 3.2.3 Two Phase Commit Protocol . . . 25
- 3.3 Advanced Transaction Models . . . 26
  - 3.3.1 Transactions with Savepoints . . . 26
  - 3.3.2 Nested Transactions . . . 27
  - 3.3.3 Multilevel Transactions . . . 28
  - 3.3.4 Sagas . . . 28
- 3.4 Advanced Transaction Metamodels . . . 28
  - 3.4.1 Gray and Reuter's Approach . . . 28
  - 3.4.2 ACTA . . . 31
- 3.5 Evaluation of Advanced Transaction Metamodels . . . 33
  - 3.5.1 Gray and Reuter . . . 34
  - 3.5.2 ACTA . . . 39
  - 3.5.3 Summary . . . 39
- 3.6 Web Service Transaction Proposals . . . 40
  - 3.6.1 Business Transaction Protocol . . . 40
  - 3.6.2 WS–Transactions and WS–Coordination . . . 41

3.6.3 Web Service Composite Application Framework 42
3.7 Evaluation of Web Service Transaction Proposals 44

II Transactional Web Service Orchestrations 48

4 Introduction into TWSO 49

5 Ubiquitous Transaction Presence 52

6 TWSO Orchestrations 55
6.1 TWSO Usage Pattern 56
6.2 Transaction Primitives 57
6.2.1 Transactions Managing Multiple Web Services 59
6.2.2 Transaction Lifecycle 60
6.3 Transaction Dependencies 61
6.4 Selected TWSO Orchestration Solutions 64
6.4.1 TWSOL – XML Implementation of TWSO 65
6.4.2 TWSO4J – Java TWSO Implementation 70

7 Transaction Monitor 75
7.1 Interface: Orchestration and Transaction Monitor 78

8 TWSO Enabled Web Services 82
8.1 TWSO WSDL 83
8.2 Interface: Transaction Monitor and Web service 87

9 TWSO Tourism Scenario Orchestrations 90
9.1 City Trip Scenario 90
9.2 Rental Car Tour Scenario 95
9.3 Summary 96

III Implementation of a TWSO Environment 99

10 TWSO Prototype Implementation 100
10.1 Overall Architecture 100
10.2 TWSO4J Implementation 101
10.3 TWSO Transaction Monitor Implementation 102
10.4 TWSO Enabled Web Services in protoTWSO 103
10.5 Limitations of Prototype Implementation 104

11 Evaluation **106**
11.1 Performed Testing 106
11.2 Performance Measurements 107
11.3 Towards an Automated Test Environment for Web Service Transaction Systems 108

IV Epilogue **111**

12 Conclusion **112**

13 Future Steps **114**

V Appendices **116**

A Web Service Description Language **117**

B TWSO Transaction Monitor WSDL **119**

C Scenario Orchestrations **123**
C.1 XPDL Scenario Orchestrations 123
C.1.1 TWSOL in XPDL 123
C.1.2 TWSOL XPDL Scenario Orchestration Documents . . 123
C.2 Java Scenario Orchestration 143

List of Figures

1.1 Web service orchestration failures 10

2.1 Non–transactional Vienna city trip orchestration 19

3.1 Common transaction usage pattern 23
3.2 Insufficient isolation scenario 25
3.3 Sufficient isolation scenario 26
3.4 Single aspect of a flat transaction system 30

4.1 Architecture of a TWSO system 50

6.1 TWSO usage pattern 56
6.2 TWSO state–transitions 61
6.3 Transaction monitor factory 71
6.4 Transaction monitor 72
6.5 TWSO4J transaction 73

9.1 Non–transactional Vienna city trip orchestration 91
9.2 Transactional Vienna city trip orchestration 91
9.3 Adapted transactional Vienna city trip orchestration 94
9.4 Transactional Crete car tour orchestration 95

11.1 Transactional Crete car tour orchestration measurements . . . 107

A.1 Elements of a WSDL definition 117

List of Tables

3.1 History of Web service transaction proposals 45

6.1 Transaction primitives and categorization 58
6.2 Basic transaction states 68
6.3 Basic transaction primitives 70

11.1 Test mock–up Web Services for Crete rental tour 110

Part I

Preliminaries

Chapter 1

Introduction

1.1 Transactions and Web Services

In contrast to single–user standalone applications, distributed computing systems as described in Definition 1.1 require high coordination efforts.

Definition 1.1. A *distributed system* is an application that consists of components running on physically different computers and/or running on different separated spheres of the same physical computer concurrently. These components are able to communicate and are designed to operate separately.[1]

Definition 1.2. A *virtual enterprise* is an organizational model in which different independent organizations work together in order to offer better services to customers that cannot be offered by a single organization. Since the cooperation may be short–term, it has to be easy to establish cooperation without great organizational or financial efforts. The members of a virtual enterprise shall collaborate mainly via the Internet. A customer will not necessarily recognize that the service is provided by different organizations. Preferably, the business processes in such a virtual enterprise should become manifest in electronic form [16].

Distributed systems allow the realization of virtual enterprises as defined in Definition 1.2 and thus have significant economical relevance. For example, consider a simple virtual enterprise that implements a booking operation in the tourism domain that consists of booking a flight, a hotel and a rental car. Flight, hotel and rental car booking occurs on different databases and either

[1]It should be noted that this definition can include, for example, a full–blown J2EE system with numerous legacy systems and relational databases on different servers as well as a simple Visual Basic application that connects to a database server, each of them running as a separated process on the same physical computer entity.

flight, hotel and rental car together or nothing at all should be booked. Such a scenario poses significant coordination efforts to guarantee a valid result. Most importantly, the following matters have to be regarded:

- If at least a single service could be booked successfully and afterward it turns out another one cannot be booked, all occurred actions have to be undone without any traces.
- When arbitrary failures emerge during the booking procedure, it has to be possible that all occurred actions are undone without any (from outside) visible traces.
- When several users perform booking actions, it has to be assured that they do not influence each other unintentionally.

To tackle such coordination tasks in distributed systems in an efficient, dependable and non–redundant way, transaction processing systems (Definition 1.3) can be employed.

Definition 1.3. According to [24], a *transaction processing system* provides tools to ease or automate application programming, execution and administration. A transaction (see Definition 1.4) is executed in the environment of a transaction processing system.

Definition 1.4. A *transaction* is a collection of operations on the physical and abstract application state [24].

The basic idea of transaction– processing is "relieving the application programmer from worrying about failure and concurrency interleaving" [20] on a high level at development time. A common transaction–oriented programming pattern (see Definition 1.5 and Subsection 3.2.1) has (therefore) evolved and contributes to this goal significantly. Correspondingly, the system administrator is relieved from worrying about invalid application states and unexpected and/or unhandled error conditions at run time.

Definition 1.5. A *pattern* records the design decisions taken by many builders in many places over many years in order to resolve a particular problem [2].

When using conventional distributed systems, transaction–oriented concepts are considered regularly. Transaction–oriented processing usually takes place in database access and other distributed systems. For most practitioners in these areas, transactions are absolutely indispensable and a great help in using and developing such systems. Two prominent case studies emphasize this as follows.

The *MySQL* relational database management system lacked transaction support for a long time. Taking Internet sources like forums, blogs, wikis, etc. into consideration to get an overview of common personal opinions, MySQL was not considered for many projects just because it lacked transaction support — although nobody really doubting MySQL's fastness and reliability (e.g. [47]). The possibility to build transaction semantics in the business logic if needed was not an alternative for most. However, the support of transactions in recent MySQL versions was considered a quantum leap and made MySQL ready for "enterprise level" applications [35].

The development of the *J2EE* standard was initially driven by satisfying the needs for simplifying the development of distributed applications that are portable, scalable, and that integrate easily with legacy applications and data [44]. A lot of other big industry players other than SUN like IBM, Microsoft, Oracle, Bea, etc. contributed to this standard by sending their most senior architects and engineers that shared their invaluable knowledge and experience [32]. Also here, transaction–oriented processing was an integral part from the very beginning. This circumstance indicates once more that transactions are highly demanded when it comes to build "serious" distributed systems.

Besides these two developments, there are countless software products that try to facilitate transaction–oriented processing in distributed systems. History has shown that the development and operation of distributed systems benefits from the employment of transaction–oriented principles to a high degree.

Definition 1.6. *Dependability* is defined as the trustworthiness of a computing system, which allows reliance to be justifiably placed on the service it delivers [33].

Transaction–related processing increases the dependability of a distributed system. The concept *dependability* as given in Definition 1.6 can be classified into different aspects, which are as follows [26]:

- Availability is the readiness of a system to accept and process requests correctly.
- Reliability is described by the persistency of the system being provided correctly.

- Safety prevents severe consequences for users and the system itself.
- Confidentiality guarantees that data is not revealed unauthorized.
- Integrity assures that no illegal system states or state transitions appear.
- Maintainability is the possibility to modify, tweak and control a running system.

To attain dependability of a system, certain measures can be taken. These measures can be classified into four categories. *Fault prevention* subsumes measures that can be taken a priori during the design and implementation of the distributed systems. This includes quality assurance during software engineering as well as quality assurance during the design of the overall system architecture and hardware. *Fault tolerance* includes measures that are taken a priori to assure that the systems works correctly even when faults emerge. When faults appear at operation and are not handled appropriately, *fault removal* can help to eliminate these faults. System diagnosis and corrections during operation may be applied here. Last not least, *fault forecasting* permanently analyzes system parameters to forecast potential faults in order to take preventive measures. Transaction–oriented systems mainly focus on fault prevention and fault tolerance. Strategies for fault removal and fault forecasting tend to be too application specific to be tackled by general models like transaction–oriented processing.

Recently, Web services are used more and more to build distributed systems. The term Web service is defined generically in Definition 1.7. This definition is good for a basic comprehension, but too wide–ranging. A more detailed and practice–oriented definition that captures the current character of Web services better and is used in this thesis is given in Definition 1.8.

Definition 1.7. *Web services* are software–powered resources or functional components whose capabilities can be accessed at an Internet URI. However, this definition is rather generic.

Definition 1.8. A *Web service* is a software functionality whose interface is described by WSDL[2] [12] and which is capable of being accessed via standard network protocols such as but not limited to SOAP[36] over HTTP. It should be noted, that a WSDL definition contains and groups multiple (related) Web service functionalities. Thus, in terms of WSDL, a Web service is a WSDL

[2]A short introduction to WSDL is given in Appendix A.

operation that is accessed at a particular port using a particular binding. It is neither a port type nor a group of related port types.

In distributed Web service systems, aggravating circumstances arise from the usually somewhat low dependability of third party Web services distributed in the Internet. Even though Internet services usually have incorporated a number of techniques for achieving high availability, user–visible failures still occur frequently [38]. Oppenheimer et al. [38] surveyed the causes of Internet services in detail and conclude, that user–visible Internet service errors originate mainly from human operator failures and errors in custom software mainly. Network failures occur, but can be hidden from users efficiently. They also conclude that more extensive testing could prevent many failures. Based on this survey, it becomes obvious that operating and developing Internet services — and therefore also Web services — is an inherently error–prone discipline. Internet services and especially Web services are still in their infancy. Despite all undertaken efforts, it seems that it is not possible to develop and operate them dependably, as it can be (and is) achieved in other computer disciplines (e.g. intra–enterprise mainframe computing). It should be noted, that pure Internet network errors are virtually non–existent. The core of the Internet consists of several large backbones, which are well–engineered and adequately provisioned. This leads to negligible delays and virtually no downtime [21, 39]. Accordingly, visible failures of Internet services virtually always stem from the service itself and not from the Internet–network, which itself is pretty much reliable. Mostly, Web service users neither have the chance to assess nor to positively influence the dependability of required Web services. Thus, it is essential to take erroneous situations into account inherently in the whole distributed Web service system. Transaction–oriented processing is a way to do so efficiently.

Depending on the characteristics of the transaction processing system, a transaction always conforms to some axiomatic semantics. The most commonly — or in practitioner groups frequently the only known — kind of transaction processing system supports solely ACID transactions, which are described more detailed in Section 3.2. It should be stressed, that in everyday database practitioner's life, the term transaction mostly refers to traditional ACID transactions. In contrast, in this thesis the term transaction is strictly used as shown in Definition 1.4, where no particular set of properties is imposed to a transaction per default.

In tightly coupled systems, transactional processing that follows the ACID semantics is ubiquitous and works well. More precisely, they are highly useful in applications with transactions that:

- have a short duration,
- influence mostly intra–organizational resources that are controllable, whose behavior is assessable, and whose availability and reliability is trustworthy, and
- act in a network that is controllable, whose behavior is assessable, and whose availability and reliability is trustworthy.

Distributed Web service systems do not adhere to the tightly–coupled system characteristics where ACID transactions would be appropriate. Mostly, Web services are out of sphere–of–control of an organization, and are nebulous regarding their behavior and dependability. In addition, Web services — especially Web services that are called in an asynchronous way[3] — possibly have a considerable extensive interaction time.

Consequently, transactions that follow ACID principles may not be practical in distributed loosely coupled systems, e.g. distributed systems composed of Web services. Potts et al [40] discuss ACID transactions regarding Web services more detailed and even assert that "transaction semantics that work in a tightly coupled single enterprise cannot be successfully used in loosely coupled multi–enterprise networks such as the Internet". For appropriate Web service transaction systems, transaction semantics that are more flexible than ACID transaction semantics are required.

Advanced transaction models (see Section 3.3) offer other transaction semantics in addition to common ACID transactions. Numerous advanced transaction models were developed [19, 24] to overcome the rigid demands of ACID transactions. For example, a particular advanced transaction model could relax the strict atomicity claim. These concepts can be used well for Web service transactions.

For Web service transaction systems, yet another aspect of advanced transaction systems comes into effect. Different (business) domains require different policies for conducting transactional processing. As stated in [42],

[3]Asynchronous invocation style has significant advantages to request–response invocation style. According to a talk from Frank Leymann on Sept. 17th 2003 hosted by Vienna University of Technology, asynchronous Web services will gain major relevance in near future.

no out–of–the–box set of advanced transaction models can satisfy all requirements of all domains that want to do inter–organizational transactional Web service processing. Therefore, a Web service transaction system should support arbitrary advanced transaction models, i.e. it should support arbitrary transaction semantics. Advanced transaction models can be described using formal metamodels. Well–known approaches for such "advanced transaction meta models" are presented in Section 3.4. Accordingly, concepts of an approach for Web service transactions should be based on such a metamodel in order to offer a reasonable solution for using arbitrary advanced transaction semantics.

These claims become more apparent when the scenario above is enhanced as follows. The booking tasks are implemented using Web services and these Web services are distributed globally and organizationally: The customer books in Austria at a local travel agency, the flight booking Web service is hosted in Moscow by Aeroflot, the hotel Web service in Germany is offered by Ibis and the rental car Web service at the destination is located in Greece. The flight booking is expected to last 5 seconds and the rental car booking 2 days. Preferably, the customer wants to book all three kinds of resources, but is also satisfied when some resources cannot be booked. Since the booking process obviously is long lasting and error–prone, considering this customer preference in detail may lead to much more successful overall results than sticking to simple all–or–nothing semantics. However, such customer preferences are subjects to change and so are the factors that lead to successful outcomes. To adequately implement this scenario, transaction semantics have to be tailored to customer preferences. Prefabricated (advanced) transaction semantics most likely may not support the customer preferences as well as the long lasting nature of some booking tasks. Moreover, an implementation of the scenario has to address the likely change of transaction semantics sufficiently. Thus, ACID transaction principles obviously are rather useless to implement this scenario.

The creation of distributed Web service systems like the scenario described above can be facilitated by employing Web service orchestration technologies, as defined in Definition 1.9.

Definition 1.9. *Web service orchestrations* tie together a set of existing Web services in order to create a completely new service by employing workflow technologies [27].

It should be noted, that the term Web service orchestration as presented in Definition 1.9 describes applications that use several Web services as well as applications that use a single Web service only. Obviously, the term "orchestration" does not describe the latter situation intuitively. However, since every orchestration technique can handle these cases by nature, the introduction of a new term is abandoned and the term Web service orchestration refers to such applications in this thesis, too.

Descriptive process languages such as XPDL[48] or BPEL4WS[3] should ease the development of Web service orchestrations. However, the use of descriptive process languages is not the only way to implement Web service orchestrations. Using traditional programming languages like Java is a possibility, too. Thus, in this thesis the term Web service orchestration does not imply a certain implementation method of the actual orchestration. A Java program that combines Web services is as well a Web service orchestration as a BPEL4WS process definition.

In Web service orchestrations that combine numerous Web services, transactional Web service processing highly gains importance because major coordination and error–management efforts have to be undertaken to deliver a valid result. Transaction–oriented processing can help to cut these efforts significantly. Additionally, as discussed above, transaction–oriented processing increases the dependability of Web service orchestrations.

A taxonomy of common failures that may occur in Web service orchestrations is depicted in Figure 1.1. Communication failures relate to erroneous network communication and consist of protocol violations and transfer failures. Protocol violations occur whenever the used communication protocol is not followed. For example, a SOAP message that does not adhere to the SOAP standard is a protocol violation. Transfer failures subsume situations where communication is not possible at all. For example, an unreachable Web service host is a transfer failure. Service failures stem from Web services. Unexpected service failures are all failures that should not occur in theory, but may happen in practice. For example, "Nullpointer Exceptions" thrown by a Java Web service or "Internal Server Errors" are unexpected service failures. Expected service failures are anticipated. Service users should be aware[4] of such failures and should take precautions. Common expected service failures are input validation related, when the delivered input does not meet the service's expectations, and resource related, when a (logical) resource that is needed for a successful service execution is not available

[4]Expected service failures are listed in the fault elements of the service's WSDL document.

adequately. For example, a booking request that fails because of ran out resources causes an expected resource related service failure.

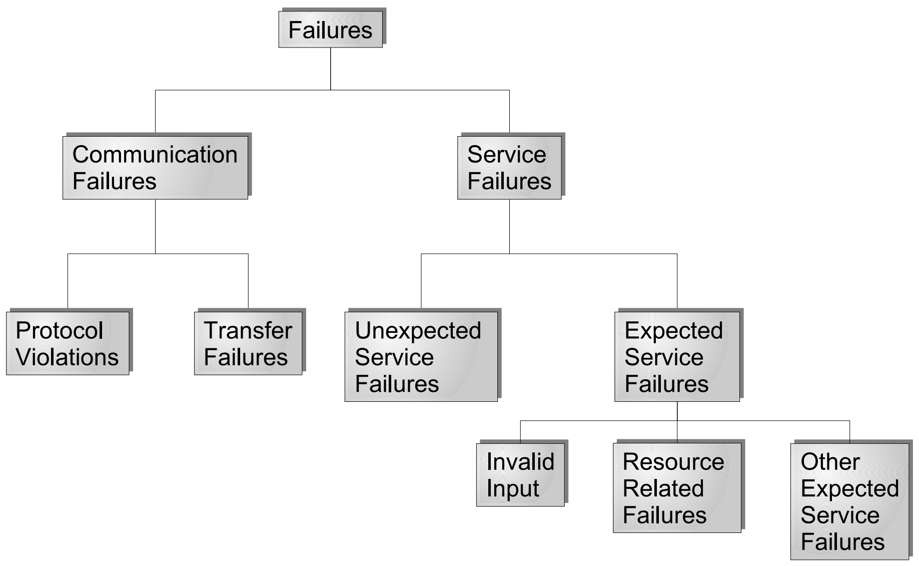
Failures
Communication Failures
Service Failures
Protocol Violations
Transfer Failures
Unexpected Service Failures
Expected Service Failures
Invalid Input
Resource Related Failures
Other Expected Service Failures

Figure 1.1: Web service orchestration failures

To react on such failures, orchestrations may apply the following general measures. Most times, a failure has to be handled explicitly by the orchestration's logic. For example, a failed booking request on service A may result in doing the same booking request on service B. In some cases, a failure can be ignored if it is assumed that it does not compromise a valid outcome of the orchestration. It also may make sense to repeat operations when it seems to be likely that the failure causing condition will disappear in the near future. For example, transfer errors are predestined to be handled that way.

It is not possible to deduce universal relationships between failure types and measures in Web service orchestrations. Such relationships highly depend on the orchestration's goal, the orchestration's domain and the particular service.

Web service orchestrations are an important technique to implement virtual enterprises, as defined in Definition 1.2. Consequently, a major application area of Web service transaction concepts is the development and operation of virtual enterprises. Implementation and run–time efforts of virtual enterprises decrease, and dependability of virtual enterprises increases.

It obviously stands to reason that the development and operation of distributed Web service systems benefit when transaction–oriented paradigms are applied. Employing transaction–oriented principles would ease distributed Web service system development and administration in terms of achieving superior dependability. The programmer of a Web service application could be absolved from the burden of worrying about failures and concurrency interleaving. The system administrator of a Web service system is relieved from worrying about invalid application states and unexpected and/or unhandled error conditions. Transaction–oriented processing in the Web service field increases the degree of dependability of distributed Web service systems significantly. Moreover, usual distributed systems or distributed applications in general require the advantages of transaction–oriented concepts. Thus, it is sound to draw the conclusion that also distributed Web service systems can potentially benefit a lot from the practical experiences and scientific results that were accumulated over time in the transactional processing and distributed systems field.

The existence of ready to use transaction–software for Web service systems would make the necessity for own, almost certainly inferior, solutions — as it is common practice nowadays — witless. Operation of distributed Web service systems will be facilitated because particular transactions in the system might be influenced directly by transaction management components. There is no need to (re)implement such transaction influencing operations in the application scope.

Spoken generally, systems that consider transaction–oriented paradigms are potentially robust when it comes to unforeseen erroneous situations at run–time. Invalid system states are avoided by nature. All these considerations strongly motivate the use of transaction–oriented concepts in modern Web service computing.

1.2 Goals

Software industry has already identified the assets that transaction–oriented processing in the Web service area could bear, and has published several proposals. As discussed in detail in Subsection 3.7, these proposals are quite alike and have weaknesses, which may be responsible for the lack of proposal implementations and the lack of transaction–oriented processing in Web service consumer applications nowadays.

The goal of this thesis is to spur on transaction–oriented processing in the Web service world by presenting an approach for transaction–aware Web service systems that overcomes certain recognized deficiencies of existing approaches. It differs from these approaches significantly. It is called *TWSO – Transactional Web Service Orchestrations*[5].

The new TWSO approach features the following core characteristics:

- Seamless integration into Web service orchestration techniques. TWSO is designed to be potentially integrateable into virtually any Web service orchestration technique.
- Usage of arbitrary advanced transaction models is taken into consideration from scratch and is an inherent characteristic of TWSO. Arbitrary advanced transaction models can be put in operation based on the particular domain's requirements without hassles.
- The usage pattern (see Definition 1.5) of TWSO resembles common transaction–oriented programming patterns. Familiarization efforts of new TWSO users are minimized.

The soundness of the TWSO approach will be verified by applying it to a scenario that is part of a real world project. This scenario consists of a virtual enterprise in the tourism domain that provides booking services across various resources. The resources are available via Web service means. The virtual enterprise integrates these resources or these Web services in an electronic business process that is implemented by using Web service orchestration technologies. Since Web service orchestrations enclose numerous challenges that can be tackled by Web service transaction approaches efficiently, this scenario is highly appropriate to check the TWSO approach thoroughly. In detail, the scenario should demonstrate that transactional processing in Web service systems with the TWSO approach:

- can be integrated with Web service orchestrations seamlessly,
- allows to put arbitrary transaction semantics into operation easily,
- employs a usage pattern that resembles the usage pattern of transaction–oriented processing commonly used nowadays,

[5]The reference to Web service orchestrations in the name stems from strengths of TWSO that can be used particularly in orchestrations that involve more Web services. Of course, TWSO is perfectly usable and reasonable for applications that call only a single Web service, too.

- increases in general the dependability of distributed Web service systems and in particular the dependability of virtual enterprises that are implemented using Web service orchestration techniques, in detail, it should be shown that TWSO is capable of increasing dependability, i.e.:
  - availability,
  - reliability,
  - safety,
  - integrity, and
  - maintainability,

  by providing effective means of:
  - fault prevention, and
  - fault tolerance.

To actually evaluate the scenario, a prototypical implementation of a TWSO transaction processing system called *protoTWSO* is developed. This implementation provides all functional aspects of a TWSO system. However, non–functional aspects like performance, stability under high–load and internal error processing and interception are regarded only to a certain extent and need to be revised for a production grade TWSO system. In addition, the implementation will be used to determine performance related influences of using transaction–oriented processing in Web service orchestrations.

1.3 Structure

Part I presents introductory contents. In Chapter 1, the background of the underlying research topic "Transactions and Web services" is introduced and goals of the thesis are defined. These goals are validated on the basis of a comprehensive real–world scenario presented in Chapter 2. State–of–the–art technology and research in the field of transactional Web service computing is presented in Chapter 3. Here, current transactional processing techniques and advanced transaction concepts are outlined. Moreover, transaction metamodels that capture arbitrary transaction semantics are presented. These metamodels are evaluated detailed in terms of applicability for Web service transaction systems, as also published in [28] and [30]. In the end, existing approaches for creating transactional Web service systems are discussed.

Part II presents the TWSO approach for transactional processing in Web service systems. For the most part, accomplishments as described here have been published in [29] and [31]. The basics of TWSO are presented in Chapters 4 and 5. Chapter 6 introduces the usage of TWSO in Web service systems in detail and Chapter 7 defines the TWSO transaction monitor component. Requirements and properties that have to be met by Web services that participate in a TWSO enabled system are described in Chapter 8. Chapter 9 discusses the implementation of the scenario using TWSO.

Part III describes a prototype implementation of a TWSO system that uses Java Web service orchestrations. Chapter 10 presents the prototype in detail and Chapter 11 provides evaluation results of the prototype system.

Finally, in Part IV, Chapter 12 draws conclusions and Chapter 13 outlines important tasks for future research that will be started based upon the results of this thesis.

Appendix A gives an understanding of WSDL that is needed to understand various specifications in the thesis and Appendix B presents the complete WSDL description of a TWSO transaction monitor. Appendix C.1 describes a method to intersperse TWSO concepts within XPDL orchestrations and gives complete listings of orchestrations that implement the scenarios of Chapter 2. Appendix C.2 shows a Java orchestration that implements selected scenario parts.

Chapter 2

Scenario

To validate the soundness of the TWSO approach, a reasonable and comprehensive scenario has to be developed. It should be shown, that the use of TWSO in such a scenario bears significant benefits.

The scenario is part of a real world project that is carried out by *EC3 - Electronic Commerce Competence Center* and *Tiscover*, a company that runs a tourism information system with a search engine and other functionalities. The project encompasses a virtual enterprise that provides a metasearch–function for miscellaneous tourism services. This search considers not only the Tiscover database, but also data–sources stemming from other Internet tourism services. The possibility to book discovered services should be provided, too. Thus, the virtual enterprise searches and combines tourism services to create a dynamic holiday package. The customer of the virtual enterprise buys such a package. The correct way to combine the individual building blocks should be calculated without human intervention. Thus, Web service composition strategies may be applied here. Web service composition software that has been developed by Electronic Commerce Competence Center and performed best in an international contest [17] may be used for this task. The particular building blocks of the dynamic holiday package stem from different resources that are globally distributed. These resources are accessible through Web service techniques. The customer of the virtual enterprise knows nothing about the individual building blocks and their global distribution and merely obtains the single journey. The scenario focuses on the booking part of the virtual enterprise only. Thus, from now on it is assumed that tourism services have been already found and composed and are ready to be booked.

To book an appropriate journey for a particular customer, strategies as follows are applied. For each destination, one or more sets of reasonable services are defined. For example, a city trip can consist of a set of services like flight, airport transfer, hotel room, public transport ticket, etc.

However, traveler preferences differ. For example, some want to have maximum flexibility and prefer to organize a holiday program on their own in an ad–hoc manner at the destination while others favor a completely pre–given holiday program for the whole stay. Which traveler type is applied may be determined by an interview or by deducing the type from other traveler attributes like age, gender, origin, profession, etc. However, the method that is used to decide on the traveler type is not covered in this thesis. It is presumed that the correct traveler type is known.

Depending on the target region and the traveler type, a particular pre–defined electronic business process is put in action. The scenario includes cases as follows.

2.1 City Trip Scenario Case

A city trip to Vienna for high convenience tourists should be created. The set of services for this trip is as follows:

- flight,
- airport transfer,
- hotel room,
- set of tickets for selected sights,
- set of tickets for selected events,
- coupons for selected restaurants to provide half–board catering.

The high convenience customer wants to reach Vienna in the most comfortable way. Thus, a flight is used. To reach the necessarily pre–booked hotel room, an airport transfer service has to be available. The customer wants a ready–made holiday program and either the sight ticket set or the event ticket set or both have to be available. Moreover, the high convenience customer does not want to worry about where to have a meal. Thus, a set of half–board restaurant coupons are needed. Expected processing times for booking the hotel room in Vienna is two days, whereas all other services can be booked within seconds.

At first, the services do not charge booking attempts but after some months, the flight service charges each booking attempt. Thus, it becomes necessary to minimize unneeded flight booking requests. Moreover, after some months it is noticed that the typical high convenience customer is not longer satisfied with only sight or event tickets and now he wants both types of tickets. This circumstance has to be considered to meet (new) customer preferences as good as possible.

2.2 Car Rental Tour Scenario Case

This Scenario encompasses a rental car tour around Crete. This trip should be offered to two types of travelers. The first type encompasses travelers that are flexible enough and enjoy to book necessary services at the destination to a certain degree on their own. But they also appreciate pre–booked services. The other type is not so flexible and expects a wide array of pre–booked services at the destination. The set of services for the Crete trip is as follows:

- flight to Heraklion,
- rental car,
- hotel in Heraklion,
- hotel in Agios Nikolaos,
- hotel in Chania.

To reach the destination, a suitable flight to Heraklion should be booked. To provide the traveler with sufficient mobility, a rental car should be provided at Heraklion airport. This trip is a tour around the whole island Crete and at adequate locations, hotel rooms may be pre–booked. For both types of travelers, the flight to Heraklion is mandatory.

For the more flexible traveler type, the following applies. It is assumed that if the flexible traveler is equipped with a car, it is very easy to find a suitable hotel room in Heraklion for her. And — the other way round — if the traveler has a pre–booked hotel room in Heraklion, she has a place to stay and can rent a car on–site without any hassles. Thus, it is required that the hotel in Heraklion and/or the car booking have to be successful for a valid journey. If both, the car booking and the hotel booking fail, a flexible traveler considers the whole journey as unacceptable. Pre–booked hotels in Chania and Agios Nikolaos are nice to have but not necessary.

The not so flexible traveler expects that there is a rental car and a hotel room provided in Heraklion. However, for these travelers it is acceptable to book at most one hotel room at the destination on their own. Thus, either the hotel room in Chania or the room in Agios Nikolaos or both should be pre–booked.

It is assumed that the car rental and hotel room bookings need approximately 2 days to finish because human processing is involved. The full automated airline reservation system needs just some seconds to provisionally reserve a free seat.

It is obvious that a traditional ACID transaction cannot support Scenario 2.1 and 2.2. It would be necessary to stick to the atomicity property: When a single service fails, the whole transaction has to fail. For example, if the car–booking fails in the Crete journey, the whole journey would fail, even though the flexible traveler accepts a journey that consists out of a flight and a hotel room in Heraklion. The isolation property (see Section 3.2) is also problematic in these cases. If the service booking durations as stated above are considered: To preserve the isolation property it would be necessary to keep a tentatively booked flight seat hidden from other transactions for days. The status of the seat remains unclear and ultimately depends on the decision of some small local organizations, like an event ticket set vendor. This situation is probably unacceptable for any airline.

To implement the scenarios without transactional concepts, massive business logic would be necessary. One would have to check numerous outcomes of operations and act accordingly with appropriate business logic in the electronic business process. An array of unforeseen erroneous situations like time–outs, temporary unavailability of services, etc. have to be handled appropriately in the electronic business process. In addition, concurrent processing (e.g. concurrent booking of tourism services) to minimize execution time imposes vast coordination measures. These measures have to be implemented in the electronic business processes, which poses a non–trivial and error–prone issue. In addition, changing requirements (as posed in the Vienna city trip) also imply massive modifications of business logics. A pos-

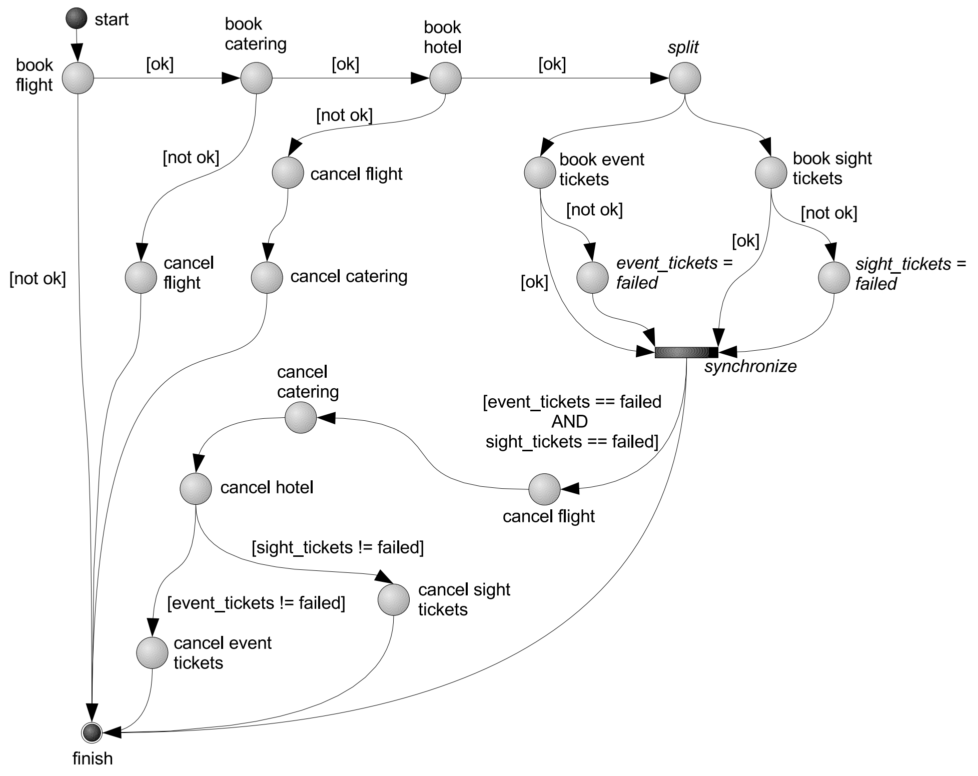

Figure 2.1: Non–transactional Vienna city trip orchestration

sible workflow that implements the first part of the Vienna Scenario 2.1 is depicted in Figure 2.1. This figure shows the vast efforts that are necessary to implement failure management and coordination when only workflow concepts are employed. Although this workflow considers concurrent processing only marginally and failure detection is simplified drastically to check "ok" or "not ok" conditions of actions.

It will be shown that using TWSO and its transactional concepts, all the fields above can be tackled in a straightforward way. Reinventing the wheel and coordination–efforts will be reduced, dependability increased.

The electronic business processes of the virtual enterprise can be implemented faster with transaction–oriented concepts. Time–to–market is decreased, the virtual enterprise can be put into operation faster. New business processes, for example, to react on varying or new tourist needs, can be deployed faster. Moreover, failure–management of the electronic business processes is improved — a key characteristic for the virtual enterprise that operates on globally distributed Web services via the Internet. As discussed in Section 1.1, erroneous situations are likely here and their causes cannot be prevented actively. The only possible remaining measure is to try to react on such situations as good as possible, and transactional processing is an effective and efficient way to do so.

From a technical point of view, proper technologies that put the electronic business processes in action are needed. Due to the fact that our virtual enterprise operates on Web service resources, Web service orchestration technologies are a rather good choice to do so. It will be shown, that the TWSO approach can be efficiently and effectively used to incorporate transactional concepts into the orchestration(s) that implement the electronic business process of the described virtual enterprise.

In short, it will be shown that TWSO is capable of:

- reducing time–to–market of the virtual enterprise,
- reducing time–to–market of the tweaked virtual enterprise,
- cutting coordination efforts in the virtual enterprise significantly,
- cutting failure management efforts in the virtual enterprise significantly,
- putting arbitrary advanced transaction semantics in the virtual enterprise into operation,
- increasing dependability of the virtual enterprise, and
- being incorporated into existing orchestration technologies efficiently and effectively.

Chapter 3

State–of–the–Art

3.1 Transaction Basics

A transaction is always executed within the bounds of a transaction processing system. Often, a particular component in the transaction processing system is dedicated to cover the management of transactions. This component is called *transaction monitor*.

