BUILDING RELIABLE AND ROBUST SERVICE-BASED SYSTEMS FOR AUTOMATED BUSINESS PROCESSES

Julian Jang-Jaccard

A thesis submitted in fulfilment of
the requirement for the degree of
Doctor of Philosophy

School of Information Technologies
University of Sydney

March 2007

Abstract

An exciting trend in enterprise computing lies in the integration of applications across an organisation and even between organisations. This allows the provision of services by automated business processes that coordinate business activity among several collaborating organisations. The best successes in this type of integrated distributed system come through use of Web Services and Service-based Architecture, which allow interoperation between applications through open standards based on XML and SOAP. But still, there are unresolved issues when developers seek to build a reliable and robust system.

An important goal for the designers of a loosely coupled distributed system is to maintain consistency for each long running business process in the presence of failures and concurrent activities. Our approach to assist the developers in this domain is to guide the developers with the key principles they must consider, and to provide programming models and protocols, which make it easier to detect and avoid consistency faults in service-based system.

We start by defining a realistic e-procurement scenario to illustrate the common problems faced by the developers which prevent them from building a reliable and robust system. These problems make it hard to maintain the consistency of the data and state during the execution of a business process in the occurrence of failures and interference from concurrent activities. Through the analysis of the common problems, we identify key principles the developers must consider to avoid producing the common problems.

Then based on the key principles, we provide a framework called GAT in the orchestration infrastructure. GAT allows developers to express all the necessary processing to handle deviations including those due to failures and concurrent activities. We discuss the GAT framework in detail with its structure and key features. Using an example taken from part of the e-procurement case study, we illustrate how developers can use the framework to design their business requirements. We also discuss how key features of the new framework help the developers to avoid producing consistency faults. We illustrate how systems based on our framework can be built using today's proven technology.

Finally, we provide a unified isolation mechanism called Promises that is not only applicable to our GAT framework, but also to any applications that run in the service-based world. We discuss the concept, how it works, and how it defines a protocol. We also provide a list of potential implementation techniques. Using some of the implementation techniques we mention, we provide a proof-of-concept prototype system.

Acknowledgements

My deepest thank goes to my supervisor Alan Fekete for his guidance and patience during the course of my doctoral studies at the University of Sydney. His broad knowledge and his clear thinking have always inspired me. Not only for the research aspect, his warm personality and friendly smile have always assured me how lucky I have been to work with someone as capable as he is. Equally, I owe the deepest gratitude to Paul Greenfield. It was Paul who hired me in CSIRO and trained me as a researcher even when I had no idea what it meant. During the last 5 years, with much happening in CSIRO, he has continuously embraced my interests and showed me the right direction: how to be a good researcher and what it means to do quality research. Alan and Paul, I am sure I'll miss very much our regular Friday meetings at 8am.

I would like to thank my colleague Surya Nepal. At many difficult moments during my research career, he has been there for me, listening to my problems and complaints. I have been able to overcome many obstacles through the constructive discussions and friendly advice from him. I also would like to thank Dean Kuo, as without him, this research project would have not started.

I would like thank to my project manager at my work John Zic for providing me an opportunity to undertake my study in conjunction with my work responsibility. I also would like thank to the supervisors and fellow PhD students of the middleware group for their constructive feedback on presentations on my thesis topic. Special thanks go to David Levy, Uwe Roehm, Shiping Chen, and Anna Liu.

This thesis is dedicated to my mom Ok-am Han, my late dad Bong-ok Jang, and my husband Frederic Jaccard. Without the frequent assurance and encouragement from my mom, I would have not made this far with my studies. My husband has been simply superb in showing me continuous love and support. My long journey would have not been the same without you my love. And I am sure that my dad watching over far from there must be feeling proud and happy for his youngest daughter to be where I am today.

Table of Contents

ACKNOWLEDGEMENTS III

INTRODUCTION 1

1.1 Problem Statement 2
1.2 Our Approach 3
1.3 Contributions: Understanding the Nature of Service-based System 4
1.4 Contributions: GAT – New Event-Driven Programming Model for Defining Business Processes 5
1.5 Contributions: Design Principles in Building a Business Process System based on GAT Model 5
1.6 Contributions: Promises – New Unified Isolation Mechanisms for Service-based System 6
1.7 Contributions: Design Principles in Supporting Promises 8
1.8 Thesis Structure 9

RELATED WORK 10

2.1 Traditional Transaction Support 10
  2.1.1 ACID properties 10
  2.1.2 Locking Mechanism 11
  2.1.3 Two-Phase Commit (2PC) 13
  2.1.4 Advanced Mechanisms for Standard ACID 15
  2.1.5 Extended Transaction Models 17
2.2 Distributed Computing Platforms 19
  2.2.1 Conventional Middleware 20
  2.2.2 Workflow Management Systems (WfMS) 22
  2.2.3 Business to Business Integration (B2Bi) and Service-oriented Architecture (SOA) 26
2.3 Summary 34

UNDERSTANDING THE NATURE OF SERVICE-BASED SYSTEMS 35

3.1 Motivating Scenario 35
  3.1.1 E-procurement When Ordering Goods 35
  3.1.2 Merchant System 36
3.2 Issues for Service-based Systems 38
  3.2.1 Time Related Issues 39
  3.2.2 No Termination 40
  3.2.3 Unprocessed Messages 40

3.2.4 Out of Order Processing of Messages 41
3.2.5 Lack of Isolation 42
3.2.6 Cancellations.. 43
3.3 Introduction to Deviations.. 44
3.3.1 Recoverable Deviations... 44
3.3.2 Unrecoverable Deviations 45
3.3.3 State-related Deviations 46
3.4 States and State Mismatch.. 47
3.4.1 States 47
3.4.2 Classification of Deviations... 49
3.5 Desired Features in Handling Deviations 54
3.5.1 Cancellations.. 54
3.5.2 Continuing to Make Forward Progress 57
3.6 Critiques of Standard Mechanisms and Supports from Current Technologies 59
3.7 Summary 60

GAT – NEW EVENT-DRIVEN PROGRAMMING MODEL FOR DEFINING BUSINESS PROCESSES 62

4.1 Payment Process 63
4.1.1 Send Invoice 63
4.1.2 Receive Payment.. 64
4.1.3 Send Receipt 66
4.1.4 Cancellations.. 66
4.2 GAT Programming Model 68
4.2.1 Structure 68
4.2.2 Key Features 71
4.3 Payment Process in GAT Model 72
4.3.1 Activity Group: sendInvoice.. 72
4.3.2 Activity Group: receivePayment.. 74
4.3.3 Activity Group: overduePayment 76
4.3.4 Activity Group: sendReceipt 78
4.3.5 Activity Group: cancellations... 78
4.4 Experiment with GAT.. 80
4.5 Evaluation Compared to Other Models 81
4.6 Summary 82

DESIGN PRINCIPLES IN BUILDING A BUSINESS PROCESS SYSTEM BASED ON GAT MODEL.................... 84

5.1 Case study.. 84
5.2 GAT Design Consideration 86
5.2.1 Control Flow of Business Activities 86
5.2.2 Atomicity/Isolation Issues 87
5.2.3 Management and Distribution of Events...................................... 87

5.3 Architecture of GAT Prototype System 88
5.3.1 User Interface 89
5.3.2 Event Handler 89
5.3.3 GAT Processor 90
5.3.4 Remote Communication Handler 90
5.3.5 Data Storage 91
5.3.6 Running Business Systems 92
5.3.7 Performance 93
5.4 Implementation of GAT model 96
5.4.1 Defining Activity Group and Execution of Activities 97
5.4.2 Supporting the GAT Event Concept 100
5.4.3 Inter-process Communication 103
5.5 Design of a General GAT Engine 104
5.5.1 GAT Specifications 106
5.5.2 Analyser 107
5.5.3 Mapper 107
5.5.4 CodeDOM 107
5.5.5 Generating C# 107
5.5.6 Compiling 107
5.6 Evaluation Compared to Other Implementation Alternatives 108
5.7 Summary 110

PROMISES – NEW UNIFIED ISOLATION MECHANISMS FOR SERVICE-BASED SYSTEMS 112

6.1 Promises 113
6.2 Resources and Predicates 116
6.2.1 Anonymous View 117
6.2.2 Named View 118
6.2.3 View via Properties 119
6.3 Atomicity and Promises 121
6.4 Implementation Techniques 122
6.5 Promise Protocol 124
6.6 Promises and Isolation 126
6.7 Other Similar Isolation Mechanisms 127
6.8 Conclusion 129

DESIGN PRINCIPLES IN SUPPORTING PROMISES 130

7.1 Design Issues and Constraints of Promises 130
7.1.1 Compatibility 131
7.1.2 Representing Promises 131
7.1.3 Promises and Schemas 131
7.1.4 Isolation and Concurrency 132
7.1.5 Dynamic Promise List 132
7.2 Structure 133
7.2.1 Messages 134

7.2.2 Components 135
7.2.3 Promise Consistency Checking 137
7.2.4 Promise Operations 138
7.3 Reflecting on our design 139
7.3.1 Compatibility 139
7.3.2 Representing Promises 139
7.3.3 Promises and Schemas 140
7.3.4 Isolation and Concurrency 140
7.3.5 Dynamic Promise List 141
7.4 Implementation 142
7.4.1 Overview of Promise Consistency Checking Interface 142
7.4.2 Implementation of Promise Operations 146
7.5 Other Alternatives 155
7.6 Summary 156
CONCLUSIONS 158

List of Figures

Figure 1 Conventional Locking Example (source from [25])........................12
Figure 2 Two-Phase Commit (2PC) Protocol..14
Figure 3 Escrow Locking Example (source from [25])..................................16
Figure 4 Long Transaction in Saga 18
Figure 5 Conventional Workflow Example...22
Figure 6 ECA Example...23
Figure 7 E-procurement Scenario..36
Figure 8 Merchant Workflow..37
Figure 9 Payment Process within the Merchant Process...............................63
Figure 10 Receive Payment 65
Figure 11 GAT Process Structure 69
Figure 12 GAT Activity Structure..70
Figure 13 Activity Group: receivePayment in GAT 86
Figure 14 Architecture of GAT Prototype System..88
Figure 15 Snapshot of the Running Prototype System..................................93
Figure 16 Performance at Each Business Activity in Milliseconds (ms)........94
Figure 17 Snapshot of the Performance Monitor...96
Figure 18 GAT Engine Concepts..105
Figure 19 GAT Engine Major Stages...106
Figure 20 Example of GAT Syntax..106
Figure 21 Outline of Ordering Process Code...127
Figure 22 Structure of Promise System...134
Figure 23 Promise Manager Flow Chart 136
Figure 24 Message Sequence on Making New Promises.............................147
Figure 25 Message Sequences on Executing Actions..................................151
Figure 26 Message Sequence on Updating Promises..................................154

Publications on which this Thesis is Based

- **Jang, J.**, Fekete, A., Greenfield, P. Delivering Promises for Web Services Applications. Technical Report of University of Sydney School of Information Technologies, TR-605 December 2006.

- Greenfield, P., Fekete, A., **Jang, J.**, Kuo, D., Nepal, S. Isolation Support for Service-based Applications. *In Proceedings of the 3rd biennial Conference on Innovative Data Systems Research (CIDR)*, pp 314-323, Asilomar, USA, January 2007.

- **Jang, J.**, Fekete, A., Nepal, S., Greenfield, P. An Event-Driven Workflow Engine for Service-based Business System. *In Proceedings of the 10th IEEE International Conference on Enterprise Computing (EDOC)*, pp 233-242, Hong Kong, China, October 2006.

- Nepal, S., Fekete, A., Greenfield, P., **Jang, J.**, Kuo, D., Shi, T. A Service oriented Workflow Language for Robust Interacting Applications. *In Proceedings of the 13th Cooperative Information Systems (CoopIS)*, pp 40-58, Cyprus, November 2005.

- Kuo, D., Fekete, A., Greenfield, P., **Jang, J.** Just What Could Possibly Go Wrong In B2B Integration? *In Proceedings of the 27th Annual International Computer Software and Applications Conference (COMSAC)*, pp 544-549, Dallas, USA, November 2003.

- Greenfield, P., Fekete, A., **Jang, J.**, Kuo, D. What are the consistency requirements for B2B systems? *High Performance Transactions Systems (HPTS) Workshop*, Asilomar, California, USA, October 2003.

- Greenfield, P., Fekete, A., **Jang, J.**, Kuo, D. Compensation is Not Enough. *In Proceedings of the 7th IEEE International Enterprise Distributed Object Computing Conference (EDOC)*, pp 232-239, Brisbane, Australia, September 2003.

- **Jang, J.**, Fekete, A., Greenfield, P., Kuo, D. Expressiveness of Workflow Description Languages. *In Proceedings of the 1st International Conference on Web Services (ICWS)*, pp 104-110, Las Vegas, USA, June 2003.

- Fekete, A., Greenfield, P., Kuo, D., **Jang, J.** Transactions in Loosely Coupled Distributed Systems. *In Proceedings of the 14th Australasian Database Conference (ADC)*, pp 7-12, Adelaide, Australia, February 2003.

- Kuo, D., Fekete, A., Greenfield, P., **Jang, J**. Towards a Framework for Capturing Transactional Requirements of Real Workflows. *The 2nd International Workshop on Cooperative Internet Computing (CIC)*, Hong Kong, August 2002.

Chapter 1

Introduction

Businesses continuously seek methods to automate tasks to reduce costs. With the advent of distributed computing, it is possible to implement Enterprise Application Integration (EAI) and Business-to-Business integration (B2Bi) solutions to automate business processes. The key to success is interoperability between loosely coupled components implemented and hosted on independent platforms. These may be within the same organisation or indeed they can be owned by different organisations. Components are written independently, and they can be combined in multiple ways. In many cases the component encapsulates a legacy IT system, such as inventory management or a financial package. The essence of such systems is the ability of collaborating organisations to allow controlled external access to their internal IT systems which raises questions familiar to database researchers, such as semantic data conversion, data integrity, and many more.

There are a number of distributed computing platforms available for implementing interoperation between application components. Some, like J2EE or .NET, are proving popular and powerful within a tightly controlled organisation. However, they lack the cross-platform interoperability needed for industry-wide collaboration. Other technologies such as CORBA have proved too heavy-weight and complex for many users.

Web Services is a maturing distributed computing platform that is currently attracting a lot of attention [25]. This approach to interoperability relies on both XML messaging and the internet. It is based on a set of standards managed by the vendor-neutral W3C with support from all major IT vendors. These standards include SOAP (Simple Object Access Protocol), WSDL (Web Services Description Language) and UDDI (Universal Description, Discovery Integration).

The Web Services standards mentioned above allow outsiders to invoke a remote application, and provide arguments and receive results in a format that can be understood on both sides. However, for effective EAI and B2Bi, we need a sequence of operations flowing in both directions providing interaction among long running business process across multiple organisations. In such situations, interacting components typically maintain per-collaboration state throughout a long collaboration. Designing a general loosely coupled distributed system thus requires description of long running business processes which exchange messages each of which is done using SOAP.

It’s important to distinguish between external and internal descriptions of a service. A description of an interface or abstract business process is the way external collaborators interact with a service. Internally, the orchestration or executable business process describes when each activity is invoked, how control flows, and how to deal with exceptions and other situations in sufficient detail. Our thesis deals with the latter: the issues involved in the orchestration level.

There are now a number of proposed standards for EAI and B2Bi solutions for long running business transactions. For example, BPEL [7] is a specification for implementing a business process from collaborating activities. Through such tool support and standards, it is fairly easy to design and construct this kind of integrated system. Current technology does not, however, make it easy to design reliable and robust applications: ones that can deal with events that cause deviations from normal processing paths, such as failures and concurrent activities, while still maintaining overall, cross-organisational consistency.

1.1 Problem Statement

An important goal for the designers of a loosely coupled distributed system is to maintain consistency for each long running business process in the presence of failures and concurrent activities. The objectives are thus similar to transaction processing for database management systems.

The environment of a loosely coupled distributed system however is very different from traditional database management systems where mechanisms used for ACID properties worked well. The loosely coupled system is constructed from pieces that need to remain autonomous, because they were written, and are run, independently. In many cases, they belong to different organisations which are competitors as well as collaborators; the organisations’ goals are not the same, and each can’t extend trust to the other. The pieces use many resources and may include human intervention, so each lasts a long time. For these reasons, it is unacceptable for one business process to hold locks in another business process that is located beyond a trust boundary. The lack of locks means that processes can’t be completely isolated from one another; also this means that one can’t follow traditional rollback, based on the restoration of before images kept in a log.

To overcome the limitations of adapting the standard mechanism from ACID, current technology tools and business process models have borrowed the exception-handler concept from programming languages and an advanced distributed transaction model based on compensation [36]. Though there is high acceptance of the compensator model as a way to provide required failure atomicity, it has limitations as a primary exception handling mechanism. This is discussed in detail in Chapter 2 Related Work.

Rather than attempting to provide the equivalent of traditional or advanced distributed transactions for the loosely-coupled Web Services world, our approach has been focused on the more modest goal of supporting the development of tools, programming models and protocols, which make it easier to detect and avoid consistency faults in service-based system.

Key questions we will address concern the nature of situations that lead to inconsistency in such systems, how designers can represent a business process including all processing needed to avoid inconsistency, how designers can express conditions that must be guaranteed to avoid interference from concurrent activities, and how these designs can be implemented effectively using today's proven technologies.

1.2 Our Approach

We will provide the answers to the questions raised in the previous section through the analysis of the nature of the service-based world and the understanding of the issues involved in the domain.

We first present a realistic e-procurement scenario to discuss the common problems faced by the developers of the service-based system which prevent them from building a reliable and robust system. These problems make it hard to maintain the consistency of the data and state during the execution of a business process in the presence of failures and interference from concurrent activities. Through the analysis of the common problems, we identify key principles that will drive our proposals.

In the second part of the thesis, we provide a framework called GAT in the orchestration infrastructure. GAT allows developers to express all the necessary processing so that they can produce reliable and robust systems despite the presence of failures and concurrent activities. We will discuss the GAT framework in detail with its structure and key features. Using the example taken from part of the e-procurement case study, we illustrate how developers can use the framework to design their business requirements. We also discuss how key features of the new framework help the developers to avoid producing consistency faults. We will also illustrate how our framework can be built using today's proven technology.

In the third part of the thesis, we provide a unified isolation mechanism called Promises that is not only applicable to our GAT framework, but also to any applications that run in the service-based world. We discuss the concept, how it works, and how it defines a protocol. We also provide a list of potential implementation techniques. Using some of these implementation techniques, we describe a proof-of-concept prototype system.

1.3 Contributions: Understanding the Nature of Service-based System

We identify some important issues which can prevent the developers of a service-based system from building reliable and robust systems for automated business processes. The issues of concern include: time related issues derived from the asynchronous nature of a service-based system, failure to terminate resulting from lack of coordination and global knowledge among autonomous systems, unprocessed messages caused by complex and sophisticated interactions among loosely-coupled components, messages which arrive and are processed out of order, lack of isolation due to activities which run for long duration across trust boundaries, and the increased chance of cancellation.

These issues are often causes of state mismatches which then produce various deviations from the expected execution path. If these deviations are not appropriately handled, the system will produce inconsistent outcomes. We give our insights into the states, starting from defining the types of state involved in a service-based system, such as real world state, abstract state and the business process state. We present different types of state mismatches in terms of these three types of state. This can provide better insights into the relationship between the key issues of any service-based system and the types of deviations.

We present what we consider to be required behaviours to be able to handle various common deviations resulting from state mismatch. In light of our list of required deviation handling behaviours, we evaluate the existing standard deviation handling mechanisms to see how well they can be used as support. We present the results of the evaluation and summaries the key requirements for describing business processes if we seek to reduce consistency problems.

1.4 Contributions: GAT – New Event-Driven Programming Model for Defining Business Processes

We propose a new programming model and notation for expressing business processes which can help designers of business system to avoid many common sources of errors. This new model is called “GAT” standing for Guard-Acton-Trigger following the name from the major elements of the model.

One of the most innovative features of GAT is that it processes normal business activities and various unusual situations (including deviations) in a uniform manner. This allows the developers to have simpler expressive ways to manage even the most complex and sophisticated deviations. Our GAT model treats deviations (such as failures and cancellations) as events, just like message arrival in normal business cases, rather than under a separate and inflexible fault-handling regime such as rollback or compensators. Once deviations are dealt with, forward progress can resume through normal business activities which allows the system to always go forward rather than having to be aborted.

The degree of knowledge of the overall system can dramatically change the behaviour of the system in dealing with deviations and other situations. Typically, existing standards and products only allow the access to Abstract States, a computer-based representation of the domain typically stored in databases (i.e. purchase order or payment), in the form of messages or variables inside business action. Business Process State, which has variables reflecting what business actions have occurred (i.e. purchase order received, payment sent), are hidden away and only implicitly used by the system to handle the failures automatically. Different to the existing approach, the GAT model makes available both Abstract States and Business Process States to the developers. The developers can explore and access both states to gain as much knowledge as possible regarding the state of the current system. This exposure of a wide range of states allows the developers to control business activities and to devise better mechanisms to handle deviations.

We have represented the whole e-procurement processing in the GAT notation, thus showing its usefulness for a business analyst.

1.5 Contributions: Design Principles in Building a Business Process System based on GAT Model

We define and solve the critical challenges that have to be addressed when designing a business process system from a description in GAT model. These include implementing control flow based on the evaluation of guards, the management and distribution of events, and enforcing atomicity constraints across the evaluation of guards and the execution of the corresponding activity. We have built a prototype system following this approach which uses available technologies such as C# and the .NET framework to produce a set of business process executables from the e-procurement scenario defined in GAT.

A key issue for a system following GAT approach is to decide how to manage the control flow, in other word, how to pick the appropriate action to perform in response to an event. We propose exploiting an event-driven publish/subscribe model to pick appropriate actions and to raise further events to make the business process go forward until it completes. We illustrate how this is done using .NET events and C# language.

In the GAT model, it is essential that the choice of which action to perform from an activity group (by evaluation of guards), the execution of the chosen action, and the evaluation of its trigger conditions and raising any further events, must all form an isolated unit. Our GAT prototype implementation uses the transaction mechanisms provided in .NET 2.0 to solve this problem. Each activity group, including the evaluation of its guards, the execution of the chosen action, and the evaluation of its trigger condition and raising further events, is constructed as a single transaction. The isolation provided by transactions guarantees that the state used by a running activity group cannot be altered by any concurrently executing business processes.

In our GAT model, events are used as communication carrier that delivers messages within a business process or across multiple business processes. In our prototype, we map these to .NET events, but there are some complexities. GAT has three different types of events: Internal Events are used to control flow among Activities within the same business process, External Events control the communication between interacting peer business processes, and Deferred Events are used when a process needs to trigger corrective actions if anticipated events have not happened by some deadline. We provide examples with coding details of how the different types of events can be implemented in our prototype system.

We also discuss how to extend these implementation techniques to a general workflow engine which can run GAT-described processes.

1.6 Contributions: Promises – New Unified Isolation Mechanisms for Service-based System

The GAT model requires the programmer to provide code to handle each possible action under every possible state. What this means is that the programmer will have to write code to handle the effect of all possible interleaving among concurrent processes. To reduce this burden, we propose an approach to providing the benefits of isolation in service-oriented applications where it is not feasible to use the traditional locking mechanism.

Our technique, called ‘Promises’, is a unified approach to describing the interactions between a client and a service where the client can make sure that some condition over resources will hold at a later time, despite concurrent activities that occur between the check and the use of the condition. We present a promise protocol: here the client application determines the conditions they need to have hold over a set of resources and express these as predicates, and the resource manager will determine if it can grant the promise and reply. Then, once a promise has been granted, the client application can continue and make changes to the resources protected by its promises, with the guarantee that they will be allowed if they are within the conditions implied, and then client applications then release their promises.

Predicates are simply Boolean expressions over resources. Our model imposes no restrictions on the form these expressions can take, and their ideal form will normally depend on the nature of the resources involved and the way we want to view them at the time. We discuss the nature of resources and the way that this defines the types of predicates that can be used in promises over them. We describe in detail three ways of viewing resources: anonymous view, named view, and view via properties. We explore the relationship between these views and predicates. We also give examples that show how applications can use these different ways of viewing resources to obtain just the degree of isolation they need.

The Promise Pattern we propose is a style of interaction, in which a client can request another service to guarantee that a predicate will remain true for a limited time into the future. The value of our proposal depends on the existence of mechanisms by which the provider can keep its promises. We list some well-known implementation techniques which could work well with promises.

1.7 Contributions: Design Principles in Supporting Promises

We define some of the implementation issues that need to be resolved in promise-based systems and discuss how we built a proof-of-concept prototype of a Promise Manager that supported promise-based isolation. The major challenge in the implementation is to ensure that the Promise Manager takes overall responsibility and coordinates the activities to maintain the validity of non-expired promises; that is, resources must be available to satisfy every predicate that the Promise Manager is committed to maintain.

One of critical decision we made in implementing Promise Manager was to avoid changes to existing applications or resource managers that the Promise Manager interacts with. This allows us to reuse existing applications and resource managers thus increasing our productivity for the development of a proof of concept system. Our solution to this constraint is to implement our Promises prototype as a layer that wrapped existing application systems and resource manager and ensured that promises could be both granted and honoured.

The Promise Manager needs to keep a persistent record of all promises that are currently in effect. Our solution is to create an object for each promise and store it as a row in an SQL database table. Some mechanism also has to be provided so that resources defined in the predicates are available. We assume that the Promise Manager is able to query the resource manager to find out the availability of resources specified in the predicate. We provide code examples how this is can be done using SQL queries for both named resources and anonymous resources. Information about promises and resource availability are stored in different places and controlled by different managers, but they are both accessed as part of promise operations. The solution we adopted to prevent problems arising from concurrent access to the promises table and shared resources is to wrap each promise operation (such as creating a new promise, when action performed or updating existing promises) in a transaction.

Promise checking is at the heart of the Promise Making system. The promise checking guarantees that resources must be available to satisfy every predicate that the Promise Manager is committed to maintain. To ensure that granted promises are not violated, the Promise Manager implements a Promise Consistency Checking mechanism where it evaluates a set of promises against the current state of resources. We illustrate two Promise Consistency Checking mechanisms to cover named resources and anonymous resources. We also demonstrate the ways Promise Consistency Checking are used in various operations, such as making new promises, executing actions, and updating existing promises, which could violate the validity of promises.

1.8 Thesis Structure

This thesis provides programming models and protocols which can make it easier to build a reliable and robust system which can deal with events that cause deviations from normal processing paths, such as failures and concurrent activities.

In Chapter 2, we introduce the background and some research works which have been proposed to solve similar consistency problems on various different computing platforms. The aim of this chapter is to survey the existing approaches and show why they cannot be used to solve our problem in the service-oriented world.

In Chapter 3, we present our understanding of the nature of service-based systems using a realistic e-procurement scenario. The aim of this chapter is to define the common problems faced by the developers of the service-based system, and to identify key principles required in any solution.

In Chapter 4, based on the key principles we identified, we propose a new process description model called GAT. This can help the developers to build more reliable and robust system despite the occurrence of the failures and interference from the concurrent activities. We discuss the innovative ideas of GAT and its key features which can help the developers to avoid consistency faults.

Chapter 5 discusses the design principles that have to be addressed when implementing a business process which is defined in the GAT model. We illustrate, with code examples, a proof-of-concept GAT prototype system following the key design issues we identified.

Chapter 6 presents a unified isolation mechanism called Promises that is applicable to provide an appropriate degree of isolation to many applications in the service-based world. We discuss the concept, how it works, how it defines a protocol, and a list of potential implementation techniques.

In Chapter 7, we define some of the implementation issues that need to be resolved in promise-based systems. We also illustrate a proof-of-concept system built using today’s proven technology.

Finally in Chapter 8, we present the conclusion of the thesis. The major contributions of each chapter are summarised and we identify the future implementation and research work we plan.

Chapter 2

Related Work

In this section we review the previous and ongoing research efforts related to our research topic. We first look at the traditional transaction concept and its implementation in the database community which it first established the concept of consistency between data items. Then we look at the distributed computing and the role of transaction support which become one of the major key components to build robust systems across multiple organisations.

2.1 Traditional Transaction Support

The concurrency control and recovery mechanisms that ensure preservation of consistency between data items within a single database were important discoveries of early database research. The very idea of an ACID transaction is an important recognition since it involves mechanisms in the infrastructure to relieve the application programmer from worrying about failure and interleaving. We look into the concept of ACID properties, and also at advanced models which have been proposed by early database research to improve the shortcomings of ACID.

2.1.1 ACID properties

ACID (atomicity, consistency, isolation, and durability) [32] are considered to be the key transaction processing properties to ensure the integrity of data. Any database transactions that meet the characteristics of these four properties are considered reliable. We examine each of these four properties in detail illustrating them with a transaction that withdraws money from one bank account and deposits it to another account.

- Atomicity refers to the ability to execute completely or not at all. There must not be any possibility that only part of a transaction is executed. We say that the transaction commits if all operations execute, and it aborts if no changes are made. In our example, we have two operations: (1) withdraw money from one account; (2) deposit it to another account. To satisfy the atomicity property, either these two operations must both execute successfully or the effect is as if nothing executes. This guarantees that one account won't be debited if the other is not credited, as might happen due to failure during the second operation.

- Consistency refers to the database being in a legal state when the transaction begins and when it ends. This means that a transaction can't break any integrity constrains. For example, if an integrity constraint states that all accounts must have a positive balance, then any transaction violating this rule will be aborted.
- Isolation refers to the ability of the application to make operations in a transaction appear as if no other transactions were running at the same time. This also means that no operation outside the transaction can ever see the data and state in an intermediate state. In our example, the balance of money in the two accounts cannot be accessed or be modified by other concurrently running operations while they are being used by the current transaction.
- Durability refers to the guarantee that once the user has been notified of success, the transaction will persist, and not be undone. Typically, all transactions are written into a log when the transaction is committed and the log can be played back to recreate the transaction in case of system failures.