Transactions are controlled by sending them special transaction commands, called *transaction primitives*. During its lifecycle, a transaction comes into different *states*. Typically, state changes of a transaction arise from issued transaction primitives.

Depending on the characteristics of a particular transaction processing system, a transaction conforms to a set of properties. Depending on the application domain, numerous combinations of types of *transaction properties* can be reasonable. Based on these properties, different transaction primitives and different transaction states exist in a transaction processing system. It should be stressed, that the term transaction does *not* imply any particular property set.

3.2 ACID Transactions

The most commonly known set of transaction properties are the ACID[1] properties. The ACID property set consists of atomicity, consistency, isolation and durability. This set is widely known and accepted and countless

[1]ACID is short for for Atomicity, Consistency, Isolation and Durability.

transaction–enabled systems support — most times even solely — transactions that adhere to the ACID properties. The exact meanings of the properties are as follows:

- Atomicity: The transaction's changes to the application state are atomic. A transaction has an "all–or–nothing" characteristic. Either all the changes or none of them happen.

- Consistency: A transaction is a correct transformation of the application state. The actions of a transaction do not violate any integrity constraints, the transaction is a "correct program". The applied integrity constraints depend on the particular application.

- Isolation: When transactions execute concurrently, for a transaction t_x the system has to pretend that t_x is the only one being executed. It has to appear that other transactions are executed either before t_x or after t_x. For example, let t_i and t_j be two transactions that are executed concurrently. For t_i it seems that it is either executed before t_j or after t_j.

- Durability: When a transaction completes successfully (e.g. a commit finishes a transaction), the changes it did to the application state have to be permanent and have to survive any kind of failure — be it a software crash, a hardware failure, a power cut or an earthquake.

Transactions that adhere to the ACID properties are suited for tightly–coupled systems, where a typical transaction is short–lived and does not act on globally distributed, organizational independent, and uncontrollable resources. Hence, ACID style transactions are very well suited for distributed systems that are intra–organizational and make use of intra–organizational resources like databases, legacy systems, middleware, etc.

To control ACID transactions, there are three distinct transaction primitives. `begin` starts a transaction. In case the related business logic of a transaction is considered to be successful, the transaction is finished with a `commit`. Changes that happened in the committed transaction's scope are made permanent (durability). Also, all measures that were taken to preserve isolation (see Subsection 3.2.2) can be withdrawn after a `commit`. When the business logic is considered to be erroneous, `abort` cancels a transaction. All changes made so far are revoked without any traces. Isolation measures — if taken — are withdrawn.

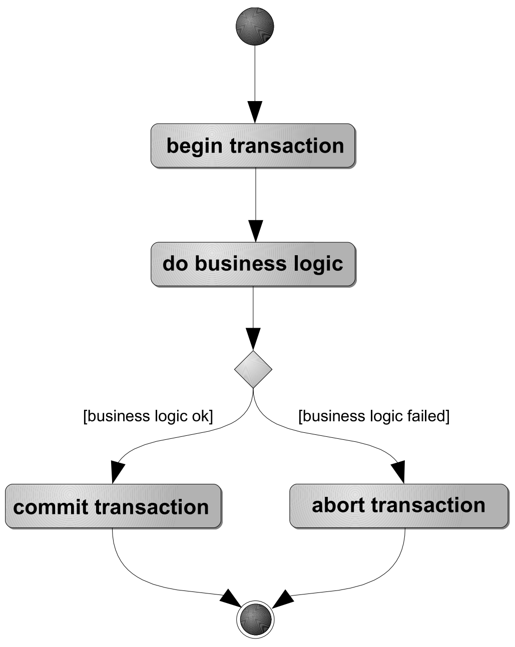

Figure 3.1: Common transaction usage pattern

Based on the primitives, an ACID transaction can have states as follows. After a transaction is started with `begin`, it is in the `in_progress` state. After a `commit`, the transaction gets into the `committed` state. Otherwise, when a transaction is aborted, it comes into the state `aborted`.

3.2.1 Usage Pattern of ACID Transactions

Current methods to do transaction oriented programming with ACID transactions follow a pattern as shown in Figure 3.1.

For example, if an application programmer wants to use a relational database in some procedural/object–oriented programming language in a transactional way, the following process is often applied. Let us assume that new records should be inserted into a `flight_reservation` and a `hotel-room_reservation` table together, i.e. a flight and a hotel room should be reserved jointly in an atomic way.

After acquiring the needed data (which flight and which hotel room and related dates), a new transaction is started. Then, business logic (as defined in Definition 3.1) is executed. First, it is checked whether the desired reservations conflict with already existing reservations. If not, reservation data is inserted into the `flight_reservation` and then into the `hotelroom_reservation` table.

Definition 3.1. *Business logic* is the set of rules, processes, and algorithms that operate on and with information (data) in the business domain, as implemented by the application [8].

Based on return messages of the database system, it is decided whether the business logic could be executed successfully or not. For example, when the database system reported an error free insertion into both tables and no conflicting reservations could be detected, the business logic would be considered to be successful. A `commit` would be used to make the changes in the `flight_reservation` and `hotelroom_reservation` persistent. Otherwise, if the database system reports that the insertion of the record in a table failed or the intended reservation conflicts with an existing reservation, the business logic would be considered to be unsuccessful. Consequently, the transaction would be aborted with an `abort` primitive. If necessary, tentatively inserted records would be undone without any traces.

3.2.2 Preserving Isolation in ACID Transactions with Locks

Considering the example before in Subsection 3.2.1, the database system would have to be configured as follows to preserve isolation. From the moment a transaction t_i performs an arbitrary operation (read, write, delete or update) on table θ_v, only t_i is able to perform operations on θ_v for the duration of t_i, i.e. θ_v is locked exclusively for t_i for all kind of operations. Operations of other transactions are queued up until t_i terminates by `abort` or `commit`. To prevent deadlock situations, timeout failures occur in case an operation is queued for a long time.

Because of the database locking configuration described above, concurrent access to the table(s) is coordinated in a semantically correct way. Isolation and consistency is guaranteed. For example, the situation in Figure 3.2 is avoided a priori. Here, user A and user B want to reserve the same flight and the same hotel room for the same date. Because they both read existing reservations concurrently at the time before any record is inserted, they assume that there is no conflicting reservation. The results are four conflicting reservation records in the database. This is an inconsistent database state

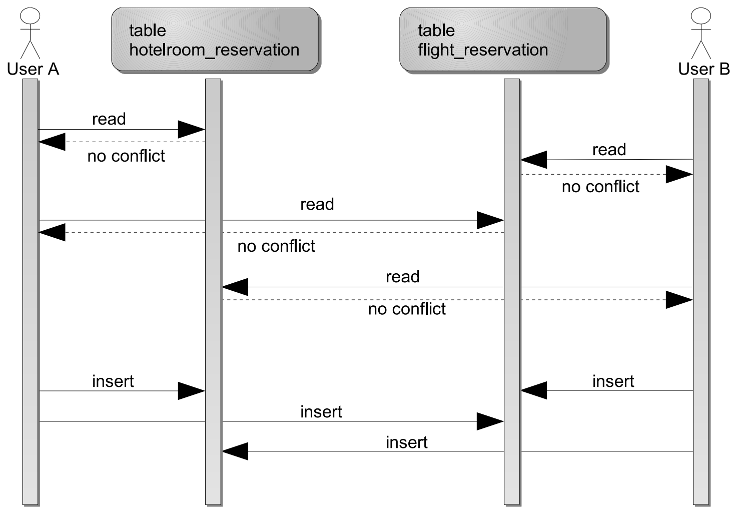

Figure 3.2: Insufficient isolation scenario

emerging from insufficient isolation.

Figure 3.3 shows the result of using transactions and locking tables after a read operation. The read operation of user B is delayed until the reservation records of user A are inserted. Consequently, the read operation notices the reservations of user A and a conflict is detected. Isolation is satisfied and a consistent database state is guaranteed.

3.2.3 Two Phase Commit Protocol

To achieve an atomic agreed outcome in a distributed system and thus to adhere to the atomicity property of an ACID transaction in a distributed system, a special communication protocol is used frequently: The Two Phase Commit protocol (2PC protocol) [24]. Here a coordination entity asks the participating entities whether they can commit or not. This phase is called *prepare phase*. If all participating entities can commit, the coordinating entity asks them to do so. If at least one participating entity tells the coordinating entity that it cannot commit, the coordinating entity tells all participating entities to abort the transaction.

The 2PC protocol is not only used in computer communications, but also used in "real life" for numerous occasions. For example, voting with veto powers is in accordance with 2PC protocol. A moderator (coordinating en-

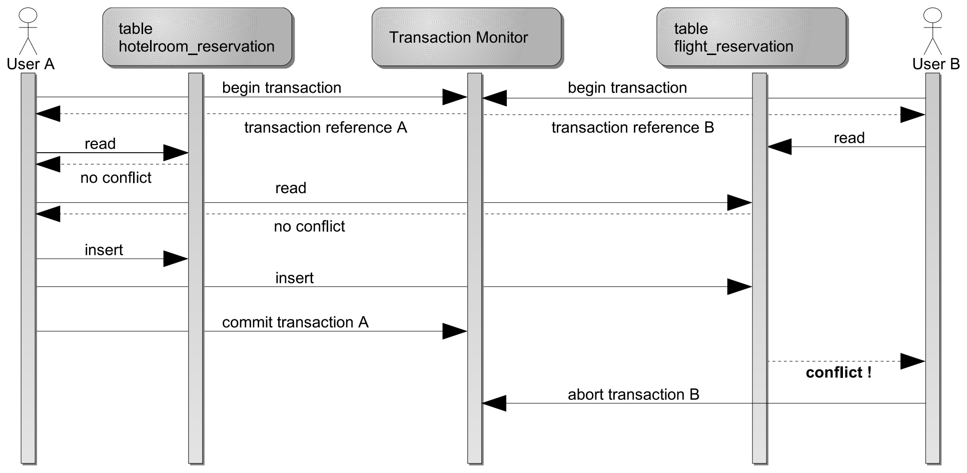

Figure 3.3: Sufficient isolation scenario

tity) asks the voters (participating entities) whether the matter (about which is voted) is ok. If at least one of the voters says “No”, the matter is aborted. A typical Christian wedding ceremony is also handled by the 2PC protocol [24]. The pastor (coordinating entity) asks each partner (participating entities) whether he or she wants to marry. If both partners agree, the marriage is committed. Otherwise, it is aborted.

3.3 Advanced Transaction Models

As discussed in Section 3.2, ACID style transactions are especially appropriate for generally simple and short–lived operations. It has been found that in other areas, the ACID concept has limited applicability [19]. Thus, to overcome the restrictions of ACID style transactions in such domains, other models for transaction–oriented processing were developed. These models are called *advanced transaction models*. Generally speaking, advanced transaction models offer concepts for transaction–oriented processing in domains where the support of one or more ACID properties is not necessary or even not possible. Some important advanced transaction models are presented here. [24] offers a detailed discussion of advanced transaction models.

3.3.1 Transactions with Savepoints

Transactions with savepoints [24] allow organizing a transaction into a sequence of actions that can be rolled back individually.

For example, a virtual enterprise in the tourism domain imports 100,000 existing user accounts from a partner company. It is important, that only the complete set of new user accounts are available in the virtual enterprise, i.e. isolation has to be considered.

Hence, 100,000 records have to be inserted, whereas each insert is a time–consuming task and takes 1 second. The whole transaction would last for a day. If the very last update fails, the whole transaction is aborted and the work of a day is lost.

Savepoints provide a solution for this scenario and relax the atomicity criterion. During a transaction, one can set savepoints and can rollback to an arbitrary savepoint if something fails. It should be noted that setting a savepoint does not commit the modifications that have been done before the savepoint, i.e. another concurrent user cannot see the modifications done so far.

For the previous example, let us assume that a savepoint is set at every 1000^{th} inserted user account, then only 15 minutes of work would have to be repeated in case the last transaction fails.

3.3.2 Nested Transactions

Nested transactions [18] can be seen as a generalization of savepoints. While transactions with savepoints organize a transaction in a sequence, nested transactions form a hierarchy of actions.

A nested transaction is a tree of transactions. The root is called top–level transaction. The superordinate of a sub–transaction is called parent, the subordinate of a transaction is called child. If a transaction rolls back, all child–transactions have to roll back, too. The commit of a child–transaction does not take effect until the parent–transaction commits. The parent–transaction is the only instance that can see the changes of a child–transaction's commit. Thus, any child–transaction can fully commit only if its parent–transaction commits. However, after a commit, the parent–transaction will see the effects of the child–transaction.

3.3.3 Multilevel Transactions

Multilevel transactions [46] are like nested transactions and build an hierarchy of transactions. However, multilevel transactions allow a complete commit of the results of subtransactions before their parent–transactions commit. The results take effect immediately. Yet, it must be possible to retract the committed results of subtransactions. This is achieved by using compensation actions, as defined below.

Definition 3.2. A *compensation action* is a "forward" action that makes some adjustments to reverse the original action. After a compensation action, the fact that the original action took place is visible. In contrast, a rollback undoes an action so that it seems like the action never took place.

3.3.4 Sagas

Put simply, a Saga [22] is a chain of transactions. Each transaction in the chain commits when it finishes its work, and provides a compensation action. If a transaction t_i fails, t_i can do an abort, and all previous transactions $t_1, \ldots, t_{i-1}$ have to start their compensation actions.

3.4 Advanced Transaction Metamodels

As indicated in Section 3.3, several advanced transaction models were developed to overcome the limitations of ACID transactions in some domains. To describe advanced transaction semantics, formal models have been developed. Here, we give a short introduction of advanced transaction metamodels.

3.4.1 Gray and Reuter's Approach

Simple transaction models can be described with finite state–machines. However, most advanced transaction models do not have a fixed number of states. Thus, finite state–machines are not appropriate, and specialized metamodels for advanced transaction models were developed. In this section we give a short overview of an advanced transaction metamodel that was introduced in [24] by Jim Gray and Andreas Reuter. In Gray & Reuter's approach, transactions are modeled as compositions of one or more atomic actions. Atomic actions have ports that identify possible signals an atomic action can receive, and final states that indicate the outcome of an action. For example, a simple atomic action a_s can have the ports `abort`, `commit`, and `begin` and the final states `aborted` and `committed`.

In a transaction, the included atomic actions can also be related. Relations can model the invocation hierarchy of the atomic actions, e.g. if atomic action a_a commits, atomic action a_b has to commit, too. Transactions impose different rules on relations among atomic actions and the effects they have on related atomic actions. For each atomic action in a transaction model, there can be one or more rules. Each rule represents a state transition an atomic action can perform. Such rules have two parts. The active part of a rule defines conditions that trigger events. For example, "commitment of a_a triggers commitment of a_b" is modeled by the active part. These events cause an atomic action to change its state. The passive part specifies the conditions for performing a state transition. For example, "commitment of a_a can only happen if a_x is ready to commit, too" can be defined by the passive part. The structure of a rule can be depicted as follows:

```
<rule identifier>:<preconditions>
<rule modifier list>, <signal list>, <state transition>
```

The rule identifier indicates the port on a target atomic action to which a signal should be sent. Preconditions are predicates that have to be fulfilled before the corresponding rule is executed.

Rule modifiers capture the dynamic behavior of a transaction model, i.e. the addition or deletion of rules. The signal list contains names of rules that are to be activated in the course of execution of the originating rule. The rule modifier list contains one or more rule modifiers. State transition is a supplementary element that gives the rule a label. The structure of a rule modifier is the following:

```
<rule modifier> ::= ((+||-) (<rule identifier>|<signal>))
||
(delete (<Atomic Action Identifier>))
```

The first clause of a rule modifier introduces means to dynamically create new rules and dependencies introduced by these new rules. It is also used to delete single rules. The second clause is a shortcut and makes it possible to dynamically delete obsolete rules pertaining to a particular atomic action.

A transaction model consists of several such rules. Whenever an event occurs, the right side of the rule that identifies the event is executed — of course only if the preconditions of the rule are met. The rule is "marked" to indicate the current state. It remains marked after its execution steps are finished until a new signal comes in. Thus, subsequent emissions of the same signal to the same action are not possible, because the port is "closed" after

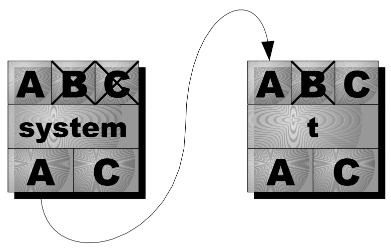

Figure 3.4: Single aspect of a flat transaction system

the first emission. Once an atomic action reaches a final state, all its rules are deleted. Note that delete actions are given only when they are essential for the described transaction model.

To illustrate Gray & Reuter's model, a simple transaction model is defined: *flat transactions*. In flat transactions we have two atomic actions: The flat transaction action itself and a system action. The `system` action can only be aborted, i.e. the system crashes for some reason. The flat transaction action can be committed and aborted. There is a dependency between the system action and the flat transaction action: If the system action aborts, the flat transaction action has to abort, too. The graphical rendering of the model depicted in Figure 3.4 describes a particular state of the flat transaction model. The figure would become too complex if we tried to describe the whole model with it. Textual rules are a better way to do that. Each atomic action has three ports (`Abort`, `Begin`, and `Commit`) and two states (`Aborted` and `Committed`). Transaction t is running, i.e. the `begin` port has been used.

Crossed–out ports in the figure cannot be used. Thus, the only ports that can be used in the current state are the `abort` port of atomic action `system` and the `abort` and `commit` port of atomic action t. If the system gets into the state `aborted`, the `abort` port of the t atomic action is signaled. This emitted signal implies an abort of t and, consecutively, a rollback of t. It should be noted that we expect a transaction that is aborted to perform a rollback. The "textual rules rendering" of the model is as follows:

S_A(system): , , System Crash (rule 1)

S_B(t): +(S_A(system)|S_A(t)), , Begin Work (rule 2)

S_A(t): (delete(S_B(t)), delete(S_C(t))), , Rollback Work (rule 3)

S_C(t): (delete(S_B(t)), delete(S_A(t))), , Commit Work (rule 4)

The notation $\texttt{S}_{\texttt{X}}\texttt{(j)}$ means signal `X` of atomic action `j`. The signals are abbreviated as follows: `A` is short for abort/rollback, `B` means begin, and `C` means commit. The first rule handles the case of a system crash: The system action is aborted. Since it does nothing, it is actually redundant. It is only given for the sake of clarity. The second rule installs the structural dependency of the atomic action `t` and the atomic action `system`, i.e. the arrow in Figure 3.4. The third rule is executed when an `abort` signal arrives. All ports are deactivated. The same is true in the case of a `commit` signal, which is written down in rule 4. Nested transactions (see Subsection 3.3.2) can be described with the following rules:

$\texttt{S}_{\texttt{B}}\texttt{(t}_{kn}\texttt{):}$ `+(`$\texttt{S}_{\texttt{A}}\texttt{(t}_{k}\texttt{)|S}_{\texttt{A}}\texttt{(t}_{kn}\texttt{))}$`, ,BEGIN WORK (rule 1)`

$\texttt{S}_{\texttt{A}}\texttt{(t}_{kn}\texttt{):}$ `, , ROLLBACK WORK (rule 2)`

$\texttt{S}_{\texttt{C}}\texttt{(t}_{kn}\texttt{):}$ $\texttt{C(t}_{k}\texttt{)}$`, , COMMIT WORK (rule 3)`

Rule 1 introduces a new atomic action and installs the dependency “if the parent atomic action aborts, the child atomic action has to abort, too”. Rule 2 establishes the abort signal and rule 3 manages the commit system: “The child can finally commit only if its parent has committed”. The rules use two atomic action identifiers: t_k and t_{kn}. t_k represents an arbitrary parent atomic action and t_{kn} an arbitrary child atomic action of t_k.

3.4.2 ACTA

ACTA is a framework that can be used to specify, analyze and synthesize advanced transaction models [13]. As in Gray and Reuter’s model, several transactions are combined to compose advanced transaction models in ACTA.

Basically, ACTA distinguishes between object events and significant events. Object events are calls on operations of objects. Significant events are invocations of transaction management primitives like `commit`, `abort`, etc. Besides that, ACTA uses the following building blocks to describe an advanced transaction model.

Inter–transaction dependencies are used to specify the relationships between transactions in a transaction model. For example, an `abort` dependency between transaction t_j and transaction t_i indicates that the abort of t_i causes the abort of t_j.

Views of transactions allow specifying the state of objects visible to a transaction at a point in time. For example, let us assume that under the control of transaction t_a an object event on object o_v has changed the state of o_v and t_a has not committed yet. Transaction views control whether it is possible for object events under a transaction t_b to operate on and/or see the not–yet–committed state of o_v.

With conflict sets it is possible to define that object events under control of a transaction cannot be called by another transaction while the object events are in–progress (in–progress object events are events that have been started but have not been committed or aborted yet).

Delegations move the responsibility for object events from one transaction to another and also delegate the responsibility for significant events from one transaction to another. Delegation of object o_d from transaction t_y to transaction t_x means that all method calls to o_d that happened before the delegation (that is t_y was de facto in control) are considered to have happened under t_x and t_x is responsible for doing significant events (e.g. `commit`) on o_d.

In [13], sample advanced transaction models are described using axioms that are expressed in predicate logic, whereas this predicate logic uses the building blocks of ACTA. The following axioms describe a simple atomic transaction:

1. $SE_t = \{\text{Begin, Commit, Abort}\}$
2. $IE_t = \{\text{Begin}\}$
3. $TE_t = \{\text{Commit, Abort}\}$
4. t satisfies the fundamental Axioms of Transactions
5. $View_t = H_{ct}$
6. $ConflictSet_t = \{p_{t'}[ob] \mid t' \neq t, \mathcal{I}nprogress(p_{t'}[ob])\}$
7. $\forall ob\ \exists p\ (p_t[ob] \in H) \Rightarrow (ob \text{ is atomic})$
8. $(\text{Commit}_t \in H) \Rightarrow \neg(tC^*t)$
9. $\exists ob\ \exists p\ (Commit_t[p_t[ob]] \in H) \Rightarrow (\text{Commit}_t \in H)$
10. $(\text{Commit}_t \in H) \Rightarrow \forall ob\ \forall p\ ((p_t[ob]) \in H) \Rightarrow (Commit_t[p_t[ob]] \in H))$
11. $\exists ob\ \exists p\ (Abort_t[p_t[ob]] \in H) \Rightarrow (\text{Abort}_t \in H)$
12. $(Abort_t \in H) \Rightarrow \forall ob\ \forall p\ ((p_t[ob] \in H) \Rightarrow (Abort_t[p_t[ob]] \in H))$

Axioms 1-3 define events and their purpose: Significant events (`begin`, `commit`, `abort`), a single initiation event (`begin`) and two termination events (`commit`, `abort`). Axiom 4 refers to basic transaction axioms, i.e. transactions can only be initiated by a single event and can only be terminated by a single event, termination can only occur if a transaction has been initiated before, and only running transactions can invoke operations on objects. Axioms 5 to 12 define further core semantics of atomic transactions. We will not describe these axioms in detail here. See [13] for a complete explanation of axioms 5 to 12.

The ACTA framework is a very comprehensive metamodel and it is unlikely that a particular idea for a custom advanced transaction model cannot be represented in ACTA. However, as one can see from the preceding example, this completeness causes complexity, and ACTA itself is neither easy to understand nor easy to use.

Specialized approaches for defining advanced transaction models that use ACTA have been proposed. For example, ASSET [6] and Bourgogne transactions [41] use the ideas of ACTA but simplify the usage of ACTA significantly. Both approaches are based on the idea to combine several basic transactions by defining their dependencies in order to compose an advanced transaction model. They also provide a set of general transaction primitives that can be applied to control transactions. These transaction primitives can be used in programming environments. ASSET focuses on O++, a database programming language, and Bourgogne transactions target the J2EE application server infrastructure. Thus, an advanced transaction model is defined by creating a program that defines one or more basic transactions, their dependencies (if necessary), and controls the transactions by means of transaction primitives.

3.5 Evaluation of Advanced Transaction Metamodels

To support arbitrary advanced transaction models (a key feature of Web service transaction systems as discussed in Section 1.1), a model that is capable to grasp semantics of advanced transactions is compulsory. To the best of our knowledge, two significant formal advanced transaction metamodels were developed in the past. These have been introduced before. In this section we analyze these advanced transaction metamodels regarding their usefulness for a Web service transaction system. To avoid redundant work, using one of these scientifically approved advanced transaction metamodels for Web service transactions is highly desirable. Thus, a decision will be finally made which of these transaction metamodels will build the base of a new approach for Web service transactions.

3.5.1 Gray and Reuter

For a Web service transaction system, Gray & Reuter's metamodel for advanced transaction models could be used to describe the application's transaction semantics. However, the rules as used in [24] are proprietary and uncommon in the Web service world. Moreover, the advanced transaction semantics have to be described in a machine–readable language. As shown below, the rule syntax satisfies these requirements only marginally. XML would be a superior solution. It is verbose enough to provide information for humans, and countless tools and libraries exist that ease the processing of XML. In addition, every standard in the Web service world is using XML, therefore, representing the model in any other way would be unreasonable. Thus, an XML representation of Gray & Reuter's model seems to be the best choice.

A straightforward approach to bring Gray & Reuter's model into the XML world is to map the rules in a one–to–one way. The "begin work" rule of a flat transaction as shown before would look something like shown in Listing 3.1.

Listing 3.1: Sample XML rule serialization

```
<rule stateTransition="begin␣work">
  <identifier>
    <signal type="begin">
      <atomicAction type="t"/>
    </signal>
  </identifier>

  <ruleModifiers>
    <add>
      <from>
        <signal type="abort">
          <atomicAction type="system"/>
        </signal>
      </from>
      <to>
        <signal type="abort">
          <atomicAction type="t"/>
        </signal>
      </to>
    </add>
  </ruleModifiers>
</rule>
```

For such a simple model, this one–to–one mapping seems to be appropriate. However, for more complex transaction models, this kind of mapping has deficits. For example, take a look at the one–to–one mapped commit rule of a nested transaction in Listing 3.2.

Listing 3.2: Simple nested transaction XML rule

```
<rule stateTransition="COMMIT␣WORK">
  <identifier>
    <signal type="commit">
      <atomicAction type="t" id="kn"/>
    </signal>
  </identifier>

  <precondition>
    <state type="commit">
      <atomicAction type="t" id="k"/>
    </state>
  </precondition>
</rule>
```

A human specialist could imagine that the identifier *kn* describes an arbitrary child atomic action and *k* its parent atomic action. The precondition for committing the child atomic action (i.e. the parent transaction has to be in the committed state) is not machine–readable, because the hierarchic relation between t_k and t_{kn} is not expressed in a machine–readable way. A similar problem arises when mapping flat transactions with savepoints (see Section 3.3.1) to XML in a one–to–one way. The abort rule and its one–to–one XML serialization for an arbitrary atomic action in a flat transaction with savepoints are as follows (let *r* be the target savepoint, i.e. the transaction should rollback to the atomic action identified by *r*), Listing 3.3 shows the corresponding XML serialization:

SA(r) : (r<S$_n$) $\rightarrow$, SA(S$_{n-1}$), ROLLBACK WORK

Listing 3.3: Savepoint transaction XML rule

```
<rule stateTransition="ROLLBACK␣WORK">
  <identifier>
    <signal type="abort">
      <atomicAction type="s" id="n"/>
    </signal>

    <arguments>
      <arg name="RollbackTargetSavepointAtomicAction">
        <constraint>
            RollbackTargetSavepointAtomicAction < Sn
        </constraint>
      </arg>
    </arguments>
  </identifier>

  <signalList>
    <emitSignal>
      <target>
        <signal name="commit">
          <atomicAction type="s" id="n-1"/>
        </signal>
      </target>
    </emitSignal>
  </signalList>
</rule>
```

Here we have an argument that defines the identifier more precisely, and a constraint, which describes the allowed values of the argument. The rule and consequently the one–to–one mapping defines this in a language that cannot be understood by a machine without difficulty. Thus, a comprehensive XML mapping should include machine–readable parameter–passing semantics, too. Another problem arises with the signal list. A human can interpret the target of the signal: the linear predecessor atomic action. Similar to the parent–child relationship problem above, the linear relationship is not expressed explicitly.

Hence we face two obvious key problems when translating Gray & Reuter's rules to XML in a straightforward one–to–one way: An explicit definition of relationship types (e.g. previous, parent, etc.) and some kind of parameter passing semantics is needed.

One has to consider that arbitrary transaction models can have arbitrary relation types between their atomic actions. While a set of basic relation types can be identified, an extension mechanism is required, too. The basic set of relation types can be separated into *linear* and *hierarchic* types. Linear types are `first`, `next`, `previous`, and `last`. The hierarchic types are `parent`, `child`, and `root`. Another special type is also needed: `self` for relations to the atomic action itself. Note that the basic set just supports transaction models that follow a linear or hierarchic structure. The names are self–describing and these basic types should be sufficient for quite a few transaction models — at least the set is sufficient for all ATMs presented in [24]. The commit rule of a nested transaction in XML is shown in Listing 3.4.

Listing 3.4: Nested transaction XML rule

```
<rule stateTransition="COMMIT␣WORK"
      xmlns:aaRelations="http://wstx.ec3.at/AA_relations">
  <identifier>
    <signal type="commit">
      <atomicAction type="t" id="kn"/>
    </signal>
  </identifier>
  <precondition>
    <state type="commit">
```

```
      <atomicAction type="t" id="k">
        <aaRelations:relationSpecification relatedTo="this"
          as="parent" />
      </atomicAction>
    </state>
  </precondition>
</rule>
```

As can be seen, the embedding of relation specifications is implemented with XML–namespaces. This provides a flexible extension mechanism for atomic action relation types. The `this` value in the `relatedTo` attribute represents the current rule. Of course, the semantics of a parent type in the `http://wstx.ec3.at/AA_relations` namespace has to be implemented in the processing software in order to do some "parent relation aware" processing. If this is the case, the processing software is aware that the parent has to commit first. Extension sets reside in other namespaces and are included by declaring their namespace and prefixing the corresponding relation element with the namespace shortcut. Again, the semantics of extension atomic action relation types has to be implemented in the processing software — at least if it is desired to process these new atomic action relation types appropriately.

Since the input parameters used in the ATM rules presented in [24] are only atomic action types, focus is laid on building a parameter passing system that considers just atomic action types for now. The allowed type of the atomic action that can be passed as well as constraints the passed atomic action instance has to respect must be known.
For the constraints, a similar technique as used before for the explicit definition of relationships is applied. It is sufficient that constraints are expressed in terms of relations to other atomic actions. Thus, the relation vocabulary is enhanced with `linearAncestor` (any previous atomic action), `linearSuccessor` (any subsequent atomic action), `treeAncestor` (on a higher tree–level), and `treeSuccessor` (on a lower tree–level). For instance, argument passing in the rollback rule of a transaction with savepoints is specified in Listing 3.5.

Listing 3.5: XML rollback rule savepoint transactions

```
<rule stateTransition="ROLLBACK␣WORK"
      xmlns:aaRelations="http://wstx.ec3.at/AA_relations">
...
  <identifier>
    <signal name="abort">
      <atomicAction type="s" id="n"/>
      <arguments>
        <arg type="s">
          <aaRelations:relationSpecification
```

```
            relatedTo="this" as="linearAncestor" />
        </arg>
      </arguments>
    </signal>
  </identifier>
...
</rule>
```

In [24], there is no explicit definition which states an atomic action can have and which signals it can accept. This is done implicitly by defining rules accordingly, i.e. if a rule is identified by signal S to atomic action t, it is assumed that t has the port S. Though it is redundant, an explicit definition of atomic action types used in the model should be added to enhance readability and to ease processing by software programs. State transitions are also defined implicitly in [24], i.e. if signal `commit` arrives at atomic action t, t gets into the `committed` state. This should be stated explicitly in the XML representation as well. A nested transaction atomic action can be represented in XML as follows:

Listing 3.6: Nested transaction XML rule

```
<signalType name="abort"/>
<signalType name="begin"/>
<signalType name="commit"/>

<stateType name="abort"/>
<stateType name="commit"/>

<atomicActionType name="t">
  <signals>
    <signal type="abort"/>
    <signal type="begin"/>
    <signal type="commit"/>
  </signals>

  <states>
    <state type="aborted"/>
    <state type="committed"/>
  </states>

  <transitions>
    <transition>
      <fromSignal type="abort"/><toState type="aborted"/>
    </transition>
    <transition>
      <fromSignal type="commit"/><toState type="comitted"/>
    </transition>
  </transitions>
</atomicActionType>
```

3.5.2 ACTA

In contrast to Gray and Reuter's advanced transaction metamodel, a one–to–one XML representation of ACTA is not eligible. The resulting complexity would be too cumbersome. There is a superior way to use the ACTA framework in the context of Web service transaction systems. Enhancing Web service orchestrations with arbitrary advanced transaction models in a way that is inspired by ASSET [6] or Bourgogne transactions [41] — both are based on ACTA — seems to be a desirable objective.

For example, ASSET uses multi–purpose transaction primitives to build arbitrary advanced transaction semantics in O++, a database programming language. Transaction primitives become an integral part of O++. Hence, O++ language constructs and transaction primitives are merged with the intention to jointly put advanced transaction semantics into operation.

Such an approach is called *programmatic transaction system* and works well in Web service environments, as shown by this work [2].

3.5.3 Summary

Gray & Reuter's metamodel is appropriate to describe the structure and basic ideas of advanced transaction metamodels. It can be especially useful to comprehend advanced transaction models in Web service systems. However, it is not well suited to facilitate developing transaction aware Web service systems directly. Major efforts would have to be taken to create a concept that enables a link between business logic and transaction semantics. For example, the model is not prepared to express that Web service ws_1 should be managed by transaction t_1. If Web service ws_1 fails for 3 times, it is considered to be unsuccessful and t_1 should be aborted. In that case, a transaction t_2 should be started that manages a Web service ws_2 and ws_2 should be called. Addressing all the limitations of the industrial Web service transaction proposals as discussed in Subsection 3.7 would require considerable efforts. Incorporation with Web service orchestrations — i.e. linking the business logic to transaction semantics — is not addressed and a well–known usage pattern is not deducible per default. Gray & Reuter's metamodel has not been developed to handle such aspects of transaction–related processing per default.

[2]A programmatic Web service transaction system is discussed in detail in Part II.

Although an exact one–to–one use of the ACTA framework would be rather ineffective to create a Web service transaction system, the ideas of ACTA can build a valuable base for Web service transactions. Using concepts of ACTA in a Web service transaction system similar as proposed in ASSET or in Bourgogne transactions is a promising undertaking. The limitations of the industrial Web service transaction proposals could be avoided. By integrating transaction primitives in the business logic, a well–known usage pattern can be provided and the incorporation of transactional processing in Web service orchestrations is enabled. Moreover, the usage of arbitrary transaction semantics is featured as an integral part.

Accordingly, it is sound to base a new approach of Web service transactions on the ideas of ACTA, similar as shown in ASSET and Bourgogne transactions. Such an approach requires significant fewer efforts to address the limitations in available Web service transaction proposals than advancing Gray & Reuter's metamodel.

3.6 Web Service Transaction Proposals

Several renowned companies published proposals that deal with transactional processing in the Web service world. These proposals are remarkably similar and differ only in details. The basic building blocks are the same.

3.6.1 Business Transaction Protocol

In the *Business Transaction Protocol (BTP)* proposal, each Web service is associated to a so–called coordinating entity. A transaction can contain several coordinating entities that are organized in a tree–hierarchy. Coordinating entities control their Web services in terms of transaction primitives and propagate transaction primitives to their sub–ordinate coordinating entity. Which transaction–primitives are sent to which Web services or subordinate coordinating entity is based on application–specific (business) logic, that is implemented in the coordinating entity. For example, a coordinating entity can abort all its Web services or subordinate coordinating entities if a single Web service or subordinate coordinating entity fails. In BTP, such behavior is called *atom* and implements atomicity in terms of the ACID properties. In contrast, a coordinating entity can tell — based on some (business) logic — some of its subordinates or Web services to abort and others to commit. This is called *cohesion*.

BTP uses transaction primitives as follows. BTP–enabled services must support provisional (tentative) state changes, called provisional effects. The provisional effect can be either made permanent (commit, in BTP called final effect) or canceled (counter effect). The actual implementation of these effects is up to the service. The counter effect can be accomplished by a rollback or a compensation. Isolation is not covered in BTP. It is up to the service how to manage isolation matters. The client decides whether a resource is visible during a transaction to all clients and how to deal with implications of this decision.

The integration in existing distributed environments is rather flexible. There are bindings for different network protocols. As for the SOAP binding, coordinating entity messages like "prepare" are sent in the SOAP body. Regarding application calls, a transaction context is transferred to the participants in SOAP header elements. Thus, a Web service can relate application calls to particular transaction instances.

3.6.2 WS–Transactions and WS–Coordination

This proposal is organized in two parts. One part is a framework for managing the transaction context between the participating activities (WS–Coordination) and the other part deals with transaction protocols (WS–Transaction).

WS–Coordination defines a framework for coordinating components (coordinators) in a transaction. It defines a coordinator's offered functionality and interactions. A coordinator produces a (transaction) context, called coordination context. The context is propagated to all involved components and contains a coordination type, which has to be supported by all components that participate in the transaction. The context is transmitted in the SOAP header and conversation flow works almost the same as in BTP: An application receives a SOAP call that includes transaction context information. Based on this information, the application registers to a coordinator and transaction specific communication with the coordinator is conducted based on protocols that are deduced from transaction models.

WS–Transaction defines communication protocols for particular transaction models. Again, the (advanced) transaction models are defined only implicitly by the communication protocol. A transaction model can offer several protocols, and a component can choose one or more protocols. Based on the component's registered protocol(s), the coordinator sends protocol conformant messages to the components. The communication protocol is arbitrary. Thus, the WS–Transaction and WS–Coordination infrastructure can be used for custom transaction model communication protocols. However, WS–Transactions defines two potentially often–used types.

The transaction model *atomic transaction (AT)* is an all–or–nothing transaction model. It fulfills the atomicity property of the ACID properties. AT offers 5 protocols that can be used by participants. For example, if a component registers to the completion protocol, the component is able to finish the transaction. It can tell the coordinator either to try to commit the transaction or force a rollback. After that, the coordinator sends a message with the transaction outcome (committed or aborted) to the participant and “forgets” about the transaction.

For long–lasting transactions, where hard isolation operations are not applicable and trust between participants is not guaranteed, the coordination type *business activity (BA)* is proposed. This type builds up a transaction tree to handle such cases. The tree can consist of both, ATs and BAs. The tree enables superordinate coordinators to control their subordinate coordinators and divides the transaction into several domains. Note that the term “control subordinates” means handling of exceptions thrown by subordinates, selection of only a few subordinates that should contribute to the overall outcome and giving transaction specific orders like “complete” or “compensate”. As in BTP, (business) logic has to be applied to coordinators, so that they know, for example, which subordinates are vital for the whole transaction. To reverse actions of participants, compensation is used.

3.6.3 Web Service Composite Application Framework

The *Web Service Composite Application Framework (WS–CAF)* contains recommendations for coordinating several Web services using transactional processing. The architecture consists of three different parts.

A Web services context service (WS–CTX) provides means to share information between several Web services that collaborate in order to accomplish a specific activity. This information is called context and is shared within those Web services.

Generic coordinating entities are defined according to the *Web Services Coordination Framework (WS–CF)* proposal. The task of a generic coordinating entity is to disseminate information to a number of participants. It should be noted that generic coordinating entities might be used for any coordination activity and are not focused on coordinating transaction processing only. WS–CTX contexts are used here to exchange information about the coordinating entities. For example, the question "How can the coordinating entity be reached?" might be answered by a WS–CTX context.

Web services transaction management (WS–TXM) defines transaction protocols. These transaction protocols are executed using the infrastructure of WS–CF and WS–CTX. Coordinators as defined in WS–CF, manage the execution of transaction protocols. For example, a tree structure of a transaction may be created using subordinate and superordinate coordinating entities. At run–time, these coordinating entities control Web services according to the hierarchy. Hence, they implement the transaction semantics. Contexts, as defined in WS–CTX, convey context information in the transaction management system. For instance, if a Web service call that comes in includes a WS–CTX context, the participating Web service may react in accordance to the context. The context could carry information about the desired transaction model and, depending on the transaction model's requirements, the Web service may register itself to a coordinating entity as also proposed in the context.

WS–TXM proposes three different transaction protocols. The ACID Transactions (AT) model provides ACID semantics for transactions. There are some tweaks to enhance the performance. For example, a Web service can indicate that it has just read operations to perform, and then is treated specifically by the coordinating entity.

For business interactions that occur over a long period, the *long running action (LRA) transaction model* is available. This model makes one important assumption about the participant's work: It has to be compensatable through compensation actions. LRAs define triggers for compensation actions and the conditions under which those triggers are executed. When a Web service participates in an LRA, it enlists a compensator Web service that is able to compensate the Web service's actions. LRAs can be nested. If an LRA inside an enclosing LRA fails, all LRAs inside the enclosing LRA have to be compensated. Thus, LRAs respect the atomicity property. However, a typical long running business interaction involves more LRAs that are not nested inside of each other. Obviously, the idea of LRAs is based on SAGAs (see Subsection 3.3.4).

The *Business Process Transactions (BPT)* transaction model is specializing primarily in inter–organizational transactions. It consists of business tasks and each task executes within a specific business domain. A BPT can either terminate successfully (i.e. all of the work could be done) or unsuccessfully (i.e. at least one task failed). In the latter case, the transaction moves to a "failure" state. In this state, work can be undone if necessary: Some tasks can be told to undo and some tasks can be told to commit. Thus, atomicity is relaxed. If some work cannot be undone, this has to be logged and further compensation operations like human intervention (business–level compensation) have to be applied. How a task undoes its work is not specified. It can be done using compensation actions like in LRAs or by some other means. Again, a tree–structure models the transaction. Each business domain is exposed as a single subordinate. Each subordinate is responsible for managing its domain in terms of making it to cooperate with the business transaction. The internal implementation of the domain is not specified and is not necessarily WS–TXM. It can also use BTP or WS–Transaction/WS–Coordination as long as its coordinating entity maps BPT messages to its implementation messages and vice versa. The basic ideas of BPT originate from multilevel transactions. In contrast to multilevel transactions, semantics of undo actions were enriched and the tree structure of the transaction does not imply cascading undo/compensation structures. What should be undone can be determined without the rules of any underlying model.

3.7 Evaluation of Web Service Transaction Proposals

As mentioned before, the published proposals for Web service transactions resemble each other to a high degree. Analyzing the participating organizations and release dates, this is no surprise. Table 3.1 shows the release dates and the most important participating organizations of each Web service transaction proposal.

BTP was the first proposal for Web service transactions, with Oracle, Sun, BEA and others involved. Only one month later, IBM and Microsoft published a competitive proposal, whereas BEA switched and obviously provided extensive know–how that originated from BTP. One year later, Oracle and Sun tried again and published WS–CAF, challenging WS–Transactions and WS–Coordination.

All these Web service transaction proposals describe an architecture of a Web service transaction system, which includes participating Web services

Table 3.1: History of Web service transaction proposals

Proposal	**Published on**	**Organizations**
Business Transaction Protocol	June 3, 2002	OASIS (Oracle, Sun, BEA, HP, Iona, Sybase)
WS–Transactions & WS–Coordination	August 9, 2002	IBM, Microsoft, BEA
Web Services Composite Application Framework	July 28, 2003	Oracle, Sun, IONA

and a hierarchy of central components that offer transaction–related services and communicate transaction–related matters to affected participating components. Based on this framework, different protocols that handle communication between all affected components are defined. These protocols adhere to certain transaction semantics. Thus, if protocol p_x conforms to transaction semantic s_x, and a transaction t_a is executed using p_x, the semantics of t_a conform to s_x. The offered transaction communication protocols are based on the semantics of advanced transaction models, as described in Section 3.3. Each proposal offers a small number of ready–to–use transaction communication protocols. The usage of additional arbitrary advanced transaction communication protocols is only possible with WS–Transactions/WS–Coordination and Web Service Composite Application Framework. Frankly spoken, the only relevant diversity of the available Web service transaction proposals lies in the offered transaction protocols.

Unfortunately, all existing Web service transaction proposals introduced in the following section suffer from weaknesses as follows.

These Web service transaction proposals focus on the specification of communication protocols between a transaction enabled Web service and transaction monitors. Such protocols exist for a few number of different (advanced) transaction models. Thus, certain default advanced transaction models are described implicitly by suggesting suitable communication protocols for them. However, the incorporation of custom advanced transaction semantics into the transaction system is left open to a high degree. On the whole, the software industry is aware of that important requirement because the Web service transaction system proposals WS–Transactions/WS–Coordination and WS–CAF support the idea of incorporating arbitrary advanced transaction models. However, to incorporate custom advanced transaction semantics, it is necessary to introduce a new communication protocol. How this can be achieved is not described in the proposals. In addition, this seems to be unwieldy. It is likely that most of the transaction system has to be modified in order to be aware of such new communication protocols. This would take huge efforts and would discourage users from using domain–specific semantics.

The architecture and communication protocols in the proposals are well specified. However, all proposals fail to present a usage pattern. It is well described that there is a network of transaction components, some participants of a transaction, and communication protocols, but it remains nebulous how an application can use such a transaction system. In contrast, if an application programmer wants to use ACID transactions in a usual transaction environment, a ubiquitous pattern is used as described in detail in Subsection 3.2.1. However, the existing proposals for Web service transactions do not provide such a usage pattern. There is no clear understanding on how the application programmer can use Web service transactions. Such an understanding would support basic ideas of transaction–oriented processing, namely "relieving the application programmer from worrying about failure and concurrency interleaving" [20]. The proposals refrain to discuss the view of the user who wants to use transaction–oriented processing for Web service computing. Only the Web service transaction system architect's view is discussed. Thus, a majority of potential Web service transaction users is not addressed.

Considering Web service orchestrations, using transaction concepts would facilitate their implementation and operation significantly as discussed in Section 1.1. Neither the Web service transaction proposals nor common Web service orchestration technologies take transactional processing into account sufficiently. To the best of our knowledge, only BPEL4WS tries to offer transactional concepts by offering explicit compensation operations. This is a beginning, but far away from a satisfying support of Web service transactions in Web service orchestrations. Thus, there is a significant gap between current proposals for Web service orchestrations and Web service transactions.

There are only few implementations for the proposals discussed above. The commercial Arjuna Transaction Service product [5] stands above all other implementations. It provides production grade implementations of

WS–CAF, WS–Transaction/WS–Coordination and also BTP[3]. For BTP, there is a technology preview from Hewlett Packard [25] and an extension of an open–source transaction environment by ObjectWeb, JOTM [37]. However, both projects seem to cease to exist sooner or later because almost no activity and interest is visible. For WS–Transaction/WS–Coordination IBM Alphaworks offers a technology preview [34] of the atomic transaction model for the Websphere application server. Apache Software Foundation develops Kandula [4], an open source implementation that currently only supports atomic transaction. However, Kandula shall support business activities in the future.

Just few usable implementations exist and Web service transactions do only gain marginal attention in the field of Web service computing. Compared to other advanced Web service proposals (e.g. proposals for Web service orchestrations) that certainly matter in the Web service area, Web service transaction proposals stagnate on a rather low level. We believe that the weaknesses discussed in this section contribute to the virtually non–existing adoption of transactions in the Web service world.

[3]BTP is no longer available on a regular basis but special arrangements seem to be possible, though.

Part II

Transactional Web Service Orchestrations

Chapter 4

Introduction into TWSO

TWSO — an acronym for transactional web service orchestrations — is a model–based and programmatic approach for doing transaction oriented processing in Web service computing. TWSO offers a model that captures essential transaction–related concepts that are well suited for Web service environments. In contrast to other approaches, TWSO tries to resemble current (programmatic) methods to do transaction oriented programming as far as possible in order to keep familiarization efforts for new users low. TWSO is based on a sound scientific foundation because its fundamental ideas stem from ACTA [13], a well–known and accredited formal metamodel for advanced transaction models. In addition, existing applications of ACTA, i.e. ASSET [6] and Bourgogne transactions [41], also inspired TWSO.

To pay tribute to the special requirements Web services pose on transaction semantics, TWSO enhances the current usage pattern of transaction–oriented processing as shown in Subsection 3.2.1 significantly. On the one hand, the set of possible transaction commands (i.e. transaction primitives) is enhanced. Standard `begin`, `abort`, and `commit` is not sufficient for Web service systems. On the other hand, TWSO features the possibility to build up comprehensive advanced transaction semantics by orchestrating several "small–scale" transactions.

It should be noted that TWSO defines primarily concepts of a Web service transaction system. These concepts can be brought into existence by implementing them for a particular Web service orchestration technique. For example, a TWSO Java API is a manifestation of TWSO concepts for Java Web service orchestrations while TWSO XML entities can manifest TWSO concepts for XML Web service orchestrations. By this means, TWSO can be integrated with virtually any Web service orchestration technology.

Figure 4.1 shows an overview of the architecture of a TWSO environment. There are three major areas of a TWSO environment: The orchestration system, the transaction monitor, and the involved Web services.

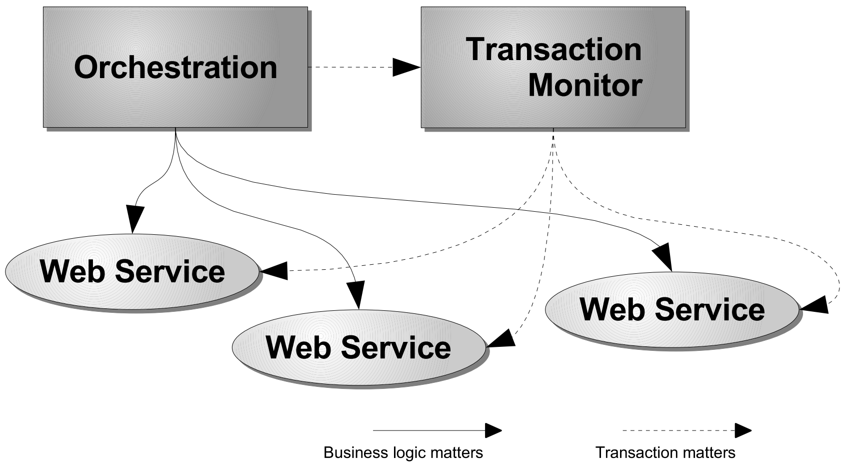

Figure 4.1: Architecture of a TWSO system

The orchestration system combines Web services in terms of workflow, i.e. it directly calls Web services in some particular order and thus invokes business logic. In addition, the orchestration system provides possibilities to setup transactions, to setup dependencies between transactions and to use transaction primitives on transactions. Such operations imply a central component that handles transaction specific tasks, a transaction monitor. The transaction monitor acts on the transaction matters that occur in the orchestration and takes charge of execution of transaction specific processing. Depending on the transaction system's status and the particular transaction matter, the transaction monitor may forward transaction primitives to affected Web services. Web services in a TWSO transaction system must conform to some requirements. They have to be able to communicate with the transaction monitor and act on the transaction monitor's commands, i.e. transaction primitives, in a sound way. Moreover, Web services in a TWSO environment must be able to associate each business logic call they receive with a particular transaction[1].

[1]This requires additional information in each Web service call that should be managed by a transaction. In TWSO, this is handled by using a transaction context, as discussed in Section 5.

There are three significant interfaces between the components of a TWSO system. To establish transaction related logic in an orchestration, the orchestration's specification (e.g. an XPDL XML document or Java source code) has to include means to specify programmatic transaction related actions. Accordingly, an interface between an orchestration's specification and an orchestration's execution engine has to be present. The orchestration's execution engine communicates transaction related matters to the transaction monitor. To do so, an interface between the transaction monitor and the orchestration engine is necessary. The transaction monitor manages TWSO enabled Web services in terms of transactional issues. Such a Web service provides an interface to facilitate the communication between the transaction monitor and itself.

This part of the thesis discusses the different components of a TWSO environment thoroughly. First, the technique to share a transaction context in a TWSO system is discussed in Section 5. TWSO concepts that are related directly to Web service orchestrations are presented in Section 6. In Section 7, TWSO transaction monitors are described. Finally, requirements for TWSO enabled Web services are specified in Section 8.

Chapter 5

Ubiquitous Transaction Presence

Many actions in a TWSO system require information about corresponding transaction instances. A ubiquitous existence of a context (as defined in Definition 5.1) that includes information of involved transactions would be useful. Such a context can be appended to all kinds of communication in a TWSO system to specify that transaction instances need to be considered for intended actions. For example, when a Web service receives a call, it has to know whether it has to consider a transaction and — if so — which transaction instance it needs to consider. If there is a transaction context included, transaction related actions like generating only tentative and tracelessly undoable state changes may be necessary.

Definition 5.1. A *context* contains information about the execution environment of a series of related interactions with a set of Web Services. Context information supplements information in application payloads [9].

Since all communication is handled using SOAP, the transaction context has to be inserted in a SOAP call. This is achieved by adding SOAP headers to SOAP messages that contain the transaction context. The transaction context is manifested in XML and specified as shown in Listing 5.1.

Listing 5.1: Transaction Context

```
<element name="TransactionContext" type="twso:TransactionContext" />

<complexType name="TransactionContext">
        <sequence minOccurs="1" maxOccurs="unbounded">
                <element name="Transaction" type="twso:Transaction"/>
        </sequence>
</complexType>
<complexType name="Transaction">
```

```
        <sequence minOccurs="1" maxOccurs="unbounded">
                <element name="SessionID" type="string"/>
                <element name="TransactionMonitorURI" type="string"/>
                <element name="TransactionID" type="string"/>
        </sequence>
</complexType>
```

A transaction context can include one or more transaction identifiers. It is possible, that a communication refers to more than one transaction. A transaction specification has to include a URI that identifies the transaction and a URI that identifies the transaction monitor that is responsible for the transaction. In addition, this URI is the SOAP endpoint that enables communication with the concerned transaction monitor. To keep track of client sessions, a session identifier may be included in the header. Thus, the transaction monitor is always aware of the transaction communication related to a particular client for the duration of a particular session. For example, the transaction monitor is always able to relate a set of running transactions to a particular client.

A Web service that receives a call to its business logic that includes a transaction context is able to unambiguously identify the corresponding transaction instance. Moreover, the Web service is also able to communicate matters that are related to the transaction instance with the concerned transaction monitor.

Listing 5.2 shows a SOAP message that triggers a flight booking. It contains a transaction context in the header. The receiver of the message has to interpret the message as follows: A flight booking has to be executed under control of transaction `423db171:108fd3442d7:-7fff` that stems from transaction monitor `http://move.ec3.at/services/twsoMonitor`.

Listing 5.2: TWSO Header SOAP Message

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">

<soap:Header>
  <twso:TransactionContext xmlns:twso="http://move.ec3.at/twso">
    <twso:Transaction>
        <twso:SessionID>
            423db171:108fd3442d7:-7ff5
        </twso:SessionID>
        <twso:TransactionMonitorURI>
            http://move.ec3.at/services/twsoMonitor
        </twso:TransactionMonitorURI>
        <twso:TransactionID>
            423db171:108fd3442d7:-7fff
        </twso:TransactionID>
    </twso:Transaction>
  </twso:TransactionContext>
</soap:Header>

<soap:Body>
        <ns1:bookFlight xmlns:ns1="urn:BookFlightService">
                <param_1 xsi:type="xsd:string">someValue</param_1>
                ...
                <param_n xsi:type="xsd:string">someValue</param_n>
        </ns1:getEmployeeDetails>
</soap:Body>

</soap:Envelope>
```

Chapter 6

TWSO Orchestrations

In this section the orchestration's view of the concepts of a TWSO transaction system is presented in detail. These TWSO concepts are inspired by the ideas of [6, 13, 41]. TWSO orchestration concepts consist of three building blocks.

Transaction primitives are fundamental commands that control particular transactions. Analogous as in ACTA, there is a fundamental discrimination between Web service calls and transaction primitives. Web service calls are operations on the Web service state. The final outcome of these operations may be influenced by transaction primitives. For example, the transaction primitive `abort` on transaction t_i that manages Web service s_u may undo all changes of the call of Web service s_u.

Transaction dependencies model interaction and organization of transactions in a TWSO orchestration. The design of ACTA and Gray and Reuter's transaction metamodel [24] is followed. A fundamental part of advanced transaction semantics is captured by creating compositions of multiple individual transactions and — if necessary — dependencies amongst them. Altogether, the combination of transaction primitives and interdependencies of transactions in an orchestration implements arbitrary advanced transaction models, of course also including the ones introduced in Section 3.3. It should be stressed that in contrast to the other Web service transaction approaches, there is no need for any modifications in the transaction system when using new transaction models in TWSO.

Furthermore, TWSO defines solely a set of concepts for Web service transaction systems. These concepts can be employed in different orchestration environments. In order to use these concepts, they have to be manifested

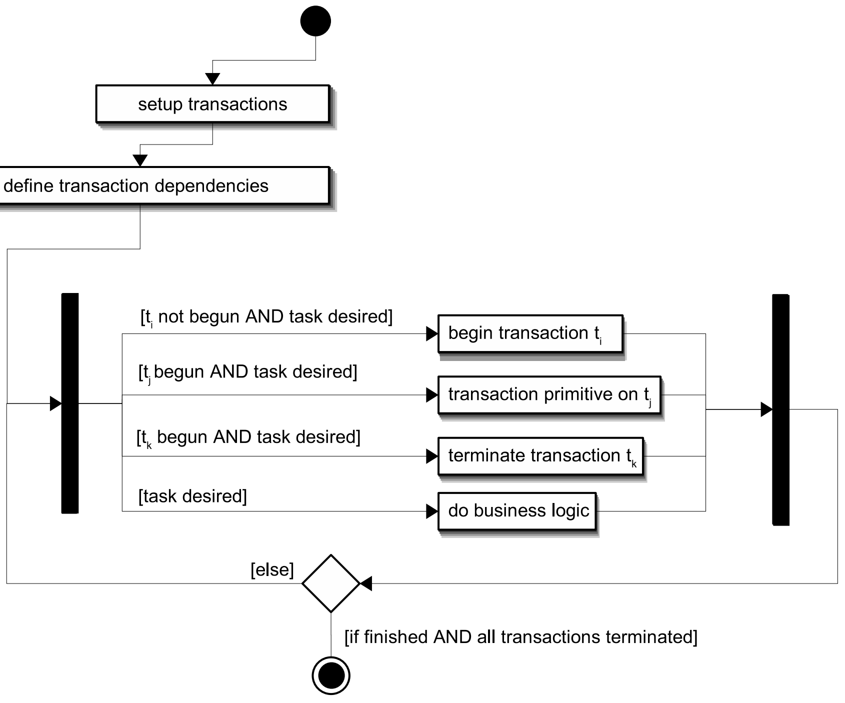
setup transactions
define transaction dependencies
[t_i not begun AND task desired]
begin transaction t_i
[t_j begun AND task desired]
transaction primitive on t_j
[t_k begun AND task desired]
terminate transaction t_k
[task desired]
do business logic
[else]
[if finished AND all transactions terminated]

Figure 6.1: TWSO usage pattern

in a form that integrates seamlessly with the particular orchestration environments. Thus, TWSO can be used in various orchestration techniques, provided that an implementation of TWSO concepts exists for the particular orchestration technique. To start with, two implementations are presented. A generic one for XML Web service orchestrations (see Subsection 6.4.1) and another for Java Web service orchestrations (see Subsection 6.4.2).

6.1 TWSO Usage Pattern

The TWSO usage pattern in Web service orchestrations roughly corresponds to the activity diagram shown in Figure 6.1.

The first action of the TWSO pattern is the setup of all participating transactions. Furthermore, Web services are associated to transactions, in order to declare which Web services are managed by which transactions. Then, dependencies between the transactions may be defined. For example, it could be stated, that if transaction t_i aborts, t_j has to abort, too. After this setup work, the real work can be done. This work consists of beginning transactions, doing business logic, issuing transaction primitives on transactions and terminating transactions. These tasks may occur in arbitrary sequence and even concurrently, provided that some basic validity constraints — e.g. a transaction can only be terminated when it has been already started — are regarded.

Examining the TWSO usage pattern it becomes clear, that the common usage pattern that is used in traditional transaction processing systems is only enhanced in TWSO. The basic paradigm, i.e. embedding business logic in a transactional embracement, remains the same. In addition to the common usage pattern, in TWSO it is possible to work with multiple transactions in concert. By orchestrating transactions through defining dependencies amongst them, it is possible to form arbitrary advanced transaction semantics. This is an inherent feature of TWSO and absolutely no part of the transaction system has to be updated to use new transaction semantics. Furthermore, TWSO provides significantly more transaction primitives.

6.2 Transaction Primitives

TWSO is based on a basic set of transaction primitives, commands that influence particular transactions and thus may influence the state of used Web services indirectly. Transaction primitives are directed primarily to particular transactions. However, depending on various circumstances in the transaction system (i.e. mainly the type of transaction primitive and the current transaction's state), the transaction primitive (or better the transaction primitive's intention) may be forwarded to affected Web service(s). For example, let t_i denote a transaction and s_u a Web service, and t_i manages s_u. The transaction primitive p_{abort} on t_i may finally impose, that s_u has to do some actions that resemble abort semantics. If a transaction primitive has the ability to be forwarded to a Web service, it is called *Web service effective*.

Solely the Web service is responsible for taking the right steps according to a received primitive. For example, a `commit` on a Web service that does not manipulate any persistent data may trigger no action at all and is correct. In a loosely coupled system like the Internet, where Web services reside, it is unlikely that the Web service user can practice any kind of control on the used Web service. Thus, as one has to trust that the Web service fulfills the business logic correctly, one also has to trust that it fulfills the transaction logic correctly.

Table 6.1: Transaction primitives and categorization

Transaction Primitive Group	**Transaction Primitive**	**Web Service Effective**
Initiation Primitives	begin	✓
In–Progress Primitives	delegate_termination	✗
Termination Primitives	abort	✓
	commit	✓
Post–Termination Primitives	compensate	✓

In the special case, when a transaction primitive fails — for example, the used Web service is of bad quality and some special circumstances provoke a runtime exception on a `commit` — the transaction has to be transitioned to an aborted state. The Web service is responsible to clean up the consequences in order to guarantee that the aborted state of the transaction is consistent with the actual state of the Web service. A thrown exception notifies the orchestration in such a case. Thus — if needed — actions can be arranged in the orchestration to handle such cases properly.

A basic set of transaction primitives presented below should be sufficient for many applications. However, if needed, the set may be enhanced. It should be noted that all participating components (e.g. transaction monitors, Web services, etc.) have to be aware of the new transaction primitives and would need according modifications.

Transaction primitives can be classified as follows. There are initiation primitives, in–progress primitives, termination primitives and post–termination primitives. Table 6.1 shows the basic set of transaction primitives, their categorization, and Web service effectiveness.

`begin` starts a transaction. After the transaction has started, business logic managed by the transaction can commence, i.e. Web services can be called and will be under transaction control.

To assign the responsibility of termination of a transaction to another transaction, the `delegate` termination primitive can be applied during the run–time of a transaction. For example, in nested transactions (see Section 3.3), `delegate` would be issued when a child transaction is ready to commit. The parent transaction takes command over termination of its child transaction. For example, when the parent transaction receives a commit, it will also `commit` its child transaction. Only termination obligations — i.e. `abort`, `commit` and also post–termination obligations like `compensate` — can be delegated. It is not possible to delegate just a single termination obligation.

After the business logic has been completed, the transaction has to be terminated. This is done using termination or post–termination primitives. Which primitive will be used depends on the outcome of the business logic. `commit` may be issued if the transaction's outcome is considered to be successful and should be finished. A Web service may, for example, persist the transaction's changes and make them visible to all users, if it receives a `commit` task.

If the outcome of the business logic is considered to be unsuccessful, `abort` may be issued to indicate that the transaction should be aborted. The affected Web service will do necessary actions that conform to the semantics of an `abort`. For example, a classic rollback could be executed by the Web service to undo the changes that happened during the run–time of the corresponding transaction.

In case the changes of an already committed transaction should be revoked, `compensate` is used to perform some forward actions that neutralize the already persisted actions. In case `compensate` is used before a transaction has been committed, the Web service decides how to act accordingly. It is reasonable that the `compensate` primitive will invoke standard `abort` actions in such a case.

6.2.1 Transactions Managing Multiple Web Services

In TWSO, a transaction can manage one *or more* Web services. The possibility to manage several Web services with a single transaction causes noteworthy consequences.

When a transaction primitive is issued to a transaction that manages multiple Web services, atomic behavior is induced and additional coordination efforts may be necessary. This depends on the kind of transaction primitive. Transaction primitives that most likely require advanced coordination efforts are Web service effective termination primitives. In the basic set, `begin`, `abort`, `commit` and `compensate` are affected. When these transaction primitives are issued to transactions that manage more Web services, they are treated — so to speak — atomically amongst the involved Web services.

Hence, it must be assured that such a primitive that is forwarded to more Web services at once leads to a consistent and valid result. This result has to be agreed among all involved Web services.

There are reliable distributed algorithms that can solve such problems. For example, the well–known two phase commit protocol, that is presented in detail in Subsection 3.2.3, can be applied successfully here.

6.2.2 Transaction Lifecycle

During its lifetime, a transaction runs through various states whereas transaction primitives cause state transitions. Only particular state transitions are valid. Accordingly, transaction primitives cannot be issued in arbitrary order because they could cause invalid state transitions. Thus, states of a transaction, valid transaction primitives on this state and state transitions based on the issued transaction primitive are defined.

The following states are defined. `initiated` indicates that a transaction was set–up. After a transaction has been begun, it is in the `in-progress` state. Based on the kind of termination, a transaction can be `committed` or `aborted`. The post–termination primitive `compensate` brings a transaction to the `compensated` state. If delegation has been applied, the corresponding transaction (i.e. the transaction from which termination obligations have been withdrawn) goes into the `delegated` state.

Figure 6.2 depicts transaction states and all valid state transitions. Regular arrows depict transaction state transitions based on an issued transaction primitive. Dashed arrows depict valid transaction primitives on the corresponding state that, however, cause no state transitions. For example, issuing a `commit` on a transaction that has already committed is considered as valid, but won't have any effect in the transaction system. The transaction system may intercept such useless commands and react by sending out only a warning without — if the transaction primitive is Web service effective — bothering the related Web service.

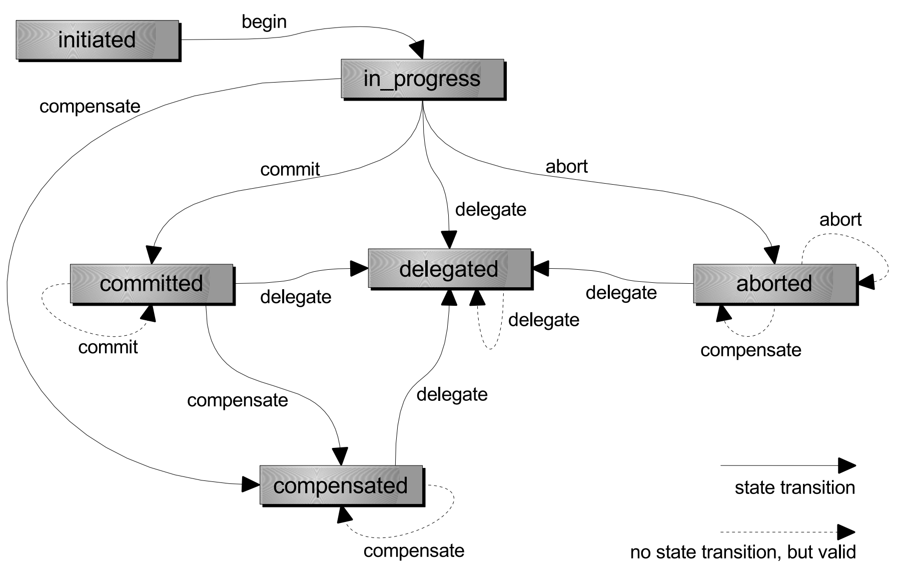

Figure 6.2: TWSO state–transitions

6.3 Transaction Dependencies

The foundation of TWSO was set by defining a collection of basic transaction primitives. The set of basic transaction primitives was enhanced to be appropriate for Web service transaction systems. Thus, a model for Web service transactions that strictly follows the common transaction usage pattern as described in Section 4 is already available.

It is possible to synthesize advanced transaction semantics using these primitives and (massive) explicit business logic of the orchestration technique. For example, to express the parent–child dependencies in nested transactions, one would have to perform something like shown in Listing 6.1 explicitly.

Listing 6.1: Parent Child Dependency in Business Logic

```
if (state(t_parent) == ABORT) {
    abort(t_child);
}
```

However, such an approach is not preferable. The significantly increased number of transaction primitives and the for advanced transaction semantics necessary concurrent use of multiple transactions impose a major growth of complexity. Using various sophisticated transaction primitives in multiple transactions concurrently instead of just using `begin`, `commit` or `abort` in a single transaction makes a significant difference. Moreover, using the basic set of transaction primitives and business logic to synthesize advanced transaction semantics mingles business logic with transaction logic to a high degree. Obviously the *separation of concerns* principle [15] is violated.

It is desirable to separate transaction from business logic as much as possible. A possibility to do so is specifying certain dependencies between transactions separately.

In TWSO, dependencies between transactions are defined as follows. A dependency consists of two parts. The *source state* specifies a particular state of the transaction system. If the transaction system gets into this state, an *effect* is triggered. The source state is composed by the state of a single transaction or a combination of transaction states. The combination of transaction states is constructed by using the logical operators *and* ($\wedge$), *or* ($\vee$) and *not* ($\neg$). For example, let t_1, t_2, and t_3 be three different transactions and a state of a transaction be $\sigma_{state\_type}(t_i)$. Possible source states could be as follows:

- $\sigma_{aborted}(t_2)$
- $\neg\ \sigma_{aborted}(t_2) \wedge (\sigma_{compensated}(t_1) \vee \sigma_{committed}(t_3))$

The first source state describes the transaction system state that comes into effect when t_2 got aborted. The second one specifies the transaction system's state when t_2 is not aborted, and t_1 is compensated or t_3 is committed. Obviously, the second source state is a combination of states of several transactions.

The effect, the second part of a rule, is the issuing of transaction primitives on one or more transactions. For example, let t_4 and t_5 be two different transactions and the issuing of a transaction primitive on a transaction t_i $p_{primitive\_type}(t_i)$. For example, effects may be as follows:

- $p_{compensate}(t_4)$
- $p_{abort}(t_4), p_{commit}(t_5)$

The first effect triggers a `compensate` primitive on transaction t_4. The second one triggers more transaction primitives, an `abort` on t_4 and a `commit` on t_5.

The concepts of transaction dependencies should be illustrated by the following example. A part of the desired advanced transaction semantic should state that as soon as t_1 gets into state $\sigma_{aborted}$ and t_2 gets into state $\sigma_{compensated}$, p_{commit} should be issued on t_3 and t_4. It is possible to express this in the business logic solely. Using Java as the orchestration technology, this could be expressed as depicted in Listing 6.2.

Listing 6.2: Transaction Dependency Using Java Only

```
if ( stateobserver.getState(t1) == States.ABORTED  &&
     stateobserver.getState(t2) == States.COMPENSATED )
{
     transactionMonitor.commit(t3);
     transactionMonitor.commit(t4);
}
```

Here, a `stateobserver` object tests the states of transaction t_1 and t_2. If both get into the desired states, a `transactionMonitor` object commits t_3, t_4. However, it should be noted that this is just a fragment of code that has to be placed in a "suitable place" in the orchestration definition. Finding or preparing such a place is not a trivial task. Besides the actual business logic of the orchestration, one would have to make an additional concurrent execution path that involves a loop that observes the states of the dependencies transactions permanently. This loop would be stopped when the source states of the dependency go into effect (the corresponding significant events are issued) or when the dependency is considered to be not relevant anymore (e.g. when the business logic is finished). In Java, this task can be tackled by using various low–level Java language constructs. In contrast, for XML Web service orchestrations that are on a much higher level, one has to get by with significantly limited constructs and will have a hard time to specify the desired semantics. Accordingly, massive efforts in the business logic are necessary to implement such advanced transaction semantics and business logic is mingled with transaction logic.

Using a transaction dependency as described in this section, the only task is to specify it in some place and in some syntax. A formal representation of the example dependency is $\sigma_{aborted}(t_1) \wedge \sigma_{compensated}(t_2) \Rightarrow p_{abort}(t_3), p_{abort}(t_4)$. This formal rule can be manifested into a form that is usable for the used orchestration technique. XML elements are a way as well as other representations. The dependency could be expressed in Java as shown in Listing 6.3. It should be noted that for the sake of readability, the definition of the dependency string does not fully conform to Java syntax.

Listing 6.3: Transaction Dependency in Java