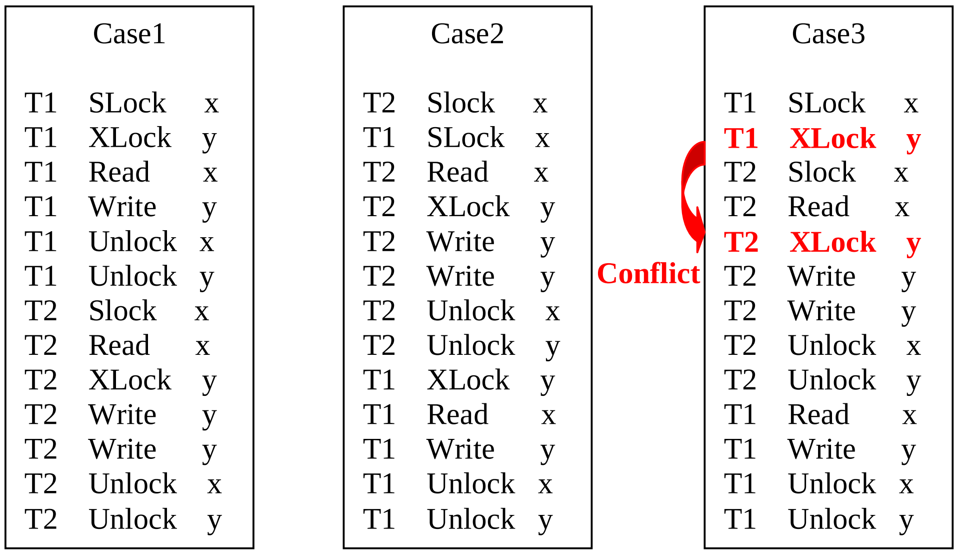
For decades, these ACID properties played an important role as the means to provide consistency required for database applications. Now we look at techniques used to guarantee the ACID properties. We first examine the locking mechanism which is provided within a local environment. Then we discuss Two-Phase Commit Protocol (2PC) as a mechanism which can guarantee ACID properties in a distributed environment where a transaction involves multiple databases which reside in multiple places.

2.1.2 Locking Mechanism

One of the key properties of transactions is “isolation” [32]. The meaning of isolation is that the executions of multiple transactions have the same effect as running the transaction serially, one after the other in sequence without having any overlap in executing among transactions. Such executions are called ‘serialisable’. Any system must guarantee serialisability to ensure there is no conflict among the data items used by concurrently running transactions. The most popular mechanism to ensure serialisability is locking.

Locking uses two types of locks, read locks and write locks. Before reading a piece of data, a transaction sets a read lock. Before writing the data, it sets a write lock. Read locks conflict with write locks, and write locks conflict with write locks. A transaction can obtain a lock only if no other transaction has a conflicting lock on the same data item. Thus, it can obtain a read lock on x only if no transaction has a write lock on x. It can obtain a write lock on x only if no transaction has a read lock or write lock on x. For ACID transactions, obtained locks must be kept until the transaction completes.

The following example in Figure 1 illustrates how two interleaving transactions can be isolated from each other. Note that we denote setting a read lock by SLock, XLock means a write lock is set, locks are released by Unlock operations at the end of each transaction.



Case1		
T1	SLock	x
T1	XLock	y
T1	Read	x
T1	Write	y
T1	Unlock	x
T1	Unlock	y
T2	Slock	x
T2	Read	x
T2	XLock	y
T2	Write	y
T2	Write	y
T2	Unlock	x
T2	Unlock	y

Case2		
T2	Slock	x
T1	SLock	x
T2	Read	x
T2	XLock	y
T2	Write	y
T2	Write	y
T2	Unlock	x
T2	Unlock	y
T1	XLock	y
T1	Read	x
T1	Write	y
T1	Unlock	x
T1	Unlock	y

Case3		
T1	SLock	x
T1	**XLock**	**y**
T2	Slock	x
T2	Read	x
T2	**XLock**	**y**
T2	Write	y
T2	Write	y
T2	Unlock	x
T2	Unlock	y
T1	Read	x
T1	Write	y
T1	Unlock	x
T1	Unlock	y

Figure 1 Conventional Locking Example (source from [32])

In Case1, two transactions T1 and T2 are isolated as they run in sequence without intervening with each other at all. In Case2, T1 and T2 interleave but their lock modes don’t conflict with each other therefore they can be called isolated. However, in Case3, T1 first places a write on the item y. Before this lock is released, T2 try to place a write lock on the same item y. Since write locks conflict with other write locks, T2’s attempt to place a write lock on the item y won’t be allowed.

Though the locking mechanism ensures the required isolation property, it has many disadvantages. One of the biggest problems is deadlocks. Deadlock refers to the situation where two or more transaction are competing for the same lock in conflicting modes, some of them will become blocked and have to wait for others to unlock their locks. For example, suppose T1 gets a read lock on x, and then T2 gets a read lock on y. Now, when T2 requests a write lock on x, it’s blocked, waiting for T1 to release its read lock on x. When T1 requests a write lock on y, it is blocked too, waiting for T2 to release its read lock on y. Since each transaction is waiting for the other one, neither transaction can make progress, so the transactions are deadlocked.

Apart from deadlocks, other problems are present in the locking mechanism. Locking mechanism is blocking which means the other transasctions have to wait until a lock held by the transaction is released. Locks are vulnerable to failures and faults. If one transaction holding a lock dies, other threads waiting for the lock may wait forever. Locks cannot scale well, as locks can only be only held within a trust boundary.

2.1.3 Two-Phase Commit (2PC)

One of the difficult problems solved by the database community was how to maintain the atomicity property across multiple sites as each machine can fail and recover independently. For example, now we assume that the update of withdrawing money takes place in the database which resides at a Sydney branch while the update of depositing money takes place in a Melbourne branch. To commit these two updates, both the one at the Sydney branch must succeed and the update in Melbourne must be successful.

However, it is possible that the update at the Sydney branch succeeds while the update at the Melbourne branch fails before the transaction commits there too. If no appropriate mechanism is in place, the failed transaction can never be recovered therefore atomicity is broken. Two-Phase Commit (2PC) solves the problem by enforcing that each task participating in the distributed environment writes its history of updates to stable storage before the transaction commits. 2PC protocol was developed in several products and later standardised by the Open Group within the X/Open specification [72]. A detailed description of 2PC is described in the Figure 2 shown in the context of our banking example.

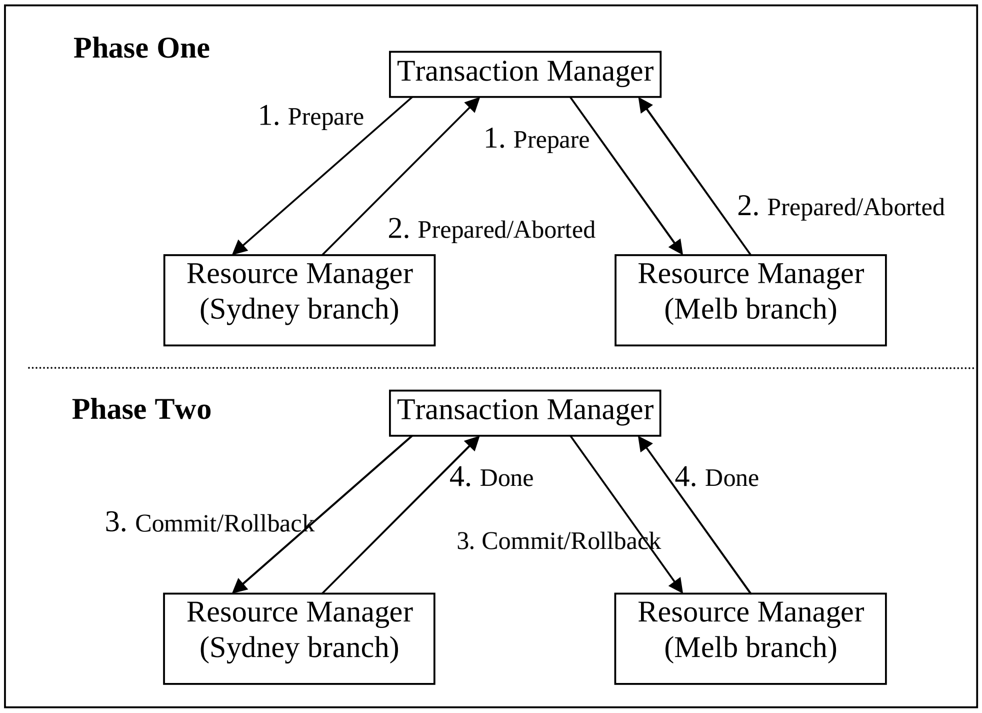


Figure 2 Two-Phase Commit (2PC) Protocol

Suppose the transaction manager has already started the transaction, and performed the two updates and that the transaction is now ready to commit. During the phase one, transaction manager sends a message ‘prepare’ to each resource manager. Each resource manager wrote its history of updates in the log as the transaction was executing. For example, the resource manager in the Sydney branch produces a log record showing the new and old values of the balance. When ‘prepare’ arrives, the resource manager makes sure these log resources are flushed to disk, so they will not be lost even if a crash occurs. The resource manager sends ‘prepared’ message back to the transaction manager once the log is successfully written to disk (otherwise it sends ‘aborted’ message). Similarly, the resource manager in the Melbourne branch sends ‘prepared’ message back to the transaction manager after saving on disk a log record showing the increased balance. The transaction manager waits till it receives ‘prepared’ from each resource manager (or till it receives ‘aborted’ from one, or till a timeout happens).

There are two different paths that can be executed by the transaction manager in phase two. One path executes if the transaction manager receives ‘prepared’ message from all resource managers during the phase one. The transaction manager sends a ‘commit’ message to all the resource managers. Each resource manager completes the task, by releasing all the locks and resources held during the transaction. Each resource manager sends a ‘done’ message to the transaction manager. This completes the transaction successfully. The other path executes if any resource manager sent an abort message during the phase one. The transaction manager sends a ‘rollback’ message to all the resource managers. Each resource manager undoes the updates which have been completed using information in the log, and then releases the resources and locks held during the transaction. Finally each resource manager sends a ‘done’ to the transaction manager.

The great disadvantage of the 2PC protocol is the fact that it is a blocking protocol. It has sometimes been called an ‘unavailability protocol’. A process will block while it is waiting for a message. This means that other processes competing for resource locks held by the blocked processes will have to wait for the locks to be released. The blocking becomes worse if the transaction manager fails permanently as some resource manager will never resolve their transactions. For example, suppose a resource manager has sent a ‘prepared’ message to the transaction manager but the transaction manager failed. The resource manager will block until a ‘commit’ or ‘rollback’ is received. If the transaction manager is permanently down the resource manager will block indefinitely. This type of blocking algorithm can only work well in a tightly coupled environment where conflicts between processes can be more easily monitored therefore resolved.

2.1.4 Advanced Mechanisms for Standard ACID

The original locking mechanism that locks the whole item was proven to be too expensive for some transactions. For example, suppose there are two transactions updating the stock on hand (soh). T1 reduces the stock by 150 (soh := soh – 150) while T2 takes 800 (soh := soh – 800). If there is stock on hand more than 950, these two transactions should be allowed to interleave but locking will prevent concurrency as each needs an XLock on the stock on hand.

Many refinements to locking mechanism to improve the level of concurrency have been proposed. The most relevant to our work is escrow locking [73]. The basic idea of the escrow locking is to preserve the truth of predicates, each of which is a condition evaluated as a Boolean value true/false, during the execution of a transaction. The escrow locking does this by recording high and low limits for the possible values of the item. If a concurrently running transaction violates the either high or low limits the transaction is rejected. Figure 3 illustrate how escrow locking operates.

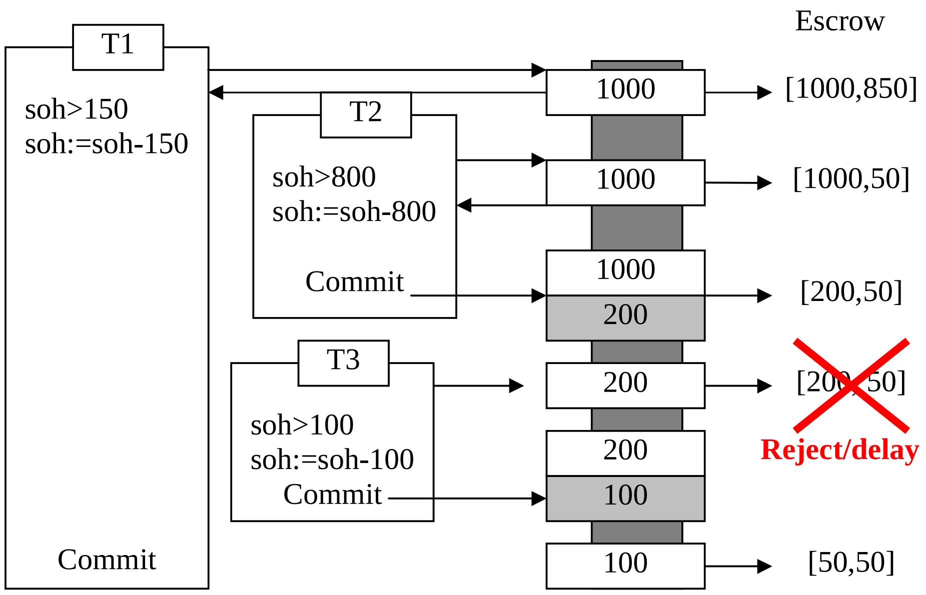


Figure 3 Escrow Locking Example (source from [32])

The operation of T1 gets executed after its predicate is satisfied (there are stock on hand of 1000 which is more than 150 the predicate tested for). The escrow records the predicate in a range between the value before the operation and the after the operation, such as [1000, 850].

Before T1 commits, T2 comes along. As this happens, before T1 commits, the stock on hand is still 1000 and the predicate of T2 is satisfied. The predicate range changes from 850 to 50 incorporating the possible changes for both T1 and T2. T2 issues commit which is granted because there are enough stock to commit for both T1 and T2. After T2 commits, the stock on hand values changes to 200.

Still before T1 commits, T3 comes along and checks the predicate which is satisfactory as there is more than 100 stocks. However, when T3 issues the commit, it cannot be granted as there is possible conflict between T1 and T3. The commit of T3 is delayed until T1 either commits or rollbacks. T3 can only commit if T1 rollbacks as this means there are still 200 stocks available. However, if T1 commits leaving only 50 stocks to be taken, T3 naturally gets rejected to maintain the integrity of the stock on hand.

This example shows how isolation can be achieved without unnecessarily locking the values that are being modified. However, the escrow locking only works for ordered numeric sets. In our work in implementing our promises isolation mechanism in Chapter 7, we use a similar technique to escrow locking.

2.1.5 Extended Transaction Models

It became clear very early on that the ACID approach was not appropriate for many activities that manipulate data in environments where assumptions such as short-living activities and trust no longer apply; therefore using mechanisms such as rollback or locking were not feasible. A range of extended transaction models were proposed aimed especially at cooperative processes like collaborative design, or long running business processes. We summarise and critique two early works that we think very significant to our research. These are Sagas [36] and ConTract [84].

2.1.5.1 Saga

The purpose of the Saga [36] is to give the atomicity property for long running transactions without having to hold locks for the whole duration. Saga structures a long running process as a sequence of smaller tasks, each of which would be done as an ACID transaction. Thus the underlying mechanism would ensure that each task ran without interference, but the tasks of one process could interleave with the tasks of another process.

The key insight of this work is in the way to respond to failure during a Saga. If a particular task fails, it can be aborted and rolled back, and then retried. However, if the Saga as a whole gets into an irretrievable difficulty, and needs to abort, what should happen? The answer proposed in [36] is that the application developer should design, for each task, a corresponding compensator. The compensator executes an operation which does the inverse of the original task. For example, the compensator for inserting a record might delete the record; the compensator for depositing to a bank account might withdraw from the same account, and a read-only task has empty compensator. To abort a Saga, the system will abort any active task of the Saga, and then invoke the compensators for each task within the Saga that had previously committed. The compensators are run in the reverse order from the order in which the tasks ran originally.

For example as seen in Figure 4, suppose a long running transaction T1 is composed of three tasks S1, S2, and S3. Programmers define compensators CS2 to undo the task done by S2, and similarly CS1 undoes the task done by S1.

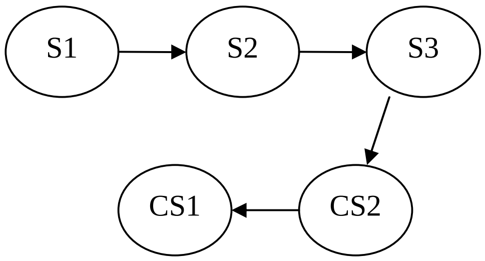


Figure 4 Long Transaction in Saga

The final outcome of this saga will be either the execution sequence:

- S1 -> S2 -> S3 and commit the saga successfully, or
- If the saga has to abort during the execution of S3, S3 is aborted and CS2 -> CS1 are executed to semantically undo the completed tasks S1 and S2,
- Similarly, an abort in S2 (after S1 has completed) will result in the execution of CS1 and thus no effect combined execution S1, CS1.

The Saga model has become a standard in transactional workflow model (see Section 2.2.2.3) and in service orchestration (see Section 2.2.3.2).

It is easy to prove that a concurrent execution of Sagas will be serialisable provided that each compensator commutes with every task and with every compensator that executed between the task and its compensator. However, it is almost impossible for many tasks to write compensators with such a strong property. In general, the drawback of the compensation approach lies in the difficulty of writing a compensator that winds back the original task from states that have changed significantly.

A compensator should remove not only the direct effect of the original task, but also any indirect effect through other activities which read the data now discovered to be inappropriate. For example, if the merchant has recorded a large order, and this has been used to calculate a bonus for the relevant region manager, then the compensator for the order task ought to recalculate the manager’s bonus (or rather, to maintain modularity, the compensator should somehow trigger a recalculation in the bonus process).

It may also be the case that the execution of one compensator ought to influence the activity of another compensator. This influence may not be possible when the compensators are run in the reverse chronological order of the original tasks. For example, in the original workflow the merchant might arrange shipment then receive payment from the customer. During compensation, any charges incurred by the merchant as it cancels the shipment, need to be deducted from the payment amount before the customer gets sent the refund.

Another difficulty with compensation-based systems lies in their assumption that the compensator always runs successfully. In real systems we have to deal with the case where we want to compensate for an overpayment, but it is possible that the recipient has already spent the money. A business process should not lead to inconsistent data when a compensator itself can’t be executed.

2.1.5.2 ConTract

In a traditional ACID transaction, each individual operation operates on data which is unchanged from that seen by earlier steps in the transaction. Thus, if one step checks the validity of a customer, then we can be sure the customer is still valid when all later steps use the customer information. In a long running business process, locks can't be held for more than a few seconds, so it is harder to ensure that the customer's validity is preserved once it has been checked. In [84], a general workflow description approach was introduced. As well as providing a language to express the sequence of steps, including the input and output parameters, a ConTract made explicit the conditions each task needed to complete successfully. These are called entry invariants of the task. The ConTract also expresses the conditions that are true at the end of a task, as exit invariants. For example, if a shipment task should only be attempted when the customer is valid; the application developer needed to state that customer validity is an entry invariant for the shipment task. The developer could also write that customer validity is an exit invariant for the validity checking task.

The syntax of ConTracts also allowed each exit invariant to indicate how it was to be preserved, and several possibilities were defined.

- Locks can be held, preventing any change to the data that was checked
- The exit condition can be preserved throughout the period, by having a check run on each interleaved activity; this other activity would be rejected if it could violate the truth of the exit condition
- The exit condition could be allowed to become false, and the system would re-run the check at the time when another task needs this condition for entry. In this situation, the developer would need to describe how to proceed if the revalidation check failed, by a 'conflict resolution' step on the task with the entry invariant. For example, the developer could indicate a procedure to call that would restore the invariant.

These techniques have not been fully implemented as they require sophisticated manipulation of logic by the workflow engine. However, we see these as offering a powerful framework to express the isolation conditions needed in application consistency. Our promises approach in Chapter 6 extends the ConTract ideas.

2.2 Distributed Computing Platforms

In this section, we examine the evolution of distributed computing platforms. We especially focus on the role played by transaction support to provide required reliability and consistency for the data being exchanged among distributed applications.

We start examining distributed computing models in the early era of procedural programming and later object technologies. Then we move to the era of workflow technology that is used as an integration tool within a single enterprise. This allows combining business logic for many steps running on multiple platforms. We discuss the research efforts to incorporate extended transaction model which tried to relax the atomicity and isolation part of ACID standards. Finally we review Web services technology and its role in integrating autonomous services that reside within different trust boundaries. We evaluate several standardization proposals to support transactions in this paradigm.

2.2.1 Conventional Middleware

Conventional middleware technology was an important milestone to facilitate and manage the interaction between applications across heterogeneous computing platforms that run on a set of servers. One of the important concepts developed in the middleware technology was to provide simple abstractions, and implementation to support such abstractions, for designing and building distributed applications. Especially, in this section we examine important concepts and abstractions developed for transaction support by middleware technology.

2.2.1.1 Communication Channel

The term distributed computing was established to refer to the model where several heterogeneous servers located in geographically different places work together to produce a common business goal.

One of the primary goals of middleware technology in the early era of distributed computing was to establish a communication channel among remote servers. In the beginning, this was done by sending messages to procedures located in other machines by using an operating system level interface called ‘socket’. The socket was an abstraction of underlying communication protocol such TCP/UDP. However, this method proved to be too problematic as different servers have different socket interfaces depending on the different operating system platform; also programmers found it tedious to deal with bit-layout and similar format issues.

Messaged based RPC was an important concept in distributed computing. This was the technology which made it possible to call procedures located on other machines in a uniform and transparent manner that looked just like conventional procedural code. RPC was introduced at the beginning of the 1980s by Birell and Nelson as part of their work on the Cedar programming environment [6]. Many important notions of distributed computing were mentioned in the paper, including client (the program that calls a remote procedure) and server (the program that implements the remote procedure being invoked). Other significant notions which are still widely used today in distributed computing system are interface definition languages (IDL), name and directory services, dynamic binding, service interface and so on.

RPC were developed in the era of procedural languages. As object-oriented languages took over, the industry required a different technology which could support remote calls among objects which reside in remote machines. Object Brokers were developed to do this job. One of the most notable object brokering technologies developed in this era is the Common Object Request Broker Architecture (CORBA) [71], which was defined and standardized by the Object Management Group (OMG). CORBA developed more advanced specifications for many aspects of object-oriented languages.

2.2.1.2 Transaction Support

Regardless of the different communication channels, all distributed computing required transactions if there is more than one call that had to work together as an atomic unit.

In the early era of TP monitors, they provided a primitive way of providing transactions using vendor specific technology which only works for certain mainframe or UNIX systems. With the advance of RPC, TP monitors such as IBM CICS [42] implemented transactional RPC which can deal with data distributed across multiple systems more uniformly and transparently. The semantics of transactional RPC is that multiple tasks can form a single unit of work as a transaction, and the transaction completes (i.e. commits) only if all tasks within the transaction successfully execute. If any tasks within the transaction fail to execute, the transaction fails. This idea gained popularity and was subsequently adapted by other TP monitors such as BEA Tuxedo [5], Encina from Transarc [19] or IBM LU 6.2 using different underlying mechanisms such as plain messages rather than RPC.

The semantics of transactional RPC was further developed into Two-Phase Commit (2PC) and subsequently became standardized by the Open Group within the X/Open specification [72]. Today, 2PC is the standard mechanism for guaranteeing atomicity for a distributed transaction.

Still, transactional RPC used synchronous interactions in which a call blocks the sender until it gets a response. Asynchronous RPC was devised by TP monitors where calls are placed in a queue and can be processed separately without having to wait for other calls to complete. The usefulness of such queuing was realized and they became middleware platforms on their own under the name of message-oriented middleware (MOM).

MOM presented a very important concept which considerably simplifies the way one supports managing errors and system failures. MOM ensures that once a message has been sent, it will be eventually delivered once and only once to recipients, even if the MOM system itself goes down. Messages are saved in a persistent storage and are made available once the MOM system is restarted. Recipients can also bundle a set of messages as an atomic unit then MOM guarantees that either the set of messages are processed altogether or none of them are processed. Some of the best-known MOM platforms include IBM MQ Series [43], MSMQ by Microsoft [60], or WebMethods [85].

2.2.2 Workflow Management Systems (WfMS)

An important class of middleware platform is used to control the execution of business processes built from many smaller activities. The term WfMS is used for this purpose.

2.2.2.1 Conventional Workflow Technologies

At the core of most workflow system is the notion of a business process. A business process is a set of business activities with a common business objective. This business process is built by linking together diverse business activities and specifying the flow of data and control among them. The following example illustrates an example of workflow for a simple ordering process.

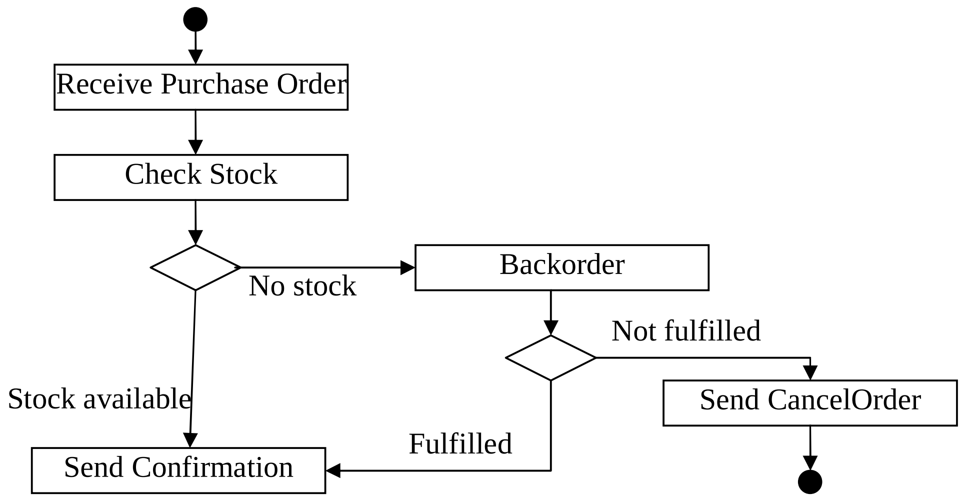


Figure 5 Conventional Workflow Example

WfMS typically provides a high level graphical orchestration tool where users can easily define flow of business activities in a style like Figure 5. The key concepts in these graph-based workflow notations include sequence (one activity starts when another finishes), decision, fork (starting several parallel threads), and join. These notations are very convenient for business analysts when describing the normal case of operation. However, they become very complicated when exceptional conditions must be handled. This is discussed in Section 2.2.2.3.

2.2.2.2 Event-Condition-Action (ECA)

As business processes become complex and sophisticated, a number of alternative orchestration models were proposed, each emphasizing different aspects of managing and controlling business activities. One notable technique most relevant to our research is ECA [15]. In ECA, a system monitors the occurrence of events rather than sequentially executing activities defined by a graph. When an event is detected, a specific action is executed to handle the situation. An action can be guarded by a condition, which is a Boolean predicate over event parameters. The condition is evaluated when an event is detected. The action gets executed if the parameters satisfy the condition.

Figure 6 illustrates an example of an ECA model where an invoice is sent to the customer when goods are reserved. After receiving the invoice, the customer sends in a payment. If full payment is received, the merchant makes a receipt and subsequently send it to the customer.

```
ON complete (goodsReserved)
IF true
THEN send Invoice

ON receive Payment
If (fullPaymentAmount == true)
THEN invoke makeReceipt

ON complete (makeReceipt)
IF true
THEN send receipt
```

Figure 6 ECA Example

One of the big advantages of the ECA model is to allow the description to be more modular and independent from other business activities. This makes it a lot easier to integrate different business activities in a loosely coupled environment which allows for dynamic joining and leaving of the business activities on demand. In addition, since any business activities can be triggered at any time during the execution, this approach can model activities that need to be started at the occurrence of any unexpected events such as deviations. ECA has been used especially to model exception handling even when groups are used for the normal case [11].

Our GAT model in Chapter 4 was inspired by this ECA model in controlling the flow of the data and business activities.

2.2.2.3 Transactional Workflows

A Workflow process is typically of long duration compared to the short running activities found in standard database transactions. Such long running processes are composed of many types of action, not just database transactions or message queuing. However, the whole set of activities often forms a single unit of work which ought to have much the same consistency provided by ACID, such as failure atomicity and isolation.

Due to the complexity involved in a workflow execution, it is often too expensive to rollback everything that had been done and to restart from scratch. Also, actions in the workflow not only involve database updates, but also they can involve actions in the other system, so rolling back some action could be very difficult because the appropriate rollback depends on the interaction of many actions. Furthermore, it is not feasible to place locks for the long lasting processes as they might cause excessive delay for other applications that need to access the same data. To overcome the inappropriateness of using mechanisms from standard database transactions, mechanisms from advanced transactional models were adapted.

WfMS maintain the state of workflow execution for each instance, such as which activities have been executed, in a persistent storage. Using this information, if the WfMS crashes and recovers, it will be able to restore the workflow up to the point of failure. The only work lost is what was performed by nodes that were active at the time of failure. This mechanism prevents the workflow system having to restart from scratch.

WfMS also provides a mechanism for use where it is not possible for the workflow to complete, and so the partial execution needs to be undone. This adopts the notion of compensation, first mentioned in Sagas [36]. The idea is to treat business activity as an atomic transaction which can commit. Each atomic transaction is attached with a compensator. In the case of workflow failure, compensators for the committed atomic transactions are executed in the reverse order.

Some WfMS prototypes developed in academic research offered additional primitives for handling exceptions. [91] shows how to provide Java-style try-catch-throw as an exception handling mechanism. The idea here is to associate a try block to an activity (or a set of activities). If the enclosing activity invokes any exception, it's captured by the catch block. Depending on the hierarchy of activities, an exception can be thrown to its parent activities. In [11], exceptional handling logic is specified by using ECA rules where the event defines the exceptional event to be captured by a certain action. The condition is a Boolean expression over the action that verifies whether the action actually corresponds to an exceptional situation that needs to be handled.

The Exotica project [2] explores the role of advanced transaction management concepts in the context of workflows. The basic idea was to provide the user with an extended workflow model that integrates Saga into the commercial workflow product Flowmark. The user could define a compensating task for each task of the workflow using a GUI tool. An Exotica engine would then translate these specifications into plain FDL (Flowmark Definition Language) by properly inserting additional compensating paths after each task or group of tasks, which are conditionally executed upon a task failure.

In [51], a transactional model for HP Changengine is presented. The model allows the definition of Virtual Transaction (VT) regions on top of a workflow graph. If a failure occurs during the execution of a task enclosed in a VT region, then all tasks in the region are compensated in the reverse order. Then, the system can retry the execution up to a maximum number of times. Then it executes an alternate path if one is present; otherwise the workflow engine terminates the entire process execution.

However, [88] showed that the concept of using compensators was not directly applicable to most real-world workflow applications. This is due to the lack of guidance in writing compensators. This left the developers to devise their own compensation logic which required intimate knowledge of how business activities interact in order to properly compensate for one activity’s execution. The use of semantic transaction models have been proposed to address this issue. [88] defined a semantically inverse task (commonly referred to as compensating tasks), or a chain of tasks that could effectively undo or repair the damage incurred by a failed task within a workflow called Information Carrier (INCA). INCA workflow model was proposed as a basis for developing dynamic workflows in distributed environments where the processing entities are relatively autonomous in nature. In this model, the INCA is an object that is associated with each workflow and encapsulates workflow data, history and processing rules. The transactional semantics of INCA procedures (or steps) are limited by the transaction support guaranteed by the underlying processing entity. The INCA itself is neither atomic nor isolated in the traditional sense of the terms. However, transactional and extended transactional concepts such as redoing of steps, compensating steps, and contingency steps, have been included in the INCA rules to account for failures and forward recovery.