```
String dependencyString =
<dependency>
    <sourceState>
        <concatenation type="and">
            <state type="aborted" transaction="t1"/>
            <state type="aborted" transaction="t2"/>
        </concatenation>
    </sourceState>

    <effect>
        <primitive type="abort" transaction="t3"/>
        <primitive type="abort" transaction="t4"/>
    </effect>
</dependency>
;

Document dependencyDoc = XMLHelper.stringToDoc(dependencyString);

transactionMonitor.setDependency(dependencyDoc);
```

Using an explicit transaction dependency definition as shown in Listing 6.3, business logic is separated from transaction logic to a significant degree. The transaction system is responsible to monitor source states and trigger effects simultaneously while the business logic executes. No additional efforts have to be undertaken in the business logic. Moreover, separation of concerns is honored and modification of transaction semantics does not impose significant changes of business logic. However, it should be noted that advanced transaction semantics are synthesized not only using transaction dependencies. Parts of the business logic have to address also transaction logic. This cannot be avoided in a programmatic transaction system approach. Accordingly, not all modifications of the transaction logic can be tackled without touching business logic and separation of concerns is not honored completely. Outsourcing transaction dependencies from the business logic just reduces mixing business logic and transaction logic to a considerable degree. It does not avoid it completely.

6.4 Selected TWSO Orchestration Solutions

The concepts of TWSO can be employed in various orchestration environments. There is no imposition to stick to a specific technology for TWSO implementations. Moreover, there is no need for a special technology to express TWSO declarations in orchestrations. XML is a possibility as well as constructs of an existing programming language, depending on the particular orchestration environment.

Important implementations of TWSO are presented below. An implementation for Java Web service orchestrations and an implementation that can be used for various XML Web service orchestration technologies are introduced. Moreover, Appendix C.1 shows the usage of the XML Web service orchestration implementation in a common Web service orchestration technology.

6.4.1 TWSOL – XML Implementation of TWSO

For various XML Web service orchestrations, it is highly reasonable to express TWSO concepts in XML. The resulting XML language is called TWSOL (Transactional Web service Orchestration Language). It is intended, that the XML elements that are defined in TWSOL are incorporated in Web service orchestration XML documents. That is, in order to embed transaction logic into XML Web service orchestrations, TWSOL XML elements are added to XML Web service orchestration documents.

By using XML namespaces [7] to discriminate TWSOL elements from the original orchestration elements, transaction logic and business logic is separated to a significant degree. Moreover, XML namespaces also allow to extend TWSOL elements in a clean way. For example, new transaction primitives (besides the ones presented in Section 6.2) can be introduced by defining them in new namespaces, and execution environments can decide on the basis of the found namespace whether they are able to execute the namespace's primitive or not.

TWSOL XML elements that can be interspersed in orchestration host–languages are specified in this section by using XML–Schema. It should be noted that global XML–Schema constructs like namespace definitions or ID/IDREF definitions are omitted for the sake of clarity. All used `twso` prefixes refer to the `http://move.ec3.at/twso/20060101` namespace.

To setup a transaction, we need a unique identifier by which the transaction is identified. Furthermore, we also need to associate the transaction to one or more orchestration work items (a transaction can control more than one work item) that it is intended to manage. This setup information is kept in the `<initiate>` element, as presented in Listing 6.4.

Listing 6.4: <`initiate`> element

```
<xs:element name="initiate">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="twso:workitemRef"
                  minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID" use="required"/>
  </xs:complexType>
</xs:element>
```

There is a single attribute `id` that defines an identifier of the transaction. The `<workitemRef>` element should include ways to reference activities that occur in the orchestration in order to specify which activities should be managed by the transaction. How this is done best depends on the XML orchestration standard to a high degree. Therefore, the XML–Schema allows any type of content in `<workitemRef>`. If the host language considers unique identifiers for workitems, `<workitemRef>` could simply contain the matching identifier. If not, techniques that identify an element unambiguously in an XML document like XPath [14] can be used. The `<workitemRef>` element is shown in Listing 6.5.

Listing 6.5: <workitemRef> element

```
<xs:element name="workitemRef"/>
```

After initiating transactions, it has to be possible to define their dependencies. The `<dependency>` element is used for that task. It is defined in Listing 6.6.

Listing 6.6: <`dependency`> element

```
<xs:element name="dependency">

    <xs:complexType>
        <xs:sequence>
            <xs:element name="sourceState" type="twso:sourceState"/>
            <xs:element name="effect" type="twso:effect"/>
        </xs:sequence>
        <xs:attribute name="id" type="xs:ID" use="required"/>
    </xs:complexType>

</xs:element>
```

`<dependency>` contains two children. The `<sourceState>` child–element specifies the state(s) that have to be in effect in order to trigger the effect. The effect is defined in the `<effect>` child–element. To reference a dependency, an identifier has to be provided in the `id` attribute.

The `twso:sourceState` type is defined as shown in Listing 6.7.

Listing 6.7: `sourceState` type

```
<xs:complexType name="sourceState">
    <xs:choice>
        <xs:element name="transactionState"
                    type="twso:transactionState"/>
        <xs:element name="operator"
                    type="twso:operator"/>
    </xs:choice>
</xs:complexType>
```

In the `twso:sourceState` type, the state of a particular transaction is specified with `<transactionState>` elements of type `twso:transaction-State`. These contain the type of the state and the identifier of the concerned transaction. It is specified in Listing 6.8.

Listing 6.8: `transactionState` type

```
<xs:complexType name="transactionState">
    <xs:sequence>
        <xs:element name="type" type="xs:string"/>
        <xs:element name="transactionID" type="xs:string"/>
    </xs:sequence>
</xs:complexType>
```

According to Subsection 6.2.2 there is a basic set of transaction states that should be sufficient for many advanced transaction models. In TWSOL, these states are referenced in the `type` attribute of the `twso:transactionState` type, as shown in Table 6.2. The content of the type attribute has to be a *qualified name* as defined in the XML namespace proposal [7]. Table 6.2 shows the transaction states that are included in the basic set, as well as the namespace URI of this set and a possible occurrence.

If the `<sourceState>` element should express states that involve the states of more than one transaction, all of theses transaction states have to be combined in some way: What arrangement of transaction states have to occur in order to trigger an effect? TWSOL offers the logical operators *and* ($\wedge$), *or* ($\vee$) and *not* ($\neg$) to combine multiple transaction states. These logical operators can be nested and are used in combination with `<transaction-State>` elements. To do so, an `<operator>` element of type `twso:operator` is introduced, as shown in Listing 6.9. `twso:operator` defines a recursion to arbitrary nest (logical) operators and `<transactionState>` elements.

Table 6.2: Basic transaction states

State	**Qualified Identifier**
initiated	`twso:initiated`
in_progress	`twso:in_progress`
committed	`twso:committed`
aborted	`twso:aborted`
compensated	`twso:compensated`
delegated	`twso:delegated`

Listing 6.9: `operator` type

```
<xs:complexType name="operator">
    <xs:sequence maxOccurs="unbounded" minOccurs="1">
        <xs:element name="operator" type="twso:operator"
                    minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="transactionState" type="twso:transactionState"
                    minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="type" type="xs:string"/>
</xs:complexType>
```

To define the effects of a dependency, the `twso:effect` type is used. It is defined in Listing 6.10. It contains one or more `<primitive>` elements, that are of type `twso:primitive`. Accordingly, if the source states go into effect, all transaction primitives that are found in a particular `<effect>` element are issued. The `twso:primitive` type is specified in Listing 6.12 and described in detail further below.

Listing 6.10: `effect` type

```
<xs:complexType name="effect">
    <xs:sequence>
        <xs:element name="primitive" type="twso:primitive"
                    minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>
```

To give an example, the dependency “$\neg\ \sigma_{aborted}(t_2) \wedge (\sigma_{compensated}(t_1) \vee \sigma_{committed}(t_3)) \Rightarrow p_{abort}(t_4), p_{abort}(t_5)$” (refer to Section 6.3 for the formal notation) is expressed in TWSOL XML in Listing 6.11.

Listing 6.11: Dependency example

```
<dependency id="exampleDependency">
  <sourceState>

    <operator type="and">

      <operator type="not">
        <transactionState transaction="t2" type="twso:aborted">
      </operator>

      <operator type="or">
        <transactionState transaction="t1" type="twso:compensated">
        <transactionState transaction="t3" type="twso:commited">
      </operator>

    </operator>

  </sourceState>


  <effect>
    <primitive type="twso:abort" target="t4">
    <primitive type="twso:abort" target="t5">
  </effect>

</dependency>
```

After the transactions are setup, a possibility to issue transaction primitives in the orchestration's business logic has to be offered, too. Adding XML elements that represent transaction primitives into suitable places of the orchestration can achieve this. In TWSOL, `<primitive>` elements of type `twso:primitive` perform this task. These elements are specified as presented in Listing 6.12.

Listing 6.12: <`primitive`> element

```
<xs:element name="primitive" type="twso:primitive"/>

<xs:complexType name="primitive">
    <xs:attribute name="type" type="xs:string" use="required"/>
    <xs:attribute name="source" type="xs:string" use="optional"/>
    <xs:attribute name="target" type="xs:string" use="optional"/>
</xs:complexType>
```

The `<primitive>` element has a `type` attribute that specifies the type of the transaction primitive. In Section 6.2, a basic set of transaction primitives that should be sufficient for many advanced transaction models is defined. These primitives are referenced in TWSOL in a `<primitive>` element in the `type` attribute. This procedure is analogous to the one for transaction states, as discussed above. Hence, the content of the type attribute has to be a *qualified name* as defined in the XML namespace proposal [7]. Table

Table 6.3: Basic transaction primitives

Primitive	**Qualified Identifier**
begin	`twso:begin`
commit	`twso:commit`
abort	`twso:abort`
compensate	`twso:compensate`
delegate	`twso:delegate`

6.3 gives an overview of transaction primitives that are included in the basic set. Possible occurrences including the namespace URI are included, too.

To specify the involved transactions, a `target` and a `source` attribute may be used. `target` defines the target transaction of the transaction primitive. If necessary, `source` can be used to define a source of a significant event. For example, if one wants to delegate responsibilities using the transaction primitive type `base_events:delegate`, both, the `source` and `target` attribute have to be given: Responsibilities will be transferred from the transaction specified in `source` to the transaction specified in `target`.

Transaction primitive elements should be interspersed in "suitable" places of the orchestration specification. Where those "suitable" places are highly depends on the orchestration technology. For example in XPDL, an XPDL activity element could contain `<primitive>` elements. Appendix C.1 illustrates TWSOL usage in XPDL orchestrations in detail.

6.4.2 TWSO4J – Java TWSO Implementation

In this section, a TWSO implementation for a Java Web service orchestration is presented. The implementation is called *TWSO4J* and is intended to be used in Java Web service orchestrations. However, since only standard object–oriented concepts are used, it can be ported to other object–oriented languages easily. TWSO4J is a Java API that consists of Java interfaces. This approach allows multiple implementations of TWSO4J that are interchangeable. TWSO4J consists of only three interfaces.[1]

[1]It should be noted that the low number of necessary interfaces emphasizes TWSO's lean design, ease of use and steep learning curve for Web service orchestration developers.

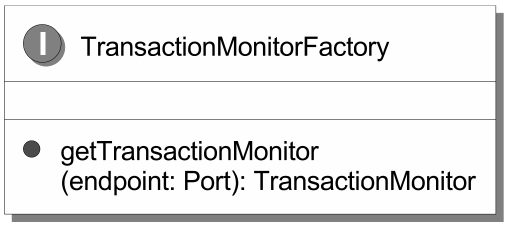
TransactionMonitorFactory
getTransactionMonitor
(endpoint: Port): TransactionMonitor

Figure 6.3: Transaction monitor factory

To establish communication between the orchestration and the component that manages transaction semantics for the orchestration (i.e. an orchestration specific transaction monitor), a proxy object called `TransactionMonitor` may be used. Such `TransactionMonitor` objects are created using a factory, called `TransactionMonitorFactory` and depicted in Figure 6.3.

The method `getTransactionMonitor()` produces an instance of a `TransactionMonitor` object. Its parameter, an endpoint of a Web service ("port"), references a Web service interface that is used to communicate with the transaction system. The endpoint parameter has to be an instance of the WSDL4J's (Java WSDL library) `Port` class. Setup procedures in the system's transaction monitor server that are needed to use the transaction monitor proxy appropriately have to be initiated while it is created. For example, a new transaction monitor client session for the newly created proxy may be generated at the server. Depending on the implementation, more aspects like authentication credentials may be added and require additional methods or parameters for the `TransactionMonitorFactory`. To conclude, `TransactionMonitorFactory` produces an individual proxy (a `TransactionMonitor` instance) for each Web service orchestration instance. This proxy is used in the orchestration to interact with the Web service transaction system.

A `TransactionMonitor` instance encapsulates the interaction between an instance of an orchestration and the Web service transaction system. It is shown in Figure 6.4.

A `WebServiceReference` can contain a URI that points to a WSDL document, a qualified name that references a binding in this WSDL document and a name for a binding operation. If only the WSDL document URI is set, the transaction should manage every service that is mentioned. If a qualified name that references a binding is present, just the services in the binding should be managed by the transaction. And when a binding operation is referenced, the transaction supervises just this single service.

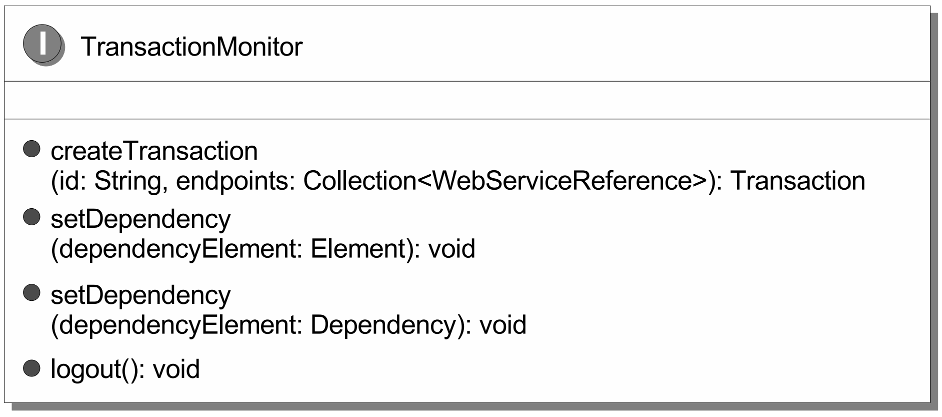
TransactionMonitor
createTransaction
(id: String, endpoints: Collection<WebServiceReference>): Transaction
setDependency
(dependencyElement: Element): void
setDependency
(dependencyElement: Dependency): void
logout(): void

Figure 6.4: Transaction monitor

The `setDependency()` methods take either an XML element or a `Dependency` object as input. The XML element describes the dependency as defined in Listing 6.6, including all its referenced XML Schema definitions. The outsourcing of dependency information from Java code to XML descriptions yields several advantages. Modifications of XML dependencies may take place without the need to recompile the Java orchestration source code. Sources of dependencies definitions can be handled flexible. Any XML source is possible: XML databases, files on the local filesystem, XML documents in the Internet or Intranet, etc. In addition, XML dependency definitions are interchangeable to a high degree. For example, it is possible to use the same XML dependency definitions in an XPDL orchestration as well as in a Java orchestration. However, it is also possible to define a dependency in Java using a `Dependency` object. The related `Dependency` class is a Java representation of the XML Schema type for a dependency as defined in Listing 6.6.

`logout()` ends a transaction monitor session and frees all related resources of the transaction monitor proxy in the transaction monitor server.

A transaction is modeled using the `Transaction` interface, as shown in Figure 6.5. It has an overloaded method that issues a transaction primitive, called `doPrimitive()`. The target of the transaction primitive is the respective `Transaction` instance. The type of the primitive is given as a `String` in the `type` parameter. If it is desired or necessary to give a source transaction, an additional corresponding `Transaction` object can be passed to `doPrimitive()`, too.

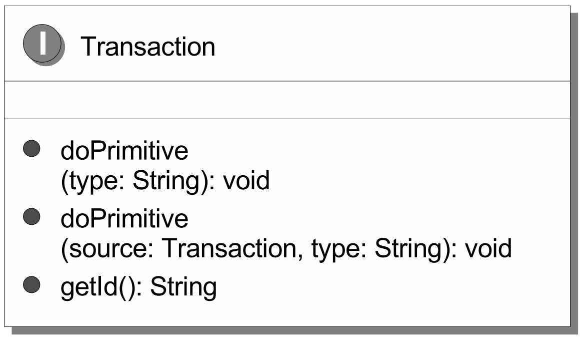

Figure 6.5: TWSO4J transaction

An example of an exemplary TWSO4J usage is shown in Listing 6.13. From line 2 to line 8, an instance of a `TransactionMonitor`, called `transactionMonitor` is created. Line 10-22 create collections of endpoints of two Web services. In line 24-27, the `transactionMonitor` instance is used to create two `Transaction` instances. They are associated to the two Web services. The method `getSimpleDependency()` creates a dependency XML element from a string. This dependency is used in line 30 to install the dependency semantics (stated in XML) in the transaction system.

Starting from line 32, business logic is executed. Some Web service calls are done on both Web services. If these calls are considered to be unsatisfactory, transaction `t1` is aborted. Consequently, `t1` gets into the state `aborted` and the dependency is triggered. This causes the `abort` of `t2`, too, and execution of business logic is stopped. Both Web services get an `abort` primitive and perform according steps to undo the actions in line 32 without any traces.

In the other case, when the first calls are considered to be satisfactory, termination obligations are transferred from `t1` to `t2` in line 39. Thus, termination transaction primitives on `t2` are also effective on `t1`.

Thereafter, line 41 does some business logic by calling Web service 2 and depending on the outcome, `t2` is committed in line 44 or aborted in line 46.

Listing 6.13: TWSO4J Example

```
1  public void doBusinessLogic() {
2          TransactionMonitorFactory factory =
3                                  new SoapTransactionMonitorFactory();
4
```

```
        TransactionMonitor transactionMonitor =
                factory.getTransactionMonitor(
                        getTransactionSystemPort()
                );

        WebServiceReference ws1Ref = getWSReference(
                        "http://ws1.ws1domain.net:8080/ws1?wsdl",
                        "http://ws1.namespace.net", "ws1SoapBinding",
                        "doWS1");
        Collection<WebServiceReference> tx1Services = new Vector<Port>();
        tx1Services.add(ws1Ref);

        WebServiceReference ws2Ref = getWSReference(
                        "http://ws2.ws1domain.net:8080/ws1?wsdl",
                        "http://ws2.namespace.net", "ws2SoapBinding",
                        "doWS2");
        Collection<WebServiceReference> tx2Services = new Vector<Port>();
        tx1Services.add(ws2Ref);

        Transaction t1 = transactionMonitor.createTransaction
                                                ("t1", tx1Services);
        Transaction t2 = transactionMonitor.createTransaction
                                                ("t2", tx2Services);

        Element simpleDependency = getSimpleDependency();
        transactionMonitor.setDependency(simpleDependency);

        //do Web service calls on Web service 1 and 2

        if (Web service 1 calls != ok) {
                t1.doPrimitive("ABORT");
                return;
        }

        t2.doPrimitive(t1, "DELEGATE");

        //do Web service calls on Web service 2

        if (Web service 2 calls == ok){
                t2.doPrimitive("COMMIT");
        } else {
                t2.doPrimitive("ABORT");
        }
}

private static Element getSimpleDependency(){
        String depString = "A␣string␣that␣contains␣XML␣as␣shown␣below...";
        //<dependency id="exampleDependency">
        //      <sourceState>
        //          <transactionState transaction="t1" type="twso:abort">
        //      </sourceState>
        //      <effect>
        //          <primitive type="twso:abort" target="t2">
        //      </effect>
        //</dependency>;
        Element dependency = XmlUtilities.stringToElement(depString);
        return dependency;
}
```

Chapter 7

Transaction Monitor

A TWSO transaction system makes use of an autonomous component that mediates transaction matters between an orchestration system and the orchestrated Web services. This component is called *transaction monitor* and is situated in the scope of the orchestration system. For example, in a virtual enterprise, a single transaction monitor is located within the scope of the central components of the virtual enterprise that combine all parts of the virtual enterprise together. In a TWSO transaction system, a single transaction monitor is sufficient. In contrast to the WS–transaction proposals introduced in Section 3.6, a TWSO transaction system does not consider a network of interoperating transaction monitors. The centric nature of a TWSO system — a single orchestration system assisted by transaction–oriented concepts ensembles multiple services — does not mandate the use of a network of transaction monitors. Thus, the huge complexity of a system of multiple distributed transaction monitors is avoided.

The advantages such a network would have (as discussed in [9, 10, 11]) are only marginally relevant for a TWSO transaction system: To reach organizational and political independence by using an own transaction monitor that operates in a network is arguable. The intra–organizational transaction monitor takes commands from transaction monitors that are beyond organizational influence and forwards these commands to intra–organizational systems. Hence, the intra–organizational transaction monitor acts on decisions that are made outside of an organization and is neither political nor organizational independent, although it acts in a network. A network of transaction monitors should contribute to avoid a single point failure. However, also a single transaction monitor can be designed to be virtually always available by adding, for example, redundancy components or other high availability techniques.

Thus — at least for a TWSO transaction system — the huge complexity of a network of multiple distributed transaction monitors outweighs its advantages by far. A single transaction monitor is highly sufficient.

[24] states, that a transaction monitor in traditional transaction processing systems must offer the following services:

- **Heterogeneity management.** If the application–level function requires access to different services, the transaction monitor has to manage transactional matters across those services.
- **Control communication.** Communication with a remote service as such has to be subject to transaction management, too.
- **Terminal management and presentation services.** Transaction services have to be available for and have to be able to be visualized by virtually all kinds of client systems.
- **Context management.** Multiple business invocations under transaction control including their involved services and transaction facilities need stable information about the corresponding transaction. A transaction context that holds this information has to be provided by the transaction monitor.
- **Start/restart.** The transaction monitor is responsible to restart service after failures.

These demands are designed for tightly coupled systems. For transaction monitors in Web service transaction systems, *control communication* and *start/restart* are not feasible. The control of Web service communication requires rules that describe situations when a Web service call should be considered to be erroneous. Such rules highly depend on the Web service itself and sometimes even on the orchestration in that the Web service is used. For example, in one orchestration, a timeout that results in a failure may occur after 2 seconds and in another orchestration after 5 minutes. In TWSO, the logic that decides when a business logic call or a communication is considered to be erroneous resides in the orchestration. Start and restart of Web services by the transaction monitor are impossible because of the organizational distributed nature of loosely coupled systems. It is unlikely that a Web service provider lets an external transaction system decide when a Web service should be restarted after a crash. Thus, in a TWSO transaction system, responsibility to restart a service lies at the service provider. However, heterogeneity management, terminal and presentation management, and context management are reasonable in loosely coupled systems with transaction support and are offered by a TWSO transaction monitor, as described here.

A TWSO transaction monitor gathers transaction related services together in order to modularize transaction concerns. The orchestration communicates simple transaction related concerns to the transaction monitor. The transaction monitor acts on these concerns accordingly. Depending on the concern, it should be noted that such an action of the transaction monitor might be rather complex. Thus, complexity of transaction related matters is hidden from orchestrations and can be called by a clean and simple interface. This interface is accessible via SOAP Web service technology. Therefore, the claim for terminal management and presentation services is satisfied. Virtually all kinds of client systems can talk to SOAP Web services and create a user friendly interface that encapsulates the SOAP interaction.

In TWSO, a single transaction can manage invocations on multiple Web services. Transaction primitives on such a transaction must cause an agreed atomic outcome across all managed operations. For example, let t be a transaction that manages all invocations on three Web services s_1, s_2, and s_3. A `commit` on t has to cause a `commit` on s_1, s_2, and s_3. However, if, for example, s_3 cannot commit, s_1 and s_2 must not commit and the transaction fails and has to be aborted. Such a case can be handled by synchronization protocols like the two phase commit, as described in Subsection 3.2.3. The transaction monitor is responsible to execute the synchronization protocol (thus guaranteeing an agreed atomic outcome), and sends outcomes to the concerning orchestration. By providing synchronization means for transactions that act across several Web services, the demand for heterogeneity management is fulfilled.

As discussed in Section 5, transaction related data is interchanged in a TWSO system between the main components using a transaction context. This context is transmitted in a TWSO transaction system in *all* operations that happen in presence of any transaction–related management. The context is produced by the transaction monitor and transmitted to the orchestration, when a transaction is generated. All communication between the transaction monitor and the orchestration is accompanied by the related transaction context. For example, the issuing of a transaction primitive contains the context of the affected transaction. This type of context handling in a TWSO system complies with the claim for context management.

7.1 Interface: Orchestration and Transaction Monitor

The interface between a TWSO enabled orchestration and the transaction monitor will be used by the orchestration to communicate transaction related concerns. It resembles the interface between the orchestration specification and the orchestration as introduced in Section 6.4. The orchestration forwards transaction related issues as specified in the orchestration's specification. Thus, the orchestration does not have to cope with transaction related logic and outsources this logic to the transaction monitor component.

The interface of a transaction monitor is accessible using SOAP. It is straightforward, lean and follows a synchronous messaging paradigm. The interface is described using WSDL descriptions. For the sake of clarity, only selected parts of WSDL definitions without type declarations and only down to the port type level are shown here. The complete WSDL definition of a TWSO transaction monitor is presented in Appendix B.

To create a transaction, the `createTransaction` operation is used. The input message consists of a significant label (the part named `name`) and a collection that contains one or more references to refer to Web services that should be executed under transaction control. Such a reference uses URI to point to the considered WSDL document and a qualified name (using the namespace URI and the local name) that references a binding in this WSDL. Additionally, a binding operation can be specified. If this is the case, the transaction should supervise just this operation. Otherwise, all operations in the binding should be managed by the related transaction. For example, to control the `bookFlight` operation of a flight booking Web service as described in Section 8.1 and Listing 8.5, one would have to give the URI of the WSDL (e.g. `http://www.lh.org/bookFlight?wsdl`), a qualified name for the binding reference (e.g. namespace `http://www.lh.org/bookFlight` and local name `LHFlightBookingSOAP`) and the binding operation name (`bookFlight`). An identifier of the created transaction is returned. This identifier is used to reference the transaction in the later processing, for example, to reference it in a transaction context. Listing 7.1 shows relevant parts of the WSDL description.

Listing 7.1: `createTransaction` operation

```
<wsdl:message name="createTransactionRequest">
        <wsdl:part  name="name" type="xsd:string" />
        <wsdl:part  name="wsReferenceCollection"
```

```
                    type="twso:wsReferenceCollection"/>
</wsdl:message>

<wsdl:message name="createTransactionResponse">
        <wsdl:part name="id" type="xsd:string" />
</wsdl:message>

<wsdl:message name="createTransactionException">
        <wsdl:part name="createTransactionException" type="xsd:string"/>
</wsdl:message>


<wsdl:portType name="TransactionMonitor">
    <wsdl:operation name="createTransaction">
            <wsdl:input message="twso:createTransactionRequest" />
            <wsdl:output message="twso:createTransactionResponse" />
            <wsdl:fault name="createTransactionException"
                    message="twso:createTransactionException">
            </wsdl:fault>
    </wsdl:operation>
</wsdl:portType>
```

The `setDependency` operation is shown in Listing 7.2. It is used to create transaction semantics by specifying dependencies between multiple small transactions. `setDependency` takes an input parameter that conforms to the dependency specification as given in Listing 6.6 and returns a status string.

Listing 7.2: `setDependency` operation

```
<wsdl:message name="setDependencyRequest">
        <wsdl:part name="dependency" type="twso:dependency"/>
</wsdl:message>

<wsdl:message name="setDependencyResponse">
        <wsdl:part name="status" type="xsd:string"/>
</wsdl:message>

<wsdl:message name="setDependencyException">
        <wsdl:part name="setDependencyException" type="xsd:string"/>
</wsdl:message>

<wsdl:portType name="TransactionMonitor">
    <wsdl:operation name="setDependency">
            <wsdl:input message="twso:setDependencyRequest"/>
            <wsdl:output message="twso:setDependencyResponse"/>
            <wsdl:fault name="setDependencyException"
                    message="twso:setDependencyException">
            </wsdl:fault>
    </wsdl:operation>
</wsdl:portType>
```

To instruct the transaction monitor to do a transaction primitive, the `doPrimitive` operation is used. It takes the type of the primitive in a format as described in Table 6.3, the identifier of the target transaction and — if needed — the identifier of the source transaction. In case the primitive fails, a special fault message is returned. Depending on the particular orchestration technique, this fault message most likely would cause a runtime–exception in the orchestration.

Listing 7.3: `doPrimitive` operation

```
<wsdl:message name="doPrimitiveRequest">
        <wsdl:part name="primitiveType" type="xsd:string"/>
        <wsdl:part name="sourceTransactionID" type="xsd:string"/>
</wsdl:message>

<wsdl:message name="doPrimitiveResponse">
        <wsdl:part name="status" type="xsd:string"/>
</wsdl:message>

<wsdl:message name="doPrimitiveException">
        <wsdl:part name="doPrimitiveException" type="xsd:string"/>
</wsdl:message>


<wsdl:portType name="TransactionMonitor">
    <wsdl:operation name="doPrimitive">
            <wsdl:input message="twso:doPrimitiveRequest"></wsdl:input>
            <wsdl:output message="twso:doPrimitiveResponse"></wsdl:output>
            <wsdl:fault name="doPrimitiveException"
                    message="twso:doPrimitiveException">
            </wsdl:fault>
    </wsdl:operation>
</wsdl:portType>
```

In addition, there are `login` and `logout` operations to enable client session handling. The `login` operation returns a session identifier that has to be included in the transaction context. Thus, the transaction monitor and the involved Web services are always aware of the particular client that uses transactions for the duration of a session. These operations are presented in Listing 7.4.

Listing 7.4: `login` and `logout` operations

```
<wsdl:message name="loginRequest"/>
<wsdl:message name="loginResponse">
        <wsdl:part name="sessionId" type="xsd:string"/>
</wsdl:message>

<wsdl:message name="logoutRequest"/>
<wsdl:message name="logoutResponse">
        <wsdl:part name="status" type="xsd:string"/>
</wsdl:message>

<wsdl:portType name="TransactionMonitor">
    <wsdl:operation name="login">
            <wsdl:input message="twso:loginRequest"/>
            <wsdl:output message="twso:loginResponse"/>
    </wsdl:operation>
    <wsdl:operation name="logout">
            <wsdl:input message="twso:logoutRequest"/>
            <wsdl:output message="twso:logoutResponse"/>
    </wsdl:operation>
</wsdl:portType>
```

Chapter 8

TWSO Enabled Web Services

Web services that are able to interoperate with a TWSO transaction system in a transactional way have to fulfill certain prerequisites. It should be noted, that Web services that do not offer TWSO transaction capabilities could be part of a transactional TWSO orchestration, too. Of course, such orchestrations have to take into account that transactional concepts cannot be applied to Web services without TWSO related capabilities.

A Web service supports a TWSO concept when it is able to act on a received TWSO matter in a semantically correct way. A TWSO transaction system cannot influence a Web service in any way and cannot validate its correct mode of operation. Thus, the transaction system has to trust the Web service, that it implements pretended capabilities in a sound way. Responsibility of correct execution of transaction related matters lies totally in the area of the Web service itself.

A Web service that supports TWSO concepts and wants to participate in TWSO transactions has to publish the supported TWSO concept in its WSDL description. This procedure enables all interested TWSO components to be aware of the Web service's TWSO capabilities. It is described in Section 8.1.

To communicate transaction matters, a Web service has to provide a standardized interface. This interface is described in Section 8.2. Of course, the concrete implementation of such an interface (as done by a TWSO Web service) has to correspond to its published supported transaction concepts. This interface will be used by the transaction monitor.

8.1 TWSO WSDL

Web services that are transaction enabled for TWSO transaction systems should publish their transaction capabilities. This way, components of a TWSO transaction system can become aware of the TWSO capabilities of a particular Web service. These capabilities are expressed by declaring the transaction related concepts a Web service or a part of a Web service supports in the WSDL description of the Web service. As this section assumes some basic knowledge of WSDL, a short introduction to WSDL is given in Appendix A.

WSDL offers an extensibility mechanism by allowing arbitrary XML elements ("extensibility elements") interspersed in particular sections of a WSDL definition. This extensibility mechanism is appropriate to publish TWSO related capabilities of a Web service.

To describe TWSO capabilities in a WSDL document, the binding section may be considered: A binding defines implementation details of an abstract port type interface definition. Moreover, WSDL extensibility elements in the binding section provide protocol specific information that applies to the port type being bound [12]. Hence, the binding section of a WSDL definition is an appropriate place for specifying TWSO capabilities of particular port type implementations.

In a binding, extensibility information can be associated either to the whole binding or to a particular binding operation of a binding. Accordingly, either the whole port type or a single operation of the port type is addressed. TWSO capabilities of a Web service can also be associated to the whole port type or to particular operations of a port type, depending on the location of the TWSO capability elements in the binding.

The XML Schema in Listing 8.1 defines TWSO related WSDL extensibility elements. Currently, supported transaction primitives and supported synchronization protocols represent TWSO capabilities. The XML Schema is designed to integrate future extensions easily.

There is an element `<tx:capabilities>` that includes a single `<tx:primitives>` element. This element contains one or more `<tx:primitiveCapability>` elements that represent the capability to act on the mentioned primitive accordingly. The `<tx:primitiveCapability>` element can contain two attributes. `type` specifies the type of the primitive, for example `twso:commit`. If available, `syncProtocol` specifies which protocol can be used to achieve an atomic execution of an issued transaction primitive across distributed services. Possible content of the `type` attribute is the same as specified in Section 6.3, when the basic set of transaction primitives is used. In a default version of TWSO, the `syncProtocol` attribute can only hold a single value to indicate that the usage of the two phase commit protocol (as described in Subsection 3.2.3) is possible: `twoPhaseCommit`.

Listing 8.1: TWSO WSDL capabilities

```
<element name="capabilities">
        <complexType>
                <sequence>
                        <element ref="tx:primitives"/>
                </sequence>
        </complexType>
</element>

<element name="primitives">
        <complexType>
                <sequence>
                        <element ref="tx:primitiveCapability"
                                 minOccurs="1"
                                 maxOccurs="unbounded"/>
                </sequence>
        </complexType>
</element>

<element name="primitiveCapability">
    <xs:complexType>
        <xs:attribute name="type" type="xs:NMTOKEN" use="required"/>
        <xs:attribute name="syncProtocol" type="xs:NMTOKEN" use="optional"/>
    </xs:complexType>
</element>
```

To conduct TWSO related communication, the Web service has to offer a special SOAP interface, as introduced in Section 8.2. If there is no further information in the WSDL, it is assumed that the network address in the port of the corresponding binding offers this interface. However, it is possible to override this default by adding a `<tx:address>` element to the port. Then, all TWSO communication will be directed to this network address. This element, which resides in the `twso` namespace, is shown in Listing 8.2.

Listing 8.2: TWSO port address

```
<element name="address">
    <xs:complexType>
        <xs:attribute name="location" type="xs:string" use="required"/>
    </xs:complexType>
</element>
```

To give an example, flight booking service WSDL descriptions including TWSO capabilities are presented. Listing 8.3 shows the port type of the flight booking service. There are two operations. `bookFlight` is used to book a flight and `bookLounge` to reserve a place in an airport lounge. It should be noted that the input and output messages are omitted for the sake of clarity.

Listing 8.3: Flight booking port type

```
<wsdl:portType name="FlightBooking">
        <wsdl:operation name="bookFlight">
                <wsdl:input message="tns:bookFlightRequest" />
                <wsdl:output message="tns:bookFlightResponse" />
        </wsdl:operation>
        <wsdl:operation name="bookLounge">
                <wsdl:input message="tns:bookLoungeRequest"></wsdl:input>
                <wsdl:output message="tns:bookLoungeResponse"></wsdl:output>
        </wsdl:operation>
</wsdl:portType>
```

The first binding that is presented in Listing 8.4 associates an implementation of the flight booking port type that offers full TWSO capabilities for both operations, whereas “full” refers to the basic transaction primitive set presented in Section 6.2. Thus, `bookFlight` and `bookLounge` business logic calls can be facilitated by `begin`, `commit`, `abort` and `compensate` transaction primitives. The `delegate` primitive is not Web service effective and needs not to be mentioned. All primitives support synchronization via the two phase commit protocol. Thus, it is possible to achieve valid atomic outcomes of this implementation together with other services.

Listing 8.4: Port type scope TWSO binding