Despite these proposals, many researchers have identified a great lack of adequate support for handling errors and failures in large-scale, heterogeneous, distributed computing environments with transactional workflow models [2],[28],[55],[76].

2.2.3 Business to Business Integration (B2Bi) and Service-oriented Architecture (SOA)

The need to integrate existing code or components is not limited to the systems within a single organization. The same advantages can be obtained by integrating multiple systems across multiple organizations. Web Services and Service-oriented Architectures (SOA) are being promoted as the best way to build such next generation Internet-scale distributed and integrated applications across multiple organizations. These are needed in business-to-business integration (B2Bi) and enterprise application integration (EAI).

These applications are made by gluing together opaque and autonomous services, possibly supplied by business partners and third party service providers, into loosely-coupled virtual applications that can span organisational boundaries and connect large-scale business processes. Services are just applications that expose some of their functionality to other applications in a particularly simple and restricted way. Services are autonomous, opaque (and probably stateful) applications that communicate with each other solely by exchanging asynchronous messages.

The key to success in this is interoperability between loosely coupled components which understand and process the messages. A set of technologies such as XML, Internet communication (such as HTTP) and standards from Web Services world makes this possible. In this section, we examine the standards from Web Services.

2.2.3.1 Basic Standards for Interoperability

This section we review three standards SOAP, WSDL, and UDDI defined by vendor-neutral W3C. An autonomous Web service can use WSDL to define its services, find other services on the Internet using UDDI, and send messages to other services using the standard message protocol SOAP.

SOAP

SOAP [82] is an XML-based messaging protocol. It defines a set of rules for structuring messages that can be used for simple one-way messaging though it is particularly useful for performing request-response style dialogue. The biggest advantage of SOAP is that it is not tied to any particular transport protocol nor is it tied to any particular operating system or programming language. This means that the clients and servers can be running on any platform and written in any language as long as they can understand and process SOAP messages. This fundamental assumption of SOAP makes it an ideal choice of messaging protocol in building a loosely-coupled service-based system.

The heart of SOAP is an envelope which contains an optional SOAP header, and a mandatory SOAP body. SOAP header is a generic mechanism for adding features to a SOAP message in a decentralized manner without prior agreement between the communicating parties. Typical examples of extensions that can be implemented as header entries are authentication, transaction management, or security. The SOAP Body element provides a simple mechanism for exchanging mandatory information intended for the ultimate recipient of the message. Typical uses of the Body element include marshalling RPC calls and error reporting.

The following example extracted from [82] illustrates how a client might format a SOAP message requesting product information for a produce with id = 827635.

```
<soap:Envelope
    xmlns:soap=http://schemas.xmlsoap.org/soap/envelope/>
    <soap:Body>
        <getProductDetails
            xmlns="http://warehouse.example.com/ws">
            <productID>827635</productID>
        </getProductDetails>
</SOAP:Body>
```

Here is an example of SOAP message that provides a response for the client request above.

```
<soap:Envelope
    xmlns:soap=http://schemas.xmlsoap.org/soap/envelope/>
    <soap:Body>
        <getProductDetailsResponse
            xmlns="http://warehouse.example.com/ws">
        <getProductDetailsResult>
            <productID>827635</productID>
            <productName>a dog mug</productName>
            <description> mug with two dogs</description>
            <price>6.50</price>
        </getProductDetailsResult>
        </getProductDetailsResponse>
</soap:Body>
```

WSDL

Web Services Description Language (WSDL) [83] is the standard format for describing a web service. It defines the functionality offered by a web service and the format of messages sent and received by the web service. A web service's WSDL document defines what services are available in the web service. The WSDL document also defines the methods, parameter names, parameter data types, and return data types for the web service. An application that uses a web service relies on the web service's WSDL document to access the web service's features. Each WSDL document contains four elements to describe a service: <portType> defines the operations performed by the service. <message> defines the messages used by the service. <types> defines the data types, and <binding> which defines the communication protocols.

The following example illustrates an example of WSDL documents for a stocking quoting service. The service defines two messages using <message> elements: one to receive a stock request and other to respond to the request. The operation "GetStockPrice" is defined by <portType> and a soap binding is specified by <binding> element.

```
<definitions name="StockQuote"

    <message name="GetStockPriceRequest">
        <part name="msg" element="xs:string"/>
    </message>
    <message name="GetStockPriceResponse">
        <part name="msg" element="xs:string"/>
    </message>

    <portType name="StockQuotePortType">
        <operation name="GetStockPrice">
           <input message="tns:GetStockPriceRequest"/>
           <output message="tns:GetStockPriceResponse"/>
        </operation>
    </portType>

    <binding name="StockQuoteSoapBinding"
        type="tns:StockQuotePortType">
        <soap:binding style="document"
            transport="http://schemas.xmlsoap.org/
            soap/http"/>
        <operation name="GetStockPrice">
           <soap:operation
               soapAction="http://example.com/
               GetStockPrice"/>
           <input>
               <soap:body use="literal"/>
           </input>
           <output>
               <soap:body use="literal"/>
           </output>
        </operation>
    </binding>
..
</definitions>
```

UDDI

UDDI [67] is a XML-based protocol that provides a distributed directory that enables businesses to list themselves on the Internet and discover other services. Similar to a telephone number, businesses can list themselves by name, product, location, or the Web services they offer. It is designed to be searched by SOAP messages and to provide access to WSDL documents describing the protocol bindings and message formats required to interact with the web services listed in its directory.

2.2.3.2 Service Orchestration

The basic web services infrastructure presented by SOAP, WSDL, and UDDI only suffices to implement simple interactions. In particular, it supports interactions where the client invokes a single operation on a Web service. When the interaction involved sequences of operations, additional support and tools are needed to ensure the correctness and consistency of the interactions.

The consistency of the interactions involved in the sequences of operations across multiple organizations has important implications from both an external (interaction) and an internal (implementation) perspective.

From the external perspective, the most important implication is how a Web service describes the set of correct and accepted message exchanges that are compatible and comparable to interacting Web services. This interaction problem among Web services has been researched in the area of service coordination. As this thesis doesn't deal with the consistency from the external perspective, we direct interested readers to work such as [33], WS-Coordination [94], WS-AtomicTransaction [92], WS-BusinessActivties [93], or WSCI [87].

Closer to the problem we are trying to solve in the thesis is how to provide the required consistency from the internal perspective. From an internal perspective, each service in the interaction must be able to execute relatively complex business activities that are compatible to the messages it exchanges with other interacting services. The important implication for the internal side is how to make it easy for developers to specify and implement complex business activities which are always compatible to messages being exchanged in the interaction. This is an implementation problem. Possible solutions will involve developing models, protocols, and tools support which can facilitate the effort of reacting correctly to a given message exchange. The term 'service orchestration' has been used to describe the efforts to devise solutions for this implementation problem.

In this section, we especially examine features from Business Process Execution Language for Web Services (BPEL). This is a primary source to discuss service orchestration and mechanisms used in supporting transactions in the Web Services area. We look at this proposal from three different angles: techniques used to control the flow of data and business activities, transactions which define transactional semantics among business activities which form a unit of work, and exception handling mechanisms which define how exceptional situations occurring during the execution of the set of business activities can be handled.

Business Process Execution Language for Web Services (BPEL)

BPEL [7] uses an orchestration model that combines the UML activity diagram [70] and the activity hierarchy (similar to flowchart) approaches which allow structured activities. These structured activities can group a set of other structured or simple activities to define ordering constraints among them. The structured activities can be sequentially executed, tested against a condition, picked in the occurrence of some event (such as the receipt of a message or the expiration of a time alarm), executed in loop, and run parallel with other structured activities.

The next example illustrates a simple purchasing order scenario in BPEL notation. The merchant receives a purchase order from a client. When PO is received, merchant runs two concurrent activities: one to calculate price, and the other to organize shipping. When these two concurrently running activities are done, the merchant sends an invoice to the customer.

```
<sequence>

// Receives a Purchase Order from a client
<receive
    partnerLink="customer"
    portType="lns:purchaseOrderPT"
    operation="sendPurchaseOrder"
    variable="PO"/>

// <flow> executes two business activities in parallel
<flow>
    // Calculates price for PO
    <sequence>
        <invoke
            partnerLink="pricing"
            portType="lns:computePricePT"
            operation="initiatePriceCalculation"
            inputVariable="PO"/>
  <receive
            partnerLink="pricing"
            portType="lns:computePricePT"
            operation="sendPrice"
            variable="Invoice"/>
    </sequence>

    // Organizes shipping to deliver goods for the PO
    <sequence>
        <assign>
            <copy>
     <from variable="PO" part="customerInfo"/>
              <to variable="shippingRequest"
                  part="customerInfo"/>
 </copy>
        </assign>
        <invoke
            partnerLink="shipping"
            portType="lns:shippingPT"
            operation="requestShipping"
            inputVariable="shippingRequest"
      outputVariable="shippingInfo"/>
   <receive
            partnerLink="shipping"
            portType="lns:shippingCallbackPT"
            operation="sendSchedule"
            variable="shippingSchedule"/>
    </sequence>
</flow>

// sends an invoice back to the customer
<reply
    partnerLink="purchasing"
    portType="lns:purchaseOrderPT"
    operation="sendPurchaseOrder"
    variable="Invoice"/>

</sequence>
```

BPEL follows a try-catch-throw approach borrowed from object oriented languages such as Java and C# to handle exceptions. Each activity implicitly defines a scope and includes one or more fault handlers, describing how a certain exception should be managed. When a fault occurs within a given scope, a BPEL engine will terminate all running activities in that scope and execute the activity specified in the fault handler for that scope. If no handler exists for a given fault, then a default handler is executed.

The next example illustrates a fault handler in BPEL. The operation which checks the purchase order details throw an exception which subsequently received by a fault handler which sends a message to the customer that the PO details is incorrect.

```
<faultHandlers>
<catch
    faultName="lns:cannotCompleteOrder"
    faultVariable="POFault">
    <reply
        partnerLink="purchasing"
        portType="lns:purchaseOrderPT"
        operation="sendPurchaseOrder"
        variable="POFault"
        faultName="cannotCompleteOrder"/>
</catch>
</faultHandlers>

<invoke name="checkOrderDetails" ..>
    <throw
        faultName="lns:cannotCompleteOrder"
        faultVariable="POFault".../>
</invoke>
```

BPEL combines exception handling approaches with techniques used in the advanced transactional models, notably from Sagas [36]. In BPEL, it is possible to define certain business activity required to semantically undo the execution of some activities in that scope. The compensation is specified by a compensation handler that will take care of performing whatever actions are needed to compensate for the execution. Every scope has a default compensation handler, whose behaviour consists of invoking the compensation handler for each enclosed scope in the reverse order of execution. Similarly, every scope also has a default fault handler, whose behaviour also consists in compensating enclosing scopes. The compensation handler for a given scope can only be invoked once the scope execution has completed normally. Its invocation can either be explicitly initiated by a compensate activity or it can occur automatically as part of the default handler. The compensate activity can only be defined within a fault handler or within a compensation handler of the scope that encloses the one to be compensated. BPEL also allows a compensation handle to be defined at the top process level. This enables the compensation of a composite service even after its completion.

The following example illustrates a compensator. When the ordering processing is cancelled after an invoice is sent, the compensator sends another invoice for a zero amount this supersedes the previously sent invoice is executed to semantically undo the effect of the completed task.

```
<scope name="sendInvoice">

    <compensationHandler>
        <reply
```

```
            partnerLink="purchasing"
            portType="lns:purchaseOrderPT"
            operation="sendPurchaseOrder"
            variable="AnotherInvoice"/>
    </compensationHandler>

  <reply
    partnerLink="purchasing"
    portType="lns:purchaseOrderPT"
    operation="sendPurchaseOrder"
    variable="Invoice"/>
</scope>
```

In Chapter 3.6, we evaluate the effectiveness of BPEL and other similar approaches, for expressing complicated deviation handling.

Other Service Composition Proposals

Before BPEL merged their ideas, IBM and Microsoft had issued alternative proposals. WSFL [95] was IBM's proposal for business process standards for web services. It uses WSDL to describe the service interfaces. A flow model describes the workflow for a process. Both control flow and data flow can be defined using a state-transition model. One innovative idea of WSFL was its handling of exceptions. WSFL supports handling different exceptions that are indicated in the content of messages by specifying transition conditions that examine the message for these exceptions. Depending on the transition conditions, different exceptions are directed to different activities.

XLANG [96] was Microsoft's proposal for business process standards for web services. Like WSFL, XLANG uses WSDL to describe the service interfaces of each participant. The behaviour is specified with a control flow that choreographs the WSDL operations. Transactions are scoped by context blocks, within which any number of business activities can be defined. Compensating blocks can be associated with each scoped context block. If a fault occurs in a scoped context block then the compensating blocks defined for the scope can be executed in the order specified by the designers of XLANG, but the default is reverse order. Exception handlers can be specified for any scoped context block and explicit recovery actions can be specified within the exception handler.

BPML [68] was a specification from the Business Process Management Initiative organization (BPMI.org). BPML supports both coordinated ACID support for short running transactions and long running transactions. A transaction can be associated with any complex activity and it can be nested. Compensation activities can be associated with both coordinated ACID and long running transactions. If a transaction is aborted, any compensating activities within the same context will be executed in reverse order.

2.3 Summary

Distributed computing support has evolved from simply invoking a single service within a single application, to arranging several invocations of many services which can be implemented by different languages and platforms located remotely outside trust boundaries.

Along with the evolution of the distributed computing platform, the requirement of transaction support has changed, from only handling short running transactions within a single trust boundary, to long running extended model transactions among autonomous applications that run across trust boundaries.

ACID was an important discovery providing mechanisms in the infrastructure to offer valuable reliability and robustness for business processes running short transactions. The mechanisms provided by the ACID model relieved the application programmer from worrying about failure and interleaving, to focus on how to automate business processes. Unfortunately, due to the nature of today's business processes which are typically autonomously written and contain long running processes connecting services across trusted boundaries, ACID cannot be adopted directly. The different assumption applied in these two environments requires different support for reliability and robustness.

The most popular transaction model adapted by current standardization proposals and commercial products is from Saga [36]. However, research has pointed out that the strong assumptions made for compensator models (such as that developers will successfully write compensators which can semantically undone the effect of completed business tasks) are too un-realistic. This thesis proposes new ways to approach this question.

Chapter 3

Understanding the Nature of Service-based Systems

In this chapter we identify some important issues which can prevent the developers of service-based system from building reliable and robust systems for automated business processes. The issues of concern include: time related issues raised from the asynchronous nature of a service-based system; various failures to terminate resulted from lack of coordination and global knowledge among autonomous systems, unprocessed messages caused by complex and sophisticated interactions among loosely-coupled components, messages which arrive out of order, lack of isolation due to activities run long duration across trust boundaries, and the increased chance of cancellations.

These issues are often causes of state mismatches which then produce various deviations from the expected execution path. If these deviations are not appropriately handled, the system will produce inconsistent outcomes. We present what we consider to be required behaviours to handle various deviations which commonly occur in the service-based system resulting from state mismatch. Based on our list of required deviation handling behaviours, we evaluate the existing standard deviation handling mechanisms to see how well they can be used as support.

3.1 Motivating Scenario

In this section, we illustrate an e-procurement scenario of ordering goods. This has been derived from a consultancy project of CSIRO. It illustrates a service-based system with the independent and stateful nature of services and the potential for concurrency. We use the e-procurement scenario as a case study through the thesis to understand the distinctive characteristics of service-based systems and the potential issues faced by the developers who are designing and writing reliable applications which run on the service-based system.

3.1.1 E-procurement When Ordering Goods

Figure 7 shows an overview of the e-procurement scenario. There are three major types of business parties involved in the scenario: a customer, a merchant, and suppliers. Each business party exposes the services it provides as Web Services. The communication between business parties are done via sending and receiving messages. The overall ordering process among three parties is as follows: the customer initiates the business process by sending a quote request. When the merchant receives a quote request, it responds with a quote or a rejection message. If the customer service decides to go ahead with the purchase, it will send a purchase order message and the merchant system will then confirm the order after reserving the goods which can honour the purchase order. After the order is confirmed, the merchant coordinates payment with customer and delivery with shippers simultaneously. The order process is completed when the customer pays for the goods and receives the goods; similarly, the merchant delivers the goods and receives the full payment. The merchant service might also exchange messages with suppliers to order goods if there were not sufficient goods on hand.

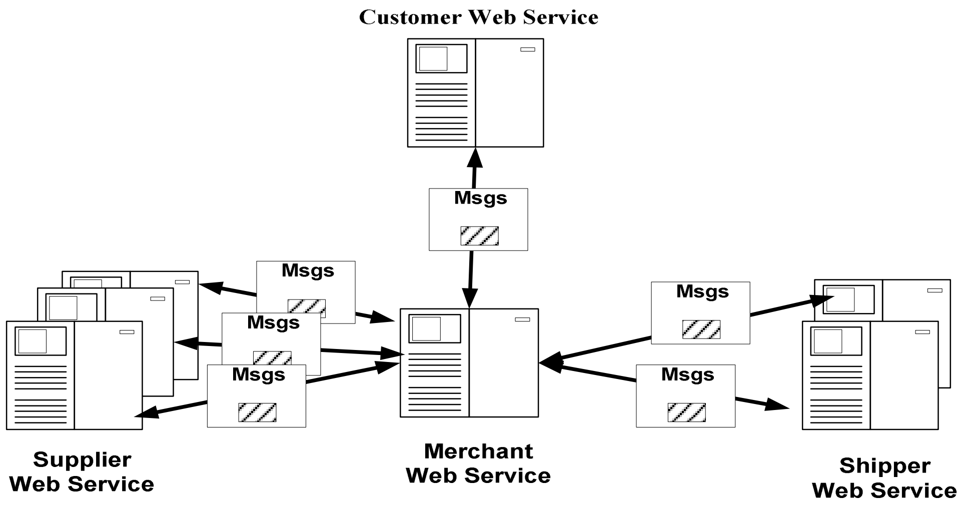


Figure 7 E-procurement Scenario

3.1.2 Merchant System

Particularly, we pay attention to the details of merchant business process within the e-procurement scenario because it requires the integration of numerous components. Internally, the merchant business process interacts with its internal catalogue and inventory system as part of enterprise application integration (EAI) strategy. As well it interacts externally with a customer process, a supplier's ordering system, and transportation booking, forming an example of business-to-business integration (B2Bi). Each external component is implemented, maintained, and managed independently by different organisations. There is only a partial trust between components running at different organisations. Also, the business activities within the merchant process typically may take days to months to complete. These characteristics of the merchant business process thus make it a good example of an application in a loosely coupled distributed system. Figure 8 shows the merchant workflow and we describe details of business requirements which have been defined as a set of tasks in the workflow.

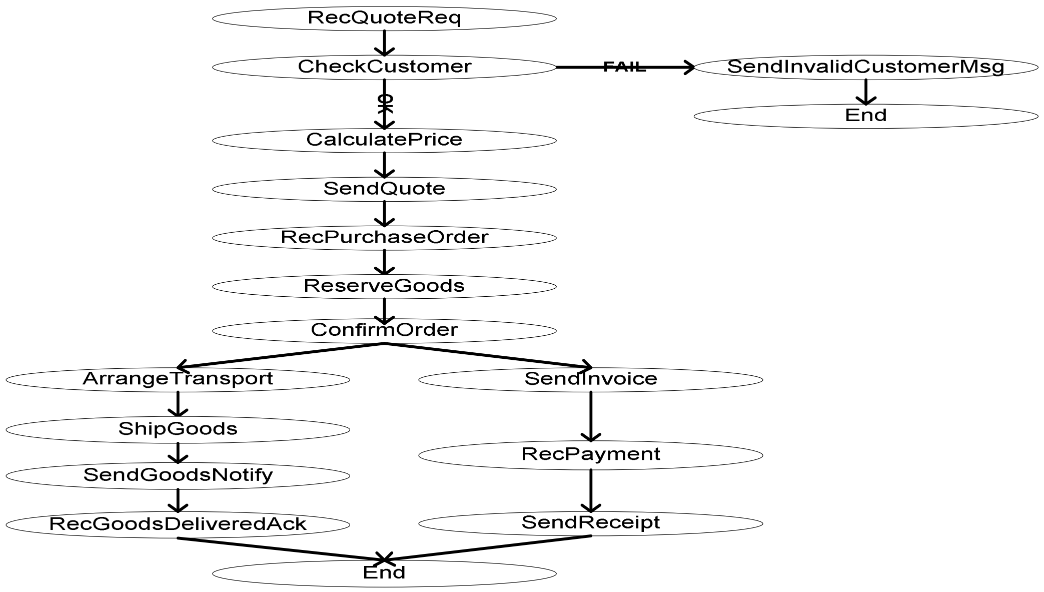


Figure 8 Merchant Workflow

Quoting

The first task in the workflow is to receive a quote request from customer (RecQuoteReq).The merchant checks that the customer is a valid customer (CheckCustomer). In our system, valid customers are ones who are registered customers with no overdue payment. If the customer is a valid customer, then the ordering process proceeds. If customers are invalid, either not registered or have overdue payment or both, the workflow sends a notification to the customer (SendInvalidCustomerMsg) and then the workflow is aborted (End). The merchant calculates the total cost of items in the quote request and generates a quote (CalculatePrice). The total cost will vary as it depends on a number of factors such as discounts available to the customer and current specials, amongst others. The quote is then sent to the customer, within 7 days since receiving a quote request (SendQuote).

Purchase Order

If the customer proceeds with the order, the customer sends a purchase order which the merchant receives (RecPurchaseOrder). The merchant reserves the goods from the warehouse's inventory system (ReserveGoods). The inventory system will return a date when the ordered products will be available from the warehouse. There may be insufficient stock in the warehouse to fulfil the order or it may require stocks to be replenished in which case an order to the supplier may be triggered. Exactly when and what other products are ordered from the supplier will depend on how the merchant manages its logistics. Once goods are reserved, the merchant sends an order confirmation to the customer (ConfrimOrder). After order confirmation is made, the merchant starts delivery and payment concurrently.

Delivery

The merchant arranges shipping with multiple shippers by sending a shipping request to each shipper. Depends on business requirements multiple shipping requests can be sent concurrently, or sequentially. After receiving shipping responses from shippers, the merchant picks the best shipper that can ship goods, for the best price with the most effective dates (ArrangeTransport). The shipper loads the goods onto the transporters and notifies this to the merchant (ShipGoods). The merchant sends a notification to the customer that the goods now are in transit (SendGoodsNotify). Customer can send an acknowledgement to the merchant that goods have been delivered when goods arrive at the customer's door and then the merchant receives this acknowledgement (RecGoodsDeliveredAck).

Payment Processing

At the same time as goods are being shipped to the customer, the merchant sends an invoice to the customer with a due date for the payment (SendInvoice). Payment is then received (RecPayment) and a receipt is sent to the customer (SendReceipt).

3.2 Issues for Service-based Systems

Any service-based system is constructed from pieces that need to remain autonomous, because they were written, and are run, independently. In many cases, they belong to different organisations which are competitors as well as collaborators; the organisations' goals are not the same, and each cannot extend trust to the other. The pieces use many resources and may include human intervention, so each lasts a long time. Although the environment of a service-based system is much more sophisticated and complex than a traditional OLTP system, still the goal of the applications built in such a system remains the same: building a reliable and robust system which can ensure that the system always finishes in consistent states. Building these applications is not trivial due to potential problems which could arise from the complex nature of service-based system, such as failures, races and other such exceptional events. Without understanding the issues of concern and devising mechanisms to deal with each issue, building a reliable and robust service-based system would not be possible. In this section, we discuss a list of key issues that the developers in this environment are required to understand.

3.2.1 Time Related Issues

In a business process, the time when something happens is often crucial to the success of the business process. In our e-procurement case study, there are many examples where timeliness is important to the success of the business process. For example, the total cost calculated for a purchase order must be based on valid unexpired quotes so the merchant and the customer have a shared understanding of the amount of payment which will be needed. The customer must have sufficient fund to pay for the goods before payment due date. The merchant must have a proper shipping arrangement in place where the shipper can deliver goods to the customer on time. The merchant needs to ensure that sufficient stocks are available when a shipper arrives to load goods, otherwise the customer might not receive the goods after payment has been made.

The time issue has become much more difficult to implement correctly in the service-based system due to the nature of services which run for long duration, and due to interaction between components which is typically asynchronous.

Concretely, we now examine how lack of timeliness can cause an e-procurement case study to produce undesirable effects, either the system ends up where customer paid for the goods but the correct items of goods were never delivered on time to the customer, or the merchant delivered the correct goods but the correct amount of payment was not received by the merchant before payment due date.

As we mentioned, e-procurement involves a quote that is only valid for a specified amount of time. Suppose the customer sends a purchase order before the expiry of the quote. But, due to network delay, it is possible that the merchant receives the purchase order only after the quote on which was based has expired. Subsequently the merchant might apply a new quote that is in effect by the time purchase order arrives. This situation can potentially cause different understanding for the payment the customer pays and the merchant receives. Another example appears when a customer sends payment before its due date but the merchant receives the payment after its due date and therefore charges the customer for a late fee.

Effect of delays in the real world can also often complicate business process implementation. For example, suppose that the merchant confirms the order because it anticipates more stock will be replenished before the shipment of the goods was provide. But, the extra stock got delayed, so the supplier was unable to stock up the necessary goods to the merchant warehouse. This could potentially leave the merchant unable to deliver the goods to the customer on time.

3.2.2 No Termination

In a distributed service-based system, the services provided by participating components are autonomous: that is, the implementation of each component is completely hidden and other components are forbidden to access any of its internal states. Multiple interactions happen simultaneously among participating components each having no knowledge of what is happening at other components. That is, there is no master component which oversees the overall interactions among multiple participating components. This introduces greater exposure to the problems due to lack of coordination and global knowledge of the system, such as deadlocks, starvation and other similar problems.

For example, the application will deadlock if it can reach the state where the merchant is waiting for payment before delivering the ordered goods while the customer is waiting for the goods to arrive before paying for them. Similarly, deadlock occurs when the merchant is waiting for a shipper’s response as whether the shipper can deliver the goods and the shipper is also waiting for the merchant for further information (such shipping date and customer address) before it can decide whether it can accept the duty of shipping.

Starvation will also occur in situations such as the following: the merchant confirms more purchase orders than can be filled with the amount of stock on hand. If this happens, some customers will wait forever for the delivery of goods which the merchant is unable to deliver. The merchant can also face a similar starvation problem if the customer sends purchase orders which require total payment more than the funds available to the customer. If so, the merchant delivers all the goods for the confirmed purchase orders to the customer and waits forever for the payment to arrive but the customer is unable to pay.

3.2.3 Unprocessed Messages

Messages play a vital role in service-based system as messages are only way to communicate between interacting components of a system [40]. Each component of a distributed service-based system potentially interacts with different sets of loosely coupled components. Each set of interaction aims to produce a different outcome depending on the business objective. For example, a merchant interacts with supplier A and B and shipper SP1 and SP2 for the e-procurement systems while the merchant interacts with supplier C and D and bank B1 for a supply-chain system. In such multi-layered interactions based on loosely coupled components, exchanging messages could become very complex and sophisticated. Due to such complexity and sophistication, the challenging issue is whether all mission critical messages will be handled appropriately. Not being able to handle a mission critical message could potentially become a lethal threat in building a reliable and robust system.

For example in our case study, the messages to ship goods and receive payment are two mission critical ones which must be processed together appropriately. In traditional OLTP systems, the operations dealing with these two messages would have been wrapped as a transaction, and the atomicity property of ACID will guarantee that the operations within the transaction are all processed completely, or none is. That means it will never create a situation where only one of the messages is processed. But in a service-based system, the operation to ship goods with the shipper and the operation to receive payment would be more likely dealt with as two completely different interactions. This is because each component keeps its autonomy and maintains a partial trust only during the interaction with its partner. In this case, the shipper wants to deal with the merchant regardless of how the merchant interacts with the customer. In such a situation, each node would not know what is happening at the other side. This might potentially create a situation where the merchant processes either the payment message or the shipping message but not both.

Even when messages do not get lost in the transmission, a message may remain unprocessed if the destination component is not expecting it. For the ordering process in our case study, the customer may send its payment before the due date and then terminate. This payment message could be delayed in transit and not arrive at the merchant until after the due date has expired. As the merchant has not received the payment by the due date, it would then send a late fee message to the customer, but this message can never be processed as the customer has already terminated.

3.2.4 Out of Order Processing of Messages

It is not enough to have all messages arrive and be processed by the destination. If messages do not arrive in the expected order, consistency can be at risk. Our e-procurement scenario illustrates many examples where undesirable effects might be produced due to messages which arrive and are processed in an unexpected order.

Suppose the merchant expect that payment is already received from the customer when it organizes shipping with a shipper. The merchant might use the payment from the customer to pay for the shipping. However, the payment has been delayed which leaves the merchant unable to pay for the shipping. The shipper now cancels the shipping. After the payment has finally arrived, the merchant is unable to ship goods to the customer.

3.2.5 Lack of Isolation

Concurrent use of shared resources is the source of many difficulties and many researchers have been working on mechanisms to isolate shared resources from other concurrently running activities [32]. One popular mechanism which has been used to solve isolation problems is locking. Here a lock is placed on a resource during the entire duration of a transaction so that the shared resources can only be accessed by the activities within the transaction, no but where else. Regrettably, this locking mechanism only works in an environment where activities run very fast and remain within a trust boundary. This assumption, however, doesn't apply in a service-based system where activities run for long durations often crossing trust boundaries. In such environments, it is unreasonable to place a lock on an interacting component's resources when there is no trust between components and when implementation details of each component are completely hidden from each other. Furthermore locking for the entire duration which could possibly last days to months is simply too expensive.

Despite the difficulties of isolating shared resources in the service-based system, a certain degree of isolation is required to prevent the system from producing undesirable effects. We list a few concrete examples from our e-procurement cast study to illustrate how the system can produce undesirable effects if there is insufficient isolation.

Before confirming an order, merchant may check the amount of available funds for a customer to ensure the customer has sufficient funds to pay for the goods. Though the customer might have enough funds at the time when the check occurs, this may not hold true by the time the customer has to pay. For example, after checking of the available funds, suppose the merchant delivers the goods. While merchant is delivering the goods, the customer might have used the funds for other orders leaving the customer unable to pay for the goods that are being delivered by the merchant. The funds of the customer, which is a shared resource, have been modified by other concurrently running orders other than one the merchant is involved in. This lack of isolation creates a situation where merchant delivers the goods but the customer is unable to pay.

Another example where lack of isolation could potentially create a problem is found with handling of stock level by the merchant. The merchant checks the availability of goods before confirming the order. While the merchant organizes shipping, other concurrently running orders could take goods the merchant was going to ship. This again will leave the merchant unable to ship goods even though the merchant checked the availability of the goods earlier.