```
<wsdl:binding name="StarAllianceFlightBookingSOAP" type="tns:FlightBooking">
        <soap:binding   style="document"
                        transport="http://schemas.xmlsoap.org/soap/http"/>
        <wsdl:operation name="bookFlight">
            <soap:operation soapAction="http://www.star.org/bookFlight"/>
            <wsdl:input><soap:body use="literal"/></wsdl:input>
            <wsdl:output><soap:body use="literal"/></wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="bookLounge">
            <soap:operation soapAction="http://www.star.org/bookFlight"/>
            <wsdl:input><soap:body use="literal"/></wsdl:input>
            <wsdl:output><soap:body use="literal"/></wsdl:output>
        </wsdl:operation>

        <tx:capabilities>
                <tx:primitives>
                        <tx:primitive   type="tx:begin"
                                        syncProtocol="tx:twoPhaseCommit"/>
                        <tx:primitive   type="tx:commit"
                                        syncProtocol="tx:twoPhaseCommit"/>
                        <tx:primitive   type="tx:abort"
                                        syncProtocol="tx:twoPhaseCommit"/>
                        <tx:primitive   type="tx:compensate"
                                        syncProtocol="tx:twoPhaseCommit"/>
                </tx:primitives>
        </tx:capabilities>

</wsdl:binding>
```

The binding shown in Listing 8.5 is for a SOAP flight booking port type implementation that supports different TWSO capabilities for each operation. `bookFlight` cannot be compensated. Hence, it offers no more than ACID style transaction support. Synchronization for atomic outcomes in combination with other services is possible using the two phase commit protocol. `bookLounge` cannot be aborted but only compensated. Consequently, state changes are final and cannot be undone without traces. In terms of business logic, reservations of lounge seats go into effect immediately and only the use of special forward actions can achieve the cancellation of reservations. Furthermore, it is not possible to synchronize the service to achieve atomic outcomes in combination with other services.

Listing 8.5: Operation scope TWSO binding

```
<wsdl:binding name="LHFlightBookingSOAP" type="tns:FlightBooking">
    <soap:binding style="document"
                  transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="bookFlight">
        <soap:operation soapAction="http://www.lh.org/bookFlight"/>
        <wsdl:input><soap:body use="literal"/></wsdl:input>
        <wsdl:output><soap:body use="literal"/></wsdl:output>

        <tx:capabilities>
                <tx:primitives>
                        <tx:primitive   type="tx:begin"
                                        syncProtocol="tx:twoPhaseCommit"/>
                        <tx:primitive   type="tx:commit"
                                        syncProtocol="tx:twoPhaseCommit"/>
                        <tx:primitive   type="tx:abort"
                                        syncProtocol="tx:twoPhaseCommit"/>
                </tx:primitives>
        </tx:capabilities>

    </wsdl:operation>

    <wsdl:operation name="bookLounge">
        <soap:operation soapAction="http://www.lh.org/bookLounge"/>
        <wsdl:input><soap:body use="literal"/></wsdl:input>
        <wsdl:output><soap:body use="literal"/></wsdl:output>

        <tx:capabilities>
                <tx:primitives>
                        <tx:primitive type="tx:begin"/>
                        <tx:primitive type="tx:compensate"/>
                </tx:primitives>
        </tx:capabilities>

    </wsdl:operation>
</wsdl:binding>
```

The Star Alliance Web service offers its TWSO communication interface at a different address than its flight–booking interface. Listing 8.6 shows the according service section of its WSDL description.

Listing 8.6: Operation scope TWSO binding

```
<wsdl:service name="StarAllianceFlightBooking">
  <wsdl:port binding="tns:StarAllianceFlighBookingSOAP"
             name="StarAllianceFlighBookingSOAP">
     <soap:address location="http://www.staralliance.org/flightBooking"/>
     <tx:address location="http://www.staralliance.org/twso/flightBooking"/>
  </wsdl:port>
</wsdl:service>
```

8.2 Interface: Transaction Monitor and Web service

To communicate TWSO related matters to Web services, a TWSO enabled Web service has to offer a standardized TWSO SOAP interface. Depending on the TWSO capabilities of a Web service, particular parts of this interface have to be present. When certain TWSO capabilities are present in a Web service's WSDL description, TWSO components (i.e. primarily transaction monitors) rely on the presence of matching operations. Using these operations require the presence of an appropriate transaction context in the SOAP call. Otherwise, the Web service cannot identify the affected transaction properly. The transaction context is described in Chapter 5.

WSDL details of the implemented TWSO SOAP interface need not to be described in the Web service's WSDL. The interface is standardized, the mapping of a capability to a SOAP operation is inherently known in the system and thus an additional occurrence would result in unnecessary redundancy. However, to be concise the interface is presented using WSDL port types and messages here.

The interface consists of three remote procedure call style operations as follows. Since a single component may offer TWSO transaction interfaces for multiple Web services, such a component has to be aware which Web service is managed by which transaction: That is, when a transaction t_i is aborted at Web service transaction component c_a, c_a has to know exactly which Web service's work has to be undone to take appropriate measures. To associate a transaction with a Web service, the `begin` operation as shown in Listing 8.7 is used. It takes a qualified name for the desired binding (`bindingNamespace` and `bindingName`) and a name for the desired binding operation (`beginOperation`).

Listing 8.7: begin operation

```
<wsdl:message name="beginRequest">
    <wsdl:part name="bindingNamespace" type="xsd:string"/>
    <wsdl:part name="bindingName" type="xsd:string"/>
    <wsdl:part name="bindingOperation" type="xsd:string"/>
</wsdl:message>

<wsdl:message name="beginResponse">
    <wsdl:part name="beginReturn" type="xsd:string"/>
</wsdl:message>

<wsdl:portType name="TWSO">
    <wsdl:operation name="begin">
        <wsdl:input message="impl:beginRequest" name="beginRequest"/>
        <wsdl:output message="impl:beginResponse" name="beginResponse"/>
    </wsdl:operation>
</wsdl:portType>
```

To execute a primitive, the `doPrimitive` operation is used. It is shown in Listing 8.8. It takes the type of the transaction primitive as described in Table 6.3 and may return a status. In case of an erroneous situation, a fault message may be thrown.

Listing 8.8: doPrimitive Operation

```
<wsdl:message name="doPrimitiveResponse">
        <wsdl:part type="xsd:string" name="status"/>
</wsdl:message>
<wsdl:message name="doPrimitiveRequest">
        <wsdl:part type="xsd:string" name="primitiveType"/>
</wsdl:message>

<wsdl:message name="primitiveFailed">
        <wsdl:part name="primitiveFailed" type="xsd:string"/>
</wsdl:message>

<wsdl:portType name="TWSO">
    <wsdl:operation name="doPrimitive">
        <wsdl:input message="twso:doPrimitiveRequest" />
        <wsdl:output message="twso:doPrimitiveResponse" />
        <wsdl:fault name="primitiveFailed" message="twso:primitiveFailed"/>
    </wsdl:operation>
</wsdl:portType>
```

In case the Web service supports the two phase commit protocol (the default synchronization protocol in TWSO systems) for synchronization, a Web service has to offer a method to support the prepare phase, as discussed in Subsection 3.2.3. The `doPrepare` operation as presented in Listing 8.9 is used for that. It takes the type of the transaction primitive (see Table 6.3) that is about to be synchronized. Thus, it is possible to achieve an agreed outcome across multiple distributed Web services on arbitrary primitives — not only on a `commit`. For example, an agreed outcome of an `abort` or `compensate` is possible. It returns a status code that indicates the readiness to execute the considered primitive. Additionally, in case of erroneous situations, a fault message can be thrown.

Listing 8.9: preparePrimitive Operation

```
<wsdl:message name="preparePrimitiveResponse">
        <wsdl:part name="status" type="xsd:string"/>
</wsdl:message>
<wsdl:message name="preparePrimitiveRequest">
        <wsdl:part name="primitiveType" type="xsd:string"/>
</wsdl:message>

<wsdl:message name="prepareFailed">
        <wsdl:part name="prepareFailed" type="xsd:string"/>
</wsdl:message>