3.2.6 Cancellations

Business activities in a service-based system are often of long duration and use asynchronous messages for communication, with complex business logic coordinating the collaboration between multiple components. This leaves many possibilities for failure, from faults in the system infrastructure to application errors, or simply changed situations in one or more components. Any of these causes can lead to the need to cancel some process that is underway or even completed. In traditional OLTP systems, cancellation is easy since all changes made within an atomic transaction can be rolled back by the system, restoring all data to its previous state. Unfortunately, it is not feasible to run an entire service-based application as a single atomic transaction because of the performance and other impacts of the locks used to ensure isolation [52], and so the applications in the service-based system need to contain application-specific mechanisms for dealing with the cancellation of an interaction.

The simplest cancellation is where the application stops the processing that was underway, and then terminates with no other action needs to be taken. For example, in the e-procurement example, taking no action is a reasonable response when the order is cancelled at any time before goods are reserved.

A slightly more complex class of cancellation is when normal processing is stopped, and some simple actions need to be performed to re-establish an appropriate state. For example, if an order is cancelled before transportation is arranged and before the invoice is sent, but after the goods have been reserved, then to cancel the order only requires that we undo the reservation. This class of cancellation is based on the Sagas model [36] where a business process is cancelled (aborted) by executing compensators in reverse chronological order for each task that has completed.

Some cases may require more complex cancellation handling. Suppose the order has to be cancelled for some reason after transportation arrangements have been made. One can try to cancel the arrangements but, if a cancellation fee is charged by the transport company, how do we define the business process used to pass on the costs onto the customer? Furthermore, how do we define a process to handle the situation where the transport company refuses to cancel its process, because the truck is already on its way to the warehouse?

Customer cancellation requests that arrive at certain points in the order process may require approval from a manager. The merchant may also need to interact with several of its partners as part of the cancellation process possibly applying different cancellation policies. For example, if an order had resulted in a back order being placed with a supplier, then this may also need to be cancelled. The manager's decision may depend on the internal state of the business (e.g. will there be too much stock in the warehouse) and whether its interacting components are also willing to accept the cancellation. If the cancellation is accepted, a fee may be charged; otherwise (when the cancel request is rejected) the normal order processing must be resumed regardless.

Cancellation after goods have been delivered introduces further complexity since we need to define processes for returning goods, checking that the correct goods are returned and ensuring that they are in acceptable condition. We also need processes to define how to handle the case when the wrong goods were returned or the returned goods are unacceptable.

3.3 Introduction to Deviations

Business processes run at each component of service-based system. Business activities that carry out instructions for business requirements are implemented within business processes. Defining a business process and modelling it would be a straightforward exercise if every activity within a business process always completed successfully and could never be cancelled. But as we examined in the section above, there are many issues to be faced by workflow modellers and software developers when activities deviate in various ways from their normal path. This is quantified in [74] where the authors report that nearly 80% of the time implementing a business process is spent on handling deviations.

The main focus of our work has been the consistency problem: ensuring that the set of autonomous components making up one service-based application always finish in consistent states. Building such applications which can maintain consistency is not easy due to various deviations which can occur from the simpler processing when everything goes well.

In this section, we look into the details of different types of deviations which a service-based system might produce. We argue that there are different types of deviations: recoverable deviations refer to deviations which have responses to correct them and return to the normal path. The responses are corrective actions which can fix the deviations as they occur then continue the process as if nothing has ever happened. Unrecoverable deviations have no responses to correct the problems. These can bring disastrous consequences such as system being terminated inappropriately. Both types of deviations can be caused by a mismatch between different states representing different aspects of system.

3.3.1 Recoverable Deviations

A business process is built from many smaller business activities. For some business activities, there is only one way to execute the business activity. If that only way of executing the activity fails, there is no way to correct things. On the other hand, for some business activities there are several different ways to execute the business activities. In such cases, if one way of execution fails it is still possible that there are other ways to execute that produce the correct result. In this scenario, when an original execution fails to execute successfully, it typically takes a business process away from its normal path. Corrective actions can be applied which take the business process back to its normal path. We call this type of deviation, which has responses which we can run to correct the situation, as recoverable deviations. Throughout the example of e-procurement, examples of recoverable deviations appear many times, some of which are listed below.

- Sudden popularity of certain goods may leave insufficient stock which results in the failure of the step for reservation of goods. Rather than rejecting the purchase order and aborting the ordering process, the merchant business process can instead trigger backorders which can replenish goods in the warehouse. Once goods are in stock, the reservations can be made successfully as if there had been enough goods in stock all along.
- Similarly, there are multiple transportation companies that can transport ordered goods to the customer. If one transportation company can not deliver, the merchant business process can find an alternative transportation company to deliver goods to the customer. From the customer point of view, the difference in transportation companies does not matter in receiving the goods so long as goods are delivered on time.
- When the merchant receives payment from a customer it is possible that the payment is less than the amount owing. This can result in an inconsistent result where the merchant shipped goods with value more than what the customer paid. To correct the situation, the merchant sends an additional invoice for a remaining amount plus some penalty, with an extended payment due date. The customer pays the remaining amount and penalty, and the ordering process then continue as if the customer paid in full in the first place.

In a traditional view, such recoverable deviations would be considered as failures. Thus they would not be recovered, but instead, subject to run one of standard deviation handling mechanisms [32] to take the system to its original state. However, taking the system to its original state in a service-based system is often either too expensive or it might not be possible due to the characteristics of the system we discussed in the Section 3.2.

3.3.2 Unrecoverable Deviations

There are some deviations where the system cannot find a way to continue forward to a reasonable. We call these unrecoverable deviations. Examples arise because of hardware or system failures, or from human error in application design.

There are system deviations where the network or server fails. In our e-procurement case study, system failures can cause ordering process to remains incomplete. For example, calculating a price can fail due to the failure of the price server. Reserving goods can fail if the reservation system is down. Arranging transport or shipping can also fail due to crashes in particular components. Sending messages (such as quote, purchase order, invoice) and receiving messages can all fail due to network failures. If one component of a service-based system fails, it is more likely that the overall interaction will fail as a consequence. Interacting components will be unable to progress, when they do not receive messages that are critical for them. For example, without receiving a payment from the customer, the merchant ordering process cannot complete.

Another common deviation in this category is due to programming bugs, a term which refers to an error, flaw, mistake, failure, or fault in a computer program. These prevent the system from behaving as intended, so producing an incorrect result. Programming bugs can come from mistakes made by people in either design or in connecting a correct design to code. For example, suppose an overdue customer has been evaluated as a valid customer even though the customer is recorded as having overdue, or a quote has been calculated based on the old price list even though new price list is available at the time of calculation, or the incorrect number of goods were reserved despite purchase order correctly stating the number. These situations can all happen due to mistakes made by developers. No matter how each component successfully processes all the messages as expected, the bug introduced by programmers will prevent the system from producing a correct result.

3.3.3 State-related Deviations

One notable type of deviations, which may be either the recoverable deviations or unrecoverable deviations, needs special attention. This is the state-related deviations. State-related deviations occur due to mismatch between states which represent different aspects of a system. For example, a state *stock_on_hand* represents the amount of stock being held in the warehouse in the real world. A state *invoice_sent* represents that a business activity has occurred (namely that an invoice was sent to the customer).

The accuracy of states is dependent on states being updated correctly every time reality changes. For example, the state *stock_on_hand* should be updated accurately every time the stock amount in the warehouse is changed. The state *invoice_sent* should only be produced after an invoice is really sent to the customer. As well, the accuracy of states may also be affected as a consequence of another state. For example, depending on the amount of funds available as represented by the state *funds_available*, the outcome of *payment_sent* will be decided. If the state *funds_available* contains sufficient money to pay for goods, the *payment_sent* will succeed. If *funds_available* contains less than what the customer owes for the goods, the *payment_sent* will fail.

But in reality, this synchronised updating between states doesn't happen perfectly all the time. Data entry errors can occur, there may be an anticipated changes in the world that are not captured in the computer system, network/server failures may present communication at the exact moment a synchronised update was supposed to happen, or the costs to maintain accuracy could be too great. However, the major causes that produce state mismatches remain the issues we discussed in Section 3.2.

Building a reliable and robust system cannot be done without handling the various deviations appropriately. In the next section, we closely examine different deviational situations which are created by the different types of state mismatches. Then we discuss the way different deviational situations can be handled, as a first step to guide the developers in handling deviations.

3.4 States and State Mismatch

The section is divided into two subsections. In the first subsection, we examine different types of states in the system that represents different aspects of the system. In the second subsection, we classify different deviational situations which can arise due to state mismatch.

We provide descriptions of how to handle various deviational situations as they occur in the example of e-procurement cast study. The description will provide a way to evaluate current and proposed B2Bi and EAI technologies on their support for deviation handling. In Section 3.6 we show that current technologies have limited support for handling various deviations.

3.4.1 States

We first define three different types of states that represent different aspect of business process – Abstract State, Business Process State and Real World State. We describe each of these types of state in detail, and we discuss the relationships between states.

3.4.1.1 Real World State

The Real World State is simply the state of the physical world, such as goods on hand and financial agreements. In our e-procurement scenario, the quantity of each product stored in the warehouse, the physical location in the warehouse where the goods are stored, the conditions of the goods (damaged or in good condition) and the locations of the warehouses are all part of the Real World State.

3.4.1.2 Abstract State

The Abstract State is a computer-based representation of the Real World State.

Each component in a loosely coupled distributed computing system has state. This state is an Abstract State as it is based on the data held within these computer-based models of the real world. This model includes information such as the expected availability of each product, the location in the warehouse where the goods are supposed to be stored and the customer's delivery address.

The Abstract State of a component is not necessarily exposed externally. This means that a component in the distributed environment may have no direct knowledge of the internal Abstract State of other components. For example, a customer would not normally know the level of stock available for any particular product; in airline reservations, a passenger would not know the exact number of seats available for a particular flight. However, it is possible for an external component to derive partial information about the Abstract State of a component by considering the component's behaviour – e.g. if a merchant accepts a purchase order then the customer knows that the merchant believes it will have at least the ordered quantity available at the time of shipment. The merchant may provide the customer with direct interfaces to query the value of the internal state, but this can never be more than a snapshot and so is potentially inaccurate.

3.4.1.3 Business Process State

A business process is defined by a set of activities and a specification of the order in which the activities are required to execute. In the e-procurement example, the business process for the merchant includes receiving a quote, verifying that the customer is a registered customer with no overdue payments, calculating a quote and sending the quote to the customer. The Business Process State is the point that the process is up to in its execution. A business process may contain forks (sets of activities that execute in parallel), thus a Business Process State can point to multiple positions in a process.

Examples of process state in e-procurement include states such as *quote request received, quote sent, invoice sent, payment received* and *receipt sent*.

There are two types of business processes. One type defines the internal activities of a component and the other defines the externally visible behaviour. In the e-procurement example, the merchant would not expose to its partners (e.g. customers and suppliers) the internal processes but would expose a sub process which consists entirely of activities that interact with its partners. There are thus two types of Business Process State: Internal and External. An External Business State encapsulates or summarizes a set of internal states. For example, the external process state *payment received* encapsulates the internal state of *received payment*, *awaiting the validation (of payment)* and *received validation*.

3.4.2 Classification of Deviations

This section classifies a number of situations where the business process deviates from normal processing as a result of state mismatch situations. Some of these deviations are recoverable while others are not. This means an architect must ensure that there are processes defined to recover from recoverable deviations and to prevent unrecoverable deviations; otherwise, unacceptable behaviour will occur that may result in adverse outcomes, such as financial loss. The events that cause a deviation from the normal processing paths can occur at any time, even when handling previous deviation, making it more difficult to ensure correct behaviour under all circumstances.

3.4.2.1 Mismatches between the Real World State and the Abstract State

An Abstract State is a representation of a Real World State. The Real World State of a warehouse is the physical state of the warehouse, such as what products and in what quantity, is stored in the warehouse as well as their storage location in the warehouse. An inventory system is an Abstract State representation of the Real World State of a warehouse.

In an ideal world, the Abstract State and the Real World State would be consistent with each other. Unfortunately, this is not feasible due to the cost and effort required to keep them synchronised at all times. In particular, the timeliness issue due to the asynchronous nature of service-based systems leads to many temporary mismatches between the Real World State and the Abstract State. As well, difference between the Real World State and the Abstract State can also be caused by real world events that are not reflected within the computer system. For example, goods can become damaged in a warehouse or can be stolen and are thus no longer available for sale. This state mismatch will persist until the inventory system is reconciled with the actual physical goods in the warehouse, something that may not happen until the next stock take.

Take an example where an Abstract State isn’t consistent with a Real World State in the e-procurement scenario. A deviation will occur when there are actually insufficient goods in the warehouse to satisfy an order but the inventory states otherwise.

As it is not feasible to keep the Abstract State and Real World State synchronised at all times, these types of deviations are unavoidable. Thus, to ensure correctness (that is, avoid unacceptable behaviours), we must be able to handle deviations caused by inaccurate abstract state.

Correct handling of deviations arising from inaccurate Abstract State is application dependent. In the e-procurement scenario, if there is insufficient stock available for delivery then there are various ways the deviation can be handled. They include:

- Delay the order until a backorder arrives and reschedule delivery;
- For orders that include other products, send all available goods as scheduled and send unavailable goods when they become available – that is, partial fulfilment of an order;
- Cancel the order.

Depending on the circumstances, how this deviation is handled may depend on the decisions and policies of the merchant and/or customer.

If this deviation is not appropriately handled then unacceptable behaviour may result. For example, goods may never be delivered to the customer but the customer is still invoiced and sends payment to the merchant; or the business process may never terminate.

3.4.2.2 Prohibited Abstract and Real World State

Integrity constraints define, via the Abstract State, that certain Real World State are prohibited. Examples of integrity constraints for e-procurement include the requirement that each customer does not exceed their credit limit, and that available stock for a particular product is not below a specified amount unless there is an active backorder.

Deviations are thrown during a business process when an integrity constraint is violated. Though guaranteeing integrity constraints is a job of application programmers, lack of isolation can often cause the deviations in this category to occur. For example, two concurrently running business activities read the credit limit of a customer at the same time thinking that each business activity only takes some amount that doesn't make the credit limit exceeding. But in truth, the sum of these two concurrently running business activities exceeds the credit limit if both are granted.

There are two different approaches to solve such problem. One is to apply an appropriate isolation mechanism which can prevent any deviations to occur due to integrity constraint being violated. We discuss the ideas for this in Chapter 6 of this thesis. Other approach is to let deviations occur, and deal with them after the fact, to keep the system from ending up in an unacceptable state such as when the customer credit limit has exceeded leaving the customer unable to pay for orders. How a deviation should be handled is application dependent. For example, if a customer exceeds his or her credit limit while placing a new order, then there are a number of ways to handle the deviation. They include: increasing their credit limit (maybe temporarily); requesting that the customer deposit funds into their account; or cancelling the order which caused their credit limit to be exceeded. Notice again that there are a number of possibilities to handle the deviation and that the business process does not have to be cancelled as a result of the deviation.

The more interesting example relates to business constraints that are important to guarantee the integrity of overall business processes, such as available stock is not below a specified level unless (or) there is an active backorder. If an order reduces the available stock to something below the acceptable level, the deviation should definitely not cancel the order but instead trigger a new backorder. If the backorder throws a deviation, perhaps because the supplier no longer stocks the ordered product, then the merchant can try to find an alternative supplier and if no such supplier can be found, they might remove the integrity constraint for this product and update the inventory to reflect that this product will no longer be available once all remaining stocks have been sold.

3.4.2.3 Prohibited Time-based Internal Process State

An interesting type of prohibited process state arises from events that are supposed to occur. A business process may specify when an activity in a business process has to occur, and if it doesn’t happen by the specified time then a deviation should be thrown.

A good example is the requirement that payment from the customer should be received by its due date. When this deadline is missed as might happen because of message delays or customer tardiness, there are many ways to handle the deviation. They include notifying the customer of overdue payment and extending the deadline for payment; charging an additional late fee and sending a new invoice; cancelling the order if the goods have not been shipped, (and possibly charging the customer a cancellation fee); and as a last resort, initiating legal action (a human oriented activity) to recover costs.

Additional complexity can be caused by the asynchronous nature of the interactions between the merchant and customer’s business processes. If the merchant sends a new invoice, including the late fee, in response to an overdue payment then it is possible that payment for the initial invoice will then be received. The merchant would then wait for payment for the late fee payment only.

If there is no appropriate mechanism in place to keep track of state of an overdue payment, the system might wait forever without termination for the arrival of a message. Care must also be taken when defining how to handle an exception between the customer and merchant; otherwise it is easy to end up with unacceptable outcomes. For example, customer may pay the original invoice, then receive another invoice (which covers the original charge plus a cancellation fee) and pay that in full as well. Inconsistency can also occur if, after the merchant sends a second invoice which includes a late fee, they receive payment from the customer for the original invoice but then forget about the late fee which is still outstanding.

3.4.2.4 Mismatch between Internal Process State and Abstract State

Successful execution of an activity from a particular Internal Process State may depend on the Abstract State having appropriate values at that time. For example, there must be sufficient funds available at the time the customer wishes to send payment to the merchant and there have to be sufficient stocks available at the warehouse for delivery when transportation arrives. The most intuitive scheme for specifying the conditions required for successful completion of an activity is by attaching predicates (pre-conditions) to activities in a long running business transaction. This was pioneered in [84] and [52] and we adopt this idea in our proposal in Chapter 4.

In a long running business activities, if an activity *A* depends on a condition to successfully execute, the business transaction will typically execute an earlier activity *A'* in the business transaction to ensure that the condition will be true when the activity *A* executes, for example, in e-procurement, the merchant will typically reserve the quantity of goods required by an order so that there will be sufficient goods available at the time of delivery. However, just because goods have been successfully reserved does not mean that the goods are already stored in the warehouse since the reserved goods may be goods that are scheduled to arrive from the supplier before the delivery date, that is, the predicate may not be actually true when the activity *A'* executes but is expected to be true by the time *A* executes. Furthermore, if *A'* successfully executes, there is actually no guarantee that the predicate will be true when activity *A* executes since the goods may have been taken out by other concurrently running activities.

The traditional approach to solve such problems is to make A and A' as a single transaction to ensure the condition (which was checked in A') is always true when the activity A executes. The lock placed during the transaction guarantees that no concurrent running activities interferes with the state A and A'. But as discussed in the Section 3.2.5, the locking mechanism is only feasible when it can be guaranteed that clients will always release locks fairly quickly; and this is not a guarantee that can be given with untrusted clients and long-running business processes. In Chapter 6, we propose an isolation mechanism which can work in such long-running business processes in the service-based system.

3.4.2.5 Mismatch between External Process States

B2B integration often requires autonomous components and long stateful interactions between numerous participants. The e-procurement scenario is a good example; the participants include a merchant, a customer, and shippers. Each component exposes its External Business Process State to its partners. However, the External Process State of one component may not be compatible with another component's external process state. Examples of such incompatibility include:

- The customer is in the *received receipt* external business process state while the merchant is in the *invoiced* state. These two external states are incompatible since the customer could not have possibly received a receipt if the merchant has not even sent the invoice. However, the external business process state *paid* for the customer is compatible with the merchant's external business process state *invoiced* since payment may be in transit.
- The customer is in the external business state *cancelled* state while the merchant is in the *successfully completed state*.
- The merchant is in an external business state which is awaiting payment but the customer is in a state which indicates that its business process has terminated.

These incompatibilities can be prevented by ensuring that the components' (dynamic) behaviour is compatible with respect to a B2B coordination protocol. That is, when one component sends a message to another component, the destination component is expecting that message, and whenever a component is awaiting the arrival of a message, some other component will (eventually) send a message of the correct type.

In a correct design and implementation of a B2Bi application, the business processes participating in the B2B interaction should never be in a situation where one business process is waiting for events (messages) that will never happen; neither should any component receive unexpected events. Such incompatibilities will cause business processes to never terminate and cause messages to be lost or queued somewhere, never to be properly processed. Although ensuring the compatibility of different external processes is an important aspect to producing a consistent system, we don't cover this topic in our thesis. We refer interested readers to work by our colleagues in [33].

3.4.2.6 Incompatible Abstract States

Even though Abstract States are internal to a component, two or more components' internal state may be incompatible in a B2B interaction. In e-procurement, examples of incompatible Abstract States include the following:

- The amount payable for an order differs in the customer and merchant's Abstract State. Similarly, if an order is cancelled, the merchant and customer may differ in their understanding of the cancellation fee that is payable.

- Similarly, the products ordered in an order differ in the merchant and customer's Abstract State.

Incompatible Abstract State becomes evident when one of the business processes throws an exception after receiving a message from another component, for example the merchant would throw an exception when it receives payment from the customer in which the amount is incorrect.

The deviation needs to be appropriately handled and the incompatibility between the Abstract States resolved; otherwise, the environment would be left in an inconsistent state. Issues of the compatibility of different processes are covered in depth in [33].

3.5 Desired Features in Handling Deviations

In Section 3.4, we have explored various deviations due to different cases of state mismatches. We also discussed potential ways of handling deviations at each case of state mismatch. This has given us an insight into a more general set of features needed in handling deviations in a service-based system. Again, we use e-procurement scenario to illustrate concrete examples of the desired features provide better understanding of each feature.

Section 3.5.1 describes the types of behaviours required when deviations cause a business process to deviate from normal processing so seriously that it is desirable to cancel the original processing entirely. We focus particularly on situations where a cancellation request is received, but there is a lot of similarity with the types of processing needed following occurrences of unrecoverable deviations.

Section 3.5.2 describes the types of behaviour required when recoverable deviations cause a business process to deviate from normal process but in a less serious way that does not require cancellation. In view of the effort already invested in a long running process, and the number of collaborators involved, a business process should only be cancelled as a last resort.

3.5.1 Cancellations

We list a number of desired deviation handling features when a business process requires terminating because the deviations are too severe to be corrected. The simplest from of terminating the business process is simply stopping the currently running business process without doing anything. A more complex handling of deviations can be that the system runs independent activities which can reverse the effects of the activities so far. A more advanced form of handling deviations is to examine all state that had been changed before deviations occurred and correct the state. This needs great understanding of which state components to examine and how to correct them. The last resort when the system is extremely complex is to notify human operators so that even the most complex and rare situations can be handled manually.

3.5.1.1 Terminate All Processes and Simple Activities

The simplest type of behaviour is to stop, that is, the abnormal situation calls for the system to terminate whatever activities are underway. Within this case, we allow for the cancellation to also execute some additional simple activities after terminating whatever is active.

An example of this type of behaviour is if the customer decides to cancel the order before the merchant has performed any significant activities that impact on the real world. Suppose the cancellation request arrives before the reservation of goods has started. The merchant may be currently calculating a price, or it may have just received the purchase order, but it has not reserved goods from the inventory system or arranged for delivery. The business process can be terminated and all that might be needed is, optionally, executing a simple activity which updates the status of the order from *active* to *cancelled*.

For this class of behaviour, notice that cancellation of a business process does not necessarily return the system back to its exact original state. For example, the customer database may have been updated. Furthermore, same activities are not undone via compensation transactions, for example, sending and receiving quotes, and calculating the quote are not compensated. All that is required is basically to terminate the business process.

3.5.1.2 Executing Compensator like Activities

A slightly more complex class of behaviour requires in dealing with deviations are reverting the effects of the deviations by running compensator like activities similar to Saga transaction model [36].

In the e-procurement scenario, if an order needs to be cancelled after goods have been reserved and transportation arranged but before an invoice has been sent and the goods shipped, then the order can be cancelled by running compensators in reverse chronological order for those activities that have successfully executed.

Compensator like activities do not always need to run in reverse chronological order, but sometimes application defined order is appropriate. In the e-procurement scenario, certain circumstances place different constraints on the order of compensators. For example, if a cancellation is received after payment has been received and transport arranged, there may be some cancellation charges from the shipper which must be passed on to the customer. Thus the compensator for receiving payment (which refunds money to the customer) should only execute after the compensator for arranging shipment has executed. Since the original processing of the delivery process was concurrent with the payment process, the completion could have been in either order. That is, in this case compensators do not necessarily occur in reverse of the original chronological order.

There are also cases where one activity has to execute before another but their compensators can execute in any order. In the general case, the compensators may be required to execute in an order that is application specific.

3.5.1.3 Executing Independent Activities

There are circumstances where the required behaviour is not to follow the traditional view of rolling back. A business process may need to execute a process whose processes are independent of the activities that have executed in the original forward processing.

For example, in the e-procurement scenario, if ordered goods have already been shipped then this will require the invocation of a return goods process. It would arrange the delivery for the unwanted goods back from the customer (either to the original warehouse or perhaps to an alternate storage site). It would also involve special checks to make sure that the goods returned were the ones originally delivered, that the goods have not been damaged, and so on.

Notice that the return goods process may itself fail and this needs to be appropriately handled.

3.5.1.4 Activities Dependent on State

The correct behaviour to handle effects of deviations in a long running process may depend on the state of other activities and data. The status may not be known at the time when a fault needs to be handled.

For example, if an order is cancelled then the cancellation fee is dependent on the state of the delivery. If there is a fee for the cancellation for delivery, then the costs are passed onto the customer. If an invoice has not been sent, then an invoice for the cancellation fee is sent to the customer; if an invoice has been sent and payment has been received, then a partial refund is sent to the customer.

The final case is a more awkward to handle as it introduces an extra dimension to the problem: the exact state is unknown. In this case, the merchant has sent the customer an invoice but has not received payment, and the merchant does not know if the customer's payment is in transit or not. The merchant has to send the customer an invoice for the cancellation fee but a payment for the original invoice may arrive after the merchant has sent the invoice. In this case, the merchant has to assume that if the customer receives an invoice for the cancellation fee and has already sent payment that the customer would ignore the invoice for cancellation fee. The customer would then wait for a partial refund from the merchant.

Care needs to be taken in defining the protocol between customer and merchant; otherwise, inconsistencies could occur where the customer never receives the correct refund or the merchant never receives due payment.

The final case can be more simply handled if the merchant does not allow orders to be cancelled, and it only sends an invoice after the goods have been shipped. That is, we can place extra constraints on the business process to avoid being in unknown states when a fault occurs.

3.5.1.5 Human Intervention

It is not realistic to expect that all deviations can be handled without human intervention since there may be very complex/special circumstances. Flexibility for handling cancellation is greatly increased if humans handle the most complex and rare situations. It is straightforward to initiate and receive notification of the outcome of human intervention activities via simple mechanisms such as sending and receiving emails.

3.5.2 Continuing to Make Forward Progress

Exception issues have been widely investigated in the workflow research community. Most techniques developed in the workflow research are similar in that each first terminates the current business process then deals with exceptions. For example, Hwang et al. [43] propose a model for handling workflow exceptions based on previous experience. When an exception occurs, a search on the previous experience in handling similar exceptions is conducted and the result is applied. Casati and Pozzi [12] describe a taxonomy of expected exceptions by categorization of similar exceptions and mapping to exception handling for each class of categorized exception. Based on this taxonomy and meta-model, Chiu et al. [15],[16],[17] developed a Web-based WFMS to support automatic resolution for expected exceptions. Fung and Hung [26] go a step further by regenerating a workflow specification which can invoke alternative Web Services to remedy the failure of mission critical business activities.

However, in a service-based system, it's often impossible to terminate a business process due to the complex relationship has been built during the long duration of interactions as well as particular environment of the system. If possible, the system should deal with the problems (i.e. deviations) as they occur and always progress forward. The simpler form of dealing with deviations is to find alternatives which can produce the same effect as the event which caused the deviations. Sometimes, it might not be possible to simply apply the alternatives from the point where a deviation just occurred, the process might need to rollback to an earlier point (where most recent stable states can be retained) and restart from there. In this situation, one may aim to reapply the event that caused the deviations or to apply alternatives. A more advanced way of handling the deviation is to continue the process despite the deviations, and also to create additional activities later which can handle the effect of the failed events.

3.5.2.1 Alternatives

Activities in a business process may fail, for example, a transportation company may not be able to deliver the goods at the required time. When a failure occurs, it is often inappropriate to take the drastic action of aborting the business process. It is possible in certain situations to execute alternative activities, and if they are successful, the business process can continue as normal such as the merchant finds another shipper.

3.5.2.2 Rollback to Earlier Points in the Processing and Redo

An activity may fail and the most appropriate cause of action is to undo via compensation like activities or other independent activities, such as alternatives, thus returning to an earlier point in the business process (savepoint) and then restarting from the savepoint.

A concrete example where this type of behaviour is required is if the delivery of goods were sent to the wrong address. The shipment is returned, and then the merchant determines the correct address and then resends the shipment. This may be done by another shipper than the one shipped the wrong goods originally. The activities in the payment process need not be undone or redone.

3.5.2.3 Continue Processing and Create Additional Process

Partial fulfilment may be required when a merchant can not provide all the goods at the time of delivery, for example if a backorder is delayed. The merchant thus ships the available goods and later it arranges transport of the other goods when they become available.

This class requires that the process spawn another process to handle currently unavailable goods; meanwhile it must continue normal processing for the goods that are already on hand. There is also a need to modify other processes such as the invoice to the customer is not for the full amount but only for goods that have been shipped. A later invoice is sent when the unavailable goods are shipped.

3.6 Critiques of Standard Mechanisms and Supports from Current Technologies

In this section, we explore in more depth the issues concerning the intrinsic shortcomings in the standard approach to dealing with deviation in systems composed from Web Services. This standard approach uses an exception-handling mechanism similar to that in programming languages, together with application defined compensators which semantically undo completed activities or all or nothing features from traditional ACID transactional model. We described this approach in Chapter 2.1.5.1.

A fundamental assumption made by the standard model is that that every completed activity can be semantically undone. That is, the model assumes that application designers can always write a correct compensator for each activity. However we saw above that it may not be possible in all cases to undo the effects of shipping some goods. The only way the standard model can deal with activities that cannot be undone is to define an *empty* compensator, but this is unsatisfactory because it is treated as successful compensation and thus enclosing scopes are not aware that the activity has not been undone correctly.

Even if some aspects of an activity can be undone, it is not always the case that we can return exactly to the original state. The compensator for an activity, such as reserving goods, is to remove the reservation. One might believe that such a compensator is guaranteed to successfully execute, but the reservation may have triggered off a back order. If the backorder plus the original reservation together would leave the merchant with an excessive quantity of goods, then this simple compensation might be unacceptable.

Furthermore, the standard model does not seem to take account of possible state-dependence in how compensation should occur. At least in the BPEL expression of the model, the compensator has access to the stored state in databases etc, and to the state captured in containers by the original activity, but it does not have access to the current state of running concurrent activities. However we have seen that the correct way to rollback a completed shipment of goods can depend on the status of the concurrent payment process.

Another flaw in the standard approach (such as BPEL described in Chapter 2) to cancellation arises from the assumption that fault-handling should involve the immediate termination of all running activities within the scope that has suffered the fault. This assumption makes sense in the traditional object-oriented programming languages where the exception-handling concept arose, but it is not valid for all cases of faults within a long-running business process. In contrast, we believe that proper handling may sometimes require the application-specific fault-handler to intervene intelligently in the running activities. It should examine the current state of the scope and then act on different activities in different ways: some may be allowed to reach a stable point, some may wish to take special preparations before termination, some may need to be killed, and others maybe should proceed to normal completion, unaffected by the fault.