<wsdl:portType name="TWSO">
    <wsdl:operation name="preparePrimitive">
        <wsdl:input message="twso:preparePrimitiveRequest"/>
        <wsdl:output message="twso:preparePrimitiveResponse"/>
        <wsdl:fault name="prepareFailed" message="twso:prepareFailed"/>
    </wsdl:operation>
</wsdl:portType>
```

Chapter 9

TWSO Tourism Scenario Orchestrations

9.1 City Trip Scenario

The requirements of this scenario are presented in detail in Scenario 2.1. Figure 9.1 depicts a suitable orchestration workflow.

Because no service charges any booking attempt in first place, performance is maximized and all bookings occur concurrently: Let $d(s_i)$ be the duration to finish the call of Web service s_i. The duration of the orchestration will be $max(d(s_{book\ flight}), \ldots, t(d_{book\ sight\ tickets}))$. The orchestration logic is completed herewith and transaction logic is left to be added. This is presented in Figure 9.2 exemplary for two Web services. The other services are embraced by completely analogous transaction logic and are indicated by "..." in the figure.

The darker dots in Figure 9.2 refer to transaction related activities. First, all to be used transactions are initiated and transaction dependencies (as discussed below in detail) have to be installed. Thereafter (for each service booking) a particular transaction is started. If the outcome of the booking is considered to be successful, the transaction is committed immediately and isolation is violated apparently. In order to guarantee isolation, services would have to lock tentatively booked resources for other customers until all other services bookings are finished. Since the hotel room booking needs approximately two days and all other bookings are finished within seconds, each service would have to hide tentatively booked resources for two days in order to wait for the hotel booking. Such tight dependencies between orga-

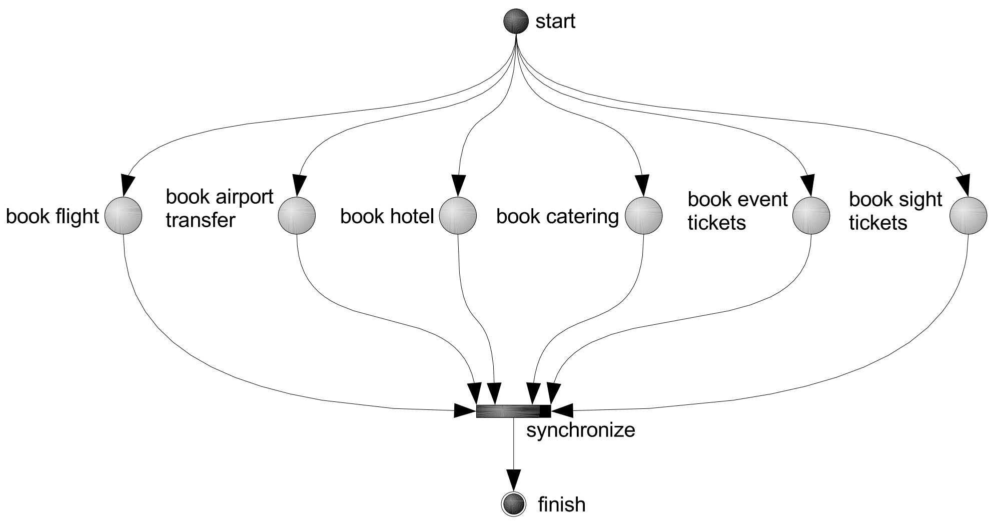

Figure 9.1: Non–transactional Vienna city trip orchestration

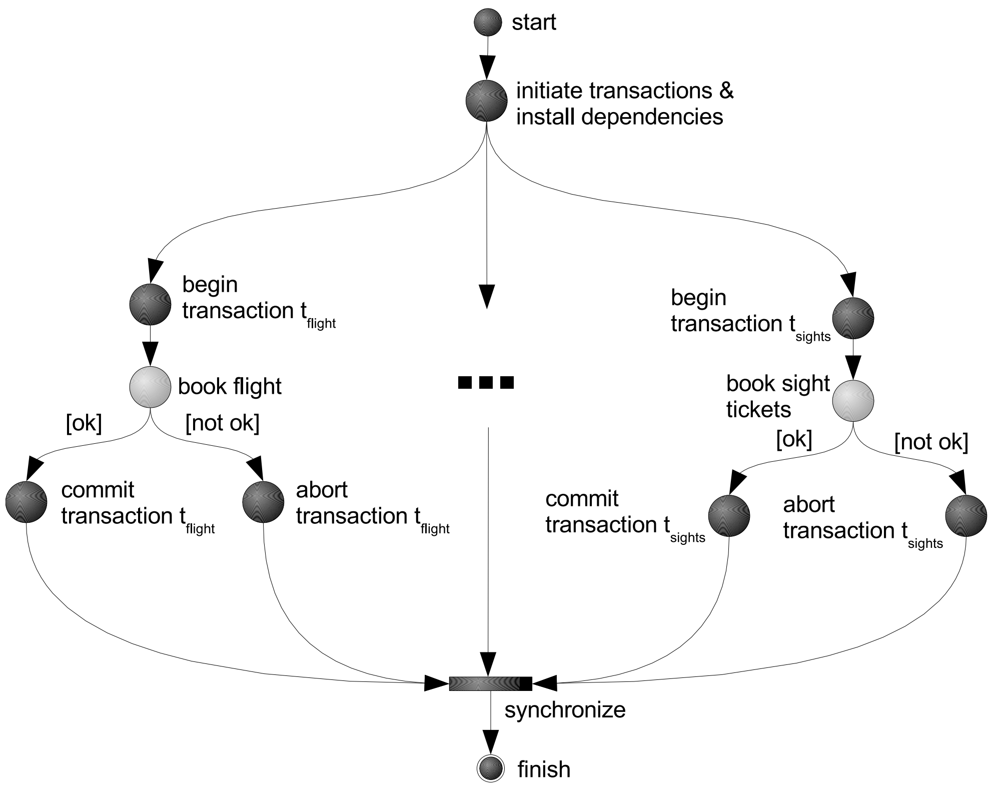
start
initiate transactions & install dependencies
begin transaction t_{flight}
book flight
[ok]
[not ok]
commit transaction t_{flight}
abort transaction t_{flight}
begin transaction t_{sights}
book sight tickets
[ok]
[not ok]
commit transaction t_{sights}
abort transaction t_{sights}
synchronize
finish

Figure 9.2: Transactional Vienna city trip orchestration

nizationally independent services are most likely deprecated. Thus, isolation can not be achieved. Consequently, compensation is used to undo committed bookings. In case a transaction is considered to be unsuccessful, it is aborted.

The tourist preferences can be expressed fully by transaction dependency logic. According dependencies are as follows, whereas the notation will be the same as used in Section 6.3: $\sigma_{statename}(t_{tx1})$ denotes that state `statename` of transaction `tx1` is in effect, and $p_{primitive}(t_{tx2})$ means issuing primitive `primitive` to transaction `tx2`. Left of "⇒" is the source state and right the effect.

- $\sigma_{aborted}(t_{flight}) \vee \sigma_{aborted}(t_{transfer}) \vee \sigma_{aborted}(t_{hotel}) \vee \sigma_{aborted}(t_{catering})$
  $\Rightarrow$
  $p_{compensate}(t_{flight}), p_{compensate}(t_{transfer}), p_{compensate}(t_{hotel}),$
  $p_{compensate}(t_{catering}), p_{compensate}(t_{events}), p_{compensate}(t_{sights})$

- $\sigma_{aborted}(t_{sights}) \wedge \sigma_{aborted}(t_{events})$
  $\Rightarrow$
  $p_{compensate}(t_{flight}), p_{compensate}(t_{transfer}), p_{compensate}(t_{hotel}),$
  $p_{compensate}(t_{catering}), p_{compensate}(t_{events}), p_{compensate}(t_{sights})$

In other words, when booking of a mandatory service (flight, airport transfer, hotel or catering) fails, all services have to be compensated. In case both, the booking of sight tickets and the booking of event tickets fail, all services have to be compensated. It should be noted that compensation has only to take place if necessary and a TWSO component acts accordingly: A service that has not been started does not need to be compensated. The same is true for a service that has been aborted. Because of the concurrent processing and the usage of transaction dependencies it can happen that — in terms of the current transaction state — invalid transaction primitives are issued. For example, it can happen that a `commit` is issued from the orchestration on a transaction that has been already compensated by a transaction dependency before. Such events are anticipated and ignored safely. It should be noted that for the sake of clarity this scenario implementation does not consider erroneous behavior of transaction related processing per se. That is, unlikely situations where transaction primitives fail are ignored and not handled.

The resulting XPDL orchestration with interspersed TWSOL elements is shown in Appendix C.1.

Scenario 2.1 needs some adjustments over time. To react accordingly to the claim that the flight booking service starts charging each booking attempt, the concurrent order of the workflow needs to be changed as shown in Figure 9.3.

Here, the flight booking happens after all other bookings have been finished to minimize flight booking attempts. In case at least one service apart from the flight booking fails, all transactions (including the transaction that manages the flight booking) are compensated. In this situation the flight booking Web service would be compensated before any (business logic) action has been performed. A business logic call to the flight booking Web service after the compensation of its related transaction is valid, but will not have any effect. Thus, it will not be charged. A p_{commit} on a transaction that is in state $\sigma_{compensated}$ (as it will be the case in this orchestration) is invalid and will cause an exception. Such an exception (as also discussed above) is anticipated and is ignored by the orchestration safely. It should be noted that the transaction logic remains exactly the same and is not affected in any way by the required modification of the orchestration.

To pay tribute to the new customer preference as required by Scenario 2.1 (i.e. customers prefer to have both, event tickets and sight tickets), it is only necessary to change the dependencies. The orchestration workflow remains completely untouched. The dependency system is as follows:

- $\sigma_{aborted}(t_{flight}) \vee \sigma_{aborted}(t_{transfer}) \vee \sigma_{aborted}(t_{hotel}) \vee$
  $\sigma_{aborted}(t_{catering}) \vee \sigma_{aborted}(t_{sights}) \vee \sigma_{aborted}(t_{events})$
  $\Rightarrow$
  $p_{compensate}(t_{flight}), p_{compensate}(t_{transfer}), p_{compensate}(t_{hotel}),$
  $p_{compensate}(t_{catering}), p_{compensate}(t_{events}), p_{compensate}(t_{sights})$

Just a single dependency is necessary now. It should be noted, that the transaction logic holds some atomic characteristics now. However, ACID transactions still cannot be used because preserving isolation would introduce tight and deprecated dependencies between the involved services, as discussed above.

The XPDL orchestration of the adapted orchestration is presented in Appendix C.2.

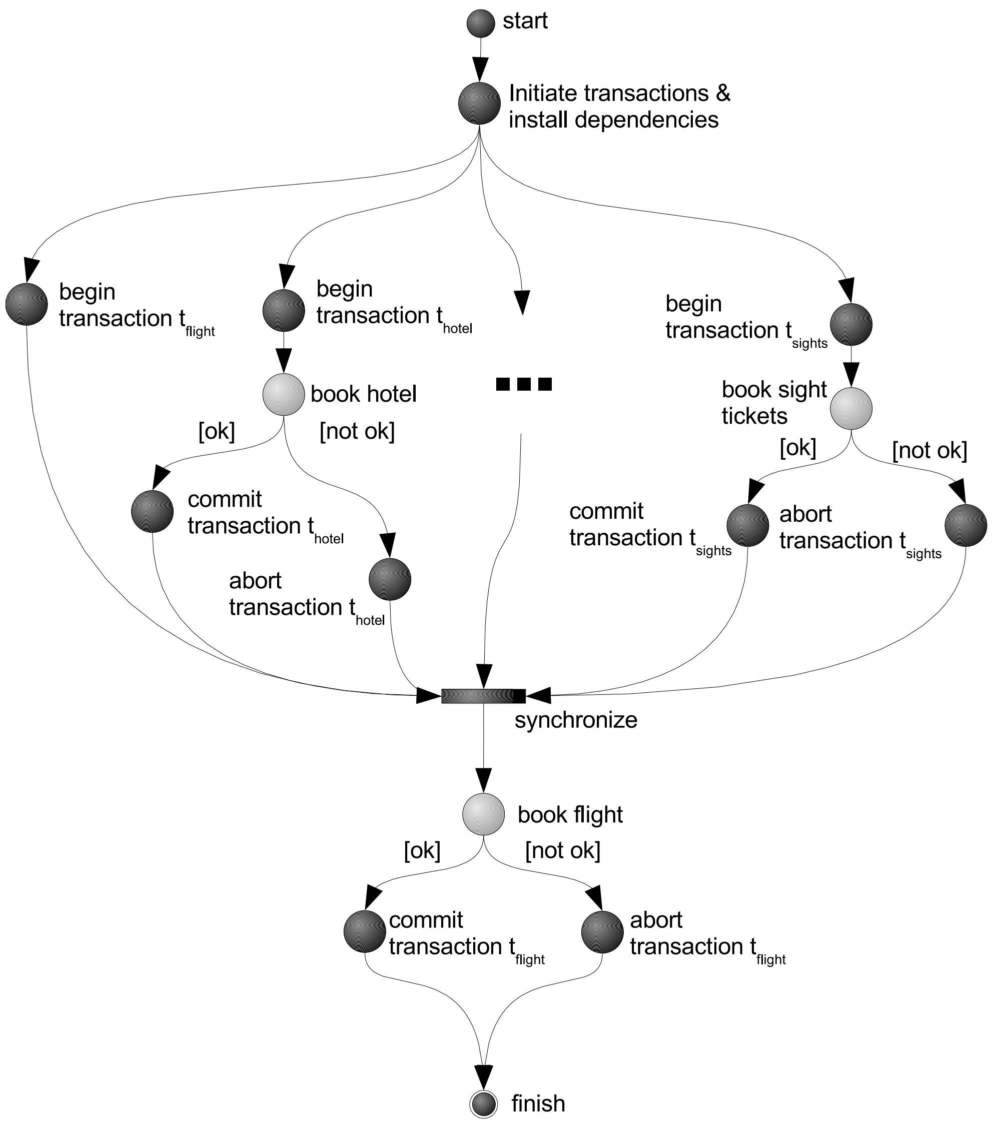
start
Initiate transactions & install dependencies
begin transaction t_{flight}
begin transaction t_{hotel}
begin transaction t_{sights}
book hotel
[ok]
[not ok]
book sight tickets
[ok]
[not ok]
commit transaction t_{hotel}
abort transaction t_{hotel}
commit transaction t_{sights}
abort transaction t_{sights}
synchronize
book flight
[ok]
[not ok]
commit transaction t_{flight}
abort transaction t_{flight}
finish

Figure 9.3: Adapted transactional Vienna city trip orchestration

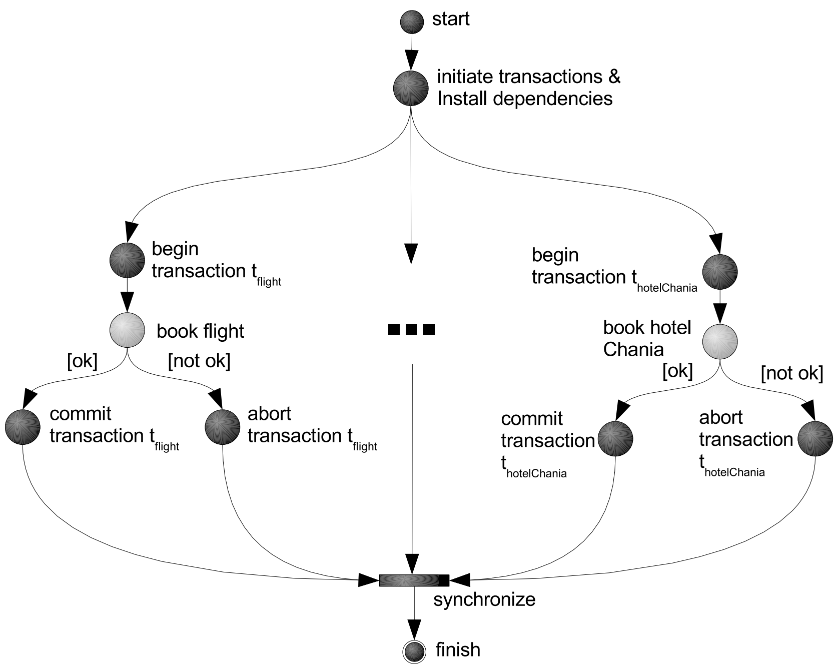
start
initiate transactions & Install dependencies
begin transaction t_flight
book flight
[ok]
[not ok]
commit transaction t_flight
abort transaction t_flight
begin transaction t_hotelChania
book hotel Chania
[ok]
[not ok]
commit transaction t_hotelChania
abort transaction t_hotelChania
synchronize
finish

Figure 9.4: Transactional Crete car tour orchestration

9.2 Rental Car Tour Scenario

For the rental car tour in Crete as introduced in Scenario 2.2, the services do not charge booking requests. Thus, the orchestration and the orchestration's transaction related parts can be adopted from the initial Vienna City Trip scenario, as described in Section 9.1 and depicted in Figure 9.2. The resulting orchestration for 2 services (the other ones are analogous and indicated using "...") is shown in Figure 9.4.

All further logic can be expressed using transaction dependencies. For the flexible traveler type, the transaction dependencies are as follows:

- $\sigma_{aborted}(t_{flight}) \Rightarrow$
  $p_{compensate}(t_{car}), p_{compensate}(t_{hotelHeraklion}),$
  $p_{compensate}(t_{hotelChania}), p_{compensate}(t_{hotelAgiosNikolaos})$

- $\sigma_{aborted}(t_{car}) \wedge \sigma_{aborted}(t_{hotelHeraklion}) \Rightarrow$
  $p_{compensate}(t_{flight}), p_{compensate}(t_{car}), p_{compensate}(t_{hotelHeraklion}),$
  $p_{compensate}(t_{hotelChania}), p_{compensate}(t_{hotelAgiosNikolaos})$

In other words, when no flight can be booked, all other services should be compensated and when both, the hotel in Heraklion and the rental car cannot be booked, all other services should be compensated, too.

For the not so flexible traveler, the orchestration remains totally the same. Only the transaction dependencies have to be adapted as shown next:

- $\sigma_{aborted}(t_{flight}) \vee \sigma_{aborted}(t_{car}) \vee \sigma_{aborted}(t_{hotelHeraklion})$
  $\Rightarrow$
  $p_{compensate}(t_{flight}), p_{compensate}(t_{car}), p_{compensate}(t_{hotelHeraklion}),$
  $p_{compensate}(t_{hotelChania}), p_{compensate}(t_{hotelAgiosNikolaos})$
- $\sigma_{aborted}(t_{hotelChania}) \wedge \sigma_{aborted}(t_{hotelAgiosNikolaos})$
  $\Rightarrow$
  $p_{compensate}(t_{flight}), p_{compensate}(t_{car}), p_{compensate}(t_{hotelHeraklion})$

Here, all services should be compensated in case the flight booking, or the car booking or the booking for the hotel in Heraklion fails. In addition, if no hotel on the tour — i.e. no hotel in Chania and in Agios Nikolaos — can be booked, all services should be compensated, too.

Please note that also this scenario implementation does not consider erroneous behavior of transaction related processing per se for the sake of clarity. Exceptions that stem from unlikely erroneous processing of transaction primitives are ignored.

An XPDL orchestration with interspersed TWSOL constructs that implements the orchestration for both, flexible and not so flexible tourists, is given in Appendix C.3. In this orchestration, an appropriate dependency set is installed dynamically, depending on the current tourist type the orchestration instance is executed for.

9.3 Summary

Coordination issues in the scenario virtual enterprise are easily tackled by using TWSO constructs. As one can see, the efficient concurrent processing in the scenario is coordinated only by few TWSO transaction dependencies. Trying to manage this concurrency with orchestration means alone would result in additional massive control flow logic expressed in the orchestration language. Consequently, a faster development process can be achieved when using TWSO instead of plain orchestration technology.

The implementation of the scenario clearly shows that TWSO can be integrated with Web service orchestrations seamlessly. The XPDL orchestrations in Appendix C.1 present a way to include TWSO constructs that fit in the hosting XPDL orchestration perfectly. This method even allows workflow engines without TWSO capabilities to execute the TWSO orchestration by omitting the transaction logic[1]. For other Web service orchestration technologies like BPEL4WS, accurate TWSO integrations are certainly feasible without any hassles, too. However, developing a TWSO integration proposal for various Web service orchestration technologies is out of scope of this thesis.

Arbitrary transaction semantics are put in operation fast and easily. For example, the alteration of user preferences in the Vienna city trip scenario causes the modification of transaction semantics. In that case, simply adjusting the transaction dependencies accomplishes this. Moreover, transaction semantics can be even installed on–the–fly, as shown in the Crete scenario. Depending on the type of tourist that initiates the orchestration, different transaction semantics go into action here.

The usage pattern of applying transaction–oriented processing in the scenario Web service orchestrations follows the common transaction usage pattern as used in various transaction systems nowadays. A transaction is begun and terminated by transaction primitives and influences the effects of executed business logic. However, the scenario also shows that the usage pattern is enriched significantly. Coordinating a system of multiple transactions by defining transaction dependencies and advanced types of transaction primitives are not part of common transaction processing systems nowadays.

By separating transaction logic and workflow logic to a high degree, separation of concerns is honored and the design of the orchestrations stays remarkably clean. Quality of the orchestration is enhanced and errors of the orchestration itself (i.e. the separated orchestration without its participating Web services) are avoided at design time: High quality fault prevention takes place in a TWSO orchestration implicitly. The high separation of workflow and transaction logic and thus the clean design is observable in the scenario orchestrations: Modifications of the transaction logic do not affect workflow logic and vice versa. Logic is implemented using two different independent layers.

[1]It should be noted that executing the bare workflows of the scenario use cases without any transaction logic would omit important semantics.

Moreover, through anticipating errors a priori at the design time in a highly effective and efficient way by using TWSO transactional concepts, emerging faults most likely do not affect the correct behavior of the orchestration and failure management is facilitated significantly. Occurring errors at runtime are detected and handled effectively. Fault tolerance of the orchestration is improved. The scenario orchestrations consequently check the outcome of Web services for faults and handle them accordingly by using TWSO concepts. Thus, correct processing of the total orchestration is hardly affected by such faults. In terms of [26], the scenario orchestrations show that availability, reliability, safety, integrity and maintainability of the orchestrations is enhanced by providing means of fault prevention and fault tolerance, whereas TWSO concepts provide such means.

Part III

Implementation of a TWSO Environment

Chapter 10

TWSO Prototype Implementation

To prove the feasibility of the theoretical foundations in Part II, a working software prototype of a TWSO system (protoTWSO) was developed. This system includes an interface for a single orchestration technology, a TWSO transaction monitor and sample Web services that are TWSO enabled. protoTWSO should form a proof of concept to evaluate current and future TWSO related concepts. It cannot fulfill all necessary requirements of a production quality TWSO system. However, since extensibility potentials and elegance of the prototype system are on a high level, protoTWSO may build a base for future production grade TWSO systems.

10.1 Overall Architecture

protoTWSO consists of three independent areas. SOAP handles communication between remote components in the entire system. protoTWSO offers an implementation of TWSO4J as described in Subsection 6.4.2. Hence, Java Web service orchestration is the only orchestration technology than can be used in a protoTWSO system. The central element of protoTWSO is a TWSO transaction monitor implementation that complies with Chapter 7. Thus, it is a Web service whose interface conforms to the WSDL definition presented in Appendix B. It should be noted that the transaction monitor could be also used with other orchestration technologies besides Java, but Java is the only orchestration technology protoTWSO supports at the moment. To test TWSO orchestration concepts effectively, protoTWSO provides means to conveniently generate TWSO enabled Web services.

For protoTWSO, orchestrations are expressed in Java. protoTWSO's TWSO4J implementation provides an application programming interface that is used in the Java orchestration and encapsulates communication between the orchestration and the transaction monitor. If necessary, the transaction monitor forwards transaction related commands to the TWSO enabled Web services. In addition, the Java orchestration calls Web service business logic, whereas these calls include a reference to a related transaction.

10.2 TWSO4J Implementation

The TWSO4J implementation of protoTWSO uses Apache AXIS Client tools for SOAP communication. It follows exactly the TWSO4J interfaces as described sufficiently in Subsection 6.4.2, besides the following noteworthy exceptions.

It is not possible to give dependencies using XML as shown in Figure 6.4. Just `Dependency` objects can be used to define transaction dependencies.

In addition to the pure TWSO4J recommendations, protoTWSO's TWSO4J includes an interface `Ec3TransactionMonitor` that extends the `TransactionMonitor` interface by convenience methods:

- `createTransactionContext()`: This method can be used to trigger the generation of up to date SOAP header context information. The context will include a session identifier as returned from the protoTWSO SOAP transaction monitor component.

- `createTransactionContext(String transactionId)`: In addition to `createTransactionContext()`, it is possible to define a transaction identifier that is included in the context.

- `getLocationURI()`: Returns the URI of the transaction monitor

- `getTransactionMonitorStub()`: Returns a low–level SOAP client stub object that can do SOAP communication. Using this object it is possible to initiate all possible low–level SOAP calls.

- `resetStub()`: Replaces the existing SOAP client stub with a completely new one. For example, it is used to delete any created SOAP header information.

All `TransactionMonitor` instances that are created by protoTWSO's `TransactionMonitorFactory` comply with both interfaces, `TransactionMonitor` and `Ec3TransactionMonitor`.

10.3 TWSO Transaction Monitor Implementation

The implementation of the transaction monitor component in protoTWSO is written in Java and externalizes its functionalities for clients via SOAP. Its WSDL definition and thus its published interfaced including methods, parameters and datatypes can be seen in Appendix B. Since the usage is covered in detail in Chapter 7, this chapter concentrates on noteworthy implementation details and strategies.

The protoTWSO transaction monitor is based on the Apache AXIS SOAP toolkit. The center of the implementation builds the WSDL document. Using AXIS WSDL tools, suitable server skeleton classes are generated from this WSDL document. Moreover, all complex XML Schema datatypes found in the WSDL are transformed automatically to Java classes using AXIS tools, too. The `TransactionMonitorServerImplementation` class implements the interface and is the starting point for all functionalities.

To keep track of particular clients, a proprietary session handling is used[1]. After logging in with the `login()` method, a session identifier is returned to the client. This identifier has to be included in the transaction context for all calls during a session as described in Chapter 5. On the server–side, stateful objects are bound to the session identifier in order to preserve the object's state across all SOAP calls in a session. The code for the server–side session handling is included in the `TransactionMonitorServerImplementation` class.

Transaction dependencies are represented using the `Dependency` class. In order to evaluate dependencies, the following strategy is used. The `BSHDependency` class extends `Dependency` and contains evaluation and execution logic for evaluating whether the source state is in effect and — if so — to execute the effect. To check whether the source state is in effect, scripting

[1]Although there are standardized approaches for Web service session management like WSRF, it was decided to create a proprietary simple session handling. Clients simply have to add the session identifier to the transaction context, which is needed anyway. The complexity of using WSRF outweighs its advantages for protoTWSO.

technologies are used. The data related to the source state in the `Dependency` object is converted to a script[2] on the fly. The script is executed and returns true or false. Additionally, it is possible to apply the effect of a dependency by using the `doEffect()` method. Source states of dependencies are evaluated immediately after they were installed and after a primitive is executed. When the evaluation reports that the source state is in effect, the dependency's effect is executed.

A primitive request on a transaction is executed mainly by forwarding it to affected Web services. This logic is located in the `PrimitiveExecutor` class. A transaction that manages multiple Web services has to assure a valid outcome across all involved Web services for a primitive. protoTWSO uses the two phase commit protocol to achieve such an outcome. When a `PrimitiveExecutor` object detects that a transaction is related to multiple Web services, it precedes a prepare phase and — when the prepare phase indicated success — forwards the primitive to all Web services.

A `delegation` for termination causes no forwarding but stores the desired delegation. In detail, a transaction has a collection of delegates and is responsible to terminate each transaction in this delegate collection. The collection is empty where there are no delegates. When termination of t_j and t_k should be delegated to t_i, t_j and t_k are added to the delegates collection of t_i. When t_i is terminated by $p_{termination\_type}$, the software iterates through t_i's collection of delegate transactions and terminates each entry in the collection with $p_{termination\_type}$. Hence, t_j and t_k are terminated by $p_{termination\_type}$. When a transaction has delegates, it must be assured that a primitive leads to an agreed and valid outcome across the transaction and its delegates. Consequently, the two phase protocol is applied in this case, too, as it is described above. It should be noted, that delegates could have delegates themselves, and recursive processing (including two phase commit actions) could take place.

10.4 TWSO Enabled Web Services in protoTWSO

protoTWSO provides a convenience class for TWSO enabled Web services. Classes that extend the `TWSOWebservice` class obtain a full TWSO compatible interface that can be published via Web service means as it is. Further-

[2]Beanshell scripting is used in protoTWSO.

more, some utility functions are provided. Besides a rudimentary but functional `doPrimitive()` operation, the Web service will include two phase commit protocol functionality. However, WSDL definitions that include TWSO capabilities as described in Section 8.1, are not created automatically and have to be developed by hand.

10.5 Limitations of Prototype Implementation

protoTWSO is a playground to test TWSO concepts. It does not assert claims to be a production grade Web service transaction system. Thus, protoTWSO has the following limitations that are irrelevant regarding the purpose of protoTWSO:

- no distribution of compiled source code,
- lack of installation/integration/deployment auxiliaries,
- Apache Tomcat on Linux is the only tested platform,
- only most essential configuration possibilities,
- no performance optimization,
- rudimentary error handling and input verification.

protoTWSO does not provide any installation auxiliaries. The components are available as sourcecode. Compilation, deployment and integration have to be performed manually without any software assistance.

Web applications of protoTWSO comply with Java Servlet specifications and definitely run on the Apache Tomcat Servlet container under Linux. It may be possible to use other Servlet containers on other operation systems as well, but this is not tested.

Moreover, only essential configuration options are provided in configuration files. There is no user–friendly interface to change these options.

Generally, protoTWSO does not focus performance related optimizations. There are several areas of code that are not optimized in terms of performance. However, these areas are marked, so that they can be spotted and refactored quickly.

Error handling and input verification are marginally implemented. The exception system is very coarse grained and lacks a common strategy on how exceptions should be propagated. Moreover, inputs from outside are not checked at all. It is assumed that all participating components behave correctly and give and receive correct data only. This is the most problematic area in protoTWSO and has to be addressed primarily when protoTWSO should be upgraded to production quality.

Chapter 11

Evaluation

To demonstrate the operativeness of protoTWSO, the Crete rental car tour scenario (as introduced in Scenario 2.2, developed in Section 9.2, and instantiated with XPDL in Appendix 9.4) was realized using protoTWSO. The resulting Java orchestration is shown and explained in detail in Appendix C.2.

11.1 Performed Testing

Numerous test runs demonstrated that ProtoTWSO is able to execute the scenario correctly. The used test design is as follows. For each booking action, a test Web service was created that pretends to offer the desired functionality. Moreover, it is possible to configure each test Web service either to pretend success or failure, whereas the failure type can be determined in detail. The possible failure types (see also Table 1.1) are as follows: Transfer failures, expected service failures, and unexpected service failures. For example, it is possible to configure the systems as follows: The booking of a hotel room in Chania delivers a Nullpointer exception (unexpected service failure), the host of the car booking service in Heraklion is temporarily unavailable (transfer failure), and there are no hotel rooms left in Heraklion (expected service failure).

Numerous test runs have been executed, which were composed as follows. Based on the assumptions above, there are 4 service outcomes (success, transfer failure, expected service failure, unexpected service failure) and 5 services (flight booking, car booking, hotel booking Heraklion, Chania, and Agios Nikolaos). Thus, a total of $4^5 = 1024$ test cases exist. Because the tests

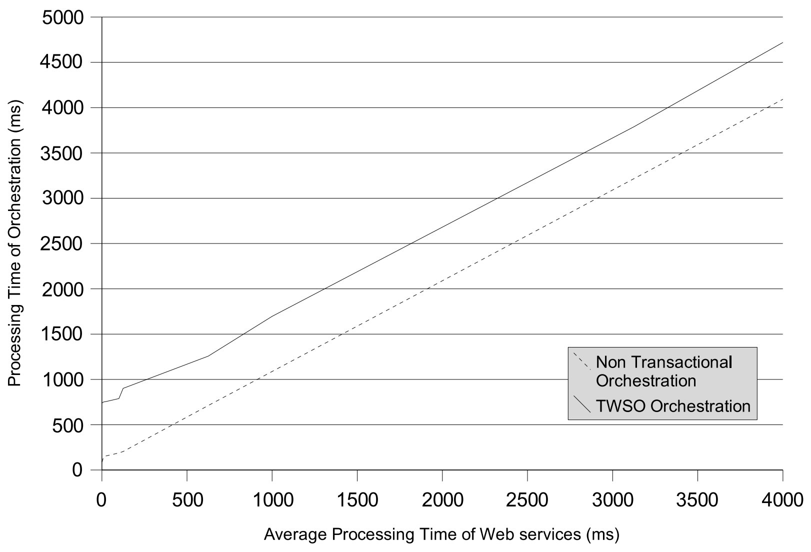

Figure 11.1: Transactional Crete car tour orchestration measurements

involve significant manual activities, it was decided to scale the test cases down reasonably. Thus, the possible outcomes were reduced to two options: Either success, or — randomly chosen — transfer failure, expected or unexpected service failure. This results in $2^5 = 32$ systematic test cases, which seem to be sufficient to validate protoTWSO.

Each test run has been applied to the flexible tourist's and not so flexible tourist's orchestration. The outcomes have been validated manually and have been all considered to be valid. Therefore, it stands to reason that the protoTWSO system acts correctly in terms of the TWSO approach.

11.2 Performance Measurements

To get an idea of performance implications when using a TWSO system, runtime behavior of the Crete rental car tour scenario implementation in protoTWSO has been measured. The protoTWSO orchestration was compared to an orchestration that does not consider transactions at all.

Figure 11.1 shows the duration of orchestration executions related to the average duration of Web service processing. There is a constant significant overhead of TWSO related processing. It stands to reason, that the longer services in an orchestration need to fulfill their tasks, the lower is the significance of TWSO related processing time. The measurements were observed on a 3.6GHz Pentium 4 Linux system that executed protoTWSO in a Tomcat 5.5 container using Sun Java 1.4.2.

11.3 Towards an Automated Test Environment for Web Service Transaction Systems

The burden of manually testing Web service transaction systems as it was necessary for the validation of protoTWSO raises the wish for an automated test environment for Web service transaction systems. On the one hand, such a test environment reduces the efforts to test Web services transaction systems. On the other hand side, standardized test environments allow meaningful comparisons of different Web service transaction systems in terms of correctness and performance. Such a test environment may be designed as follows.

To objectively validate the correctness of the outcome of a transactional Web service orchestration instance executed in a transactional orchestration system, the following factors have to be taken into account.

First of all, the set of final states of the participating Web services has to be checked. Since the types of possible final Web service states depend on the particular domain, orchestration, and Web service, a universal set of final Web service states cannot be defined. For example, a particular flight booking Web service may have the states `success`, `flight_unavailable`, and `failure` whereas a Web service of a hotel provides only the states `success` and `failure`.

Also, the final states of the transaction instances have to be considered because they have to be consistent with the final states of the participating Web service. For example, let transaction t_i manage s_u. A final state $\sigma_{failure}$ for Web service s_u and $\sigma_{failure}$ for transaction t_i may imply an erroneous inconsistency. However, again the consistency conditions depend on the particular domain, orchestration, and Web service(s).

Since no universal relationships between final Web service states, final transaction states, and orchestration validity can be deduced, validity constraints have to be developed for each particular testing. Given a particular orchestration, such constraints should express a combination of final Web service states and final transaction states that indicate a valid orchestration's outcome.

To provide valuable test settings, the test environment must be able to influence the activities of participating Web services. Mock–up Web services, that do not provide any business related functionality, but can be configured to simulate relevant behavior for testing purposes, are necessary. Three configuration dimensions of such Web services are noteworthy.

The states a Web services can have and a realistic external visible behavior related to a state must be simulated. For example, a mock–up Web service for the hotel booking stated above has two states and returns a success message when it enters the `success` state and a failure message when it enters the `failure` state.

A statistical concept should be used to determine which kind of state the mock–up Web service should enter when it is called. For example, it should be possible to specify that the flight booking mock–up Web service should go into the `flight_unavailable` state with a probability of 80%, into the `success` state with 18% probability, and into the `failure` state with a probability of 2%.

Also, the time dimension should be considered to provide a realistic test environment. A mock–up Web service should simulate the response time as subject to some kind of statistical concept and the intended state. For instance, when the hotel booking mock–up is intended to enter the `success` state, the response time should adhere to a particular Gaussian distribution.

To test Web service transaction systems on the basis of the Crete rental car tour Scenario for the not so flexible tourist, the test framework may be composed as follows. It should be noted, that this test configuration tries to approximate the scenario as good as possible.

There are 5 mock–up Web services that are shown in Table 11.1. All these Web services are configured to have some states, which are entered with some probability and have a response time that adheres to a certain statistical distribution. A set of validity constraints in natural language may

Table 11.1: Test mock–up Web Services for Crete rental tour

Mock–Up	**Success**		**Expected Service Failure**		**Unexpected Service Failure**		**Transfer Failure**	
	P_{entry}	D_{time}	P_{entry}	D_{time}	P_{entry}	D_{time}	P_{entry}	D_{time}
Flight	0.8	N($\mu = 2$, $\sigma^2 = 0.01$)	0.1	N($\mu = 3$, $\sigma^2 = 0.02$)	0.05	$\chi^2(k = 3)$	0.05	$\chi^2(k = 3)$
Car	0.5	N($\mu = 7000$, $\sigma^2 = 0.5$)	0.2	N($\mu = 4000$, $\sigma^2 = 0.4$)	0.1	$\chi^2(k = 5)$	0.2	$\chi^2(k = 4)$
Hotel Heraklion	0.7	N($\mu = 1000$, $\sigma^2 = 0.1$)	0.2	N($\mu = 1400$, $\sigma^2 = 0.3$)	0.02	$\chi^2(k = 4)$	0.08	$\chi^2(k = 5)$
Hotel Chania	0.4	N($\mu = 2000$, $\sigma^2 = 0.25$)	0.3	N($\mu = 500$, $\sigma^2 = 0.5$)	0.2	$\chi^2(k = 3)$	0.1	$\chi^2(k = 4)$
Hotel Agios Nikolaos	0.3	N($\mu = 3000$, $\sigma^2 = 0.4$)	0.1	N($\mu = 3500$, $\sigma^2 = 0.2$)	0.2	$\chi^2(k = 4)$	0.4	$\chi^2(k = 5)$

P_{entry}: Probability of a mock–up service to enter a state
D_{time}: Response time distribution
χ^2(k=n): Chi–Square distribution with n degrees of freedom
N($\mu = i$, $\sigma^2 = k$): Gaussian distribution with mean i and standard deviation k

be as follows:

- Definition: A Web service *failed* when it is in state `expected service failure`, `unexpected service failure`, or `transfer failure`.
- The transactional orchestration is valid when
  - flight service and car service succeeded, and
  - hotel booking in Heraklion succeeded, and
  - hotel booking in Chania or in Agios Nikolaos succeeded.
- All transactions that are associated to failed services must have been aborted or compensated.

Moreover, to allow systematic testing, a generator should be able to configure the mock–up Web services during a test session according to some given strategy. The Crete rental car scenario, for example, has 1024 possible combinations of Web service end states as described above. A generator could generate all these end states without manual intervention to test the transaction system entirely.

Part IV

Epilogue

Chapter 12

Conclusion

Web services offer possibilities to invoke functionalities on remote systems using the Internet. As the maturity of basic Web service technology meets the requirements of many domains by now, demands for combining the functionalities of several Web services — i.e. using Web service and Internet technologies for complex distributed systems like virtual enterprises — emerge. A first step towards the support of these demands is a simple combination of multiple Web services by so–called Web service orchestration technologies.

Transactional processing is an imperative for established distributed systems technologies like CORBA or Enterprise Java (J2EE). These technologies offer such functionality as a matter of course. However, existing transactional processing concepts can only be used marginally in distributed Web service systems since they were built for intra–organizational and short–time interactions. Distributed Web service systems typically cross the boundaries of organizations and involve long–time interactions. Thus, approaches for transactional processing in Web service systems have been developed and published as proposals. However, they did not succeed until now. These proposals lack sufficient integration in Web service orchestrations, sufficient semantical adaptability for usage in arbitrary domains, and a clear usage concept.

The major result of this thesis is TWSO, a transaction–processing system for Web service orchestrations. TWSO is based on a sound scientific base and forms a programmatic transaction processing system for Web services. The TWSO approach presents concepts that overcome the shortcomings of recent Web service transaction proposals.

To be able to use TWSO in virtually any domain, it is possible to define arbitrary transaction semantics. Virtually anything, ranging from traditional ACID transactions over Sagas to own custom transaction semantics, is possible. TWSO concepts were developed to enable integration with Web service orchestration technologies. Consequently, integration with virtually any kind of Web service orchestration technology like BPEL4WS, XPDL or even Java is achievable in a straightforward way. This thesis presents an XPDL integration as well as a Java integration. Moreover, TWSO provides a clear usage pattern that resembles the usage pattern that is used in common current transaction systems. Thus, Web service orchestration designers are able to grasp the advantages of, understand, and use a TWSO system without major difficulties quickly.

The goals defined in Section 1.2 have been achieved by this thesis. On the basis of relevant scenarios it has been shown, that TWSO can be integrated in Web service orchestrations in order to fit them out with transactional behavior, and that arbitrary transaction logic can be arranged without hassles. Moreover, it has been demonstrated that the resulting orchestrations provide fault tolerance and fault prevention and hence enhance dependability. The scenario implementations also emphasized the clear usage pattern of TWSO. Other goals related to the use of TWSO in virtual enterprises as presented in Section 2.2 were accomplished, too. It has been shown, that TWSO is able to cut coordination efforts in virtual enterprises and has the ability to accelerate the creation and adaptation of virtual enterprises significantly.

Chapter 13

Future Steps

Future work will focus on a production quality TWSO system implementation. The prototype presented in Part III allows efficient and accurate hands–on testing of TWSO concepts, but lacks features required for production quality like stability, performance and ease of use. Thus, the prototype will be consequently improved and enhanced to offer a production grade Web service transaction system that can be used in real world distributed systems.

The usage of such a production grade TWSO system in real world systems like a virtual enterprise in the tourism domain as introduced in Chapter 2 may provide significant feedback. Thus, another future goal is to gather as much feedback as possible. These valuable experiences will be integrated in future versions of TWSO systems. Again, feedback of future versions will be collected and used for improvements, and so on. Consequently, the TWSO approach and its implementations will be improved by a continuous integration process.

Furthermore, an automatic test environment for Web service transaction systems as proposed in Chapter 11 will be developed. Such a test environment will offer invaluable support for validating and comparing Web service transaction systems. Moreover, to spur on Web service transaction computing in the Web service community, contests similar to other Web service composition contests like [23] should be held. In such a contest, different Web service transaction systems would compete and be ranked based on the results achieved in the test environment.

The developed prototype provides a TWSO integration for Java Web service orchestrations only. Compared to Java Web service orchestrations, XML Web service orchestrations operate on a higher conceptual level to reduce complexity. In many domains, this reduced complexity is sufficient. XML Web service orchestrations may be preferred to Java orchestrations in such domains. Thus, it is necessary to provide TWSO integration concepts with prominent XML Web service orchestration technologies. For XPDL, such a concept has been developed and is shown in Appendix C.1. Besides XPDL, another crucial orchestration technology at the moment is BPEL4WS. An approach for an integration of TWSOL in BPEL4WS orchestrations is absolutely needed and will be created in a next step.

Moreover, to execute TWSO XML Web service orchestrations adequately, TWSO enabled orchestration engines are necessary. A reasonable possibility to create TWSO enabled orchestration engines is the extension of existing Web service orchestration engines. For XPDL and BPEL4WS, there are high–quality Web service orchestration engines, that are distributed as open–source software. It is possible to add TWSO functionality to those orchestration engines without major hurdles. To support TWSO in BPEL4WS processes, the ActiveBPEL [1] engine will be enhanced. For XPDL orchestrations including TWSO concepts, Enhydra's XPDL engine Shark [45] will be extended.

Part V

Appendices

Appendix A

Web Service Description Language

Web services are described using the Web Service Description Language (WSDL) [12]. The main components of a WSDL description are as described below and presented in Figure A.1.

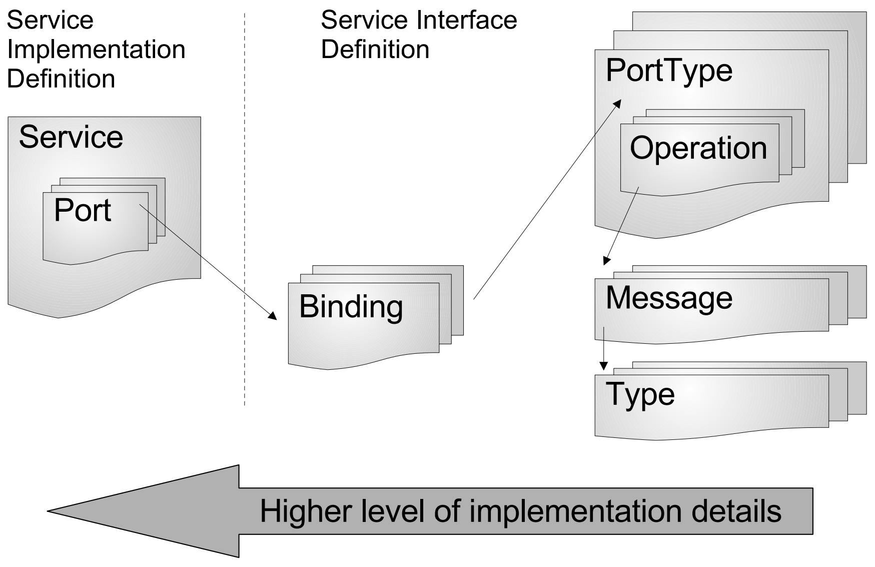
Service Implementation Definition
Service Interface Definition
Service
Port
Binding
PortType
Operation
Message
Type
Higher level of implementation details

Figure A.1: Elements of a WSDL definition

- The hub of a WSDL Web service description is the *PortType*, which represents an implementation neutral interface to some service.

- A PortType consists of one or more *operations*. An operation operates on an input message and returns an output message. These messages are of particular types, which are normally defined using XML–Schema [43].
- *Bindings* define message formats and protocol details for operations and messages defined by a particular portType. It is possible that there are multiple bindings for a single PortType. In this case, there are multiple implementations of a PortType.
- *Ports* define network endpoints of bindings. In case of a SOAP binding, a port contains the URL of the SOAP Web service. A binding can have multiple ports and a single port can have multiple bindings. Thus, a port can implement multiple bindings and a binding can have different alternative implementations.
- *Services* group such related ports together. For example, ports that are associated to bindings that share the same PortType are related and grouped in a single service to express that these ports can be used alternatively.

Appendix B

TWSO Transaction Monitor WSDL

Listing B.1 presents the complete WSDL interface description of a TWSO transaction monitor. TWSO transaction monitors are described in detail in Chapter 7.

Listing B.1: Transaction Monitor WSDL Definition

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="TransactionMonitor"
  targetNamespace="http://move.ec3.at/twso/TransactionMonitor/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:twso="http://move.ec3.at/twso/TransactionMonitor/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <wsdl:types>
    <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      targetNamespace="http://move.ec3.at/twso/TransactionMonitor/">

      <xsd:complexType name="wsReferenceCollection">
        <xsd:sequence>
          <xsd:element name="webServiceReference" type="twso:webServiceReference" minOccurs="1"
            maxOccurs="unbounded">
          </xsd:element>
        </xsd:sequence>
      </xsd:complexType>

      <xsd:complexType name="webServiceReference">
        <xsd:sequence>
          <xsd:element name="wsdlURI" type="xsd:string"></xsd:element>
          <xsd:element name="binding" type="twso:qualifiedName"></xsd:element>
          <xsd:element name="bindingOperation" type="xsd:string" minOccurs="0"></xsd:element>

        </xsd:sequence>
      </xsd:complexType>

      <xsd:complexType name="qualifiedName">
        <xsd:sequence>
          <xsd:element name="namespace" type="xsd:string"></xsd:element>
          <xsd:element name="localName" type="xsd:string"></xsd:element>
        </xsd:sequence>
      </xsd:complexType>

      <xsd:complexType name="transactionState">
        <xsd:sequence>
          <xsd:element name="type" type="xsd:string"></xsd:element>
          <xsd:element name="transactionID" type="xsd:string"></xsd:element>
        </xsd:sequence>
      </xsd:complexType>
      <xsd:complexType name="operator">
        <xsd:sequence maxOccurs="unbounded" minOccurs="1">
          <xsd:element name="operator" type="twso:operator" minOccurs="0" maxOccurs="unbounded"/>
          <xsd:element name="transactionState" type="twso:transactionState" minOccurs="0"
            maxOccurs="unbounded">
          </xsd:element>
        </xsd:sequence>
        <xsd:attribute name="type" type="xsd:string"></xsd:attribute>
      </xsd:complexType>

      <xsd:complexType name="dependency">
        <xsd:sequence>

          <xsd:element name="sourceState">
            <xsd:complexType>
              <xsd:choice>
                <xsd:element name="transactionState" type="twso:transactionState"></xsd:element>
                <xsd:element name="operator" type="twso:operator"></xsd:element>
              </xsd:choice>
            </xsd:complexType>
          </xsd:element>

          <xsd:element name="effect">
            <xsd:complexType>
              <xsd:sequence>
                <xsd:element name="primitive" minOccurs="0" maxOccurs="unbounded">
                  <xsd:complexType>
                    <xsd:sequence></xsd:sequence>
                    <xsd:attribute name="type" type="xsd:string"></xsd:attribute>
                    <xsd:attribute name="source" type="xsd:string"></xsd:attribute>
                    <xsd:attribute name="target" type="xsd:string"></xsd:attribute>
                  </xsd:complexType>
                </xsd:element>
              </xsd:sequence>
            </xsd:complexType>
          </xsd:element>

        </xsd:sequence>
      </xsd:complexType>
    </xsd:schema>
  </wsdl:types>

  <wsdl:message name="createTransactionResponse">
    <wsdl:part name="id" type="xsd:string" />
  </wsdl:message>
  <wsdl:message name="createTransactionRequest">
    <wsdl:part name="name" type="xsd:string" />
    <wsdl:part name="wsReferenceCollection" type="twso:wsReferenceCollection" />
  </wsdl:message>
  <wsdl:message name="doPrimitiveResponse">
    <wsdl:part name="status" type="xsd:string" />
  </wsdl:message>
  <wsdl:message name="doPrimitiveRequest">
    <wsdl:part name="primitiveType" type="xsd:string" />
    <wsdl:part name="sourceTransactionID" type="xsd:string" />
  </wsdl:message>
  <wsdl:message name="setDependencyResponse">
    <wsdl:part name="status" type="xsd:string" />
  </wsdl:message>
  <wsdl:message name="setDependencyRequest">
    <wsdl:part name="dependency" type="twso:dependency" />
  </wsdl:message>
  <wsdl:message name="doPrimitiveException">
    <wsdl:part name="doPrimitiveException" type="xsd:string" />
  </wsdl:message>
  <wsdl:message name="primitiveOnInvalidTransactionStateException">
    <wsdl:part name="primitiveOnInvalidTransactionStateException" type="xsd:string"/>
  </wsdl:message>
  <wsdl:message name="setDependencyException">
    <wsdl:part name="setDependencyException" type="xsd:string" />
  </wsdl:message>
  <wsdl:message name="createTransactionException">
    <wsdl:part name="createTransactionException" type="xsd:string" />
  </wsdl:message>
  <wsdl:message name="loginResponse">
    <wsdl:part name="sessionId" type="xsd:string" />
  </wsdl:message>
  <wsdl:message name="loginRequest" />
  <wsdl:message name="logoutResponse">
    <wsdl:part name="status" type="xsd:string" />
  </wsdl:message>
  <wsdl:message name="logoutRequest" />
```

```
<wsdl:portType name="TransactionMonitor">
  <wsdl:operation name="createTransaction">
    <wsdl:input message="twso:createTransactionRequest" />
    <wsdl:output message="twso:createTransactionResponse" />
    <wsdl:fault name="createTransactionException" message="twso:createTransactionException" />
  </wsdl:operation>
  <wsdl:operation name="doPrimitive">
    <wsdl:input message="twso:doPrimitiveRequest" />
    <wsdl:output message="twso:doPrimitiveResponse" />
    <wsdl:fault name="doPrimitiveException" message="twso:doPrimitiveException" />
    <wsdl:fault name="primitiveOnInvalidTransactionStateException"
              message="twso:primitiveOnInvalidTransactionStateException"/>
  </wsdl:operation>
  <wsdl:operation name="setDependency">
    <wsdl:input message="twso:setDependencyRequest" />
    <wsdl:output message="twso:setDependencyResponse" />
    <wsdl:fault name="setDependencyException" message="twso:setDependencyException" />
  </wsdl:operation>
  <wsdl:operation name="login">
    <wsdl:input message="twso:loginRequest" />
    <wsdl:output message="twso:loginResponse" />
  </wsdl:operation>
  <wsdl:operation name="logout">
    <wsdl:input message="twso:logoutRequest" />
    <wsdl:output message="twso:logoutResponse" />
  </wsdl:operation>
</wsdl:portType>


<wsdl:binding name="TransactionMonitorSOAP" type="twso:TransactionMonitor">
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
  <wsdl:operation name="createTransaction">
    <soap:operation soapAction="http://move.ec3.at/twso/TransactionMonitor/createTransaction" />
    <wsdl:input>
      <soap:body use="literal" namespace="http://move.ec3.at/twso/TransactionMonitor/" />
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal" namespace="http://move.ec3.at/twso/TransactionMonitor/" />
    </wsdl:output>
    <wsdl:fault name="createTransactionException">
      <soap:fault namespace="http://move.ec3.at/twso/TransactionMonitor/" use="literal"
        name="createTransactionException" />
    </wsdl:fault>
  </wsdl:operation>
  <wsdl:operation name="doPrimitive">
    <soap:operation soapAction="http://move.ec3.at/twso/TransactionMonitor/doPrimitive" />
    <wsdl:input>
      <soap:body use="literal" namespace="http://move.ec3.at/twso/TransactionMonitor/" />
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal" namespace="http://move.ec3.at/twso/TransactionMonitor/" />
    </wsdl:output>
    <wsdl:fault name="doPrimitiveException">
      <soap:fault namespace="http://move.ec3.at/twso/TransactionMonitor/" use="literal"
        name="doPrimitiveException" />
    </wsdl:fault>
    <wsdl:fault name="primitiveOnInvalidTransactionStateException">
        <soap:fault namespace="http://move.ec3.at/twso/TransactionMonitor/" use="literal"
                name="primitiveOnInvalidTransactionStateException" />
    </wsdl:fault>
  </wsdl:operation>
  <wsdl:operation name="setDependency">
    <soap:operation soapAction="http://move.ec3.at/twso/TransactionMonitor/setDependency" />
    <wsdl:input>
      <soap:body use="literal" namespace="http://move.ec3.at/twso/TransactionMonitor/" />
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal" namespace="http://move.ec3.at/twso/TransactionMonitor/" />
    </wsdl:output>
    <wsdl:fault name="setDependencyException">
      <soap:fault namespace="http://move.ec3.at/twso/TransactionMonitor/" use="literal"
        name="setDependencyException" />
    </wsdl:fault>
  </wsdl:operation>
  <wsdl:operation name="login">
    <soap:operation soapAction="http://move.ec3.at/twso/TransactionMonitor/login" />
    <wsdl:input>
      <soap:body use="literal" namespace="http://move.ec3.at/twso/TransactionMonitor/" />
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal" namespace="http://move.ec3.at/twso/TransactionMonitor/" />
    </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="logout">
      <soap:operation soapAction="http://move.ec3.at/twso/TransactionMonitor/logout" />
      <wsdl:input>
        <soap:body use="literal" namespace="http://move.ec3.at/twso/TransactionMonitor/" />
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal" namespace="http://move.ec3.at/twso/TransactionMonitor/" />
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>


  <wsdl:service name="TransactionMonitor">
    <wsdl:port binding="twso:TransactionMonitorSOAP" name="TransactionMonitorSOAP">
      <soap:address location="http://move.ec3.at:8080/twso/services/TransactionMonitorSOAP" />
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Appendix C

Scenario Orchestrations

C.1 XPDL Scenario Orchestrations

C.1.1 TWSOL in XPDL

To intersperse TWSOL elements into XPDL orchestrations (i.e. to enrich XPDL orchestrations with TWSO transactions) an existent XPDL extensibility mechanism is used. XPDL provides `<ExtendedAttribute>` elements to express additional entity characteristics that are not offered by XPDL itself. TWSOL elements are inserted in `<ExtendedAttribute>` elements of XPDL `<Activity>` elements. Thus, when a TWSO(L)–enabled workflow engine executes an activity and comes across a TWSOL extended attribute in this activity, it fulfills the stated TWSO related tasks.

C.1.2 TWSOL XPDL Scenario Orchestration Documents

This section presents XPDL orchestrations that implement the scenarios in Section 2.1 and 2.2. These orchestrations use TWSOL to tackle coordination and error management tasks. For the sake of clarity, exceptions that stem from transaction processing per se are ignored and as parameter passing details are omitted. Listing C.1 shows the initial Vienna city trip scenario orchestration and Listing C.2 the adapted one. The Crete rental car tour scenario orchestration is presented in Listing C.3.

Listing C.1: Initial Vienna City Trip in XPDL

```
<?xml version="1.0" encoding="UTF-8"?>
<Package Id="viennaCityTour" Name="viennaCityTour" xmlns="http://www.wfmc.org/2002/XPDL1.0"
  xmlns:twso="http://move.ec3.at/twso/20060101"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.wfmc.org/2002/XPDL1.0
```

```
␣␣http://wfmc.org/standards/docs/TC-1025_schema_10_xpdl.xsd">
  <PackageHeader>
    <XPDLVersion>1.0</XPDLVersion>
    <Vendor>Together</Vendor>
    <Created>2006-01-31 17:22:28</Created>
  </PackageHeader>

  <WorkflowProcesses>
    <WorkflowProcess Id="viennaCityTour1" Name="viennaCityTour1">
      <ProcessHeader>
        <Created>2006-01-31 17:23:32</Created>
      </ProcessHeader>
      <Participants>
        <Participant Id="OrchestrationEngine">
          <ParticipantType Type="SYSTEM" />
        </Participant>
      </Participants>

      <Applications>
        <Application Id="starAllianceBooking" Name="starAllianceBooking">
          <ExternalReference location="http://webservices.staraliance.com/flightBooking"
            namespace="http://webservices.staraliance.com/flightBooking" xref="bookFlight" />
        </Application>
        <Application Id="viennaAirport" Name="viennaAirport">
          <ExternalReference location="http://webservices.viennaairport.com/"
            namespace="http://webservices.viennaairport.com" xref="bookCityTransfer" />
        </Application>
        <Application Id="tiscover" Name="tiscover">
          <ExternalReference location="http://ws.tiscover.com/booker"
            namespace="http://ws.tiscover.com/booker" xref="bookAccomodation" />
        </Application>
        <Application Id="lokaltipp" Name="lokaltipp">
          <ExternalReference location="http://www.lokaltipp.at/webservice"
            namespace="http://www.lokaltipp.at" xref="book_coupons" />
        </Application>
        <Application Id="events" Name="events">
          <ExternalReference location="http://soap.events.at/" namespace="http://www.events.at"
            xref="tickets" />
        </Application>
        <Application Id="vienna_service" Name="vienna_service">
          <ExternalReference location="http://www.wien.gv.at/soap"
            namespace="http://www.wien.gv.at/" xref="purchase_sight_tickets" />
        </Application>
      </Applications>



      <Activities>
        <Activity Id="installDependencies" Name="installDependencies">
          <Implementation>
            <No />
          </Implementation>
          <Performer>OrchestrationEngine</Performer>

          <TransitionRestrictions>
            <TransitionRestriction>
              <Split Type="AND">
                <TransitionRefs>
                  <TransitionRef Id="viennaCityTour1_tra1" />
                  <TransitionRef Id="viennaCityTour1_tra4" />
                  <TransitionRef Id="viennaCityTour1_tra13" />
                  <TransitionRef Id="viennaCityTour1_tra14" />
                  <TransitionRef Id="viennaCityTour1_tra15" />
                  <TransitionRef Id="viennaCityTour1_tra16" />
                </TransitionRefs>
              </Split>
            </TransitionRestriction>
          </TransitionRestrictions>

          <ExtendedAttributes>
            <ExtendedAttribute Name="twsol">

              <twso:initiate id="txFlight">
                <twso:activityRef>book_flights</twso:activityRef>
              </twso:initiate>
              <twso:initiate id="txTransfer">
                <twso:activityRef>book_airport_transfer</twso:activityRef>
              </twso:initiate>
              <twso:initiate id="txHotel">
                <twso:activityRef>book_hotel</twso:activityRef>
              </twso:initiate>
              <twso:initiate id="txCatering">
                <twso:activityRef>book_catering</twso:activityRef>
              </twso:initiate>
              <twso:initiate id="txEvents">
                <twso:activityRef>book_event_tickets</twso:activityRef>
              </twso:initiate>
              <twso:initiate id="txSights">
                <twso:activityRef>book_sight_tickets</twso:activityRef>
              </twso:initiate>

              <twso:dependency>
                <twso:sourceState>
                  <twso:operator type="or">
                    <twso:transactionState type="twso:aborted" transactionID="txFlight" />
                    <twso:transactionState type="twso:aborted" transactionID="txTransfer" />
                    <twso:transactionState type="twso:aborted" transactionID="txHotel" />
                    <twso:transactionState type="twso:aborted" transactionID="txCatering" />
                  </twso:operator>
                </twso:sourceState>

                <twso:effect>
                  <twso:primitive type="twso:compensate" target="txFlight" />
                  <twso:primitive type="twso:compensate" target="txTransfer" />
                  <twso:primitive type="twso:compensate" target="txHotel" />
                  <twso:primitive type="twso:compensate" target="txCatering" />
                  <twso:primitive type="twso:compensate" target="txSights" />
                  <twso:primitive type="twso:compensate" target="txEvents" />
                </twso:effect>
              </twso:dependency>

              <twso:dependency>
                <twso:sourceState>
                  <twso:operator type="and">
                    <twso:transactionState type="twso:aborted" transactionID="txSights" />
                    <twso:transactionState type="twso:aborted" transactionID="txEvents" />
                  </twso:operator>
                </twso:sourceState>

                <twso:effect>
                  <twso:primitive type="twso:compensate" target="txFlight" />
                  <twso:primitive type="twso:compensate" target="txTransfer" />
                  <twso:primitive type="twso:compensate" target="txHotel" />
                  <twso:primitive type="twso:compensate" target="txCatering" />
                  <twso:primitive type="twso:compensate" target="txSights" />
                  <twso:primitive type="twso:compensate" target="txEvents" />
                </twso:effect>
              </twso:dependency>

            </ExtendedAttribute>
          </ExtendedAttributes>

        </Activity>

        <Activity Id="synchronize">
          <Route />
          <TransitionRestrictions>
            <TransitionRestriction>
              <Join Type="AND" />
            </TransitionRestriction>
          </TransitionRestrictions>
        </Activity>

        <Activity Id="book_flights">
          <Implementation>
            <Tool Id="starAllianceBooking" Type="APPLICATION" />
          </Implementation>
          <Performer>OrchestrationEngine</Performer>
          <TransitionRestrictions>
            <TransitionRestriction>
              <Split Type="XOR">
                <TransitionRefs>
                  <TransitionRef Id="viennaCityTour1_tra3" />
                  <TransitionRef Id="viennaCityTour1_tra6" />
                </TransitionRefs>
              </Split>
            </TransitionRestriction>
          </TransitionRestrictions>
        </Activity>

        <Activity Id="book_airport_transfer">
          <Implementation>
            <Tool Id="viennaAirport" Type="APPLICATION" />
          </Implementation>
          <Performer>OrchestrationEngine</Performer>
          <TransitionRestrictions>
```

```
        <TransitionRestriction>
          <Split Type="XOR">
            <TransitionRefs>
              <TransitionRef Id="viennaCityTour1_tra12" />
              <TransitionRef Id="viennaCityTour1_tra9" />
            </TransitionRefs>
          </Split>
        </TransitionRestriction>
      </TransitionRestrictions>
    </Activity>

    <Activity Id="book_event_tickets">
      <Implementation>
        <Tool Id="events" Type="APPLICATION" />
      </Implementation>
      <Performer>OrchestrationEngine</Performer>
      <TransitionRestrictions>
        <TransitionRestriction>
          <Split Type="XOR">
            <TransitionRefs>
              <TransitionRef Id="viennaCityTour1_tra24" />
              <TransitionRef Id="viennaCityTour1_tra36" />
            </TransitionRefs>
          </Split>
        </TransitionRestriction>
      </TransitionRestrictions>
    </Activity>