The standard approach does not provide sensible code reuse between the default handlers and customised ones written by the application developer. The programmer must either rely entirely on the default (which simply runs compensators in reverse chronological order), or they must write the entire handler from scratch. There is no opportunity to do some preliminary activity and then invoke the default handler, nor can programmer access information used by the default handler, such as the order of completion of the sub-activities. For example, one couldn't write a customized handler which runs the compensator of the last completed sub-activity, but not any others.

3.7 Summary

In this Chapter, we started with the question of what makes it difficult for developers to build a reliable and robust service-based system. To answer this question, we needed to look at the environment of a service-based system. Some of major concerns of a service-based environment are:

- The asynchronous nature of interaction among autonomous components which made it harder to deal with the timeliness issue.
- The complex web of relationships between multiple components, which deal with different partner components depending on business context, makes it difficult to process all messages elegantly and correctly. This potentially leads the system to halt without a proper termination or terminate with unprocessed messages.
- There is a high possibility of interference from concurrently running business activities due to lack of appropriate isolation mechanisms.
- And possible cancellation requests at various stages of the running system can leave the developers in a challenging position.

We explored different types of deviations and we observed one important point needing special attention: states. The issues we identified for service-based systems have contributed to the number of possible state mismatches. This in turn produces many chances for the system to deviate from its normal path. We looked at the different deviational situations produced in each type of state mismatch. For each deviation, we also provided descriptions of possible ways to deal with the situation using our e-procurement case study. This is important because without proper mechanisms in place to deal with deviations it would be difficult for the application programmers to build reliable and robust systems [74].

We then used our understanding of how deviations need to be handled, to discuss requirements on the developers' description of deviation handling. We showed the drawbacks in the standard mechanisms as found in BPEL and the others. In brief, the key requirements for describing business processes are:

- Need to represent many different approaches including both forward progress and cancellations.
- Need to allow deviation handling to interact in sophisticated ways with aspects of state including business process state.

This led us to devise a new model which allows the developers to define all sorts of handling mechanisms cleanly and declaratively. We present the model in the next chapter.

Chapter 4

GAT – New Event-Driven Programming Model for Defining Business Processes

In the previous chapter, we examined the issues which make it difficult for the developers to build a reliable and robust service-based system which can always finish in consistent states. We discussed the more frequent and complex types of deviations which occur due to state mismatches. Our evaluation of the existing standard mechanisms for expressing business process definitions appeared to be pessimistic about their support for designers who seek robustness. One of the key problems with existing mechanisms is that they treat normal activities and deviational events differently using separate handling techniques. This means that there is very limited support for dealing with recoverable deviational events, where we found were important.

In this Chapter, we propose a new model and notation for expressing business processes. This can help designers of business systems to avoid many common sources of errors in handling various deviations resulting from state-mismatches. This new model is called GAT, standing for Guard-Acton-Trigger following the name from the major elements of the model. Unlike most existing standard mechanisms and current technology tools, our GAT model does not separate the normal business activities from deviational cases, nor does it use a special handling mechanism for deviations. In addition, GAT allows each activity to access the wider range of states such as Abstract States and Business Process State. This makes it easier for the developers to make use of more accurate information about the current state of the system. This can greatly help the developers when they plan how to handle different types of deviations.

Using a payment process taken from the e-procurement scenario presented in the previous chapter, we describe some common difficulties in defining the business process. Then we show how features of the GAT model can help the developers in these cases. We give all of the payment process written in the GAT model. Finally, we show how GAT relates to previous proposals for expressing business processes.

4.1 Payment Process

In Chapter 3, we used an e-procurement scenario of a merchant business process to discuss various issues faced by application developers in designing a service-based system. In this chapter we use just the payment section of this larger example (seen in the red circle in the Figure 9) as a case study to discuss our proposed GAT model and to demonstrate how one can write a business process using GAT. We also illustrate how some features of GAT can guide the application programmers to write a business process that can maintain data and state consistency.

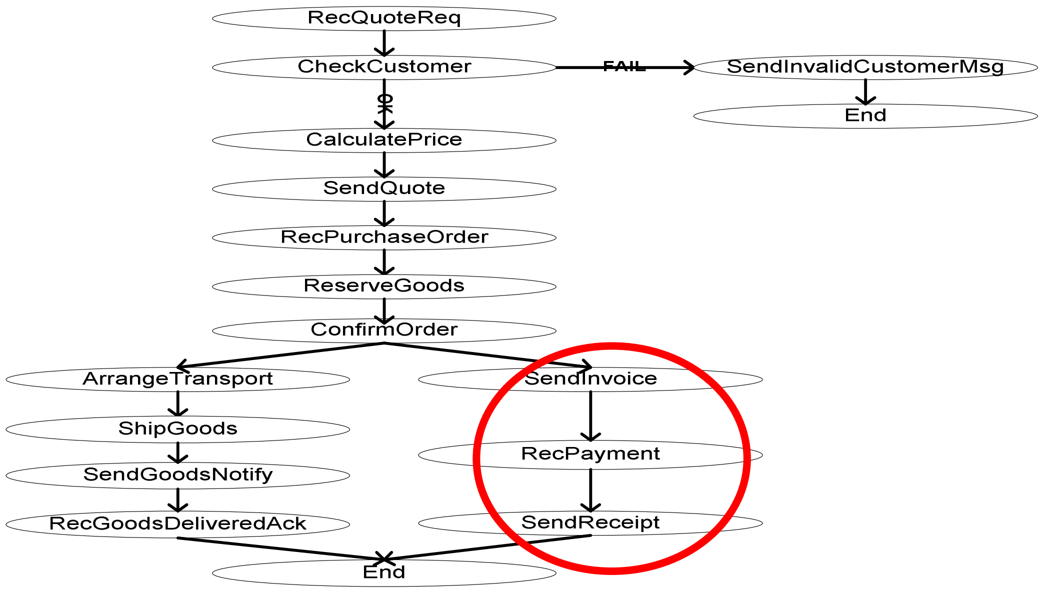


Figure 9 Payment Process within the Merchant Process

We describe four scenarios derived from the overall payment process in this section. They describe procedures that are followed in dealing with payment which include: sending invoice, receiving payment, sending receipt and dealing with cancellation requests (the last is not shown in the diagram). We do not consider at the moment any temporal events and related constraints other than these four procedures. At each scenario we also describe potential problems which can cause the payment system to produce inconsistent outcomes if they are not handled properly. Then in Section 4.3, we illustrate how features of GAT help the application programmers to write a business process avoiding the problems we have mentioned.

4.1.1 Send Invoice

The merchant sends a confirmation to the customer after goods have been successfully reserved. Then, merchant starts two parallel business processes:

shipping process to deliver reserved goods to the customer, and the payment process which deals with the payment.

The first business step after starting the payment process is to send an invoice to the designated customer. The merchant at this stage checks the details of the customer to ensure there is enough information, such as billing address, an invoice to send to a designated customer. If any details of the customer are incorrect the payment system must send a notification to Customer Management System (CMS) to deal with the incorrect information, for example by contacting the customer by phone to get the correct address.

A potential problem that may cause the system to produce an inconsistent outcome appears when customer doesn't respond to the invoice: that is, they do not send payment within the due date. The merchant cannot wait forever for payment to arrive as this will cause the payment system to never terminate.

The merchant should define a procedure which can deal with the lack of response events. For example, when payment is overdue, business logic may require the merchant to extend the payment due date and send a reminder, with a penalty, to the customer. This procedure can be invoked as soon as the payment due date elapses so that customer is notified of the overdue payment. The merchant may allow sending such reminder up to 3 times. Once the overdue payment is received (within 3 reminders), the payment process continues its normal path and other parts of the merchant's system can be notified that the payment process has completed successfully. If overdue payment is not received after 3 reminders, the payment system sends an alarm message to a (human) manager.

4.1.2 Receive Payment

This business scenario describes the procedure to deal with payment sent by the customer. There are three possible paths that can be followed in response to the payment depending on the amount of the payment as seen in the Figure 10:

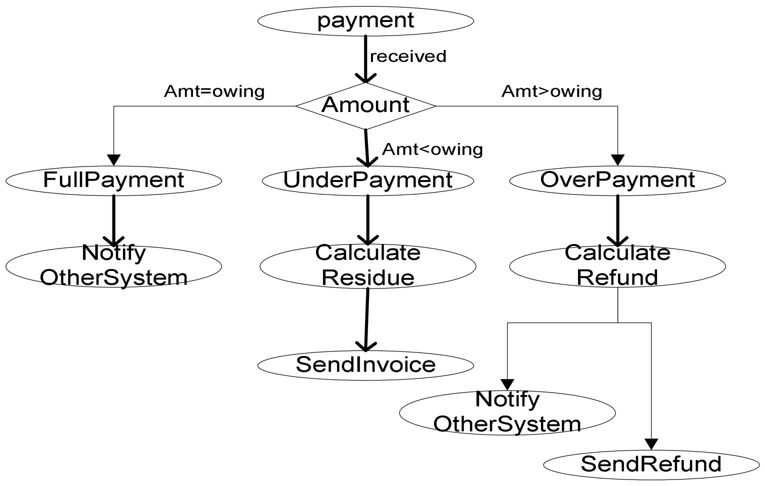


Figure 10 Receive Payment

- *Full payment:* the payment received is equal to the amount owed. The rest of the merchant's application is notified that this phase of the procurement cycle has completed successfully.
- *Under-payment*: the payment received is less than the amount owed. In this case, the amount still owing is calculated and an additional invoice is sent to the customer. Depends on the types of customers, the calculation of the remainder owning can vary. For example, if under-payment is made by premium customers, no late fee is applied. If a normal customer makes the under-payment, a late fee is added to the additional invoice.
- *Over-payment*: the payment received is for more than the amount owed. In this case, there are two further actions to be taken. One is to calculate the over-payment and refund it. The other action notifies the rest of the merchant's application that the customer has paid in full.

There are a number of potential problems in this simple example that could lead to inconsistent outcomes and business process deviations. For example, a customer under-payment has to be handled as an event that needs correction rather than as a deviation leading to the whole payment process being aborted. In this case, the customer should receive an invoice for the residual amount so the correct payment can be made. Similar to the overdue example in the above scenario, the payment process should continue down the normal path and other parts of the merchant's system can be notified that the payment process has completed successfully once the complete amount has been received.

Concurrent unprotected actions can also result in inconsistent outcomes and process failures. Without suitable protection, actions can check that it is safe for them to proceed, only to have a concurrent action invalidate this decision by changing critical shared state between the check and the code that depended on it. For example, the action that calculates the residual amount and sends an additional invoice runs because the system has checked received < owing. If an incoming payment changes the state before the extra invoice is sent, an inconsistent state may be produced. Similar when payment that is more than the money owing is received, the calculation of refund must not be interrupted; otherwise the amount of refund might be altered by concurrently running activities that access the same state (the amount of refund).

The different possibilities for the payment amount should all be considered and properly handled. A merchant payment system must have mechanisms in place that can deal with all different amount of payment received. For example, if merchant system only deals with exact payment amount, that is when the payment received equals the money owing, it can result in the customer paying more than what they owed without getting refund, or the merchant receiving insufficient payment.

The payment process defines different procedures for different types of customers who underpay and this should be measured correctly in the process system. For example, premium customers should only get a remainder for the remaining amount when they underpay whereas non-premium customers must incur a penalty.

Business processes in a service-based world are built by putting together different parts of systems from legacy code to newly developed code. It is most likely organisations implementing business system reuse their legacy code wherever possible. If the merchant payment system already has code that deals with payment, the code should be reusable.

4.1.3 Send Receipt

As soon as customer pays the full payment (that is equals to the amount specified in the invoice) a receipt is sent to the customer.

4.1.4 Cancellations

The overall payment process depicted in Figure 10 only shows the normal path which is straightforward. However, there are many different ways payment process is forced to cancel, such as due to the overall ordering process being cancelled, or the customer simply has changed his/her mind and sent cancellation requests, or system and network failures occurred.

A business transaction typically has some persistent effect on the overall system state, even if a cancellation has occurred. For example, just rolling back is not an appropriate response to an order being cancelled. The existence and outcome of the attempted order have to be recorded, and there may be other consequences such as the imposition of cancellation fees.

In most situations, a component participating in a B2B transaction cannot unilaterally cancel (abort) a business transaction, rather all possible current states of the system by the time of cancellation should be considered, for example, once a merchant has accepted a purchase order, neither the customer nor merchant can unilaterally cancel the order. Even if the order can be cancelled, the subsequent process is often complex, perhaps goods may need to be returned to the merchant and checked before the process can continue, and there is no guarantee that the return goods process will be successful. We need to be able to deal with problems that can arise during cancellation of an order and cannot just assume that a cancellation will always be successful.

When a cancellation occurs in a business process, there are typically many possible ways to handle the deviation and it is not always the case that the business transaction itself has to be cancelled. In the following examples, we illustrate different ways to handle cancellation requests sent by the customer. Notice the different behaviours required to handle such deviations at the various stage of the payment process.

- *Payment not received and before goods in transit*: The simplest cancellation scenario is where the cancellation request is received anytime before the payment is received, the payment process can be terminated without taking any further action. We assume the merchant payment process imposes a cancellation fee if the cancellation is received after 7 days since an order confirmation has been sent.
- *Payment received before goods in transit*: A more complex class of cancellation is the one where the cancellation request is received after the payment is received, but before the goods in transit. The cancellation of the process then only requires the refund of the received amount or the remaining amount after deducting the cancellation fee. The payment process terminates after refund is sent to the customer.
- *Goods in transit*: Handling the cancellation once goods in transit could be very complex. For example, goods need to be returned from the customer to the merchant once they arrive at the customer's door, the cost of returning goods needs to be calculated and paid by the merchant, and a refund needs to be made to the customer if customer has already paid, but a cancellation fee may be subtracted from the refund. These descriptions all depend on business needs. As our aim for this case study is to show the different paths possible for cancellation requests which arrive during different stages of the system. We simply assume that the merchant rejects the cancellation once goods are in transit rather than trying to describe complex

business scenarios. In this case, the merchant payment process merely continues to progress as if nothing has happened.

Being able to access to a wider range of states is a key factor in providing more flexible handling mechanisms for different deviations (including cancellations) that occur at different stage of the payment process. For example, the payment process does not only access the Abstract State representing payment (amount of payment) but also access the Business Process State of knowing how far the shipping process has progressed so that the state of shipping can be taken into consideration when deciding on a desired handling mechanisms for a cancellation requested during the payment process.

4.2 GAT Programming Model

This section first gives an overview of the orchestration framework called GAT, which stands for Guard, Action, and Trigger, following the critical components of the model. We then illustrate how the payment process procedures discussed in Section 4.1 are expressed in this framework in Section 4.3.

Defining processes specifies their behaviour. Defining these processes is relatively straightforward for the normally expected case when everything goes according to plan. Unfortunately, many problems can arise during an execution as seen in the above examples. The defined processes must specify how each of these problems is to be handled, and no problems can be omitted or neglected without risking serious consequences.

The process description framework GAT is based on an event model to define processes. We believe such event based model is more appropriate and effective than traditional graph based models, such as BPEL (Business Process Execution Language for Web Services). Graph based models are sufficiently expressive to describe a process for the normal, expected case; however they lack the flexibility required to clearly and precisely specify how various deviations should be handled as described in Chapter 2.2.2.3.

We now look at the details of GAT programming model from its structure to the major features that GAT offers.

4.2.1 Structure

In GAT model a Process is written as a set of Activity Groups. Each activity group consists of an Event and a set of related Activities, as shown in Figure 11.

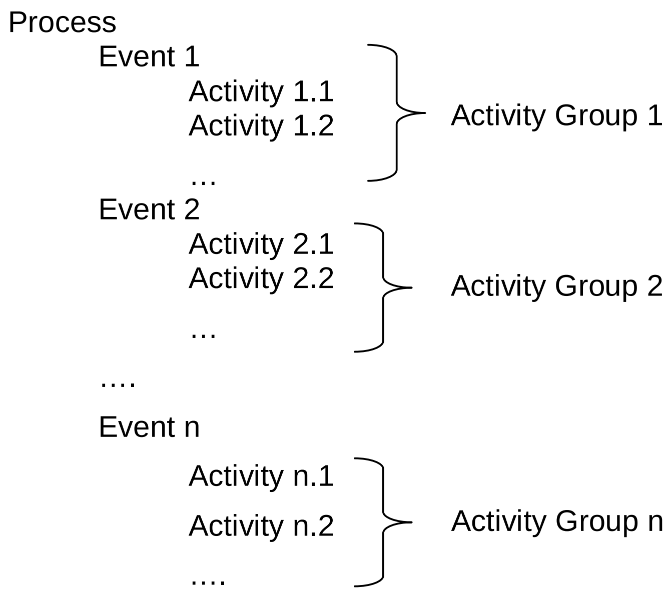
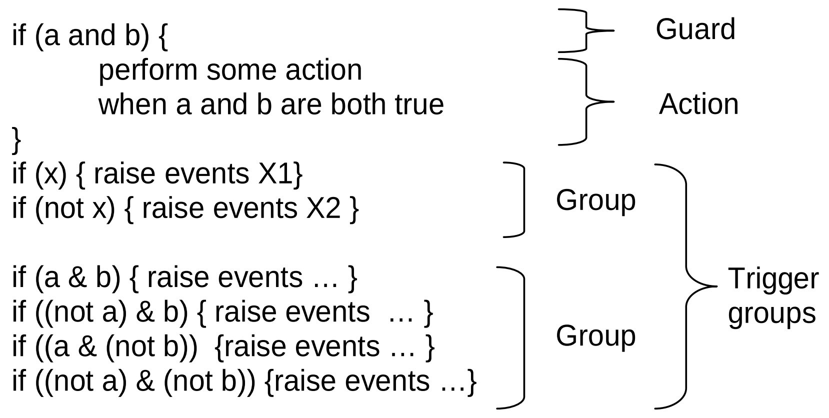


Figure 11 GAT Process Structure

The GAT model defines three types of Events: Internal, External and Deferred. Internal Events are generated and consumed within a single business process. External Events are used to communicate between peer business processes. Deferred Events are internal or external events that are generated in the normal way but only sent if certain other events have not occurred within a specified time period. Other than specifying a required time period an event to happen, no other conditions required to trigger a deferred event. All events are treated uniformly, regardless of whether they are internal or external, and whether or not they occur on what could be regarded as normal or exceptional processing paths.

Each activity represents one possible response to an event and consists of a Guard, an Action and a set of Trigger Groups. Each Trigger Group consists of a set of Trigger. Figure 12 shows an Activity composed of a Guard, an Action and two Trigger Groups, the first containing two Triggers and the second containing four Triggers.

Guards are Boolean expressions that control whether or not their corresponding action should be executed as part of the response to the event. The Action part of an activity is conventional code whose task is to handle the incoming event under the conditions specified by the related guard expression. Triggers complete the handling of an incoming event by raising any follow-on events that are needed to continue the business process.



```
if (a and b) {
        perform some action
        when a and b are both true
}
if (x) { raise events X1}
if (not x) { raise events X2 }

if (a & b) { raise events … }
if ((not a) & b) { raise events  … }
if ((a & (not b))  {raise events … }
if ((not a) & (not b)) {raise events …}
```

Figure 12 GAT Activity Structure

Triggers consist of conditions and corresponding sets of events. After the action has completed, each associated trigger condition expression is evaluated in turn and the corresponding events are raised if the condition is true. These events can be sent immediately or deferred.

The guard expressions in any one activity group are closed, meaning that the guard of exactly one activity in an activity group has to be true. That is, within a single activity group, whenever an event is received, the Boolean expression of one of the activities must be true and all the other Boolean expressions must be false. This coverage property lets us guarantee that exactly one action will be taken every time an event is received.

The trigger expressions in each trigger group are also closed. That is, in a single trigger group, exactly one trigger expression must be true and only the events corresponding to that trigger expression will be raised as a result. Activities can have multiple independent trigger groups, each corresponding to different and parallel possible courses of action that will be taken by the process.

As discussed in Chapter 2.2.2, more traditional approaches to describing business processes are based on graphs (effectively flowcharts) where the nodes are actions and the edges specify ordering constraints and the flow of control. These graph based approaches work well when used to describe processes where there are few deviations that can divert the path of execution away from its normal path. Using these approaches to specify how to handle such deviations that can occur in any state at any time can be much more difficult and complex. The event based model presented here overcomes this limitation by treating deviations uniformly with other events. This makes it easy to specify how to correct problems and return processing back to the normal path. Guards always define the correct action to take when an event occurs, taking into account the current system state. The closure properties for activities ensure that no combinations of events and system state can be omitted from the definition of a procedure. The closure property for trigger expressions ensures that the result of an execution can also not be omitted. The result is that all deviations must be handled by some activity.

4.2.2 Key Features

The GAT programming model offers several features that can help software developers avoid a number of common mistakes. These include:

- *Uniform processing:* there is no separation of the normal case from deviations. All arriving messages, whether they correspond to normal or deviation processing cases, are treated equally**.**
- *Resumption:* business processes can continue to execute normal actions after executing actions that could be regarded as corresponding to exceptional cases.
- *Access to state:* there is no hidden or implicit state. Both Abstract State and Business Process States can be freely examined by guards and updated from within actions.
- *Uniform outcome:* there is no inbuilt notion of returning to an initial state or of compensation for the business process as a whole. Individual actions may act as atomic transactions, and abort and rollback, but the whole GAT process merely continues executing actions as events arrive until the process completes in some way.
- *Coverage:* alternate actions for the same event are grouped together and guard conditions specify which of these actions should be executed. Simple closure tests on these conditions can guarantee that at least one action will be executed whenever any event is raised.
- *Protected actions:* each action has a guard that is sufficient to ensure its code runs without errors. No other concurrent process can cause this guard to become false while the action is executing. This support for isolation means that an action can rely on the truth of a property (such as the existence of a customer) that was checked during guard evaluation.
- *Response to non-occurrence of events:* the trigger mechanism normally used to raise events that drive business processes forward can also be used to define events that are raised when expected events fail to arrive in time.
- *Integration:* the raising of new events is separated from the action code that modifies state, allowing legacy code to be used as actions. This provides the same type of uncoupling of control flow and processing steps that is found in graph-based workflow models.

The significant contribution of the GAT model is in the way that it combines these features to assist developers in the construction of more robust distributed applications.

One of the most important robustness properties of a business process is being able to guarantee that the application will always terminate in one of a specified set of consistent ‘acceptable’ states. Some of these acceptable states may correspond to successful outcomes, such as goods received and paid for in full, while others represent ‘less desirable’ outcomes, such as purchase order cancelled and cancellation fee paid. These consistent outcomes are all acceptable to all the participants in the distributed application, and none of them are treated as failures or successes [30]. This property is supported by the “*Uniform outcome*” feature of the GAT model.

The robustness of a business process also depends on the completeness of the application specification. The application must be able to handle all possible events at any stage of its execution, even if they arrive when they are no longer expected or allowed by the application. This property is supported by the “*Coverage*” and “*Access to state*” features of the GAT model. The application must also be able to deal with non-arrival of expected events and this is supported by the “*Response to non-occurrence*” feature of the GAT model.

It is vital for a robust business process that each piece of code is only executed when the system is in an appropriate state. For example, code that looks up a customer’s balance must not run if there is no record for the customer present in the database. The “*Protected action*” feature of the GAT model supports this requirement. The necessary state conditions can be specified as a guard, ensuring that the code will only be invoked when its preconditions are known to be satisfied.

4.3 Payment Process in GAT Model

This section defines the payment process described in Section 4.1 in the event-based GAT programming model with the description of how features of GAT helps the developers to avoid common mistakes for each business scenario.

4.3.1 Activity Group: sendInvoice

The first activity group in this process definition handles preparing and sending an invoice to a customer. This activity group is invoked as soon as merchant reserves goods for the customer successfully. If customer details are correct an invoice is sent and it sets an event which can be invoked if the customer does not respond to the invoice, that is, customer doesn’t send a payment within the due date. If customer details are incorrect a notification message is sent to Customer Management System (CMS).

There are two activities in this activity group. The first activity sends an invoice to the customer when its guard evaluates that customer details are correct. The action part of this first activity prepares an invoice specifying the amount customer has to pay for the goods and the due date for the payment. When this action completes successfully, one trigger group generates an event to send the invoice to the customer while a second trigger group generates an overdue event which will be raised only if customer does not send payment by its due date. The second activity of the sendInvoice group sends a notification to Customer Management System when its guard evaluates that customer details are incorrect.

Group: sendInvoice	
Event	goodsReserved
Activity: invoicing	
Guard	*The customer details correct*
Action	*prepare invoice message* invoice
	set balance *to be the invoiced amount;*
	set duedate *for the payment from the customer to* 30 days*;*
	construct the overduePayment *messag*
Trigger Group	(True) *send* invoice *to the customer*
Trigger Group	(True) *set* overduePayment *to be sent back to its own process if the full payment is not received by the due date*
Activity: invalid customer	
Guard	*The customer details are not correct*
Action	*construct a* notifyCMS *message*
Trigger Group	(True) *send* notifyCMS *to Customer Management System*

One of the motivations in the design of the GAT model was to help designers avoid some common classes of errors. One particular type of error that the model can prevent is where a business process ‘hangs’ (stops making forward progress prior to termination) because it is waiting for a message that will never arrive or because the message arrived when the system was not expecting it. Another common error is ambiguity, where it is unclear which of several different pieces of code needs to be executed in a particular situation. The GAT model forces the designer to define which action is to be executed to handle every event in every situation, through the completeness properties and closure properties.

The completeness property ensures that every possible event (internal and external) must have one or more activity groups which define how that event is to be handled when it occurs. If this completeness constraint is violated, then the particular event could never be correctly handled and the application system would hang or fail if it did occur.

The Guard Closure property ensures, within an activity group, exactly one of the activities must be invoked in response to the occurrence of a single event.

This guarantees that there will be at most one action that handles the event. The Trigger Closure property ensures that exactly one of the trigger conditions must fire after the related action has completed. This guarantees that there will normally be at least one follow-on event produced as an outcome of an activity and that the overall business process will continue making progress. In the case where the business process reaches termination as a result of completing the action, there will be no associated trigger group defined as there are no more events to raise or consume. Now we check how these completeness and closure properties are defined in the activity group.

There are two activities defined in the activity group whose guards are closed: either customer details are correct or incorrect. Depending on the status of the customer details, only one activity will have guard evaluates true, but never both so only one activity will execute. This guard closure of the activity group ensures that there is always one activity that responds in the event when goods have reserved.

The trigger expressions in each trigger group are also closed. That is, in a single trigger group, exactly one trigger expression must be true and only the events corresponding to that trigger expression will be raised as a result. When the customer has the correct details, two trigger groups are defined. Each trigger group will trigger at least one trigger. However, since each trigger group defines a special trigger condition {true} which always triggers an event; both trigger groups will always trigger an event. An event to send an invoice is always generated by the first trigger group. The second trigger group also always generates a deferred event overdue which is to be raised if payment is not received by its due date. These triggers ensure that there are follow-up events as a result of executing the activity.

4.3.2 Activity Group: receivePayment

Customer responds to an invoice by sending back a payment as message. This response contains the amount of payment which the customer pays for the invoice. The amount of payment can fall in three cases: the payment amount equals to the amount owing as specified in the invoice, the payment amount is less than the owing, and finally the payment is more than the debt.

This activity group contains three activity groups corresponding to the three possible scenarios of dealing with different amount of the payment. The first activity executes if the payment equals to the amount owing. The action part of this activity records the full payment. The trigger sends out an event to other parts of merchant’s system such as Accounting notifying that full payment has been received.

The second activity executes if the payment amount is less than what is owed. The action records the received payment amount, and then calculates the residual the customer has to pay. For the premium customers, an invoice for the residual is constructed. For non-premium customers, an additional late fee is charged on top of the residual. Depending on the type of the customer, the relevant invoice is sent by the trigger.

The third activity executes if the payment amount is more than the debt. The action first records the full payment amount in the database then calculates the refund to send back to the customer. Two trigger groups are defined to generate two separate events. An event to notify to the Accounting system that full payment has been received is triggered by the first trigger group. Another trigger to send refund to the customer is generated by the second trigger group.

Group: receivePayment	
Event	Payment
Activity: process the full payment	
Guard	*Payment equals to amount owing*
Action	*Record full payment has been received* *Construct a* PaidInFull *event.*
Trigger Group	*(*true*) raise the* PaidInFull *event to notify other parts of the merchant's system that full payment has been received.*
Activity: process the under payment	
Guard	*Payment is less than amount owing*
Action	*Record payment amount* *Calculate residual amount* *Check the customer type* *If* (premium customer*)* *no late fee applies* *Construct* ResidualInvoice *If not* (premium customer*)* *late fee added to residual* *Construct* ResidualInvoiceWithLateFee
Trigger Group	*(*premium customer*) send* ResidualInvoice *to the premium customer* Not(premium customer) *send* ResidualInvoiceWithLateFee *to non premium customer*
Activity: process over payment	
Guard	*Payment is more than amount owing*
Action	*Record full payment has been received* *Calculate the refund amount* *Construct* Refund *event* *Construct a* PaidInFull *event*
Trigger Group	*(*true*) send* Refund *to the customer*
Trigger Group	*(*true*) raise the* PaidInFull *event to notify other parts of the*

	merchant's system that full payment has been received.

The key features of GAT model mentioned deal with the potential problems illustrated earlier in the Section 4.1 to help designers build robust systems. For example, if an under-payment occurs, the GAT model allows the business process to deal with it as an activity like any other (in our case by sending an invoice for the residual amount) (*Uniform Processing*). Because this case is not treated as a deviation, normal payment handling can continue once the residue has been paid (*Resumption*).

The GAT model lets designers ensure that code is never run if the system is in an inappropriate state. Consider the calculation of a refund in the activity that processes an overpayment. This code clearly can only be run when the customer's payments are less than or equal to the amount owing. In GAT, the designer places the required pre-condition (amount paid > amount owing) in the guard for the activity which calculates the refund (*Protected Action*). The ability for a guard expression to refer to database state, as well as business process state, is used here (*Access to State*). In this case, the atomicity of evaluating the guard and performing the activity means that the condition will still hold when the code is executed.

The business process designer also needs to ensure that all events will be safely and properly handled in all possible situations. In the GAT model, this goal can be achieved by ensuring that the guards associated with an event are complete and cover all possibilities. The Payment event in this example passes this test as it has activities whose guards cover all possible relationships between the amount owing and the amount paid (*Coverage*).

Existing legacy code can be easily used GAT allows the legacy code to be used as Actions (*Integration*). The feature of GAT model that separates the action from raising new events makes possible decoupling between the control flows (triggers) and processing steps (actions).

4.3.3 Activity Group: overduePayment

The next activity group ensures that an invoice is not inadvertently forgotten. If a payment is not received before the payment due date has expired, another invoice with the late fee for the overdue payment is sent to the customers. If no response is received to this overdue invoice, then the overdue invoice with additional late fee is re-sent. If no response is received after three overdue invoices have timed out, then an alarm notification is sent to a manager.

There are two activities in this activity group. The first activity sends an overdue invoice which contains an extended due date and additional late fee if there has been no response to the original invoice and fewer than 3 overdue invoices have been sent so far. The overdue invoice retry loop is implemented by sending another overdue invoice and having the activity re-raises its own deferred overdue payment timeout event. The second activity sends an alarm to a higher authority such as a manager when its guard condition determines that no response has been received and that three overdue invoices have already been sent.

Group: overduePayment	
Event	overduePayment
Activity: send reminder	
Guard	*Full payment has not been received from the customer and fewer than 3 overdue invoices sent*
Action	*Extend the* duedate *by* 7 days*;* *Apply late fee.* *increment the* sent overdueInvoice *counter;* *construct the* overdueInvoice *message;* *construct the* overduePayment *event*;
Trigger Group	(true) *send* overdueInvoice *to the customer*
Trigger Group	(true) *send* overduePayment *message to be sent back to its own process if the full payment is not received by the extended due date ;*
Activity: no full payment received	
Guard	*Full payment has not been received after sending three overdue invoices*
Action	*construct an alarm message* alarmMsg *to be sent to manager*
Trigger Group	(true) *send* alarmMsg *to be sent to manager*

Timeouts are a critical part of most business processes. Without a proper mechanism in place, business processes can wait forever for incoming messages to arrive which could attribute the business processes unable to terminate. Through the use of Deferred Events in GAT model, the developers can enforce some limit placed on how long it can wait for an event to happen, if the event has not happened till a wait times out a different path of execution may be defined to deal with the situation (*Response to non-occurrence of event*).

This activity group illustrates that GAT model not only deals with the situation where payment has been received as expected, but also it can deal with the situation where payment is delayed or not received by its due date by having an activity that deals with overdue payment.

4.3.4 Activity Group: sendReceipt

The next activity group defined is to send a receipt to a customer who has paid in full according to the amount specified in the invoice. The activity group contains just a single activity whose action we always want to execute (full paying customers always get receipts), and so its guard condition has to always evaluate to true to satisfy the closure property of guards. The action constructs a receipt and it is sent by the trigger defined by the single trigger group.

Group: sendReceipt	
Event	PaidInFull
Activity: send receipt	
Guard	*True*
Action	*Construct a* Receipt *event;*
Trigger Group	(true) *send* Receipt *to the customer*

4.3.5 Activity Group: cancellations

We illustrate three possible ways to handle cancellation requests (deviations) sent by the customer at different stage of payment process, in this activity group.

The simplest cancellation handling mechanism is implemented in the first activity when a cancellation has been requested before payment has not been received and goods have not been shipped to the customer (as evaluated by the guard condition). The action simply cancels the payment process unless cancellation fees are involved. If cancellation has been requested 7 days after order confirmation, the business scenario of the merchant imposes a cancellation fee. This is reflected in the action and an additional invoice for this late fee is sent to the customer.

The cancellation handling mechanism becomes little more complex if it is requested after payment has been received (but before goods are in transit). The second activity implements the cancellation handling mechanism when its guard condition evaluates that payment has been received but before goods are in transit. The merchant calculates the refund amount, that is, if the cancellation has been requested before 7 days from order confirmation all payment is refund; otherwise the merchant deduces the cancellation fee from the refund. There are two trigger groups defined in this activity: one trigger group sends the refund to the customer while the other trigger group sends a cancellation confirmation message.

We assume the merchant simply rejects the cancellation request if its guard condition evaluates that goods are already in transit.

Group: cancellations	
Event	cancelRequest
Activity: payment not received	
Guard	*Payment not received and goods not in transit*

Action	*set* cancelled *to be* True;
	set cancelFee *to be* True *if the cancellation occurs after 7 days of order confirmation;*
	*if (*cancelFee*) constructs a message* CanceallationInvoice*;*
	construct an event CancelConfirm*;*

Trigger Group	(cancelFee) *send* CanceallationInvoice *to the customer* Not(cancelFee) *// do nothing*
Trigger Group	(true) *send* cancelConfirm *to the customer*
Activity: payment received	
Guard	*Payment received and goods not in transit*
Action	*Set* canclled *to be* True*;* *Calculate* Refund *amount;*
Trigger Group	(true) *send* Refund *to the customer*
Trigger Group	(true) *send* cancelConfirm *to the customer*
Activity: goods in Transit	
Guard	*Goods in transit (payment received or not)*
Action	*Construct a cancellation rejection message* rejectCancel
Trigger Group	*(*true*) send the* rejectCancel *to send to the customer*

One of the innovations of GAT model is to have uniform processing between normal business cases and deviations. Unlike the traditional approach, deviations are not treated as special cases which cause the whole system to abort and going back to its original state. A more effective and desired way of handling deviations is to fix the deviations as they occur using the same flexibility of normal business scenarios which consider the current state of the system and then continue on the normal path (*Uniform Processing*).

One of the key features of GAT is to allow accessing different types of states, both Abstract State (such as payment recorded in the database) and the Business Process State (such as progress of payment process and the shipping process) which gives better flexibility for the developers to discover more accurate current state of the running system (*Access to State*). This allows the developers to design and implement more desired behaviours for the deviations when they occur.

The rejection of cancellation requests has not been well handled by the standard deviation approaches. As we discussed in Chapter 3.6, existing systems support can throw a fault, perhaps runs compensators, and terminate the whole process. However, as we see in the above examples, termination of a whole process isn't always a desired behaviour or may not be possible if additional activities are created such as cancellation fees. In GAT, the merchant payment process can simply reject the cancellation and continue the next business steps (*Resumption*).

4.4 Experiment with GAT

So far, we presented a GAT expression for only the payment process part of the e-procurement. This was enough to show how features of GAT model can help the designers minimize common mistake and errors in designing service-based applications. In fact, we have written the whole e-procurement scenario in GAT. In this section, we report on this our experience, to illustrate the value of GAT for a designer of business processes.

As described in Chapter 3.1.1, the e-procurement scenario consists the interactions among three partner systems namely customer, merchant, and shipper. For each partner system, we define a set of business processes, each of which contains a set of Activity Groups and events. For example, the customer system contains four business processes dealing with the different aspects of business: Quote Process, Purchase Order Process, Payment Process, and Delivery Acknowledgement Process. Similarly, the merchant system contains five business processes: Quote Processing Process, Purchase Processing Process, Payment Processing Process, Delivery Processing Process, and Cancellation Process. Lastly, the shipper system contains a single business process: Shipping.

Here is a summary of statistics when all business processes in the e-procurement were written in GAT model.

- For the customer system, four business processes were defined having 24 Activity Groups and 24. 29 Activities were defined among all Activity Groups, each of which contains a guard and an action and a set of trigger groups. The total number of triggers produced by all Activities was 27. Out of 29 Activities, 24 Activities define what we consider normal path while 5 Activities define the cases where execution deviates from the normal path.
- The merchant system is defined by 29 Activity Groups with 64 enclosed Activities. 63 triggers were produced by all Activities. 29 Activities define the normal path and 35 Activities define deviational cases.
- The shipper system had 6 Activity Groups. 8 Activities and 9 triggers were defined in all. 6 Activities define a normal path and 2 Activities define deviations.

Note the dramatic increase in the number of Activities defined for deviations for the merchant system in proportion to the number of Activity Groups. In other partner systems, such as the customer and the shipper, the number of Activities defined is only slightly higher than the number of Activity Groups reflecting a low number of Activities for deviation handling. But in the merchant system, the number of Activities is about twice the number of Activity Groups. This is due to the fact that the merchant system is a lot more complex and sophisticated than those of the customer and the shipper as it takes an active role of processing requests (from the customer and the shipper). Thus, the merchant system defines a high number of Activities to deal with various deviational situations, such as invalid purchase order, when a customer is not registered, insufficient goods, if payment gets delayed, when goods are out of stock, and so on. In contrast, the customer and the shipper define only a few deviations as the complexity is relatively low compared to the merchant system. Even the merchant, though, is far from the typical situation with graph-based models, where deviation-handling is overwhelmingly more voluminous than the normal path.

The number of triggers is proportional to the number of Activities. This doesn't mean that one activity always fires one trigger. Among almost 100 Activities between all partner systems, there were about 15 Activities which create more than one trigger. However, there were also about 10 Activities which do not produce any triggers (mainly because they were the last Activities in a process). This tends to even out the difference between the number of Activities and the triggers.

One of the difficulties we faced in writing a GAT model for the e-procurement case study was to think about the cases where activities deviate from the normal path. Without an intimate knowledge of the system, it was difficult to devise all possibilities for deviational cases. However, the difficulty seems reduced once we focused on state. Once we figured out which state to pay attention to, we could easily list all the possible values for each state. We then simply wrote Activities for each possible value.

4.5 Evaluation Compared to Other Models

There is a huge literature on notations for describing business processes, workflows, or long-running activities. Many of these notations have been implemented in systems offering workflow-management or business process support [29]. The dominant approach in commercial systems presents a graph controlling the flow of control between steps, or as a simplification of this, a block-structured language with fork and join constructs as well as sequential flow and conditional branching. This is also the model used in standards such as BPEL [7], and WSCI [87]. A smaller group of research papers and prototypes however have considered an event-based approach similar to the one in our GAT model.

The initial impetus for event-based control flow came from the flurry of research in active databases [90] where triggers are used to respond to situations whenever they occur. The first paper to adopt this idea for managing control flow in a long-running activity was Dayal et al. [15], where the ECA (Event-Condition-Action) notation was proposed. The idea has proved especially valuable for building prototypes of distributed workflow execution engines such as C^{2}offein [50], IRules [81], EVE [81] and WIDE [12]. Semantics for the ECA model are proposed by Geppert et al. [30]. Key features of our GAT model not found in these ECA systems include the grouping of actions with closure property on the conditions (to ensure coverage), the capacity to raise events which will occur later but only if some appropriate condition does not happen in the meantime (in order to provide time-outs), the concept of final events, and the idea of uniform outcome.

One paper does suggest some grouping and coverage condition: Knolmayer et al. [49] have proposed an ECAA (Event-Condition-Action-AlternateAction) model, which is in essence an activity group of exactly two actions with conditions that are complementary to one another. The paper mentions the possibility of larger numbers of actions being grouped, but gives no details.

Several other proposals have used ECA rules for dealing with exceptional conditions, while graph models are used for normal case processing (these are often called the “blue sky” paths). These proposals focus especially on the need to adapt and vary the way exceptions are handled, as the system evolves. Casati et al. have defined a language (Chimera-Exc) and implemented a system FAR [11], Hagen and Alonso built OPERA [37], and Muller et al describe AGENT WORK [64]. Because these systems do not offer uniform handling, and they terminate the normal case when the exception is raised, thus they have difficulties in all the situations we described above where resumption and access to state are needed to decide the proper response to a cancellation or other exception.

4.6 Summary

We have proposed a new model and notation for expressing interacting business processes. It has a number of key features which together help the designer avoid many common sources of errors, including inconsistent outcomes. Unlike most existing business process modelling languages used for expressing business processes, we do not separate the normal case from exceptional activity, nor do we treat exceptional events as deviations that require special handling mechanisms such as compensation.

Defining a normal behaviour in a process is easy and straightforward in the standard graph-based languages. But the same can not be said for the various deviations. The main reason is that the standard approach uses fault handlers or compensation handlers to deal with most deviations. Such approaches can handle certain class of deviations, but not all possible deviations. Moreover, handling deviations leads to the termination of a process. Describing all possible execution paths including deviations using just fault and compensation handlers becomes either clumsy or impractical.

The event-based GAT model presented in this chapter overcomes the limitation of existing business process modelling languages by making no distinction between the exception and normal processing. Guards always define the correct action to take when an event occurs, taking into account of the current system state. The closure properties for activities ensure that no combinations of events and system state can be omitted from the definition of a process. The closure property for trigger expressions ensures that the result of an execution can also not be omitted. The result is that the specification is complete that is every possible event (internal and external) must have one or more activity groups which define how that event is to be handled when it occurs.

We have carried out an extensive case study based on an e-procurement application. In this chapter, we extract the payment process from the e-procurement application and illustrate how a lot of common mistakes and errors from the developers in the service-based applications can be minimized if GAT model is applied. This work shows that our model enables programmers to write the individual services participating in a distributed application in such a way that they deliver consistent outcomes despite various deviations. In the next chapter we explore the possibility of implementing our GAT model using today's proven technology, to verify its usability and practicability.

Chapter 5

Design Principles in Building a Business Process System based on GAT Model

In the previous chapter, we proposed a new way to describe a long running business process. Our GAT approach helps the designer achieve a reliable and robust system. In particular, GAT makes it easier for the designer to describe sensible processing for all possible deviations from the simplest normal path.

This chapter discusses how one can implement service-based applications that are defined according to the GAT model. We propose implementation techniques for key features of GAT. These include implementing control flow based on the evaluation of guards, the management and distribution of events, and enforcing atomicity across the evaluation of guards and the execution of the corresponding activities. We have built a prototype system following this approach for a particular example business process. Our approach uses available technologies, such as C# and the functionalities provided by .NET framework. We also discuss how to build a generic GAT engine which can produce executable business processes from various business scenarios expressed in GAT model.

5.1 Case study

We illustrate our implementation proposal on the payment process present in Chapter 4. To recap from the previous chapter, the receivePayment Activity Group represents that part of the e-procurement process where the merchant processes payments made by customer. The merchant's process receives incoming payment messages and checks the amount being paid against the amount owed. There are then three possible business scenarios that can be triggered in response:

- Full payment has been received: the payment received is equal to the amount owed. The rest of the merchant's application is notified that this phase of the procurement cycle has completed successfully.

- Under-payment has been received: the payment received is less than the amount owed. In this case we calculate the amount still owing and send an additional invoice to the customer. If this is for a premium customer, send a reminder; otherwise send an invoice for the remaining amount plus penalty.
- Over-payment has been received: the payment received is for more than the amount owed. In this case, there are two further actions to be taken, each expressed as a separate trigger group. One is to calculate the over-payment and refund it. The other action notifies the rest of the merchant's application that the customer has paid in full.

This has been shown as Activity Group "receivePayment using GAT programming model " in the Figure 10 taken from the previous chapter.

Group: receivePayment	
Event	Payment
Activity: process the full payment	
Guard	*Payment equals to amount owing*
Action	*Record full payment has been received*
	Construct a PaidInFull *event.*
Trigger Group	*(*true*) raise the* PaidInFull *event to notify other parts of the merchant's system that full payment has been received.*
Activity: process the under payment	
Guard	*Payment is less than amount owing*
Action	*Record payment amount*
	Calculate residual amount
	Check the customer type
	If (premium customer*)*
	no late fee applies
	Construct Reminder
	If not (premium customer*)*
	late fee added to remaining
	Construct InvoiceWithPanalty
Trigger Group	*(*premium customer*) send* Reminder *to the premium customer*
	Not(premium customer) *send* InvoiceWithPanalty *to non premium customer*
Activity: process over payment	
Guard	*Payment is more than amount owing*
Action	*Record full payment has been received*
	Calculate the refund amount
	Construct Refund *event*
	Construct a PaidInFull *event*
Trigger Group	*(*true*) send* Refund *to the customer*
Trigger Group	*(*true*) raise the* PaidInFull *event to notify other parts of*

	the merchant's system that full payment has been received.

Figure 13 Activity Group: receivePayment in GAT

5.2 GAT Design Consideration

We identify a number of key aspects of the GAT model that have an impact on the design of the GAT prototype system in this section, and then discuss their implementation in more detail in the following section.

5.2.1 Control Flow of Business Activities

In conventional graph-based or block-structured workflow languages, control flow between stages of the workflow is defined in the syntax or its graphical representation. Recall that control flow in GAT is determined dynamically as events are raised and corresponding activities are invoked. A key issue for a GAT prototype system is how to manage the control flow: how to pick the appropriate action to perform in response to an event. Our current implementation makes use of the .NET Event mechanism which allows code to subscribe to events and be invoked whenever these events are raised, but does not support the concept of conditional guards that are used in GAT to choose the one appropriate action from an activity group. In Section 5.4.1, we describe the way each GAT activity group is represented in our prototype by a method which contains code to successively evaluate the guards of the contained actions.

A further complexity for our implementation comes from our decision to allow guard expressions to refer to any aspect of state. GAT guards can refer to both Business Process State (which has variables reflecting what actions have occurred) and Abstract State (a computer-based representation of the domain, typically stored in databases or in variables used by the code of particular actions). In contrast, conventional workflow languages use only Business Process State (often implicit) when deciding control flow, and allow references to Abstract State only from inside the individual actions.

The GAT model defines in detail how execution must take place when multiple activity groups are dealing with the same event. This is especially significant because the guard conditions may refer to state that can be changed inside actions. The GAT model defines that when an event occurs, all the activity groups related to this event will be selected; one of these activity groups will be chosen and its guards evaluated to determine which of its activities is to be executed. This activity is then executed, and this may (through the triggers) raise further events. Another of the originally selected activity groups is then processed in the same way, until all selected activity groups have been considered. After this, if additional events have been raised, each is processed in the same way in turn. The GAT prototype system implements this required behaviour using multiple subscribers to .NET events.

5.2.2 Atomicity/Isolation Issues

In the GAT model, it is essential that the choice of which action to perform from an activity group (by evaluation of guards), the execution of the chosen action, and the evaluation of its trigger conditions and the raising of any further events, must all form a single isolated unit of execution. Without such isolation, concurrently running business activities could alter some critical shared state referenced by the activity group and this could result in the process failing or terminating in an inconsistent state. For example, the guard on the action to deliver goods may have been evaluated and shown that Alice has sufficient funds to pay for the goods, allowing the action to proceed safely. Without proper protection on Alice's account, it is possible that a concurrently running action could use some of Alice's funds leaving Alice no longer able to make payment for the goods that are being delivered. In this case, the balance in Alice's account has been changed between the time it was checked in a guard and when the funds were needed in the action. Including both the evaluation of guards and the execution of the chosen action in a single isolated unit of execution can prevent such problems.

Our GAT prototype implementation uses the transaction mechanisms provided in .NET 2.0 to provide this required level of isolation. Each activity group, including the evaluation of its guards, the execution of the chosen action, and the evaluation of its trigger condition and raising further events, is constructed as a single transaction. The isolation provided by transactions guarantees that the state used by a running activity group cannot be altered by any concurrently executing business processes.

5.2.3 Management and Distribution of Events

As noted above, events play a vital role in driving the GAT programming model. While the underlying .NET event publish/subscribe mechanism can be used to implement the basic control flow mechanism, the GAT event concept is somewhat more complex and the prototype system needs to have special mechanisms for the various types of GAT events. GAT model defines three different types of events that cover different aspects of communication in different circumstances.

- Internal Events are used to control flow among Activity Groups within the same business process. In our engine implementation, these are mapped directly to .NET Events. Section 5.4.2 discusses the various classes and methods which we define to make this work.
- In the GAT model, External Events control the communication between interacting peer business processes. In a service-based model, communication between parties is handled solely by exchanging

messages, using technologies such as SOAP or .NET Remoting. The GAT prototype needs to convert between messages and events and we did this by representing each external GAT event twice in our prototype: both as a message and as an internal .NET event. In Section 5.4.3 we show how our implementation can convert between these two forms as external events are received or raised.

- GAT Deferred Events are used to provide a way of initiating activities whenever other events and activities do not occur before the expiration of a deadline. For example, when the payment has not been received by its due date (so corrective action, such as sending a reminder or alarming a human operator are needed), we represent this as a deferred event. Our prototype implements deferred events through the combination of .NET events (which are raised and processed immediately) and .NET timers.

5.3 Architecture of GAT Prototype System

We have implemented a prototype system for the whole of the e-procurement case study, which was written in GAT model. The prototype implementation follows the design approaches we mentioned above. The overall architecture of the system is pictured in Figure 14 followed by the descriptions of major components of the system.

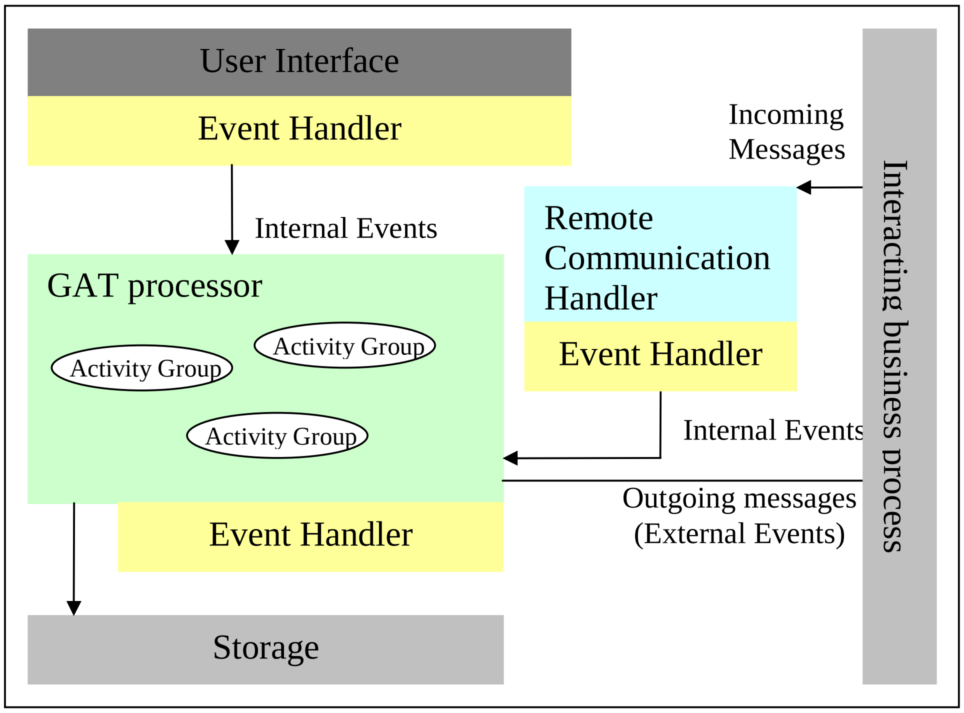


Figure 14 Architecture of GAT Prototype System

5.3.1 User Interface

User interface is a front end presentation layer to capture data from human operators (such as purchase order form, payment form) or to render a collection of data received from the processes. Each business party uses this layer to monitor the business activities such as what messages have been received from, or sent to, other services. We use Windows Form to capture the user requirements as well as to display the progress of business activities.

5.3.2 Event Handler

The GAT model is based on an event driven approach where communication among various activities are done by sending/receiving events. To realize this fundamental principle of GAT model, the prototype system contains an Event Handler component which controls all aspects of events: it generates events from various event sources, creates required event arguments for each event to carry, and it notifies to various activities about the events.

For this particular prototype system which implements the e-procurement scenario, there are three possible event sources: User Interface responding to actions by the human operators, Activity Groups as managed by GAT Processor and external messages received by Remote Communication Handler.

- Human operators creating messages, such as a purchase order, via User Interface, are one event source. For example, when customer enters the details of a purchase order using purchase order form and clicks the submit button, an event PO is generated with the details of purchase order as an event argument.
- Activities handled by GAT processors are another event source to create events. For example after receiving full payment from the customer, the Activity to process the full payment executes generating an event "PaidInFull" which is raised to notify other parts of system (such as Accounting System) that full payment has been received.
- External messages received from interacting parties (such as customer or shipper) are also an event source where events are generated. Details of handling external messages are described in the Section 5.4.3.

In our prototype, different event sources generate different event types. Events generated by User Interface and GAT processors are mostly Internal Events and Deferred Events as their likely recipients are located inside the single business process. On the other hand, External Events are generated from messages received from externally interacting business processes.

The major role of the Event Handler is three fold. It first receives messages from various event sources. Then it generates platform specific events. Our current implementation generates .NET Events which are understood by GAT Processors written in C# language. Once events are generated, they are raised immediately thus these events can be notified to event subscribers as soon as they are generated. In our implementation, Activity Groups act as event subscribers to receive .NET Events.

5.3.3 GAT Processor

A GAT Processor is a core part of our GAT prototype implementation. It is where the business logic is defined. The GAT Processor includes code for business process in C# with a structure corresponding to a set of activity groups and activities defined using GAT model. For example, the payment process from Chapter 4.1 can be written as a GAT processor which contains five Activity Groups “sendInvoice”, “receivePayment”, “sendReceipt”, “overduePayment” and “cancelRequest” each enclosing the respective Activities.

Activity Groups in the GAT Processor are event subscribers which consume events raised by the Event Handler. For example, the Activity Group “sendInvoice” subscribes to an event Invoice. When an event Invoice is generated and raised by Event Handler via event source GUI User Interface, the event Invoice is notified to Activity Group “sendInvoice” as its subscription matches to the event being raised.

Once the subscribed Activity Groups are chosen, GAT processor picks up an Activity from each Activity Group and executes each Activity in turn. More detailed description of how we define an Activity Group and how we execute Activities are given in Section 5.4.

Each Activity picked up within an Activity Group generates a new set of events. New events can be of any event types, Internal, External or Deferred. If new events need to be sent within the same business party, the new events are made as Internal Events or Deferred Events. If new events need to be sent across interacting business parties, the new events are sent as outgoing messages. Outgoing messages are .NET Remoting objects which are understood by a remoting server implemented at each party.

5.3.4 Remote Communication Handler

Each business party implements a single Remote Communication Handler to deal with incoming messages sent from interacting business parties. Though WS-Events [89] provides a standardized way to generate events in a Web Services environment, at the time when this prototype implementation was being carried out, the standard was still at an early stage of development and it was even hard for us to obtain a reference implementation. Thus we did not incorporate this in our design.

Our implementation approach is to use .NET remoting mechanism in conjunction with Event Handler. If there are External Events required to be sent to external parties, these external events are converted to .NET remoting objects and sent to the external parties as serialized objects. Each party implements Remote Communication Handler which continuously listens to incoming .NET remoting objects. Once an incoming .NET remoting object arrives, Remote Communication Handler de-serializes the remoting object and works with Event Handler to convert it to a corresponding Internal Event. Then in turn, this Internal Event is notified and subsequently consumed by designated event subscribers (Activity Groups).

We explain this process more concretely with an example from e-procurement case study. Suppose the customer sends a purchase order to the merchant. The purchase order is sent as .NET remote object from the customer. The purchase order, now as .NET remote object, is received by Remote Communication Handler implemented in the merchant system. The merchant's Remote Communication Handler now works with Event Handler to generate ePO .NET Event. The ePO Event is then received by the Activity Group 'processPurchaseOrder' which records the details of purchase order and examines whether the purchase order can be accepted or rejected.

5.3.5 Data Storage

Each business party stores Abstract State which might represent many mission critical aspects of business, such as a purchase order, invoice, or payment, into a persistent storage such as a database. This Abstract State stored in the merchant database is then accessed and manipulated by other Activity Groups that are running in the same trust boundary. For example, merchant service stores the details of invoice (Abstract State) as soon as an invoice is sent to the merchant. When payment arrives from the customer, the merchant retrieves the invoice from the merchant database to evaluate guard conditions to decide which activity must run. If payment is less than the amount owning stated in the invoice, the merchant service runs an activity that deals with under payment, and so on.

For our prototype implementation, we use Microsoft SQL 2005 database to store abstract state. Activity Groups use ADO.NET to access and manipulate the data inside SQL database.

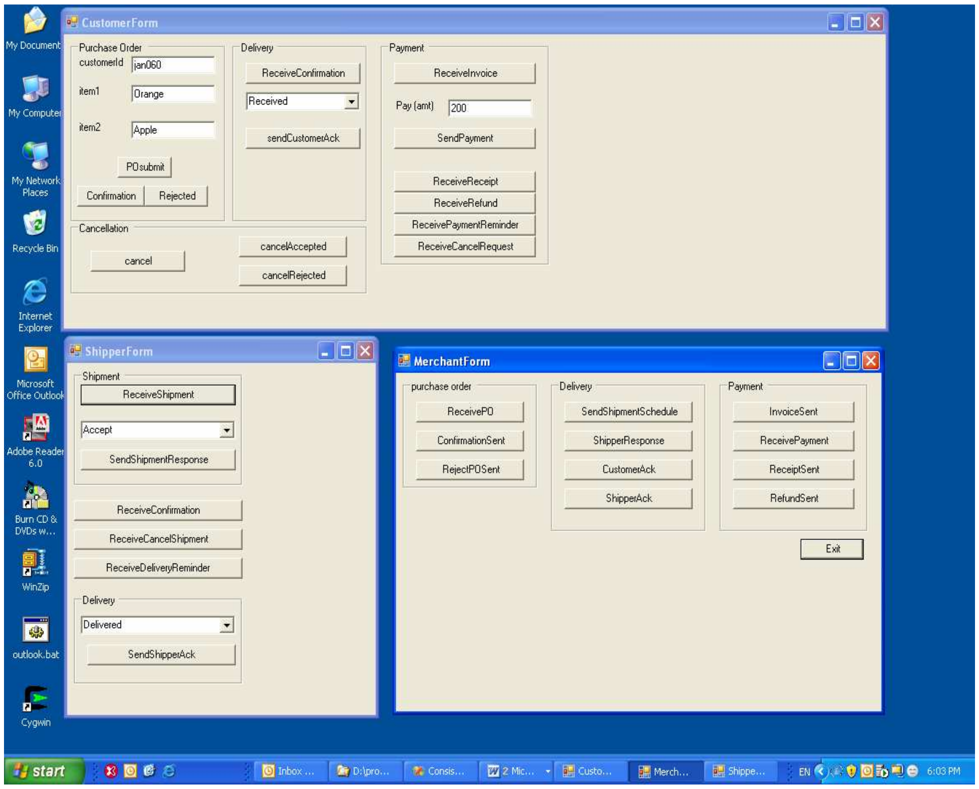
The current prototype implementation does not, however, store Business Process State such as *invoice sent* or *payment received*, in the persistent storage such as database. Rather Business Process State is stored simply in the memory for each instance of business process and accessed and manipulated by other Activity Groups in the same manner as Abstract State. When a cancellation request is received, its processing is different depending on which variables are true. If *payment received* is false and *goods in transit* is false, and *confirmation send* is true the cancellation is quite simple. Other values of these variables may mean the cancellation request has to be rejected.

5.3.6 Running Business Systems

Following the architecture described the above a prototype system has been built which consists of three interacting business parties in our e-procurement case study (namely, customer, merchant and shipper).

The prototype system has been built on .NET framework v2.0 using C# language on Windows XP platform with SP2. There is a single GUI user interface which interacts with human operators at each business system. Also each business system implements one module of Remote Communication Handler to deal with incoming messages from interacting parties. Event Handler sits along with GUI, Remote Communication Handler to convert messages to events. These events are received and consumed by Activity Groups controlled by GAT processors. GAT processors also generate events as results of executing Activities.