    <Activity Id="book_catering">
      <Implementation>
        <Tool Id="lokaltipp" Type="APPLICATION" />
      </Implementation>
      <Performer>OrchestrationEngine</Performer>
      <TransitionRestrictions>
        <TransitionRestriction>
          <Split Type="XOR">
            <TransitionRefs>
              <TransitionRef Id="viennaCityTour1_tra34" />
              <TransitionRef Id="viennaCityTour1_tra21" />
            </TransitionRefs>
          </Split>
        </TransitionRestriction>
      </TransitionRestrictions>
    </Activity>

    <Activity Id="book_hotel">
      <Implementation>
        <Tool Id="tiscover" Type="APPLICATION" />
      </Implementation>
      <Performer>OrchestrationEngine</Performer>
      <TransitionRestrictions>
        <TransitionRestriction>
          <Split Type="XOR">
            <TransitionRefs>
              <TransitionRef Id="viennaCityTour1_tra33" />
              <TransitionRef Id="viennaCityTour1_tra20" />
            </TransitionRefs>
          </Split>
        </TransitionRestriction>
      </TransitionRestrictions>
    </Activity>

    <Activity Id="book_sight_tickets">
      <Implementation>
        <Tool Id="vienna_service" Type="APPLICATION" />
      </Implementation>
      <Performer>OrchestrationEngine</Performer>
      <TransitionRestrictions>
        <TransitionRestriction>
          <Split Type="XOR">
            <TransitionRefs>
              <TransitionRef Id="viennaCityTour1_tra35" />
              <TransitionRef Id="viennaCityTour1_tra22" />
            </TransitionRefs>
          </Split>
        </TransitionRestriction>
      </TransitionRestrictions>
    </Activity>

    <Activity Id="commit_txFlight">
      <Implementation>
        <No />
      </Implementation>
      <Performer>OrchestrationEngine</Performer>
      <ExtendedAttributes>
        <ExtendedAttribute Name="twsol">
          <twso:primitive type="twso:commit" target="txFlight" />
        </ExtendedAttribute>
      </ExtendedAttributes>
    </Activity>

    <Activity Id="commit_txAirportTransfer">
      <Implementation>
        <No />
      </Implementation>
      <Performer>OrchestrationEngine</Performer>
      <ExtendedAttributes>
        <ExtendedAttribute Name="twsol">
          <twso:primitive type="twso:commit" target="txTransfer" />
        </ExtendedAttribute>
      </ExtendedAttributes>
    </Activity>

    <Activity Id="commit_txHotel">
      <Implementation>
        <No />
      </Implementation>
      <Performer>OrchestrationEngine</Performer>
      <ExtendedAttributes>
        <ExtendedAttribute Name="twsol">
          <twso:primitive type="twso:commit" target="txHotel" />
        </ExtendedAttribute>
      </ExtendedAttributes>
    </Activity>

    <Activity Id="commit_txCatering">
      <Implementation>
        <No />
      </Implementation>
      <Performer>OrchestrationEngine</Performer>
      <ExtendedAttributes>
        <ExtendedAttribute Name="twsol">
          <twso:primitive type="twso:commit" target="txCatering" />
        </ExtendedAttribute>
      </ExtendedAttributes>
    </Activity>

    <Activity Id="commit_txEvents">
      <Implementation>
        <No />
      </Implementation>
      <Performer>OrchestrationEngine</Performer>
      <ExtendedAttributes>
        <ExtendedAttribute Name="twsol">
          <twso:primitive type="twso:commit" target="txEvents" />
        </ExtendedAttribute>
      </ExtendedAttributes>
    </Activity>

    <Activity Id="commit_txSights">
      <Implementation>
        <No />
      </Implementation>
      <Performer>OrchestrationEngine</Performer>
      <ExtendedAttributes>
        <ExtendedAttribute Name="twsol">
          <twso:primitive type="twso:commit" target="txSights" />
        </ExtendedAttribute>
      </ExtendedAttributes>
    </Activity>

    <Activity Id="begin_txFlight">
      <Implementation>
        <No />
      </Implementation>
      <Performer>OrchestrationEngine</Performer>
      <ExtendedAttributes>
        <ExtendedAttribute Name="twsol">
          <twso:primitive type="twso:begin" target="txFlight" />
        </ExtendedAttribute>
      </ExtendedAttributes>
    </Activity>

    <Activity Id="abort_txFlight">
      <Implementation>
        <No />
      </Implementation>
      <Performer>OrchestrationEngine</Performer>
      <ExtendedAttributes>
        <ExtendedAttribute Name="twsol">
          <twso:primitive type="twso:abort" target="txFlight" />
        </ExtendedAttribute>
      </ExtendedAttributes>
    </Activity>

    <Activity Id="abort_txAirportTransfer">
      <Implementation>
        <No />
      </Implementation>
      <Performer>OrchestrationEngine</Performer>
      <ExtendedAttributes>
        <ExtendedAttribute Name="twsol">
          <twso:primitive type="twso:abort" target="txTransfer" />
        </ExtendedAttribute>
      </ExtendedAttributes>
    </Activity>

    <Activity Id="begin_txAirportTransfer">
      <Implementation>
        <No />
      </Implementation>
      <Performer>OrchestrationEngine</Performer>
      <ExtendedAttributes>
        <ExtendedAttribute Name="twsol">
          <twso:primitive type="twso:begin" target="txTransfer" />
        </ExtendedAttribute>
      </ExtendedAttributes>
    </Activity>

    <Activity Id="begin_txHotel">
      <Implementation>
        <No />
      </Implementation>
      <Performer>OrchestrationEngine</Performer>
      <ExtendedAttributes>
        <ExtendedAttribute Name="twsol">
          <twso:primitive type="twso:begin" target="txHotel" />
        </ExtendedAttribute>
      </ExtendedAttributes>
    </Activity>

    <Activity Id="abort_txHotel">
      <Implementation>
        <No />
      </Implementation>
      <Performer>OrchestrationEngine</Performer>
      <ExtendedAttributes>
        <ExtendedAttribute Name="twsol">
          <twso:primitive type="twso:abort" target="txHotel" />
        </ExtendedAttribute>
      </ExtendedAttributes>
    </Activity>

    <Activity Id="begin_txCatering">
      <Implementation>
        <No />
      </Implementation>
      <Performer>OrchestrationEngine</Performer>
      <ExtendedAttributes>
        <ExtendedAttribute Name="twsol">
          <twso:primitive type="twso:begin" target="txCatering" />
        </ExtendedAttribute>
      </ExtendedAttributes>
    </Activity>

    <Activity Id="abort_txCatering">
      <Implementation>
        <No />
      </Implementation>
      <Performer>OrchestrationEngine</Performer>
      <ExtendedAttributes>
        <ExtendedAttribute Name="twsol">
          <twso:primitive type="twso:abort" target="txCatering" />
        </ExtendedAttribute>
      </ExtendedAttributes>
    </Activity>

    <Activity Id="abort_txEvents">
      <Implementation>
```

```
        <No />
      </Implementation>
      <Performer>OrchestrationEngine</Performer>
      <ExtendedAttributes>
        <ExtendedAttribute Name="twsol">
          <twso:primitive type="twso:abort" target="txEvents" />
        </ExtendedAttribute>
      </ExtendedAttributes>
    </Activity>

    <Activity Id="begin_txEvents">
      <Implementation>
        <No />
      </Implementation>
      <Performer>OrchestrationEngine</Performer>
      <ExtendedAttributes>
        <ExtendedAttribute Name="twsol">
          <twso:primitive type="twso:begin" target="txEvents" />
        </ExtendedAttribute>
      </ExtendedAttributes>
    </Activity>

    <Activity Id="begin_txSights">
      <Implementation>
        <No />
      </Implementation>
      <Performer>OrchestrationEngine</Performer>
      <ExtendedAttributes>
        <ExtendedAttribute Name="twsol">
          <twso:primitive type="twso:begin" target="txSights" />
        </ExtendedAttribute>
      </ExtendedAttributes>
    </Activity>

    <Activity Id="abort_txSights">
      <Implementation>
        <No />
      </Implementation>
      <Performer>OrchestrationEngine</Performer>
      <ExtendedAttributes>
        <ExtendedAttribute Name="twsol">
          <twso:primitive type="twso:abort" target="txSights" />
        </ExtendedAttribute>
      </ExtendedAttributes>
    </Activity>
  </Activities>



  <Transitions>
    <Transition From="installDependencies" Id="viennaCityTour1_tra4" To="begin_txFlight" />
    <Transition From="installDependencies" Id="viennaCityTour1_tra1"
      To="begin_txAirportTransfer" />
    </Transition>
    <Transition From="begin_txFlight" Id="viennaCityTour1_tra2" To="book_flights" />
    <Transition From="book_flights" Id="viennaCityTour1_tra3" To="commit_txFlight" />
    <Transition From="commit_txFlight" Id="viennaCityTour1_tra5" To="synchronize" />
    <Transition From="book_flights" Id="viennaCityTour1_tra6" To="abort_txFlight">
      <Condition Type="EXCEPTION" />
    </Transition>
    <Transition From="abort_txFlight" Id="viennaCityTour1_tra7" To="synchronize" />
    <Transition From="begin_txAirportTransfer" Id="viennaCityTour1_tra8"
      To="book_airport_transfer" />
    <Transition From="book_airport_transfer" Id="viennaCityTour1_tra9"
      To="commit_txAirportTransfer" />
    <Transition From="commit_txAirportTransfer" Id="viennaCityTour1_tra10" To="synchronize" />
    <Transition From="abort_txAirportTransfer" Id="viennaCityTour1_tra11" To="synchronize" />
    <Transition From="book_airport_transfer" Id="viennaCityTour1_tra12"
      To="abort_txAirportTransfer1">
      <Condition Type="EXCEPTION" />
    </Transition>
    <Transition From="installDependencies" Id="viennaCityTour1_tra13" To="begin_txHotel" />
    <Transition From="installDependencies" Id="viennaCityTour1_tra14" To="begin_txCatering" />
    <Transition From="installDependencies" Id="viennaCityTour1_tra15" To="begin_txSights" />
    <Transition From="installDependencies" Id="viennaCityTour1_tra16" To="begin_txEvents" />
    <Transition From="begin_txHotel" Id="viennaCityTour1_tra17" To="book_hotel" />
    <Transition From="begin_txCatering" Id="viennaCityTour1_tra18" To="book_catering" />
    <Transition From="begin_txSights" Id="viennaCityTour1_tra19" To="book_sight_tickets" />
    <Transition From="book_hotel" Id="viennaCityTour1_tra20" To="commit_txHotel" />
    <Transition From="book_catering" Id="viennaCityTour1_tra21" To="commit_txCatering" />
    <Transition From="book_sight_tickets" Id="viennaCityTour1_tra22" To="commit_txSights" />
    <Transition From="begin_txEvents" Id="viennaCityTour1_tra23" To="book_event_tickets" />
    <Transition From="book_event_tickets" Id="viennaCityTour1_tra24" To="commit_txEvents" />
```

```
        <Transition From="commit_txHotel" Id="viennaCityTour1_tra25" To="synchronize" />
        <Transition From="abort_txHotel" Id="viennaCityTour1_tra26" To="synchronize" />
        <Transition From="commit_txCatering" Id="viennaCityTour1_tra27" To="synchronize" />
        <Transition From="abort_txCatering" Id="viennaCityTour1_tra28" To="synchronize" />
        <Transition From="commit_txSights" Id="viennaCityTour1_tra29" To="synchronize" />
        <Transition From="abort_txSights" Id="viennaCityTour1_tra30" To="synchronize" />
        <Transition From="commit_txEvents" Id="viennaCityTour1_tra31" To="synchronize" />
        <Transition From="abort_txEvents" Id="viennaCityTour1_tra32" To="synchronize" />
        <Transition From="book_hotel" Id="viennaCityTour1_tra33" To="abort_txHotel">
          <Condition Type="EXCEPTION" />
        </Transition>
        <Transition From="book_catering" Id="viennaCityTour1_tra34" To="abort_txCatering">
          <Condition Type="EXCEPTION" />
        </Transition>
        <Transition From="book_sight_tickets" Id="viennaCityTour1_tra35" To="abort_txSights">
          <Condition Type="EXCEPTION" />
        </Transition>
        <Transition From="book_event_tickets" Id="viennaCityTour1_tra36" To="abort_txEvents">
          <Condition Type="EXCEPTION" />
        </Transition>
      </Transitions>
    </WorkflowProcess>
  </WorkflowProcesses>
</Package>
```

Listing C.2: Adapted Vienna City Trip in XPDL

```
<?xml version="1.0" encoding="UTF-8"?>
<Package Id="viennaCityTour" Name="viennaCityTour" xmlns="http://www.wfmc.org/2002/XPDL1.0"
  xmlns:twso="http://move.ec3.at/twso/20060101"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.wfmc.org/2002/XPDL1.0
␣␣http://wfmc.org/standards/docs/TC-1025_schema_10_xpdl.xsd">
  <PackageHeader>
    <XPDLVersion>1.0</XPDLVersion>
    <Vendor>Together</Vendor>
    <Created>2006-01-31 17:22:28</Created>
  </PackageHeader>

  <WorkflowProcesses>
    <WorkflowProcess Id="viennaCityTour2" Name="viennaCityTour2">
      <ProcessHeader>
        <Created>2006-01-31 17:23:32</Created>
      </ProcessHeader>
      <Participants>
        <Participant Id="OrchestrationEngine">
          <ParticipantType Type="SYSTEM" />
        </Participant>
      </Participants>

      <Applications>
        <Application Id="starAllianceBooking" Name="starAllianceBooking">
          <ExternalReference location="http://webservices.staraliance.com/flightBooking"
            namespace="http://webservices.staraliance.com/flightBooking" xref="bookFlight" />
        </Application>
        <Application Id="viennaAirport" Name="viennaAirport">
          <ExternalReference location="http://webservices.viennaairport.com/"
            namespace="http://webservices.viennaairport.com" xref="bookCityTransfer" />
        </Application>
        <Application Id="tiscover" Name="tiscover">
          <ExternalReference location="http://ws.tiscover.com/booker"
            namespace="http://ws.tiscover.com/booker" xref="bookAccomodation" />
        </Application>
        <Application Id="lokaltipp" Name="lokaltipp">
          <ExternalReference location="http://www.lokaltipp.at/webservice"
            namespace="http://www.lokaltipp.at" xref="book_coupons" />
        </Application>
        <Application Id="events" Name="events">
          <ExternalReference location="http://soap.events.at/" namespace="http://www.events.at"
            xref="tickets" />
        </Application>
        <Application Id="vienna_service" Name="vienna_service">
          <ExternalReference location="http://www.wien.gv.at/soap"
            namespace="http://www.wien.gv.at/" xref="purchase_sight_tickets" />
        </Application>
      </Applications>
```

```
<Activities>
  <Activity Id="installDependencies" Name="installDependencies">
    <Implementation>
      <No />
    </Implementation>
    <Performer>OrchestrationEngine</Performer>

    <TransitionRestrictions>
      <TransitionRestriction>
        <Split Type="AND">
          <TransitionRefs>
            <TransitionRef Id="viennaCityTour2_tra1" />
            <TransitionRef Id="viennaCityTour2_tra4" />
            <TransitionRef Id="viennaCityTour2_tra13" />
            <TransitionRef Id="viennaCityTour2_tra14" />
            <TransitionRef Id="viennaCityTour2_tra15" />
            <TransitionRef Id="viennaCityTour2_tra16" />
          </TransitionRefs>
        </Split>
      </TransitionRestriction>
    </TransitionRestrictions>

    <ExtendedAttributes>
      <ExtendedAttribute Name="twsol">

        <twso:initiate id="txFlight">
          <twso:activityRef>book_flights</twso:activityRef>
        </twso:initiate>
        <twso:initiate id="txTransfer">
          <twso:activityRef>book_airport_transfer</twso:activityRef>
        </twso:initiate>
        <twso:initiate id="txHotel">
          <twso:activityRef>book_hotel</twso:activityRef>
        </twso:initiate>
        <twso:initiate id="txCatering">
          <twso:activityRef>book_catering</twso:activityRef>
        </twso:initiate>
        <twso:initiate id="txEvents">
          <twso:activityRef>book_event_tickets</twso:activityRef>
        </twso:initiate>
        <twso:initiate id="txSights">
          <twso:activityRef>book_sight_tickets</twso:activityRef>
        </twso:initiate>

        <twso:dependency>
          <twso:sourceState>
            <twso:operator type="or">
              <twso:transactionState type="twso:aborted" transactionID="txFlight" />
              <twso:transactionState type="twso:aborted" transactionID="txTransfer" />
              <twso:transactionState type="twso:aborted" transactionID="txHotel" />
              <twso:transactionState type="twso:aborted" transactionID="txCatering" />
              <twso:transactionState type="twso:aborted" transactionID="txSights" />
              <twso:transactionState type="twso:aborted" transactionID="txEvents" />
            </twso:operator>
          </twso:sourceState>

          <twso:effect>
            <twso:primitive type="twso:compensate" target="txFlight" />
            <twso:primitive type="twso:compensate" target="txTransfer" />
            <twso:primitive type="twso:compensate" target="txHotel" />
            <twso:primitive type="twso:compensate" target="txCatering" />
            <twso:primitive type="twso:compensate" target="txSights" />
            <twso:primitive type="twso:compensate" target="txEvents" />
          </twso:effect>
        </twso:dependency>

      </ExtendedAttribute>
    </ExtendedAttributes>

  </Activity>

  <Activity Id="join">
    <Route />
    <TransitionRestrictions>
      <TransitionRestriction>
        <Join Type="XOR" />
      </TransitionRestriction>
    </TransitionRestrictions>
  </Activity>

  <Activity Id="book_flights">
    <Implementation>
```

```
      <Tool Id="starAllianceBooking" Type="APPLICATION" />
    </Implementation>
    <Performer>OrchestrationEngine</Performer>
    <TransitionRestrictions>
      <TransitionRestriction>
        <Join Type="AND" />
        <Split Type="XOR">
          <TransitionRefs>
            <TransitionRef Id="viennaCityTour2_tra3" />
            <TransitionRef Id="viennaCityTour2_tra6" />
          </TransitionRefs>
        </Split>
      </TransitionRestriction>
    </TransitionRestrictions>
  </Activity>

  <Activity Id="book_airport_transfer">
    <Implementation>
      <Tool Id="viennaAirport" Type="APPLICATION" />
    </Implementation>
    <Performer>OrchestrationEngine</Performer>
    <TransitionRestrictions>
      <TransitionRestriction>
        <Split Type="XOR">
          <TransitionRefs>
            <TransitionRef Id="viennaCityTour2_tra12" />
            <TransitionRef Id="viennaCityTour2_tra9" />
          </TransitionRefs>
        </Split>
      </TransitionRestriction>
    </TransitionRestrictions>
  </Activity>

  <Activity Id="book_event_tickets">
    <Implementation>
      <Tool Id="events" Type="APPLICATION" />
    </Implementation>
    <Performer>OrchestrationEngine</Performer>
    <TransitionRestrictions>
      <TransitionRestriction>
        <Split Type="XOR">
          <TransitionRefs>
            <TransitionRef Id="viennaCityTour2_tra24" />
            <TransitionRef Id="viennaCityTour2_tra36" />
          </TransitionRefs>
        </Split>
      </TransitionRestriction>
    </TransitionRestrictions>
  </Activity>

  <Activity Id="book_catering">
    <Implementation>
      <Tool Id="lokaltipp" Type="APPLICATION" />
    </Implementation>
    <Performer>OrchestrationEngine</Performer>
    <TransitionRestrictions>
      <TransitionRestriction>
        <Split Type="XOR">
          <TransitionRefs>
            <TransitionRef Id="viennaCityTour2_tra34" />
            <TransitionRef Id="viennaCityTour2_tra21" />
          </TransitionRefs>
        </Split>
      </TransitionRestriction>
    </TransitionRestrictions>
  </Activity>

  <Activity Id="book_hotel">
    <Implementation>
      <Tool Id="tiscover" Type="APPLICATION" />
    </Implementation>
    <Performer>OrchestrationEngine</Performer>
    <TransitionRestrictions>
      <TransitionRestriction>
        <Split Type="XOR">
          <TransitionRefs>
            <TransitionRef Id="viennaCityTour2_tra33" />
            <TransitionRef Id="viennaCityTour2_tra20" />
          </TransitionRefs>
        </Split>
      </TransitionRestriction>
    </TransitionRestrictions>
  </Activity>
```

```
    <Activity Id="book_sight_tickets">
      <Implementation>
        <Tool Id="vienna_service" Type="APPLICATION" />
      </Implementation>
      <Performer>OrchestrationEngine</Performer>
      <TransitionRestrictions>
        <TransitionRestriction>
          <Split Type="XOR">
            <TransitionRefs>
              <TransitionRef Id="viennaCityTour2_tra35" />
              <TransitionRef Id="viennaCityTour2_tra22" />
            </TransitionRefs>
          </Split>
        </TransitionRestriction>
      </TransitionRestrictions>
    </Activity>

    <Activity Id="commit_txFlight">
      <Implementation>
        <No />
      </Implementation>
      <Performer>OrchestrationEngine</Performer>
      <ExtendedAttributes>
        <ExtendedAttribute Name="twsol">
          <twso:primitive type="twso:commit" target="txFlight" />
        </ExtendedAttribute>
      </ExtendedAttributes>
    </Activity>

    <Activity Id="commit_txAirportTransfer">
      <Implementation>
        <No />
      </Implementation>
      <Performer>OrchestrationEngine</Performer>
      <ExtendedAttributes>
        <ExtendedAttribute Name="twsol">
          <twso:primitive type="twso:commit" target="txTransfer" />
        </ExtendedAttribute>
      </ExtendedAttributes>
    </Activity>

    <Activity Id="commit_txHotel">
      <Implementation>
        <No />
      </Implementation>
      <Performer>OrchestrationEngine</Performer>
      <ExtendedAttributes>
        <ExtendedAttribute Name="twsol">
          <twso:primitive type="twso:commit" target="txHotel" />
        </ExtendedAttribute>
      </ExtendedAttributes>
    </Activity>

    <Activity Id="commit_txCatering">
      <Implementation>
        <No />
      </Implementation>
      <Performer>OrchestrationEngine</Performer>
      <ExtendedAttributes>
        <ExtendedAttribute Name="twsol">
          <twso:primitive type="twso:commit" target="txCatering" />
        </ExtendedAttribute>
      </ExtendedAttributes>
    </Activity>

    <Activity Id="commit_txEvents">
      <Implementation>
        <No />
      </Implementation>
      <Performer>OrchestrationEngine</Performer>
      <ExtendedAttributes>
        <ExtendedAttribute Name="twsol">
          <twso:primitive type="twso:commit" target="txEvents" />
        </ExtendedAttribute>
      </ExtendedAttributes>
    </Activity>

    <Activity Id="commit_txSights">
      <Implementation>
        <No />
      </Implementation>
      <Performer>OrchestrationEngine</Performer>
      <ExtendedAttributes>
        <ExtendedAttribute Name="twsol">
          <twso:primitive type="twso:commit" target="txSights" />
        </ExtendedAttribute>
      </ExtendedAttributes>
    </Activity>

    <Activity Id="begin_txFlight">
      <Implementation>
        <No />
      </Implementation>
      <Performer>OrchestrationEngine</Performer>
      <ExtendedAttributes>
        <ExtendedAttribute Name="twsol">
          <twso:primitive type="twso:begin" target="txFlight" />
        </ExtendedAttribute>
      </ExtendedAttributes>
    </Activity>

    <Activity Id="abort_txFlight">
      <Implementation>
        <No />
      </Implementation>
      <Performer>OrchestrationEngine</Performer>
      <ExtendedAttributes>
        <ExtendedAttribute Name="twsol">
          <twso:primitive type="twso:abort" target="txFlight" />
        </ExtendedAttribute>
      </ExtendedAttributes>
    </Activity>

    <Activity Id="abort_txAirportTransfer">
      <Implementation>
        <No />
      </Implementation>
      <Performer>OrchestrationEngine</Performer>
      <ExtendedAttributes>
        <ExtendedAttribute Name="twsol">
          <twso:primitive type="twso:abort" target="txTransfer" />
        </ExtendedAttribute>
      </ExtendedAttributes>
    </Activity>

    <Activity Id="begin_txAirportTransfer">
      <Implementation>
        <No />
      </Implementation>
      <Performer>OrchestrationEngine</Performer>
      <ExtendedAttributes>
        <ExtendedAttribute Name="twsol">
          <twso:primitive type="twso:begin" target="txTransfer" />
        </ExtendedAttribute>
      </ExtendedAttributes>
    </Activity>

    <Activity Id="begin_txHotel">
      <Implementation>
        <No />
      </Implementation>
      <Performer>OrchestrationEngine</Performer>
      <ExtendedAttributes>
        <ExtendedAttribute Name="twsol">
          <twso:primitive type="twso:begin" target="txHotel" />
        </ExtendedAttribute>
      </ExtendedAttributes>
    </Activity>

    <Activity Id="abort_txHotel">
      <Implementation>
        <No />
      </Implementation>
      <Performer>OrchestrationEngine</Performer>
      <ExtendedAttributes>
        <ExtendedAttribute Name="twsol">
          <twso:primitive type="twso:abort" target="txHotel" />
        </ExtendedAttribute>
      </ExtendedAttributes>
    </Activity>

    <Activity Id="begin_txCatering">
      <Implementation>
        <No />
      </Implementation>
```

```
        <Performer>OrchestrationEngine</Performer>
        <ExtendedAttributes>
          <ExtendedAttribute Name="twsol">
            <twso:primitive type="twso:begin" target="txCatering" />
          </ExtendedAttribute>
        </ExtendedAttributes>
      </Activity>

      <Activity Id="abort_txCatering">
        <Implementation>
          <No />
        </Implementation>
        <Performer>OrchestrationEngine</Performer>
        <ExtendedAttributes>
          <ExtendedAttribute Name="twsol">
            <twso:primitive type="twso:abort" target="txCatering" />
          </ExtendedAttribute>
        </ExtendedAttributes>
      </Activity>

      <Activity Id="abort_txEvents">
        <Implementation>
          <No />
        </Implementation>
        <Performer>OrchestrationEngine</Performer>
        <ExtendedAttributes>
          <ExtendedAttribute Name="twsol">
            <twso:primitive type="twso:abort" target="txEvents" />
          </ExtendedAttribute>
        </ExtendedAttributes>
      </Activity>

      <Activity Id="begin_txEvents">
        <Implementation>
          <No />
        </Implementation>
        <Performer>OrchestrationEngine</Performer>
        <ExtendedAttributes>
          <ExtendedAttribute Name="twsol">
            <twso:primitive type="twso:begin" target="txEvents" />
          </ExtendedAttribute>
        </ExtendedAttributes>
      </Activity>

      <Activity Id="begin_txSights">
        <Implementation>
          <No />
        </Implementation>
        <Performer>OrchestrationEngine</Performer>
        <ExtendedAttributes>
          <ExtendedAttribute Name="twsol">
            <twso:primitive type="twso:begin" target="txSights" />
          </ExtendedAttribute>
        </ExtendedAttributes>
      </Activity>

      <Activity Id="abort_txSights">
        <Implementation>
          <No />
        </Implementation>
        <Performer>OrchestrationEngine</Performer>
        <ExtendedAttributes>
          <ExtendedAttribute Name="twsol">
            <twso:primitive type="twso:abort" target="txSights" />
          </ExtendedAttribute>
        </ExtendedAttributes>
      </Activity>
    </Activities>



    <Transitions>
      <Transition From="installDependencies" Id="viennaCityTour2_tra4" To="begin_txFlight" />
      <Transition From="installDependencies" Id="viennaCityTour2_tra1"
        To="begin_txAirportTransfer">
      </Transition>
      <Transition From="begin_txFlight" Id="viennaCityTour2_tra2" To="book_flights" />
      <Transition From="book_flights" Id="viennaCityTour2_tra3" To="commit_txFlight" />
      <Transition From="commit_txFlight" Id="viennaCityTour2_tra5" To="join" />
      <Transition From="book_flights" Id="viennaCityTour2_tra6" To="abort_txFlight">
        <Condition Type="EXCEPTION" />
      </Transition>
      <Transition From="abort_txFlight" Id="viennaCityTour2_tra7" To="join" />
```

```
        <Transition From="begin_txAirportTransfer" Id="viennaCityTour2_tra8"
          To="book_airport_transfer" />
        <Transition From="book_airport_transfer" Id="viennaCityTour2_tra9"
          To="commit_txAirportTransfer" />
        <Transition From="commit_txAirportTransfer" Id="viennaCityTour2_tra10" To="book_flights" />
        <Transition From="abort_txAirportTransfer" Id="viennaCityTour2_tra11" To="book_flights" />
        <Transition From="book_airport_transfer" Id="viennaCityTour2_tra12"
          To="abort_txAirportTransfer">
          <Condition Type="EXCEPTION" />
        </Transition>
        <Transition From="installDependencies" Id="viennaCityTour2_tra13" To="begin_txHotel" />
        <Transition From="installDependencies" Id="viennaCityTour2_tra14" To="begin_txCatering" />
        <Transition From="installDependencies" Id="viennaCityTour2_tra15" To="begin_txSights" />
        <Transition From="installDependencies" Id="viennaCityTour2_tra16" To="begin_txEvents" />
        <Transition From="begin_txHotel" Id="viennaCityTour2_tra17" To="book_hotel" />
        <Transition From="begin_txCatering" Id="viennaCityTour2_tra18" To="book_catering" />
        <Transition From="begin_txSights" Id="viennaCityTour2_tra19" To="book_sight_tickets" />
        <Transition From="book_hotel" Id="viennaCityTour2_tra20" To="commit_txHotel" />
        <Transition From="book_catering" Id="viennaCityTour2_tra21" To="commit_txCatering" />
        <Transition From="book_sight_tickets" Id="viennaCityTour2_tra22" To="commit_txSights" />
        <Transition From="begin_txEvents" Id="viennaCityTour2_tra23" To="book_event_tickets" />
        <Transition From="book_event_tickets" Id="viennaCityTour2_tra24" To="commit_txEvents" />
        <Transition From="commit_txHotel" Id="viennaCityTour2_tra25" To="book_flights" />
        <Transition From="abort_txHotel" Id="viennaCityTour2_tra26" To="book_flights" />
        <Transition From="commit_txCatering" Id="viennaCityTour2_tra27" To="book_flights" />
        <Transition From="abort_txCatering" Id="viennaCityTour2_tra28" To="book_flights" />
        <Transition From="commit_txSights" Id="viennaCityTour2_tra29" To="book_flights" />
        <Transition From="abort_txSights" Id="viennaCityTour2_tra30" To="book_flights" />
        <Transition From="commit_txEvents" Id="viennaCityTour2_tra31" To="book_flights" />
        <Transition From="abort_txEvents" Id="viennaCityTour2_tra32" To="book_flights" />
        <Transition From="book_hotel" Id="viennaCityTour2_tra33" To="abort_txHotel">
          <Condition Type="EXCEPTION" />
        </Transition>
        <Transition From="book_catering" Id="viennaCityTour2_tra34" To="abort_txCatering">
          <Condition Type="EXCEPTION" />
        </Transition>
        <Transition From="book_sight_tickets" Id="viennaCityTour2_tra35" To="abort_txSights">
          <Condition Type="EXCEPTION" />
        </Transition>
        <Transition From="book_event_tickets" Id="viennaCityTour2_tra36" To="abort_txEvents">
          <Condition Type="EXCEPTION" />
        </Transition>
      </Transitions>
    </WorkflowProcess>
  </WorkflowProcesses>
</Package>
```

Listing C.3: Crete Rental Car Tour in XPDL

```
<?xml version="1.0" encoding="UTF-8"?>
<Package Id="viennaCityTour" Name="viennaCityTour" xmlns="http://www.wfmc.org/2002/XPDL1.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.wfmc.org/2002/XPDL1.0
␣␣http://wfmc.org/standards/docs/TC-1025_schema_10_xpdl.xsd">
  <PackageHeader>
    <XPDLVersion>1.0</XPDLVersion>
    <Vendor>Together</Vendor>
    <Created>2006-01-31 17:22:28</Created>
  </PackageHeader>
  <WorkflowProcesses>

    <WorkflowProcess Id="creteCarTour" Name="creteCarTour">
      <ProcessHeader>
        <Created>2006-01-31 17:23:32</Created>
      </ProcessHeader>

      <Participants>
        <Participant Id="OrchestrationEngine">
          <ParticipantType Type="SYSTEM" />
        </Participant>
      </Participants>

      <Applications>
        <Application Id="starAllianceBooking" Name="starAllianceBooking">
          <ExternalReference location="http://webservices.staraliance.com/flightBooking"
            namespace="http://webservices.staraliance.com/flightBooking" xref="bookFlight" />
        </Application>
        <Application Id="heraklionRentalCars" Name="heraklionRentalCars">
          <ExternalReference location="http://webservices.rentalcars.heraklion.com/"
            namespace="http://webservices.rentalcars.heraklion.com/" xref="bookCar" />
        </Application>
        <Application Id="creteHotels" Name="creteHotels">
          <ExternalReference location="http://ws.creteHotels.org/accomodationBooking"
            namespace="http://ws.creteHotels.org/" xref="book" />
        </Application>
      </Applications>



      <Activities>

        <Activity Id="initiate_transactions">
          <Implementation>
            <No />
          </Implementation>
          <Performer>OrchestrationEngine</Performer>
          <TransitionRestrictions>
            <TransitionRestriction>
              <Split Type="XOR">
                <TransitionRefs>
                  <TransitionRef Id="creteCarTour_tra38" />
                  <TransitionRef Id="creteCarTour_tra37" />
                </TransitionRefs>
              </Split>
            </TransitionRestriction>
          </TransitionRestrictions>

          <ExtendedAttributes>
            <ExtendedAttribute Name="twsol">

              <twso:initiate id="txFlight">
                <twso:activityRef>book_flights</twso:activityRef>
              </twso:initiate>
              <twso:initiate id="txCar">
                <twso:activityRef>book_airport_transfer</twso:activityRef>
              </twso:initiate>
              <twso:initiate id="txHotelHeraklion">
                <twso:activityRef>book_hotel</twso:activityRef>
              </twso:initiate>
              <twso:initiate id="txHotelChania">
                <twso:activityRef>book_catering</twso:activityRef>
              </twso:initiate>
              <twso:initiate id="txHotelAgiosNikolaos">
                <twso:activityRef>book_event_tickets</twso:activityRef>
              </twso:initiate>

            </ExtendedAttribute>
          </ExtendedAttributes>

        </Activity>

        <Activity Id="flexibleTouristDependencies" Name="flexibleTouristDependencies">
          <Implementation>
            <No />
          </Implementation>
          <Performer>OrchestrationEngine</Performer>
        </Activity>

        <Activity Id="synchronize">
          <Route />
          <TransitionRestrictions>
            <TransitionRestriction>
              <Join Type="AND" />
            </TransitionRestriction>
          </TransitionRestrictions>

          <ExtendedAttributes>
            <ExtendedAttribute Name="twsol">

              <twso:dependency>
                <twso:sourceState>
                  <twso:transactionState type="twso:aborted" transactionID="txFlight" />
                </twso:sourceState>

                <twso:effect>
                  <twso:primitive type="twso:compensate" target="txCar" />
                  <twso:primitive type="twso:compensate" target="txHotelHeraklion" />
                  <twso:primitive type="twso:compensate" target="txHotelChania" />
                  <twso:primitive type="twso:compensate" target="txHotelAgiosNikolaos" />
                </twso:effect>
              </twso:dependency>
              <twso:dependency>
                <twso:sourceState>
                  <twso:operator type="and">
                    <twso:transactionState type="twso:aborted" transactionID="txCar" />
                    <twso:transactionState type="twso:aborted" transactionID="txHotelHeraklion" />
                  </twso:operator>
                </twso:sourceState>

                <twso:effect>
                  <twso:primitive type="twso:compensate" target="txFlight" />
                  <twso:primitive type="twso:compensate" target="txCar" />
                  <twso:primitive type="twso:compensate" target="txHotelHeraklion" />
                  <twso:primitive type="twso:compensate" target="txHotelChania" />
                  <twso:primitive type="twso:compensate" target="txHotelAgiosNikolaos" />
                </twso:effect>
              </twso:dependency>
            </ExtendedAttribute>
          </ExtendedAttributes>
        </Activity>