The snapshot of all three running business systems in our prototype is shown below in Figure 15.



customerId jan060
item1 Orange
item2 Apple
Received
Pay (amt) 200
Accept
Delivered

Figure 15 Snapshot of the Running Prototype System

5.3.7 Performance

We have run various performance tests to observe the feasibility and practicability of business processes implemented following our GAT model approach. In this section we illustrate the performance results we obtained on DELL Optiplex GX270SMT with a 3 GHz Intel Pentium IV processor and 1GByte RAM.

First we measured the latency of the overall prototype system to evaluate how long it takes from when a client sends a purchase order request to the server until the client receives a final response from the server. In this measurement, we exclude time spent inside physical processes (such as shipping goods) or waiting for human interaction (such as approving the choice of shipper). For the purpose, we used a high resolution counter QueryPerformanceCounter [40] with the counter frequency of 1/2999272000 seconds, which returns system clock counters approximately every 10 microseconds (us) on typical Intel Pentium IV processor. Such high performance counter gives 1000 times better accuracy than typical counters which only return the system clock counters every 10 milliseconds (ms). The result of measuring the latency using the high resolution counter for our prototype system was on average 23ms.

Figure 16 shows how overall latency of 23ms is spent at different stages of business activities in our prototype system.

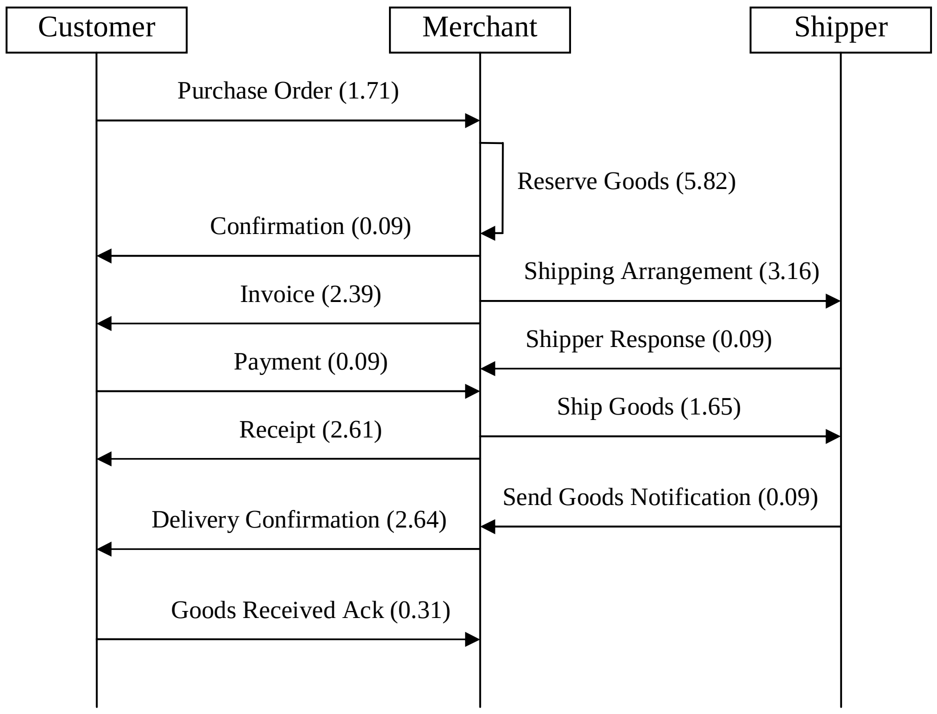


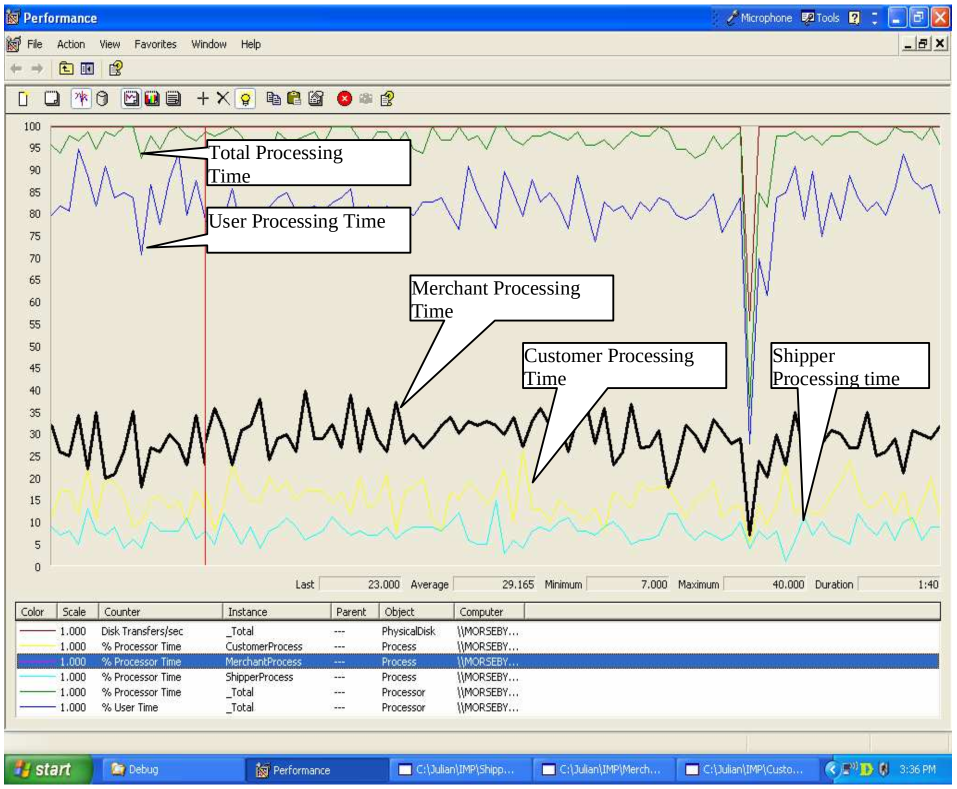
Figure 16 Performance at Each Business Activity in Milliseconds (ms)

The measurement of each stage is done in terms of computing the time between the following sequences of steps: An event is received. Activity Group evaluates guard conditions and finds an Activity which satisfies the guard. Action within the chosen Activity is then executed and following event(s) are triggered. For example, the time measured in Receipt (2.61ms) is a result of the Activity Group 'receivePayment' being notified of an event payment after payment message is received from the customer. The Activity Group 'receivePayment' evaluates its guard conditions by fetching the invoice amount from the SQL database to compare with the payment amount. If payment amount equals the invoice amount, the Action of the Activity to process the full payment is executed which updates the payment amount in the database and constructs a "PaidInFull" event, which is subsequently raised.

As expected, any business activities with database transactions take longer than those simply processing events by Activity Groups. The business activities with database transactions such as Purchase Order (1.71), Reserve goods (5.82), Shipping Arrangement (3.16), Invoice (2.39), Ship Goods (1.65), Receipt (2.61), and Delivery Confirmation (2.64) all take longer processing time than business activities which simply process events such as Confirmation (0.09), Shipper Response (0.09), Payment (0.09), Send Goods Notification (0.09), and Goods Received Ack (0.31). Also as the number of database transactions increase, the processing time proportionally increases. For example, the activity Reserve Goods (5.82) contains the highest number of database transactions with 7 selects and updates in contrast Shipping Arrangement (3.16), Invoice (2.39), Receipt (2.61), and Delivery Confirmation (2.64) contains on average 2-3 updates. Purchase Order (1.71) and Ship Goods (1.65) have one update each having the least processing time among business activities with database transaction. Note that the business activities with no database transaction all have relatively smaller number of processing time ranging from 0.09ms to 0.31ms. This clearly indicates that database access and manipulation remains the major factor in the overall processing time.

Note that there is approximately 2.5ms of overhead when sending remote messages across different components. An example is sending of a purchase order as remote object. This overhead is not measured in our system as network latency will greatly vary depending on the network topology and location of interacting components. We consider this case beyond the scope of the simple performance study of GAT.

We look at the processing time of each component of our prototype system in Figure 17.



Last 23.000 Average 29.165 Minimum 7.000 Maximum 40.000 Duration 1:40

Color	Scale	Counter	Instance	Parent	Object	Computer
——	1.000	Disk Transfers/sec	_Total	---	PhysicalDisk	\\MORSEBY...
——	1.000	% Processor Time	CustomerProcess	---	Process	\\MORSEBY...
——	1.000	% Processor Time	MerchantProcess	---	Process	\\MORSEBY...
——	1.000	% Processor Time	ShipperProcess	---	Process	\\MORSEBY...
——	1.000	% Processor Time	_Total	---	Processor	\\MORSEBY...
——	1.000	% User Time	_Total	---	Processor	\\MORSEBY...

Figure 17 Snapshot of the Performance Monitor

As can be seen, the full processing time indicate by the green line includes the processing time used by the system itself and the users of the system. Among the overall processing time, 85% is user processing time, indicated by the blue line. Among the user time, about 25% is consumed by the merchant process (the black line) as it contains the most processing intensive business activities such as goods (5.82), Shipping Arrangement (3.16), Invoice (2.39), Receipt (2.61), and Delivery Confirmation (2.64). Approximately 10% processing time is used by the customer process (the yellow line) followed by the smallest processing time 5% used by the shipper process (the skyblue line) which contains a relatively small number of business activities and no database updates.

5.4 Implementation of GAT model

In this section, we discuss details of the implementation of each of the key features of GAT that were covered in Section 5. 2. We first explain how each business process, represented in GAT as a set of activity groups, can be expressed as a collection of C# classes and their methods. We then discuss the event-handling and message-passing infrastructure which provides the equivalent of conventional control flow mechanisms for GAT.

5.4.1 Defining Activity Group and Execution of Activities

A business process is defined in the GAT model as a set of activity groups, where each activity group consists of an event and a set of related activities to handle that event in different situations.

Each Activity Group is translated into a single C# method within a class that corresponds to the business process. Each activity within the group is represented as one if-then-else block inside this method. An activity group is run as a transaction by using System.Transactions mechanisms. The Guard condition of the Activity is, naturally, the Boolean condition that is tested by the 'if' clause. The action part of each Activity in GAT typically contains the piece of code that fulfils a business step, such as checking customer validity. For simplicity, our current prototype system implements C# code for the Action, so no conversion is needed to produce the body of the 'then' clause. Each execution of an activity is followed by further 'if' statements that represent the triggers which generate new events.

The result of this translation process is shown in the following pseudo code corresponding to the example from Figure 10. There are three different activities in this pseudo code for handling payments as discussed previously, allowing the application to respond appropriately to the payment amount received.

```
Method receive payment
begin transaction

  // activity group: process the full payment
  if payment amount equals to the amount owing
    update the payment
    generate an event paidInFull

  // activity group: process the under payment
  if payment amount is less than the amount owing
    update the payment and calculates remaining
    if premium customer, generate an event Reminder
    if non premium customer, generate an event
      RamainingInvoiceWithPenalty

  // activity group: process over payment
  if payment amount is more than the amount owing
    update the payment and calculates refund
    generate an event paidInFull
    generate an event Refund

commit/rollback transaction
```

The Activity Group ‘receivePayment’ gets executed when an event Payment arrives from the customer. There are three different Activities in this example for handling payments as discussed previously, allowing the application to respond appropriately to the payment amount received. The GAT specification selects from amongst these alternatives using Guard expressions, that are translated into the conditions for each of the top-level ‘if’ expressions. Once an activity is chosen, the prototype runs its Action code and then falls into the series of ‘if’ statements that represent this activity’s trigger expressions that will send out any follow-on events. The following code excerpt shows the detailed implementation illustrated in the above pseudo code.

```
//Activity Group: ReceivePayment
public void ReceivePayment (PaymentEventArgs e)
{

    // Make ActivityGroup transactional so the evaluation
    // of the guard and execution of corresponding
    // activity is run as a transaction
    using(TransactionScope scope=new TransactionScope())
    {
        //retreive the invoice to compare the payment
        // with owing
        Invoice inv = fetchInvoice(e.payment.Id);

        double owing = invoice.amount ;

        //Activity : processFullPayment

        //Guard: payment amount equals to the amount
        //owing
        if (invoice == e.payment.Paid)
        {
            //ACTION: record the payment
            UpdatePayment (e.payment);

             //TRIGGER GROUP:
             // A single Trigger is defined for an event
             // PaidINFull to notify Account System that
             // full payment has been received.
            argsPaidInFull =
                 new paidInFullEventArgs(invoice);
            OnEPaidInFull(argsPaidInFull);
        }

        // Activity : processUnderPayment
        // GUARD : Payment received is less than owing
        if (invoice > e.payment.Paid)
        {
             //ACTION: record the payment
             //
             // and construct events send
             // to different types of customers.
             UpdatePayment (e.payment);
             int remaining = invoice – e.payment.Paid

             // construct a message reminder to send to
             // premium customers
             If (e.payment.customer == “PREMIUM”)
             {
                 Reminder = constructReminder(remaining);
             }

             // construct a remaing invoice with penalty
            // send to non_premium customers
             Else If (e.payment.customer == “NON_PREMIUM”)
             {
                 RemainingInvoiceWithPenalty =
                  constructRemainingInvoiceWithPenalty
                 (remaining)
             }

            //TRIGGER GROUPS:
            // A single trigger group contains two
            // triggers, only one of them is trigger.
            // If customer is a premium customer a
            // reminder is sent. If customer is a
            // non_premium customer an invoice with
            // remaining owing + penalty is sent
             If (e.payment.customer == “PREMIUM”)
             {
                  Cust.receiveReminder(Reminder);
             }
             Else If (e.payment.customer == “NON_PREMIUM”)
             {
                 Cust.receiveRIWP(
                     RemainingInvoiceWithPenalty);
             }
     }

     // Activity : processOverPayment
     //GUARD : payment is more than amount owing
     if (invoice < e.payment.Paid)
     {
         //ACTION: record the payment and construct
         // Refund
         UpdatePayment (e.payment);
          Int refundamt = invoice – e.payment.Paid

         // Construct Refund to send to the customer
         refund = constructRefund(refundamt);

         //TRIGGER GROUPS:
         // Two Trigger Groups are defined. Each Trigger
         //Group triggers a single event. One trigger is
         // is trigger to send a refund to the customer.
         // Another trigger notifies Accounting System
         // that full payment has been received.
        Cust.receiveRefund(Refund);
```

```
        argsPaidInFull =
            new PaidInFullEventArgs(invoice);
        OnEPaidInFull(argsPaidInFull);
    }

    scope.Complete();
    }
}
```

5.4.2 Supporting the GAT Event Concept

There are three types of events in the GAT model: Internal Events, External Events, and Deferred Events. We describe significant differences in the way they are handled.

5.4.2.1 Internal Events

Internal events are used to communicate between activities within a single business process. New Events are published by Trigger Groups after the execution of Action code. These Events will be consumed by any Activity Groups whose subscriptions match its event type. Both the GAT event concept and its .NET implementation follow the publish/subscribe model.

There are four classes involved in the implementation of event communication. The class that raises the event is called the *event sender*. The class that consumes the event and responds to it is called the *event receiver*. The event sender does not know which object will receive the events it raises. An intermediary class called a *delegate* connects the event sender and the event receiver, and an additional class is used as an *event argument* to pass data between the event sender and the event receiver. In our GAT implementation, the Triggers and Remote Communication Handler which raise events are mapped into event senders in .NET, while Activity Groups are mapped into event receivers.

The next code excerpt illustrates payment process which creates a new event "PaidInFull" after payment sent by the customer is enough to cover the amount owing. The payment process is written as following way in order to create .NET events.

- Defines an *event argument.*
- Defines an event *delegate* which can connect from an *event sender* to *event receiver.*
- Define a method which maps the event and the *event argument.*
- Defines a method named On*Event_Name* which maps the event (with argument) and the event delegate.

```
// Defining an event argument class for the event
// PaidInFull
```

```
public class PaidInFullEventArgs : EventArgs
{
    private Invoice inv;
    public PaidInFullEventArgs(Invoice inv)
      {this.inv = inv;}
     public Invoice paidInFull {get {return inv;}}
}

// Defining event delegate for the event ePaidInFull
public delegate void PaidInFullEventHandler(
    PaidInFullEventArgs e);
}

public class PaymentProcess
{

    // Use the event delegate class created for
    // ePaidInFull
    public event PaidInFullEventHandler ePaidInFull;

    // Use the event argument class created for
    // ePaidInFull
    public PaidInFullEventArgs argsPaidInFull;

     // The method receivePayment raises theh event
     // paidInFull
    public void ReceivePayment (PaymentEventArgs e)
    {

         if (invoice == e.payment.Paid)
          {
               …
               argsPaidInFull =
                    new PaidInFullEventArgs(invoice);
               OnEPaidInFull(argsPaidInFull);
          }
          …
    }


     // definition of pre-defined ONEvent_name method to
     // raise ePaidInFull
    protected virtual void OnEPaidInFull(
        PaidInFullEventArgs e)
    {
         // The event delegate is combined with the event
         if (ePaidInFull != null) ePaidInFull(e);
    }
```

When the event PaidInFull is raised, it is received by the method SendReceipt so that the method SendReceipt can generates a receipt to send to the customer.

```
// The event delegate invokes Activity Group
// “SendReceipt” when event ePaidInFull is received
```

```
ePaidInFull += new PaidInFullEventHandler(SendReceipt);

public void SendReceipt(PaidInFullEventArgs e)
{
    //Activity: Send a receipt: Note no guard condition
    //as merchant always send a receipt whenever a full
    //payment is received

    Receipt receipt = new Receipt ();
    receipt.Id = e.paidInFull.Id ;
    receipt.Amount = e.paidInFull.Paid ;

    //Send the receipt to the customer as extenal event
    cs.ReceiveReceipt(receipt);
}
```

5.4.2.2 External Events

In GAT, External Events are used to communicate between activities in different business processes. The GAT prototype represents these External Events using .NET Remote objects. These message objects are converted to local Internal Events when they arrive at the receiving system. This mechanism is discussed in detail in Section 5.4.3 below.

5.4.2.3 Deferred Events

Deferred Events are used when a process needs to trigger corrective actions if anticipated events have not happened by some deadline. In our GAT example, a deferred event OverduePayment is raised as soon as an invoice is sent. This event may be received later on by the Activity Group that handles the overdue payment process - but only if the anticipated payment was not received within the due period. In contrast .NET events are consumed immediately. Thus we cannot simple represent each deferred GAT event as a .NET event.

The next code sample shows how the SendInvoice Activity sets a timer that waits until payment due period has expired and then raises the Elapsed event. The timer is turned off if the payment is received within the payment due period. The Elapsed event is received by the Activity Group Overdue that then sends a reminder to the customer and notifies to accounting officer.

```
// Define a timer that checks the payment due
Timer paymentdue = new Timer ();

// As soon as an invoice sent, sets a timer for the
// duraton of the payment due
public void SendInvoice(GoodsReservedEventArgs e)
{
    ...
    paymentdue.Elapsed +=
      new ElapsedEventHandler(Overdue);
    paymentdue.Interval = returnPaymentDueDuration();
    paymentdue.enable = true;
}

// the timer is turned off if payment is received
public void ReceivePayment(PaymentEventArgs e)
{
    paymentdue.enable = false;
    …
}

// if payment is not received till the payment due, the
// timer elapses generating an event overduePayment which
// subsequently received by Overdue method which can
// handle the overdue such as sending reminder of the
// payment and notifying the acoounting office
public void Overdue (OverduePaymentEventArgs e)
{
    //Activity: send a reminder
    ...
    //Activity: send an alarm to an accounting
    // officer that the payment is overdued
    ...
}
```

5.4.3 Inter-process Communication

The GAT prototype contains three components (customer, merchant and shipper) of the e-procurement case study. These communicate with one another by transferring objects using .NET Remoting. This use of .NET Remoting means that business processes can be location-transparent; they may be on the same computer, on different computers on the same network, or on computers across separate networks. In this section we show how this works.

As mentioned above, external events in the GAT model are implemented as object parameters passed using .NET Remoting. The underlying Remoting mechanism also supports different transport and communication protocols. Currently, our GAT prototype allows code to be produced which uses either of two transport protocols, TCP and HTTP.

The following code excerpt illustrates a customer sending a purchase order as a remote object to the merchant service.

```
// Defines a proxy for MerchantService
MerchantService ms =
    Activator.GetObject(
        typeof(MerchantService),

"http://remotehostname:7001/MerchantService.rem");

// customer sending a purchase order
public void SendPurchaseOrder(POEventArgs e)
{
     …
    // Construct a PurchaseOrder from the event ePO
    PurchaseOrder po = new PurchaseOrder();
    po.id = e.ePO.id;
    po.items = e.ePO.items;
    po.totalPrice = e.ePO.totalPrice;
    po.deliveryDate = e.ePO.deliveryDate ;

    // Sends the purchase order as a remote object to the
    // merchant remote service
    ms.ReceivePO (po);
}
```

The prototype implements a remote service class for Remote Communication Handler for each business component. A service method in each of these classes receives incoming messages as a parameter and raises ordinary local .NET Events according to what type of messages was received. Local activity groups will then consume these events, possibly sending out messages in response as parameters to calls on other remote objects. The following code excerpt illustrates the receiving of a purchase order as a remote object. The method ReceivePO, uses the purchase order object as an event argument and raises an event ePO.

```
// The merchant remote service that receives all
// arriving messages as remoting objects from peer
// business processes and turn them into internal events
MerchantService: MarshalByRefObject
{

    // Receives a purchase order
    public void ReceivePO(PurchaseOrder po){

        // Constructs an event argument using purchase
        // order object
        POEventArgs argsPO = new POEventArgs(po);

        // Raise an event ePO
        OnEPO(argsPO);
    }
}
```

5.5 Design of a General GAT Engine

The GAT prototype system we explained so far only works in a situation where developers write code for a particular application in a stylized way. They implement business processes starting from GAT descriptions of these procedures. This type of prototype system is enough to show the feasibility of the GAT model approach for building reliable and robust systems. However, this requires developers to learn a particular style of coding. To overcome the limitation, we explored the possibility of developing a general GAT engine which can produce executable business processes from any business scenarios written in GAT model.

In this section, we describe our initial work on developing the general GAT engine which converts from GAT specifications, which are any business scenarios written in the GAT model, to C # applications running on the .NET platform. Business analysts will represent the intended business requirements in the GAT specifications. The GAT engine reads these GAT specifications, and translates into code written as C# classes which make use of the support provided by Microsoft's .NET environment. The engine then automatically compiles and executes these C# programs on the .NET platform. This separation of specification from implementation is similar to the approach taken by MDA (Model driven Architecture) [61]. Figure 18 illustrates the overall operation of the engine.

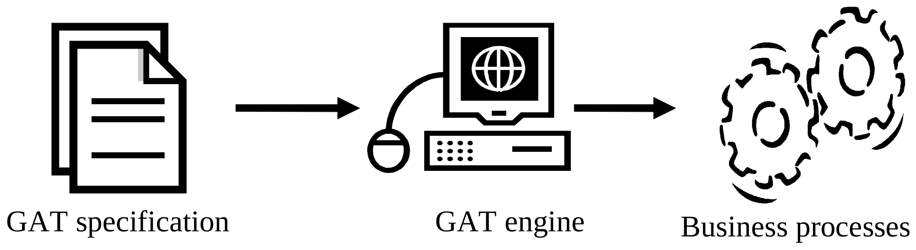


Figure 18 GAT Engine Concepts

We look in more detail at the internal structure of the engine as it carries out the conversion from GAT syntax to executable binaries. The GAT engine has two major stages: translation and generation.

- The translating stage reads specifications written in GAT syntax and translates them to corresponding C# code. As each line of GAT syntax is read, the engine creates an 'analyser' table representing the C# components corresponding to the GAT specification. It also creates a 'mapper' table that contains enough information to define the specific .NET mechanisms needed for each C# components.
- The generation stage is based on the mapper table and uses CodeDOM [59] to create C# classes with matching methods and variables. These C# files are compiled into executable DLL files by the engine.

Figure 19 shows the major steps the engine goes through to produce tables of structures, classes, and files.

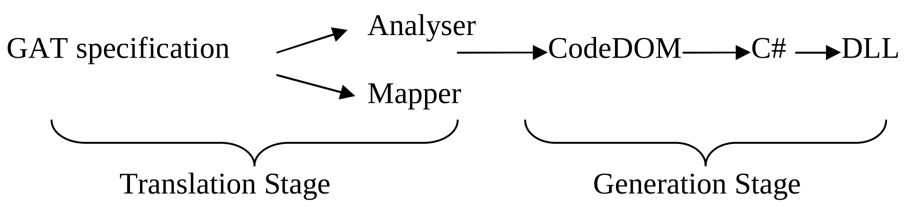


Figure 19 GAT Engine Major Stages

5.5.1 GAT Specifications

The GAT model describes each participating application within a distributed business activity as a set of Activity Groups. The GAT presentation form shown in Figure 10 is unsuitable for machine translation and instead we use the more concrete GAT syntax shown in Figure 20. This syntax was devised just for the purpose of showing that it is possible to construct business processes using the GAT model and that these processes could be translated into executable code.

```
ACTIVTYGROUP: ReceivePayment

EVENT: Payment

ACTIVITY: processFullPayment
GUARD: Invoice.amt = payment.amt
ACTION: FullPaymentAction(payment)
TRIGGER GROUP: (true)[INT] PaidInFull

ACTIVITY: processUnderpayment
GUARD: Invoice.amt > payment.amt
ACTION: UnderpaymentAction(payment)
TRIGGER GROUP: (true) [EXT]ResidueInvoice

ACTIVITY: processOverpayment
GUARD: Invoice.amt < payment.amt
ACTION: OverpaymentAction(payment)
TRIGGER GROUP: (true) [EXT]Refund
TRIGGER GROUP: (true)[INT]PaidInFull

ACTIVITY: processOverdue
GUARD: Not(payment)
ACTION: OverdueAction()
TRIGGER GROUP: (true)[EXT]PaymentReminder
```

Figure 20 Example of GAT Syntax

The notation such as [INT] and [EXT] is used to distinguish between internal and external events. Further work is still to be done on improving the GAT syntax and building better tools for defining GAT-based applications. One could envision sophisticated support for syntax checking, automatic generation of GAT syntax from diagrams and other such development tools; however our current initial work on a GAT engine simply asks the system developer to produce text such as that shown above.

5.5.2 Analyser

Analyser reads a GAT specification file, parses GAT syntax, and stores information in table structures. For example, Activity Groups, complete with their Events and Activities, are parsed and stored in the activity group table. Information about activities are held in the activity table, each entry referring to a guard table entry with its conditional expression, a method name for the code that implements the action (normally pre-written legacy code) and a reference to a trigger table entry.

5.5.3 Mapper

The role of the Mapper is to map GAT structures into the corresponding C# syntax. For example, an Activity Group is converted into a method call containing 'if-then-else' blocks representing each Activity statement within the group. GAT events are translated directly into .NET Events and code is generated to subscribe Activity Groups to these events. Any outgoing events are made as .NET remoting objects.

5.5.4 CodeDOM

The core of the GAT engine generates the source code needed to run the business process corresponding to the given GAT specification. The GAT engine uses the CodeDOM [59] source code generation tool which is designed to work with .NET framework. This tool generates source code from a language-neutral defined set of statements and the GAT engine builds CodeDOM structures from information held in the Mapper table, including class names, member variables and member methods.

5.5.5 Generating C#

The end result of the CodeDOM builder is a fully populated CodeDOM tree. Developers can then generate source code in any .NET languages such as VBScript, J#, or C# from these CodeDOM trees.

5.5.6 Compiling

The GAT engine provides its own code-compiling utilities that can access a C# complier. The C# compiler is then used to generate executables after the successful compilation of generated C# code. The GAT engine then runs generated executables to produce the running business systems. Alternatively, these generated executables can be taken anywhere that runs .NET platform and can be executed manually.

5.6 Evaluation Compared to Other Implementation Alternatives

In this section we review implementation alternatives that can provide similar functionalities to the .NET based solutions as found in our GAT prototype system.

As previously mentioned, defining Activity Groups and Activities are done through the use of C# methods within a class and one if-then-else block inside this method. Guard and Trigger conditions are expressed as Boolean condition that are tested by ‘if’ clause. Action code is expressed as a set of C# function that executes when Guard ‘if’ clause evaluates true. All other procedural and OO languages (java, python, C++ etc) provide similar features and so could be used for code generation.

Events in GAT model follow the feature of publish/subscribe service. A publish/subscribe service is a logically centralized infrastructure that intermediates the communication between publishers and subscribers of information. The information is represented as events, sets of data provided at a particular point in time. GAT is based on a distributed publish/subscribe model where information sources (publishers) generate and publish events, while information consumers (subscribers) manifest interest in sets of events. Hence, the notification service collects and routes events from their producers to the appropriate subscribers, delivering them as notifications. Many modern systems such as CORBA Event/Notification service [69], Java Messaging Service (JMS) [48], Elvin [23], Siena [10], and Herald [9] all provide a type of Publish/Subscribe service though each system distinguishes itself by providing different subscription mechanisms. The big advantage of .NET events over other similar pub/sub model is that the publishers and subscribers are decoupled by the use of *delegates*. The subscriber can change its implementation of how it detects events without breaking any other subscribing classes. The subscribing classes can change how they respond to events without breaking the publisher. The implementations of two classes can be rewritten independently of one another, which make for code that is easier to maintain.

We used .NET remoting for inter-process communication. Java RMI or CORBA IIOP channel also allow programs and software components to interact across application domains, processes, and machine boundaries. This enables applications to take advantage of remote resources in a networked environment. All these three cross-platform communication technologies allow using binary communication over TCP channel which allows a high performance. The biggest disadvantage of .NET remoting and Java RMI is that they are proprietary technologies that rely on specific programming language and platform such as .NET framework or J2EE platform. Though

CORBA IIOP is devised by a standard body such as OASIS, its implementation is considered difficult due to its lack of supporting tools.

Other communication approaches are possible. The Web services technology enables cross-platform integration by using HTTP, XML and SOAP for communication thereby enabling true business-to-business application integrations across firewalls. Because Web services rely on industry standards to expose application functionality on the Internet, they are more suited for the users looking for vendor and platform agnostic solution. However, verbose implementation of XML makes Web services relatively slower as a communication protocol [66]. WS-Events [89] defines a set of XML syntax and rules for advertising, producing and consuming Events for Web Services applications. An Event is an abstract concept that is physically represented by a Notification. Notifications flow from Event producer to Event consumer using asynchronous or synchronous delivery modes. This might be a good fit for implementing External Events for our GAT model. However, at the time of this prototyping, WS-Events was only a proposal from WWW with no implementation support.