       <Activity Id="inflexibleTouristDependencies" Name="inflexibleTouristDependencies">
          <Implementation>
            <No />
          </Implementation>
          <Performer>OrchestrationEngine</Performer>

          <ExtendedAttributes>
            <ExtendedAttribute Name="twsol">

              <twso:dependency>
                <twso:sourceState>
                <twso:operator type="or">
                  <twso:transactionState type="twso:aborted" transactionID="txFlight" />
                  <twso:transactionState type="twso:aborted" transactionID="txCar" />
                  <twso:transactionState type="twso:aborted" transactionID="txHotelHeraklion" />
                </twso:operator>
                </twso:sourceState>

                <twso:effect>
                  <twso:primitive type="twso:compensate" target="txFlight" />
                  <twso:primitive type="twso:compensate" target="txCar" />
                  <twso:primitive type="twso:compensate" target="txHotelHeraklion" />
                  <twso:primitive type="twso:compensate" target="txHotelChania" />
                  <twso:primitive type="twso:compensate" target="txHotelAgiosNikolaos" />
                </twso:effect>
              </twso:dependency>

              <twso:dependency>
                <twso:sourceState>
                  <twso:operator type="and">
                    <twso:transactionState type="twso:aborted" transactionID="txHotelChania" />
                    <twso:transactionState type="twso:aborted" transactionID="txHotelAgiosNikolaos" />
                  </twso:operator>
                </twso:sourceState>

                <twso:effect>
                  <twso:primitive type="twso:compensate" target="txFlight" />
                  <twso:primitive type="twso:compensate" target="txCar" />
                  <twso:primitive type="twso:compensate" target="txHotelHeraklion" />
                  <twso:primitive type="twso:compensate" target="txHotelChania" />
                  <twso:primitive type="twso:compensate" target="txHotelAgiosNikolaos" />
                </twso:effect>
              </twso:dependency>
            </ExtendedAttribute>
          </ExtendedAttributes>

        </Activity>

        <Activity Id="join">
          <Route />
          <TransitionRestrictions>
            <TransitionRestriction>
              <Join Type="XOR" />
              <Split Type="AND">
                <TransitionRefs>
                  <TransitionRef Id="creteCarTour_tra13" />
                  <TransitionRef Id="creteCarTour_tra1" />
                  <TransitionRef Id="creteCarTour_tra4" />
                  <TransitionRef Id="creteCarTour_tra14" />
                  <TransitionRef Id="creteCarTour_tra15" />
                </TransitionRefs>
              </Split>
            </TransitionRestriction>
          </TransitionRestrictions>
        </Activity>

        <Activity Id="book_flights">
          <Implementation>
            <Tool Id="starAllianceBooking" Type="APPLICATION" />
          </Implementation>
          <Performer>OrchestrationEngine</Performer>
          <TransitionRestrictions>
            <TransitionRestriction>
              <Split Type="XOR">
                <TransitionRefs>
                  <TransitionRef Id="creteCarTour_tra3" />
                  <TransitionRef Id="creteCarTour_tra6" />
                </TransitionRefs>
              </Split>
            </TransitionRestriction>
          </TransitionRestrictions>
        </Activity>

        <Activity Id="book_rental_car">
          <Implementation>
            <Tool Id="heraklionRentalCars" Type="APPLICATION" />
          </Implementation>
          <Performer>OrchestrationEngine</Performer>
          <TransitionRestrictions>
            <TransitionRestriction>
              <Split Type="XOR">
                <TransitionRefs>
                  <TransitionRef Id="creteCarTour_tra12" />
                  <TransitionRef Id="creteCarTour_tra9" />
                </TransitionRefs>
              </Split>
            </TransitionRestriction>
          </TransitionRestrictions>
        </Activity>

        <Activity Id="book_hotel_heraklion">
          <Implementation>
            <Tool Id="creteHotels" Type="APPLICATION" />
          </Implementation>
          <Performer>OrchestrationEngine</Performer>
          <TransitionRestrictions>
            <TransitionRestriction>
              <Split Type="XOR">
                <TransitionRefs>
                  <TransitionRef Id="creteCarTour_tra33" />
                  <TransitionRef Id="creteCarTour_tra20" />
                </TransitionRefs>
              </Split>
            </TransitionRestriction>
          </TransitionRestrictions>
        </Activity>

        <Activity Id="commit_txFlight">
          <Implementation>
            <No />
          </Implementation>
          <Performer>OrchestrationEngine</Performer>
          <ExtendedAttributes>
            <ExtendedAttribute Name="twsol">
              <twso:primitive type="twso:commit" target="txFlight" />
            </ExtendedAttribute>
          </ExtendedAttributes>
        </Activity>

        <Activity Id="commit_txRentalCar">
          <Implementation>
            <No />
          </Implementation>
          <Performer>OrchestrationEngine</Performer>
          <ExtendedAttributes>
            <ExtendedAttribute Name="twsol">
              <twso:primitive type="twso:commit" target="txRentalCar" />
            </ExtendedAttribute>
          </ExtendedAttributes>
        </Activity>

        <Activity Id="commit_txHotelHeraklion">
          <Implementation>
            <No />
          </Implementation>
          <Performer>OrchestrationEngine</Performer>
          <ExtendedAttributes>
            <ExtendedAttribute Name="twsol">
              <twso:primitive type="twso:commit" target="txHotelHeraklion" />
            </ExtendedAttribute>
          </ExtendedAttributes>
        </Activity>

        <Activity Id="begin_txFlight">
          <Implementation>
            <No />
          </Implementation>
          <Performer>OrchestrationEngine</Performer>
          <ExtendedAttributes>
            <ExtendedAttribute Name="twsol">
              <twso:primitive type="twso:begin" target="txFlight" />
            </ExtendedAttribute>
          </ExtendedAttributes>
        </Activity>

        <Activity Id="abort_txFlight">
          <Implementation>
            <No />
          </Implementation>
          <Performer>OrchestrationEngine</Performer>
          <ExtendedAttributes>
            <ExtendedAttribute Name="twsol">
              <twso:primitive type="twso:abort" target="txFlight" />
            </ExtendedAttribute>
          </ExtendedAttributes>
        </Activity>

        <Activity Id="abort_txRentalCar">
          <Implementation>
            <No />
          </Implementation>
          <Performer>OrchestrationEngine</Performer>
          <ExtendedAttributes>
            <ExtendedAttribute Name="twsol">
              <twso:primitive type="twso:abort" target="txRentalCar" />
            </ExtendedAttribute>
          </ExtendedAttributes>
        </Activity>

        <Activity Id="begin_txRentalCar">
          <Implementation>
            <No />
          </Implementation>
          <Performer>OrchestrationEngine</Performer>
          <ExtendedAttributes>
            <ExtendedAttribute Name="twsol">
              <twso:primitive type="twso:begin" target="txRentalCar" />
            </ExtendedAttribute>
          </ExtendedAttributes>
        </Activity>

        <Activity Id="begin_txHotelHeraklion">
          <Implementation>
            <No />
          </Implementation>
          <Performer>OrchestrationEngine</Performer>
          <ExtendedAttributes>
            <ExtendedAttribute Name="twsol">
              <twso:primitive type="twso:begin" target="txHotelHeraklion" />
            </ExtendedAttribute>
          </ExtendedAttributes>
        </Activity>

        <Activity Id="abort_txHotelHeraklion">
          <Implementation>
            <No />
          </Implementation>
          <Performer>OrchestrationEngine</Performer>
           <ExtendedAttributes>
            <ExtendedAttribute Name="twsol">
              <twso:primitive type="twso:abort" target="txHotelHeraklion" />
            </ExtendedAttribute>
          </ExtendedAttributes>
        </Activity>

        <Activity Id="abort_txHotelChania">
          <Implementation>
            <No />
```

```
          </Implementation>
          <Performer>OrchestrationEngine</Performer>
           <ExtendedAttributes>
            <ExtendedAttribute Name="twsol">
              <twso:primitive type="twso:abort" target="txHotelChania" />
            </ExtendedAttribute>
          </ExtendedAttributes>
        </Activity>

        <Activity Id="commit_txHotelChania">
          <Implementation>
            <No />
          </Implementation>
          <Performer>OrchestrationEngine</Performer>
           <ExtendedAttributes>
            <ExtendedAttribute Name="twsol">
              <twso:primitive type="twso:commit" target="txHotelChania" />
            </ExtendedAttribute>
          </ExtendedAttributes>
        </Activity>

        <Activity Id="book_hotel_chania">
          <Implementation>
            <Tool Id="creteHotels" Type="APPLICATION" />
          </Implementation>
          <Performer>OrchestrationEngine</Performer>
          <TransitionRestrictions>
            <TransitionRestriction>
              <Split Type="XOR">
                <TransitionRefs>
                  <TransitionRef Id="creteCarTour_tra201" />
                  <TransitionRef Id="creteCarTour_tra331" />
                </TransitionRefs>
              </Split>
            </TransitionRestriction>
          </TransitionRestrictions>
        </Activity>

        <Activity Id="begin_txHotelChania">
          <Implementation>
            <No />
          </Implementation>
          <Performer>OrchestrationEngine</Performer>
          <ExtendedAttributes>
            <ExtendedAttribute Name="twsol">
              <twso:primitive type="twso:begin" target="txHotelChania" />
            </ExtendedAttribute>
          </ExtendedAttributes>
        </Activity>

        <Activity Id="abort_txHotelAgiosNikolaos">
          <Implementation>
            <No />
          </Implementation>
          <Performer>OrchestrationEngine</Performer>
           <ExtendedAttributes>
            <ExtendedAttribute Name="twsol">
              <twso:primitive type="twso:abort" target="txHotelAgiosNikolaos" />
            </ExtendedAttribute>
          </ExtendedAttributes>
        </Activity>

        <Activity Id="commit_txHotelAgiosNikolaos">
          <Implementation>
            <No />
          </Implementation>
          <Performer>OrchestrationEngine</Performer>
           <ExtendedAttributes>
            <ExtendedAttribute Name="twsol">
              <twso:primitive type="twso:commit" target="txHotelAgiosNikolaos" />
            </ExtendedAttribute>
          </ExtendedAttributes>
        </Activity>

        <Activity Id="book_hotel_agiosNikolaos">
          <Implementation>
            <Tool Id="creteHotels" Type="APPLICATION" />
          </Implementation>
          <Performer>OrchestrationEngine</Performer>
          <TransitionRestrictions>
            <TransitionRestriction>
              <Split Type="XOR">
                <TransitionRefs>
                  <TransitionRef Id="creteCarTour_tra332" />
                  <TransitionRef Id="creteCarTour_tra202" />
                </TransitionRefs>
              </Split>
            </TransitionRestriction>
          </TransitionRestrictions>
        </Activity>

        <Activity Id="begin_txHotelAgiosNikolaos">
          <Implementation>
            <No />
          </Implementation>
          <Performer>OrchestrationEngine</Performer>
          <ExtendedAttributes>
            <ExtendedAttribute Name="twsol">
              <twso:primitive type="twso:begin" target="txHotelAgiosNikolaos" />
            </ExtendedAttribute>
          </ExtendedAttributes>
        </Activity>
      </Activities>



      <Transitions>
        <Transition From="join" Id="creteCarTour_tra1" To="begin_txRentalCar" />
        <Transition From="begin_txFlight" Id="creteCarTour_tra2" To="book_flights" />
        <Transition From="book_flights" Id="creteCarTour_tra3" To="commit_txFlight" />
        <Transition From="commit_txFlight" Id="creteCarTour_tra5" To="synchronize" />
        <Transition From="book_flights" Id="creteCarTour_tra6" To="abort_txFlight1">
          <Condition Type="EXCEPTION" />
        </Transition>
        <Transition From="abort_txFlight1" Id="creteCarTour_tra7" To="synchronize" />
        <Transition From="begin_txRentalCar" Id="creteCarTour_tra8" To="book_rental_car" />
        <Transition From="book_rental_car" Id="creteCarTour_tra9" To="commit_txRentalCar" />
        <Transition From="commit_txRentalCar" Id="creteCarTour_tra10" To="synchronize" />
        <Transition From="abort_txRentalCar" Id="creteCarTour_tra11" To="synchronize" />
        <Transition From="book_rental_car" Id="creteCarTour_tra12" To="abort_txRentalCar">
          <Condition Type="EXCEPTION" />
        </Transition>
        <Transition From="join" Id="creteCarTour_tra13" To="begin_txHotelHeraklion" />
        <Transition From="begin_txHotelHeraklion" Id="creteCarTour_tra17"
          To="book_hotel_heraklion" />
        <Transition From="book_hotel_heraklion" Id="creteCarTour_tra20"
          To="commit_txHotelHeraklion" />
        <Transition From="commit_txHotelHeraklion" Id="creteCarTour_tra25" To="synchronize" />
        <Transition From="abort_txHotelHeraklion" Id="creteCarTour_tra26" To="synchronize" />
        <Transition From="book_hotel_heraklion" Id="creteCarTour_tra33"
          To="abort_txHotelHeraklion">
          <Condition Type="EXCEPTION" />
        </Transition>
        <Transition From="join" Id="creteCarTour_tra4" To="begin_txFlight" />
        <Transition From="initiate_transactions" Id="creteCarTour_tra37"
          To="inflexibleTouristDependencies">
          <Condition Type="OTHERWISE" />
        </Transition>
        <Transition From="initiate_transactions" Id="creteCarTour_tra38"
          To="flexibleTouristDependencies">
          <Condition Type="CONDITION">flexibleTourist==true</Condition>
        </Transition>
        <Transition From="inflexibleTouristDependencies" Id="creteCarTour_tra39" To="join" />
        <Transition From="flexibleTouristDependencies" Id="creteCarTour_tra40" To="join" />
        <Transition From="book_hotel_chania" Id="creteCarTour_tra201" To="commit_txHotelChania" />
        <Transition From="book_hotel_chania" Id="creteCarTour_tra331" To="abort_txHotelChania">
          <Condition Type="EXCEPTION" />
        </Transition>
        <Transition From="begin_txHotelChania" Id="creteCarTour_tra171" To="book_hotel_chania" />
        <Transition From="book_hotel_agiosNikolaos" Id="creteCarTour_tra202"
          To="commit_txHotelAgiosNikolaos" />
        <Transition From="book_hotel_agiosNikolaos" Id="creteCarTour_tra332"
          To="abort_txHotelAgiosNikolaos">
          <Condition Type="EXCEPTION" />
        </Transition>
        <Transition From="begin_txHotelAgiosNikolaos" Id="creteCarTour_tra172"
          To="book_hotel_agiosNikolaos" />
        <Transition From="join" Id="creteCarTour_tra14" To="begin_txHotelChania" />
        <Transition From="join" Id="creteCarTour_tra15" To="begin_txHotelAgiosNikolaos" />
        <Transition From="commit_txHotelChania" Id="creteCarTour_tra16" To="synchronize" />
        <Transition From="abort_txHotelChania" Id="creteCarTour_tra18" To="synchronize" />
        <Transition From="commit_txHotelAgiosNikolaos" Id="creteCarTour_tra19" To="synchronize" />
        <Transition From="abort_txHotelAgiosNikolaos" Id="creteCarTour_tra21" To="synchronize" />
      </Transitions>
    </WorkflowProcess>
  </WorkflowProcesses>
```

```
</Package>
```

C.2 Java Scenario Orchestration

The Java orchestration starts by generating objects that represent references to Web services and creating transactions for each booking action. Depending on the type of tourist (flexible or not flexible), a suitable object that models a transaction dependency is created. Then the transactions are started using `begin`. In case an exception occurs at this point, the orchestration cannot be executed and is aborted. The orchestration is shown in Listing C.4.

Java threads are used to do concurrent processing of the booking actions. For that reason, a class called `ThreadedServiceCall` as presented in Listing C.5 is introduced. This class includes the logic to book a resource, and because it extends Java's `Thread` class, it is possible to execute this logic concurrently, i.e. multithreaded. The booking logic in a `ThreadedServiceCall` object is as follows. First, a Web service is invoked. In case an exception[1] occurs, the Web service should be aborted. That `abort` can cause two types of exceptions. `abort` may be invalid because it is issued on a transaction that is currently in an inappropriate state for an `abort`. This case is anticipated and considered as correct, as described in detail in Section 9.1. Other exceptions should not occur but cannot be ignored safely. Thus, such exceptions are stored when they occur for later treatment. In case the Web service call succeeds, i.e. no exception occurs, it is committed. Exception handling related to a `commit` is the same as when doing an `abort`, as described above.

For each booking action, an appropriate `ThreadedServiceCall` is created and started to do booking actions concurrently. After that, a loop waits until all booking actions have finished, which means that the orchestration terminates.

Listing C.4: Crete Rental Car Tour in Java

```
public class RentalTourOrchestration {

  private static final int THREAD_POLL_INTERVAL = 100;
```

[1]It should be noted that such an exception may stem from both, business logic reasons (e.g. no more free resources) or technical reasons (e.g. service is not reachable due to network errors).

```
private static final String ABORTED = "aborted";

private static final String COMPENSATE = "COMPENSATE";

private static final String BEGIN = "BEGIN";


public static void main(String[] args) throws Exception {

  boolean isFlexibleTourist = true;

  int iterations = 5;

  doTransactionalCreteCarTourOrchestration(isFlexibleTourist);

}


private static void doTransactionalCreteCarTourOrchestration(
  boolean flexibleTourist) throws WSDLException,
  TransactionMonitorCreationException, TransactionCreationException,
  DependencyCreationException, InterruptedException {

  // get WSDL4J port of used transaction monitor
  Port endpoint = Utilities.getTransactionMonitorPort();

  // create transaction monitor proxy
  TransactionMonitor monitor = new SoapTransactionMonitorFactory()
    .getTransactionMonitor(endpoint);

  // create Web service references for transactions
  WebServiceReference starAllianceBookingRef = Utilities
    .createWSReference(
      "http://ec3-55.ec3.at:8080/twso_scenario/services/starAllianceBooking?wsdl",
      "http://ec3-55.ec3.at:8080/twso_scenario/services/starAllianceBooking",
      "starAllianceBookingSoapBinding", "FlightBookerService", "FlightBooker");

  WebServiceReference heraklionRentalCarsRef = Utilities
    .createWSReference(
      "http://ec3-55.ec3.at:8080/twso_scenario/services/heraklionRentalCars?wsdl",
      "http://ec3-55.ec3.at:8080/twso_scenario/services/heraklionRentalCars",
      "heraklionRentalCarsSoapBinding", "CarBookingService", "CarBooking");

  WebServiceReference creteHotelsRef = Utilities.createWSReference(
    "http://ec3-55.ec3.at:8080/twso_scenario/services/creteHotels?wsdl",
    "http://ec3-55.ec3.at:8080/twso_scenario/services/creteHotels",
    "creteHotelsSoapBinding", "BookService", "Book");

  // initiate transactions
  Transaction t_flight = monitor.createTransaction("t_flight",
    new WebServiceReference[] { starAllianceBookingRef });
  Transaction t_car = monitor.createTransaction("t_car",
    new WebServiceReference[] { heraklionRentalCarsRef });
  Transaction t_hotelHeraklion = monitor.createTransaction(
    "t_hotelHeraklion", new WebServiceReference[] { creteHotelsRef });
  Transaction t_hotelChania = monitor.createTransaction("t_hotelChania",
    new WebServiceReference[] { creteHotelsRef });
  Transaction t_hotelAgiosNiklaos = monitor.createTransaction(
    "t_hotelAgiosNiklaos", new WebServiceReference[] { creteHotelsRef });

  // install transaction dependencies dynamically (depends on type of
  // tourist)
  if (flexibleTourist) {
    Dependency dep = new Dependency();

    DependencySourceState sourceState = new DependencySourceState();
    sourceState.setTransactionState(Utilities.createTransactionState(
      t_flight, ABORTED));

    DependencyEffectPrimitive[] effects = new DependencyEffectPrimitive[] {
      Utilities.createEffect(t_car, COMPENSATE),
      Utilities.createEffect(t_hotelHeraklion, COMPENSATE),
      Utilities.createEffect(t_hotelChania, COMPENSATE),
      Utilities.createEffect(t_hotelAgiosNiklaos, COMPENSATE) };

    dep.setSourceState(sourceState);
    dep.setEffect(effects);

    monitor.setDependency(dep);

  } else {
    Dependency dep = new Dependency();
    DependencySourceState sourceState = new DependencySourceState();
```

```
      Operator booleanAND = new Operator();
      booleanAND.setType("and");

      booleanAND.setTransactionState(new TransactionState[] {
        Utilities.createTransactionState(t_flight, ABORTED),
        Utilities.createTransactionState(t_car, ABORTED) });

      sourceState.setOperator(booleanAND);

      DependencyEffectPrimitive[] effects = new DependencyEffectPrimitive[] {
        Utilities.createEffect(t_flight, COMPENSATE),
        Utilities.createEffect(t_car, COMPENSATE),
        Utilities.createEffect(t_hotelHeraklion, COMPENSATE),
        Utilities.createEffect(t_hotelChania, COMPENSATE),
        Utilities.createEffect(t_hotelAgiosNiklaos, COMPENSATE) };

      dep.setSourceState(sourceState);
      dep.setEffect(effects);

      monitor.setDependency(dep);
    }



    // Try to begin all transaction. When the bgein of a singel transaction
    // fails, exit orchestartion
    try {

      t_flight.doPrimitive(BEGIN);
      t_car.doPrimitive(BEGIN);
      t_hotelHeraklion.doPrimitive(BEGIN);
      t_hotelChania.doPrimitive(BEGIN);
      t_hotelAgiosNiklaos.doPrimitive(BEGIN);

    } catch (Exception beginFailure) {
      beginFailure.printStackTrace();

      return;
    }

    // For concurrent Web service calls, create Thread objects
    // ("ThreadedServiceCall") that
    // do the booking.
    ThreadedServiceCall flightBooker = new ThreadedServiceCall(
      t_flight, starAllianceBookingRef, "bookFlight", new Object[] { "Vienna",
        "Heraklion", "2006-07-01", "2006-07-15", "2pax" });

    ThreadedServiceCall carBooker = new ThreadedServiceCall(
      t_car, heraklionRentalCarsRef, "bookCar", new Object[] { "2006-07-01",
        "2006-07-15", "C" });

    ThreadedServiceCall heraklionHotelBooker = new ThreadedServiceCall(
      t_hotelHeraklion, creteHotelsRef, "hotelRoomBooking", new Object[] {
        "2006-07-01", "2006-07-4", "Heraklion", "2pax" });

    ThreadedServiceCall chaniaHotelBooker = new ThreadedServiceCall(
      t_hotelChania, creteHotelsRef, "hotelRoomBooking", new Object[] {
        "2006-07-04", "2006-07-09", "Chania", "2pax" });

    ThreadedServiceCall agiosNikolaosHotelBooker = new ThreadedServiceCall(
      t_hotelAgiosNiklaos, creteHotelsRef, "hotelRoomBooking", new Object[] {
        "2006-07-09", "2006-07-15", "Agios␣Nikolaos", "2pax" });

    // Start threaded calls
    flightBooker.start();
    carBooker.start();
    heraklionHotelBooker.start();
    chaniaHotelBooker.start();
    agiosNikolaosHotelBooker.start();

    // wait for all calls to finish
    while (flightBooker.isAlive() || carBooker.isAlive()
      || heraklionHotelBooker.isAlive() || chaniaHotelBooker.isAlive()
      || agiosNikolaosHotelBooker.isAlive()) {
      Thread.sleep(THREAD_POLL_INTERVAL);
    }
  }

}
```

Listing C.5: Booking Thread Class Rental Car Tour

```
package at.ec3.move.twso.creteRentalTour;


public class ThreadedServiceCall extends Thread {

        private static final String COMMIT = "COMMIT";
        private static final String ABORT = "ABORT";

        private Transaction transaction;
        private WebServiceReference wsRef;
        private String wsOperation;
        private Object[] wsParams;
        private Vector transactionLogicExceptions = new Vector();

        public ThreadedServiceCall(Transaction transaction,
                        WebServiceReference wsRef, String wsOperation, Object[] wsParams) {
                super();
                setTransaction(transaction);
                setWsRef(wsRef);
                setWsOperation(wsOperation);
                setWsParams(wsParams);
        }

        public void run() {

                /*
                 * business logic, business logic level exceptions (that are much more
                 * likely than transaction logic exceptions) are caught and result in an
                 * abort
                 */
                try {

                        System.out.println("Booking␣" + wsRef.getWsdlURI());
                        Service service = new Service();
                        Call call = (Call) service.createCall();
                        call.setTargetEndpointAddress(wsRef.getBinding().getNamespace());
                        call.setOperation(getWsOperation());
                        call.invoke(wsParams);

                } catch (Exception e) {
                        e.printStackTrace();

                        /*
                         * abort transaction, transaction logic related exceptions (although
                         * they should be extremely unlikely) are caught and stored
                         */
                        try {
                                getTransaction().doPrimitive(ABORT);
                                return;
                        } catch (PrimitiveOnInvalidTransactionStateException invalidStateEception) {
                                // Exception is anticipated and considered ok
                                return;
                        } catch (PrimitiveException abortFailure) {
                                abortFailure.printStackTrace();
                                getTransactionLogicExceptions().add(abortFailure);
                                return;
                        }
                }

                /*
                 * business logic succesful commit transaction, transaction logic
                 * related exceptions (although they should be extremely unlikely) are
                 * caught and stored
                 */
                try {
                        getTransaction().doPrimitive(COMMIT);
                } catch (PrimitiveOnInvalidTransactionStateException invalidStateException) {
                        // Exception is anticipated and considered ok
                }

                catch (PrimitiveException commitFailure) {
                        commitFailure.printStackTrace();
                        getTransactionLogicExceptions().add(commitFailure);
                }

        }

        public Transaction getTransaction() {
                return transaction;
        }

        public void setTransaction(Transaction transaction) {
                this.transaction = transaction;
        }

        public String getWsOperation() {
                return wsOperation;
        }

        public void setWsOperation(String wsOperation) {
                this.wsOperation = wsOperation;
        }

        public Object[] getWsParams() {
                return wsParams;
        }

        public void setWsParams(Object[] wsParams) {
                this.wsParams = wsParams;
        }

        public WebServiceReference getWsRef() {
                return wsRef;
        }

        public void setWsRef(WebServiceReference wsRef) {
                this.wsRef = wsRef;
        }

        public Vector getTransactionLogicExceptions() {
                return transactionLogicExceptions;
        }

        public void setTransactionLogicExceptions(Vector transactionLogicExceptions) {
                this.transactionLogicExceptions = transactionLogicExceptions;
        }

}
```

Bibliography

[1] ActiveBPEL LLC. ActiveBPEL Engine – Open Source BPEL Server. `http://www.activebpel.org/`, 2005. cited on 2006-03-08.

[2] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A Pattern Language*. Oxford University Press, New Jork, USA, 1st edition, 1977.

[3] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services Version 1.1. `http://www.ibm.com/developerworks/library/ws-bpel/`, May 2003. cited on 2005-01-28.

[4] Apache Software Foundation. Apache Kandula. `http://ws.apache.org/kandula/`. cited on 2006-01-19.

[5] Arjuna Technologies. Arjuna Transaction Service for Web Services. `http://www.arjuna.com/products/arjunats/ws.html`. cited on 2006-01-19.

[6] A. Biliris, S. Dar, N. H. Gehani, H. V. Jagadish, and K. Ramamritham. ASSET: A System for Supporting Extended Transactions. In R. T. Snodgrass and M. Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 44–54, Minneapolis, Minnesota, 1994.

[7] T. Bray, D. Hollander, and A. Layman. Namespaces in XML. `http://www.w3.org/TR/REC-xml-names/`, 1999. cited on 2005-01-28.

[8] S. Brown. Simon Brown's Weblog. `http://www.simongbrown.com/blog/2003/04/22/how_do_you_define_business_logic.html`, 2003. cited on 2006-01-17.

[9] D. Bunting, M. Chapman, O. Hurley, M. Little, J. Mischkinsky, E. Newcomer, J. Webber, and K. Swenson. Web Services Composite Application Framework. `http://developers.sun.com/techtopics/webservices/wscaf/primer.pdf`, 2003. cited on 2005-01-28.

[10] L. F. Carbrera, G. Copeland, M. Feingold, T. Freund, J. Johnson, C. Kahler, J. Klein, D. Langworthy, F. Leymann, A. Nadalin, D. Orchard, I. Robinson, J. Shewchuk, T. Storey, and S. Thatte. Web Services Coordination, Web Services Business Activity Framework, Web Services Atomic Transaction. `http://www.ibm.com/developerworks/library/specification/ws-tx/`, 2004. cited on 2005-01-28.

[11] A. Ceponkus, S. Dalal, T. Fletcher, P. Furniss, A. Green, and B. Pope. Business Transaction Protocol. `http://www.oasis-open.org/committees/business-transactions/documents/specification/2002-06-03.BTP_cttee_spec_1.0.pdf`, 2002. cited on 2005-01-28.

[12] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. `http://www.w3.org/TR/wsdl`, 2001. cited on 2005-10-20.

[13] P. K. Chrysanthis and K. Ramamritham. Synthesis of Extended Transaction Models Using ACTA. *ACM Transactions on Database Systems*, 19(3):450–491, 1994.

[14] J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0. `http://www.w3.org/TR/xpath`, 1999. cited on 2005-02-04.

[15] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, October 1997.

[16] J. Dorn. Management and Optimization of Business Processes in Virtual Enterprises. Research note, EC3 - Electronic Commerce Competence Center, 2001.

[17] J. Dorn, P. Hrastnik, and A. Rainer. Web Service Discovery and Composition with Move. In *Proceedings of the 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE '05)*, pages 791–792, Washington DC, USA, March 2005. IEEE Computer Society.

[18] J. Eliot and B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, MIT, Dept. of Electrical Engineering and Computer Science, 1981.

[19] A. K. Elmagarmid. *Database Transaction Models for Advanced Applications.* Morgan Kaufmann Publishers, San Mateo, California, 1992.

[20] A. Fekete, P. Greenfield, D. Kuo, and J. Jang. Transactions in Loosely Coupled Distributed Systems. In *Proceedings of the Fourteenth Australasian database conference on Database technologies*, volume 17, Adelaide, Australia, 2003. Australian Computer Society, Inc.

[21] C. Fraleigh, S. Moon, B. Lyles, C. Cotton, M. Khan, D. Moll, R. Rockell, T. Seely, and C. Diot. Packet–Level Traffic Measurements from the Sprint IP Backbone. *IEEE Network Magazine*, 17(6), November 2003.

[22] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 249–259, New York, USA, 1987. ACM Press.

[23] Georgetown University. Web Service Challenge. `http://www.ws-challenge.org/`, 2005. cited on 2006-03-17.

[24] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, San Francisco, California, 9th edition, 2002.

[25] Hewlett Packard. HP Web Services Transactions. `http://www.hpmiddleware.com/HPISAPI.dll/hpmiddleware/products/webservices_transactions/default.jsp`, 2005. cited on 2006-01-19.

[26] M. Hillebrandt, J. Götze, and P. Müller. Creating Dependable Web Services Using User–Replica. In *Next Generation Web Services Practices*, pages 303–312, Seoul, Korea, August 2005. IEEE.

[27] P. Hrastnik. Execution of Business Processes Based on Web Services. *International Journal of Electronic Business*, 2(5):550–556, 2004.

[28] P. Hrastnik and W. Winiwarter. An Advanced Transaction Meta–Model For Web Services Environments. In *The Sixth International Conference on Information Integration and Web–based Applications & Services (iiWAS2004)*, pages 303–312, Jakarta, Indonesia, September 2004. Austrian Computer Society.

[29] P. Hrastnik and W. Winiwarter. TWSO – Transactional Web Service Orchestrations. In *Next Generation Web Services Practices*, pages 45–50, Seoul, Korea, August 2005. IEEE.

[30] P. Hrastnik and W. Winiwarter. Using Advanced Transaction Meta-Models for Creating Transaction-Aware Web Service Environments. *International Journal of Web Information Systems*, 1(2), 2005.

[31] P. Hrastnik and W. Winiwarter. TWSO – Transactional Web Service Orchestrations. *Journal of Digital Information Management*, 4(1):56–62, 2006.

[32] J. Kannegaard. Foreword to Designing Enterprise Applications with the J2EE Platform. `http://java.sun.com/blueprints/guidelines/designing_enterprise_applications/foreword.html`, 2000. cited on 2005-10-10.

[33] H. Kopetz. *Real-Time Systems.* Kluwer Academic Publishers, Norwell, Massachusetts, USA, 1st edition, 1997.

[34] D. Matthews. Web Services Atomic Transaction for WebSphere Application Server. `http://www.alphaworks.ibm.com/tech/wsat`, 2003. cited on 2006-01-19.

[35] M. Mickos. MySQL Database Now Provides Full Transaction Support. `http://www.mysql.com/news-and-events/press-release/release_2002_11.html`, 2002. cited on 2005-10-10.

[36] N. Mitra. SOAP Part 0: Primer. `http://www.w3.org/TR/soap12-part0/`, 2003. cited on 2005-10-20.

[37] ObjectWeb. ObjectWeb Forge: Project Info - JOTN. `http://forge.objectweb.org/projects/jotm/`. cited on 2006-01-19.

[38] D. Oppenheimer, A. Ganapathi, and D. Patterson. Why Do Internet Services Fail, And What Can Be Done About It? In *In Proc. 4th USENIX Symposium on Internet Technologies and Systems (USITS '03)*, Seattle, WA, March 2003.

[39] K. Papagiannaki, S. Moon, C. Fraleigh, P. Thiran, and C. Diot. Measurement and Analysis of Single–Hop Delay on an IP Backbone Network. *IEEE Journal on Selected Areas in Communications*, 21(6), August 2003.

[40] M. Potts, B. Cox, and B. Pope. Business Transaction Protocol Primer. `http://www.oasis-open.org/committees/business-transactions/documents/primer/`, 2002. cited on 2005-01-28.

[41] M. Prochazka. *Advanced Transactions in Component-Based Software Architectures*. PhD thesis, Charles University, Faculty of Mathematics and Physics, Department of Software Engineering, Malostranske namesti 25, 118 00 Prague 1, Czech Republic, 2002.

[42] J. Roberts and K. Srinivasan. Tentative Hold Protocol Part 1: White Paper. `http://www.w3.org/TR/2001/NOTE-tenthold-1-20011128/`, 2001. cited on 2005-10-12.

[43] C. M. Sperberg-McQueen and H. Thompson. XML Schema. `http://www.w3.org/TR/xml-schema/`, 2006. cited on 2006-03-17.

[44] SUN Microsystems. Java 2 Platform, Enterprise Edition (J2EE). `http://java.sun.com/j2ee/index.jsp`, October 2005. cited on 2005-10-10.

[45] Together Teamsolutions. Open Source Java XPDL Workflow. `http://www.enhydra.org/workflow/shark/index.html`, 2006. cited on 2006-03-08.

[46] G. Weikum and H. J. Schek. Multi–Level Transactions and Open Nested Transactions. In *Data Engineering*, volume 14, pages 60–64, Los Alamitos, California, March 1991. IEEE Computer Society Press.

[47] Wikipedia. MySQL – Criticisms. `http://en.wikipedia.org/wiki/MySQL#Criticisms_of_MySQL`, 2005. cited on 2005-10-10.

[48] Workflow Management Coalition. Workflow Process Definition Interface – XML Process Definition Language. `http://www.wfmc.org/standards/docs/TC-1025_10_xpdl_102502.pdf`, 2002. cited on 2005-01-28.

Curriculum Vitae

Personal Data

Name:	Peter Hrastnik
Date of Birth:	1975-05-18
Place of Birth:	Salzburg, Austria
Gender:	male
Nationality:	Austria
E–Mail:	peter@hrastnik.at
Address:	Aegidigasse 7-11/2/23, A-1060 Vienna, Austria

Education

1981 – 1984:	Elementary school at Volksschule Hallwang bei Salzburg
1984 – 1993:	High school at Christian Dopplergymnasium Salzburg
1994 – 1999:	Master studies (Business Informatics) at University of Vienna and Vienna University of Technology Master thesis: XML/EDI for Machine–To–Machine Communication
2003 – 2006:	PhD studies at Vienna University of Technology

Career Overview

1992 – 1998:	Internships at Bundesländer Insurances Company, Reader's Digest, Salzburg Festivals, Österreichische Lotterien Cooperation, Datenwerk Corporation
1999 – 2001:	tele.ring Telecom Service Corporation (Developer of Dataservices)
since 2001:	EC3 – E-Commerce Competence Center (Software Developer, since 2003 Researcher)