Another group of potential implementation mechanisms are the workflow processing systems. BizTalk [58] server provides a solution as a general framework to create business processes that unite separate systems into a coherent much the same goal as envisaged by our GAT engine. The BizTalk server provides two main parts: (1) a messaging component that provides the ability to communicate with a range of other software by the use of various adapters. (2) Support for creating and running graphically-defined processes called *orchestrations*. Messages are received and converted as XML format used by BizTalk engine. These converted XML messages are then delivered into a database called the MessageBox, which is implemented using SQL Server. The logic that drives a business process is implemented as one or more orchestrations using a graphical tool provided by the orchestrations. Each orchestration creates subscriptions to indicate the kinds of messages it wants to receive. When an appropriate message arrives in the MessageBox, that message is dispatched to its target orchestration using the pub/sub model of MSMQ message queuing technology incorporated with BizTalk engine, then takes whatever action business process requires. The result of this processing is typically another message, produced by the orchestration and saved in the MessageBox. This message may convert it to the format required by its destination then sent out to the destination. Many business cases defined in GAT model can be directly implemented using BizTalk as it supports a wide set of technologies that can deliver messages to appropriate event subscribers. However, deferred events are not supported by BizTalk. The biggest disadvantage of BizTalk server engine is the use of orchestration as a tool to define business processes. Through our evaluation [53], we found that the

BizTalk orchestration tool suffers from a significant lack of simplicity in expressing various deviations that are common to such system.

In this section, we reviewed several technologies that can be used to build an executing system for a business processes described in the GAT model. We note that one can use any procedural or OO languages to describe Activity Groups and Activities as a set of methods and if-then-else blocks to capture business requirements. Transaction support is required to form an isolated unit from an activity group (by evaluation of guards), the execution of the chosen action, and the evaluation of its trigger conditions and the raising of any further events to prevent the interference from concurrently running business activities. We also demonstrated that we need an event system that can accept Activity Groups as event subscribers and Trigger Groups and arriving messages as event consumers. The mediation between the event consumers and the event subscribers is also required to connect events generated by the event publishers to the event subscribers. Care must be taken since there may exist multiple event subscribers which subscribe to the same event. If this is the case, all the subscribers must be notified when the event is raised. Inter-process communication technology is another essential requirement for building a system that follows the GAT model.

5.7 Summary

We have presented a list of design considerations of a system that executes long-running business processes defined according to the GAT model.

We demonstrated that it is possible use today's proven technology to build a system following the GAT model. The prototype system we described in this chapter contains executable business processes for the e-procurement scenario we discussed in Chapter 3. We have shown in detail; how each GAT construct can be implemented using C# methods and classes, how control flow among business activities can be implemented using the basic .NET event mechanism where Activity Groups are made event consumers which are notified when events are raised which generate further events, and how GAT events are expressed as .NET events and also as .NET remote objects when communication carries them to other business processes.

We also presented our initial work on developing a general GAT engine which can read from GAT specifications (that is any business scenarios written in GAT syntax) and generate executable business processes which can run on .NET platform environments.

Though our GAT model provides a framework where the developers can represent a reliable and robust design, the GAT model requires the programmers to provide code to handle all possible actions under every possible state. This includes many conditions affected by the interference of concurrent activities. In the next chapter we discuss our work on providing isolation mechanisms to reduce this part of the developer's task.

Chapter 6

Promises – New Unified Isolation Mechanisms for Service-based Systems

In our GAT approach, normal activities and exceptional (deviational) events are processed equally without any limitations on how to handle different types of exceptional events. This increases the chances of producing robust and reliable service-based distributed applications by allowing the developers to define often complex and sophisticated behaviours required to deal with various deviations that could occur during the execution of applications.

Using our GAT model, different activities which deal with different situations of business (both normal activities and exceptional events) can be defined so each will not meet problems, relying on its guard condition. Once the activity which can deal with the current state of system is executed, further events are triggered to notify the follow-up business activities. This model allows the system to execute business activities that are appropriate for the current state of system, so it deals with problems (such as deviations) as they occur without having to abort or roll back to the original or earlier state.

However, the GAT model requires the programmer to provide code to handle each possible event under every possible state. That is, many different activities must be written each with its own guard condition and these guards must cover all possibilities (the closure property). For example, after the merchant checks that sufficient stock is available at the warehouse, it organises transportation to ship the goods to the customer. At the time a truck from the transportation company arrives at the warehouse, there must be code for every possible stock level, such as (1) if there is sufficient stock when the truck arrives (and so the truck can load the goods), or (2) if there is not sufficient stock due to perhaps stock taken for other order requests (in this case the merchant service might have to trigger a backorder to its suppliers and return the truck back to its suppliers).

What would be better for the developers in this situation is that after the merchant checked stock levels when the order was accepted, it could then rely on having sufficient stock available throughout the rest of the order process, regardless of any concurrent orders or other activities. The challenge we deal with in this chapter is providing a useful degree of isolation in a services-based world where autonomy and lack of trust meant that traditional lock-based isolation mechanisms could not be used. Our technique, called ‘Promises’, provides a uniform mechanism that clients can use to ensure that they can rely on the values of information resources remaining unchanged in the course of long-running operations. The Promises approach covers a wide range of implementation techniques on the service side, all allowing the client to first check a condition and then rely on that condition still holding when performing subsequent actions.

6.1 Promises

A Promise is an agreement between a client application (a ‘promise client’) and a service (a ‘promise maker’). By accepting a promise request, a service guarantees that some set of conditions (‘predicates’) will be maintained over a set of resources for a specified period of time.

In the conceptual model discussed in this chapter, promises are granted and guaranteed by a Promise Manager rather than directly by services. A promise manager sits between clients and application services and implements Promise functionality on behalf of a number of services and resource managers. The job of a promise manager is to work with application services and resource managers to grant or deny promise requests, check on resource availability and ensure that promises are not violated.

Client applications can determine what resources they need to have available in order to always complete successfully, express these as a precise set of predicates and send them to the relevant promise manager as a promise request. The promise manager will examine both the complete set of existing promises and the availability of the requested resources, and either grant or reject the promise request. Once a promise request is granted, the client application is isolated from the effects of concurrent activities with respect to the resources protected by its promises. For example, the merchant order-handling process we mentioned above can now ask the manager of the stock resource for an initial promise that the goods required to meet an order will not be sold to anyone else for the duration of the order handling process. Once this promise has been obtained, the order-handling process can proceed with the knowledge that the required stock will be available when needed, even though concurrent order processes may be also selling the same type of goods to other customers.

Traditional lock-based isolation can be seen as a very strong and monolithic form of promise, one where the resource manager is guaranteeing that no other concurrent process can alter, or possibly even examine, the state of a protected resource for the duration of an operation. The proposed promise-based isolation mechanism is weaker but can be just as effective because it can be more precise. The predicates contained within a promise specify a client application's exact resource requirements, allowing other promises covering the same resources to be granted concurrently as long as they do not conflict with any already granted promises.

Promises do not last forever. The client and promise manager agree on the period of time for which a promise will be valid as part of the promise request/granting process, and promises will expire at the end of this time. Promise managers return 'promise-expired' errors to clients that attempt to perform operations under the protection of expired promises.

Promise-aware applications can be written with the knowledge that the resources they need for successful completion will always be available, and any unavailability exceptions can be treated as serious errors rather than as part of the normal processing flow. Applications can always perform actions that are not protected by promises, but resource changes that violate promises will be detected by the promise manager and undone in order to honour the guarantees it has made.

Promises are an abstract way for a client to specify the resources they need to ensure that they can complete successfully. A granted promise guarantees that the requested resources will be available when needed by later actions, but does not necessarily guarantee that any particular instance of the resource will be used to meet this promise. For example, a client may request a promise that a 5th floor room will be available on the requested date. The response to this promise will be that a room matching the requirements will be available, not that the client has been assigned room 512. The messages and services used in the application have to reflect this level of abstraction, in this case by later making a booking for a 5th floor room, rather than trying to confirm a booking for room 512.

Promises are both a pattern and a protocol that supports this pattern. The pattern is simply that client applications determine the constraints they need to have hold over a set of resources and express these as predicates that are sent within promise requests to a promise manager. The promise manager will consult with resource managers to determine whether a promise can be granted, and reply with either a granted or rejected response. Once a promise has been granted, the client application can continue and call services that will make changes to the resources protected by its promises with the guarantee that they will be successful if they are within the constraints implied by its promises. Client applications then release their promises by sending promise release messages to their promise managers. Promise release requests can be combined with application request messages. In this case the promise release and the application request form an atomic unit, and the promise will only be released if the associated action succeeded.

The Promises model places no limitations on the nature or form of predicates nor on the way that promise managers should implement these predicates to guarantee that they hold despite concurrent updates to the same resources. This flexibility means that promise managers and resource managers are free to implement what ever form of constraint checking or isolation mechanism is best for the type of resource being protected.

Some forms of promises could be implemented using the common business practice sometimes called ‘soft locks’. This approach uses a field in the database record to show whether an item has been allocated or reserved for a client. The record is not locked against access once the allocation has been made; instead applications read this field when looking for available resources and ignore any record that has been already allocated. Different forms of promises, such as guaranteeing that there will be enough money in an account to pay for a future purchase, could best be implemented using techniques such as escrow locking [73].

The Promise pattern accommodates both of these ways of implementing isolation, but it is more general, separating the model and its supporting protocol from any specific implementation or resource schema considerations. The flexibility that results lets us also support more general predicates where the actual allocation of a particular resource to a client is delayed to long after the promise is made, and also to support promises over pools of different but acceptable resources that export the same set of properties. Section 6.4 discusses a range of implementation alternatives.

The motivation behind the development of the Promises approach to isolation was to provide application programmers with something similar to the simplicity that comes from the traditional ACID transaction model. By implementing weaker but effective constraints over shared resources, we wanted to let programmers establish those resource-based pre-conditions needed to ensure their application can complete successfully, letting them then write their application code with the guarantee that concurrent activities could not violate these promises. Promise violation is still possible for other reasons (an accident might damage previously-promised stock or a third party may default on a promise they have made) but these incidents can now be treated as serious exceptions. This is very far from the situation without isolation where the effects of concurrency are common enough that they need to be included throughout the normal processing paths.

The promises obtained by clients conceptually place constraints on the behaviour of the services that they invoke. Clients get promises about resource availability and the services they then call should only make changes to protected resources that comply with these promises. For example, if a client obtains a promise that 5 pink widgets will be available to fulfil an order, then the services it calls can complete the order process for these promised goods, or the client can release the promise. The client should not use the promise for pink widgets to ask the order service to deliver some un-promised blue widgets. This restriction on the behaviour of services could be largely theoretical, being more like a design pattern than a type-safety mechanism, or the restrictions could be enforced to some degree by promise and resource managers.

Our proposed Promise protocol fits very naturally into the SOAP protocol and the Web Services model. All of our promise protocol messages can be transferred as elements in SOAP message headers and the associated actions can be carried within the body of the same SOAP messages. The fit between the Promise protocol and SOAP is discussed more fully in Section 6.5.

Section 6.7 compares our ideas to previous work in this area. Our key innovations lie in the analysis of the variety of resources and conditions, in considering how to atomically combine several related aspects of managing a single promise, and in integrating these ideas into the services-oriented message exchange framework.

6.2 Resources and Predicates

This section discusses several different ways that resources can be viewed by client applications, and how these differences are reflected in the types of predicates that can be used in promises over the availability of these resources. Applications can use these different types of resource availability predicates to obtain just the degree of isolation they need for their purposes, without needing to resort to using traditional locking techniques.

Predicates are simply Boolean expressions over resources. Our model imposes no restrictions on the form these expressions can take, and in practice their form will depend on the application involved, nature of the resources and the way we want to view these resources at the time.

The simplest form of predicate expression is an application-dependent request for resources, such as asking for ‘room 212, Sydney Hilton, 12/3/2007’. In this case there is a close coupling between the application, the promise manager and the resource schema, and the promise manager is responsible from translating from this application-dependent predicate to any necessary queries and updates on the room availability data held by the resource manager. The relationship between predicates, applications and resources can be much more abstract than shown in this simple example, and complex applications could define their own resource predicate language and implement their own promise managers to guarantee resource availability.

In their most general and complex form, predicates can be general Boolean expressions over defined resource availability data that is specified using standard schemas. In this case, the client would be responsible for understanding resource schemas and how resource availability is represented, and for constructing suitable predicates in the agreed standard syntax. The promise manager in this case can be completely general purpose, knowing nothing about the applications, schemas or resource availability. All that the promise manager has to be able to do is maintain sets of predicate expressions represented in this standard syntax, check them for consistency, and evaluate them with the assistance of the appropriate resource manager. For example, we could send and maintain resource availability predicates written in a standard language such as XPath using XML or SQL, and have these query expressions evaluated by a compatible resource manager whenever the promise manager needs to check for resource availability or predicate violation.

Predicates are expressions over resources but the form and structure they take in any particular application can depend on the way we regard the resources involved. Different applications may want to treat the same physical resource, such as a particular airline seat or an individual pink widget, in different ways, and so will want to use different types of predicates to achieve the required level of isolation from any other applications that might be using the same or related resources at the same time.

In this section we discuss three different ways of regarding resources: Anonymous View, Named View, and View via Properties. These abstractions were derived from a study of different isolation mechanisms commonly used in existing business practices. These different ways of viewing resources influence the sort of predicates that clients will need to use in order to achieve the level of isolation they require to always operate correctly.

6.2.1 Anonymous View

From the point of view of client applications, some resources can naturally be regarded as pools of indistinguishable and identical resource instances, any of which could meet a client application’s requirements. All the resources in the same pool have the exactly same values for the set of attributes that are relevant to the client and it is not important to the client which items from the pool it is allocated and when this allocation takes place.

Most retail goods can be regarded as anonymous for many purposes. Barnes and Noble may have many copies of each book title in stock, and a client who wants a promise that a book will be available does not care which physical copy they are given when the order is dispatched. In this case, the book title represents a resource pool, consisting of many identical and indistinguishable copies, and all that the retailer needs to track in order to be able to make promises about availability is the number of copies they have available for sale.

Financial applications, such as banking, use anonymous resources all the time. For example, if a promise is made that a client application will be able to withdraw $500 from an account, the bank is not obliged to set aside five specific $100 bills, uniquely identified by their serial numbers.

There can be any number of promises outstanding on anonymous resources, the only constraint being that the sum of all promised resources should not exceed the resources that are actually available. For example, our bank can grant many promises against Alice’s account, just as long as the account will not be overdrawn if all of these promises are followed by withdrawal requests.

The availability of anonymous resources is usually explicitly tracked and recorded in an attribute associated with each resource pool. These attributes are traditionally called something like ‘quantity on hand’ or ‘account balance’.

6.2.2 Named View

Clients using a named view of a resource know that each instance of the resource is unique and possesses an identifier, such as a serial number or some other set of distinguishing characteristics that can be used to refer to it. Clients can obtain a promise about the availability of a resource based on this identifier, and they can later make use of that resource instance, knowing that the promise will ensure it will be available when needed.

Some resources are naturally unique and there is only one instance of a given resource. For example, used cars could be considered unique and not interchangeable, as each one is distinguishable by the distance it has travelled and its condition. A client who gets a promise on a particular vehicle is expecting to get that one, not an ‘equivalent’ substitute. Conversely, new cars and hire cars would normally be accessed anonymously by model or category as they can be considered identical for the purposes of selling or hiring.

Resources such as airline seats or hotel rooms are another common class of named resources. These are virtual resources which represent the opportunity to use a (more or less) physical resource at a specific time. For example, ‘Room 212, Sydney Hilton’, 12/3/2007’ names a specific room instance, and the date is the necessary part of the unique identifier that distinguishes one booking for the room from another.

The concepts of named and anonymous resources are about the way client applications view the resources, not about the resources themselves. A group of related named resources might be accessed anonymously in some situations, and by their unique names in others. For example, each seat on a flight has a unique name (e.g. seat 24G on QF1 departing on 8/10/2007). Some client applications may let customers try to book specific seats on a flight, and so need named access to the seat instance. In many cases though, all economy seats will be regarded as equivalent, and client applications will be using anonymous access to get promises about the availability of economy class seats on that flight.

The availability of named resources will often be tracked by the use of something like free/busy attributes associated with each resource instance. Many resources will support both anonymous and named views at the same time, allowing some clients to obtain promises on specific resources instances while others are getting promises over a collection of such resource instances.

A single named resource instance cannot be promised to more than one client application at the same time, regardless of the predicates being used and how resources are being viewed by client applications. For example, if one client is promised 'seat 24G on QF1 departing on 8/10/2007', this seat must not be included in the considerations leading to the granting of a promise for an arbitrary economy-class seat on the same flight.

6.2.3 View via Properties

The concepts of named and anonymous resource views we just discussed are really based the properties (or attributes) exposed by a resource, and the characteristics of these properties are what determine the type of promise predicates can be requested over these resources. If a set of properties can be used to always uniquely determine a specific resource instance, we can use these properties in predicates where we want a named view of the resources. If a set of properties inherently determine a set of resource instances, then we could use these properties when we want anonymous access to a pool of acceptable and interchangeable resources.

An individual resource or collection of resources would normally expose multiple properties, many of which could be of interest to clients and potentially be the target of promise predicates. For example, a hotel booking service would maintain a collection of rooms and information about their availability on specific dates. Each of these rooms has a number of properties, such as the size and type of beds, whether or not smoking is allowed in the room, whether or not there is a view, and which floor it is on. All of these properties can be used in promise predicates by client applications wanting to determine room availability.

Different client applications, acting on behalf of different customers, can make concurrent requests over the same collection of rooms and use different sets of these properties in their promise predicates. For example, one customer may be asking for a room with a view, while another might be requesting any 5th floor room. Room 512 could be a suitable available resource that would allow the promise manager to grant either of these requests, but the manager has to ensure that the same room is not allocated to both requests at once. The use of different properties in the two competing promise requests makes this task more difficult as it may not be straightforward to see that their predicates are effectively overlapping.

Users may regard some properties as essential and others as desirable but not required, and this could be reflected in their promise predicates. The interplay between essential and desirable properties when obtaining a promise may be complicated and could lead to systems where the promise requestor and the promise maker negotiate to find a promise that is both satisfiable and maximally desirable. For example, the client may initially request a non-smoking room with a view and twin beds, and eventually accept a promise for a room with just twin beds.

Another interesting possibility is that the values of certain properties could be treated as ordered in acceptability, with it being understood that a promise can be satisfied either by a resource that meets the precise value for a property as requested or by one offering a 'better' value. For example, a customer who holds a promise for an economy class airline seat will not normally complain if, when they fly, they are upgraded to business class.

Predicates are expressions over the values of abstract properties of resources, not over concrete fields in database tables. This abstraction gives rise to the possibility of treating resources polymorphically, allowing a single predicate to cover any number of acceptable resources as long as they all expose the required properties. For example, a hotel booking service could aggregate availability information from a number of providers, each with their own schemas for describing available rooms. A single predicate could be used to obtain a promise from any of these providers, as long as they all exported the set of properties required by the predicate (or if the properties they do export can be transformed to the required ones by the promise manager).

6.3 Atomicity and Promises

In this section we identify three important atomicity requirements for the implementation of promises and promise managers. While the autonomy of service-providers means that there is no way to demand atomicity across long duration business processes, it is feasible to require that specific atomicity guarantees apply during the handling of a single Promise message. These requirements are:

- *Request guarantees on several predicates at once.* While it may be common to seek a single guarantee such as ‘ensure that at least 5 widgets are available when I decide to buy them’, sometimes a client will want to ensure that several different properties (perhaps involving several resources) will all be true when the resources are required at later stages of the application’s execution. The classic example is from travel planning, where a client may want a promise that a flight and a rental car and a hotel room will all be available. By treating the evaluation and granting of all the predicates carried in a single promise request as an atomic unit, the client can ensure that they will either get all the resources they need or none of them. As an aside here, the travel agent client could also build up the set of required promises needed by obtaining them one at a time, trying alternative resources and predicates when other promise requests are rejected.
- *Perform an action which depends on, but violates, a previously promised condition, together with releasing the promise.* One common pattern where promises are useful is where a promise of resource availability is used to protect a later operation which consumes the resource (and thus makes it not available any more). Suppose an art gallery service has promised a client that a particular painting will be available, and the client then goes ahead and buys the painting. When the purchase occurs, the gallery service is released from the promise (the client cannot expect the painting to still be available after they themselves bought it!); however if the purchase fails for some reason (perhaps no shipper is available that day) then the promise should remain in force. In this case, the promise release and the action which depends on the promise form a unit and both parts must succeed or fail together.
- *Modify the predicate whose preservation is promised, by obtaining a new promise and releasing a previous one atomically.* An important use-case is where the client requests changes to promises they have already been granted. The requested change can be to upgrade the promises, or to weaken them. For example, if a client has obtained a promise that an account will have a balance of at least \$100, they may find that their anticipated later withdrawal has changed to \$200 (a stronger promise is needed) or to \$50 (a weaker promise). In either case, it would be too restrictive to force the service to honour the new

guarantee as well as the previous one, nor would the client want to release the previous one until the new one was obtained. Thus obtaining a new promise should be atomic with releasing the old one, and the previous one should be retained if the service can't guarantee the modified request.

6.4 Implementation Techniques

The Promise Pattern we are proposing allows clients to ask a service to guarantee that a supplied predicate will remain true for some specified time into the future. The usefulness of this proposal depends on the existence of mechanisms which will allow the provider to guarantee that they can honour these promises, regardless of other promise requests that may be made and any other actions that may take place against the same set of resources. In this section we describe several well-known techniques that could be used in the implementation of promises.

These implementation techniques are not meant to be exposed to clients through the language used to express promise predicates. This principle means that clients can express their resource requirements by using abstract predicates over resource properties, and the promise manager that receives these requests can then use whatever techniques it wants to implement the promises and meet the guarantees it has made. This approach lets the client deal in the abstractions of predicates and resources, and gives the promise manager the ability to implement these abstractions in whatever way is best at the time, and to change these implementations over time without forcing corresponding changes in client applications.

- *Resource Pool*: In managing anonymous interchangeable resources, it is common to keep the available instances of each resource in a pool, and move them to a separate 'allocated' pool to ensure that a promise can be honoured. For example, when we promise that we can supply 10 widgets, we remove 10 widgets from the pool of available widgets and place them in the allocated pool. The digital equivalent can be implemented by keeping a count of available and allocated items in the record corresponding to each type of resource. This technique is similar to escrow locking [73].
- *Allocated Tags*: In the case of resources that are accessed via a named view, we can keep an availability status field as part of the data used to describe the resource instance. This field would be set to something like 'available' initially and then to 'promised' when the instance was provisionally allocated to a client as a result of making a promise. It would then be either set to 'taken' by a subsequent action, or would be reset back to 'available' if the promise is released and the client has no further use for the resource.

- *Satisfiability Check*: The promise manager keeps a record of all the promises it is currently committed to honouring and also has access to the current state of all resources covered by these promises. Whenever a new promise request is received, the manager checks that it and all relevant existing promises can be honoured based on the current state of the resources involved. Similarly, a check is performed after every client-requested operation has completed to be sure that the state afterwards still allows all existing promises to be honoured.

  If property-based access is used, the decision about which resource will be used to honour a granted promise can be delayed until the execution of the operation which takes the resource. In this approach, the promise manager needs to be able to check the compatibility of a set of promises with the state of the resources. This might be done by finding a matching in a bipartite graph where edges link the untaken resources to the promise predicates that they can satisfy.

  One consequence of this model is that the availability of a resource is indicated by the presence (or absence) of a covering predicate, as well as (possibly) fields in the resources themselves. In contrast to the 'allocated tag' mechanism just described above, we now have the situation where the availability field in the resource now only indicates whether or not the resource has been definitely taken. This means that status information for a single set of resources is now distributed between the promise and resource managers, and special care will be needed to ensure consistency.
- *Tentative allocation*: This is a hybrid mechanism, where property-based promise requests are met by marking the chosen resource instances as 'promised', and also remembering the specific predicate that resulted in this resource allocation. If a later promise request is not satisfiable from the pool of unallocated instances, the manager can consider rearranging these tentative allocations to allow it continue to meet all previous promises as well as granting the new request. For example, a request for a hotel room with a view may lead to tentatively allocating room 512 (on the basis that it has a view). When a later request is made to promise a 5th floor room, the system may reallocate 512 to the new request as long as a different room with a view can be still be provided to meet the earlier request.
- *Delegation*: Promises are made that rely on the promises of third parties. For example, a purchase order can be accepted by the merchant if it has received a promise from the distributor that a backorder will be fulfilled on time. In this scenario, the promise is delegated from the merchant to the merchant's supplier.

As mentioned earlier, the architectural model we are using here has promises being granted and guaranteed by a Promise Manager. This system component acts as an intermediary between clients and services by receiving and granting promises, working with resource managers to help determine availability and passing application requests on to services for execution.

In this model, client applications always send both promise messages and application requests to an intermediate promise manager rather than directly to services or resource managers. The promise manager will act on the promise messages, consulting with applications and resource managers as needed to determine if promises can be granted. Application requests pass through the promise manager so that they can be rejected if any associated promises cannot be granted or if executing the request would cause existing promises to be violated.

This is only a conceptual model, although it is the one implemented in our prototype which we describe in Chapter 7. Actual implementations are free to implement the required promise functionality in any way at all. Implementations could move all promise functionality into the application services, letting them use whatever application-dependent mechanisms they wish to express predicates, record promises and determine resource availability. Another alternative would be to move the responsibility for granting and enforcing promises to the resource managers where they could be implemented as a form of dynamic integrity constraint.

6.5 Promise Protocol

This section discusses the structure of some protocol elements that could be used in a SOAP-based implementation of the Promise Pattern. In this protocol, clients and promise managers exchange promise-related information using <promise> and <environment> message header elements. <Promise> elements are used in the creation and release of promises. <Environment> elements are used to specify the promise context that applies for the SOAP service requests carried in the associated message body.

A <promise> element can have zero or more <promise-request> elements; each representing one request for the recipient to make a promise that will guarantee the included predicates for a certain period of time. A <promise> element can also include zero or more <promise-response> elements which are used to return outcomes from previous requests that flowed in the reverse direction. Each participating service can act as both client and promise-maker, so a single <promise> element can include both <promise-request> and <promise-response> elements.

A <promise-request> defines:

- A *request identifier* that can uniquely identify each promise-request. This request identifier is used to correlate promise-requests and promise-responses.
- A set of *predicates* that specify the conditions on which the client will rely in a later interaction and that the promise-maker must maintain.
- A set of *resources* that specify the subjects of the promise.
- A *promise duration* that indicates how long the client wants the promise to be kept.
- An optional set of *promise identifiers* that refer to existing promises that can be released if this new promise request is successfully granted.

Each promise-request must be treated atomically. All of the predicates over the specified resources must be promised or the entire promise must be rejected. A promise request may hand back previous promises in exchange for new promises, and if these new promises cannot be granted, the existing promises must continue to hold.
Promise makers send promise responses back to promise requestors to inform them whether their promise requests have been accepted or rejected. The elements of a <promise response> are:

- A *promise identifier* that the promise maker uses to uniquely identify this promise.
- A *promise result* that says whether a promise request is accepted or rejected. Promise responses could also return other results, such as 'pending' or 'accepted with the condition XX' but these possibilities have still to be investigated.
- A *promise duration* that indicates how long the promise manager will guarantee to keep this promise. This may be the same as the duration which was requested, but the promise manager might, for example, offer a guarantee that expires sooner than the client wished.
- A *promise correlation* which is the *request identifier* of the earlier promise request.

Successful promise requests establish promise environments. Application requests can specify that they must be executed within a specific promise environment (with the set of resource guarantees defined by its promises) by including an <environment> element in the associated message header. An <environment> must define;

- A set of *promise identifiers* that define which promises will apply for the execution of the request.
- A corresponding set of *promise release options* that indicate whether the associated promises should be released after the request has completed.

We note that each message may contain any subset of the different elements relating to promises, and these may be related to the message body or unrelated. For example, we allow an application message from A to B to contain a related request for B to make a promise, and it can also carry a piggybacked response reporting on the outcome of a previous request that B had sent to A.

6.6 Promises and Isolation

The key contribution of the Promise pattern is that it allows a client to check for the availability of resources and then later make service requests with the assurance that these operations will not fail because the required resources are no longer available (except for very rare catastrophic situations that might need human intervention). Programmers are relieved of the need to consider the frequent but unwelcome situation where concurrent activity has changed the truth of relied-on conditions after they were checked.

We will illustrate how applications can use promises to achieve the precise degree of isolation they require through two examples based on the merchant example mentioned earlier. Both of these examples make use of the Promise Pattern but differ in the resources involved, the way they view them and the predicates they use.

The first example Figure 21 shows how the ordering process can check for the availability of goods using a promise and then be guaranteed that these goods will continue to be available for purchase, regardless of any concurrent activities, until the order is completed or abandoned. In this example, the customer is trying to order 5 pink widgets. As our customer doesn't care exactly which 5 of the many identical pink widgets in stock they will receive as a result of this order, we will use the anonymous access view defined in Section 6.2.2 for this example.

Order process	***Promise manager***
Determine we need 5 pink widgets to be in stock	
Send promise request that (quantity of 'pink widgets' >= 5)	
	Check stock levels of pink widgets
	Accept promise if >=5 currently available and not promised elsewhere.
	Record promise as predicate over stock levels, guaranteeing that at least 5 units will always be available. This predicate will be checked before any further promises are granted or purchases are performed.

	Send ‘accept’ <promise response>
	Reject promise request otherwise *Send ‘reject’ <promise response>*
If promise rejected *Terminate order process saying goods unavailable* *If promise accepted* *Continue processing order (organise payment, shippers)*	
Send ‘purchase stock’ request to promise manager *and release promise to keep stock level >= 5*	
	Pass ‘purchase stock’ to application service *(Release 5 pink widgets for delivery and reduce stock-on-hand by 5)* *Remove this promise from the set of predicates over the pink widget stock level*

Figure 21 Outline of Ordering Process Code

The second example is more complex and illustrates the flexibility of promise predicates. In this example, our merchant offers ‘next day’ shipping to its customers for a fixed additional cost on all orders. The order process asks the promise manager for the shipping component for a promise of next day delivery, with the predicate making no assumptions about how this promise will be implemented or needing any information about the structure of the shipping component and its internal states. The shipping promise manager could implement the promise by obtaining soft-locks on warehouse and shipping capacity but other implementations are possible. The merchant may even have a number of shipping alternatives available, each with different capacity and cost structure, and the actual choice of which shipper to use could be deferred until shipping is required in order to reduce costs and optimise utilisation. This flexibility is not visible to the order process or the customer, all that they need to know is that the shipping component has promised next-day delivery and guarantees that this will occur.

6.7 Other Similar Isolation Mechanisms

One of our inspirations in this project was the early ConTract work of Wachter and Reuter [84]. This introduced the importance of expressing preconditions (‘entry invariants’) needed to allow actions within a workflow to execute successfully. The authors identified several different styles of ensuring that these preconditions still hold at the time when applications rely on them later in an execution. Among the styles proposed was the use of semantic locks to preserve conditions and notifying the client when a checked condition changes. Our work extends the semantic lock ideas of ConTract to the services world with its interacting autonomous participants. Our consideration of atomically combining steps is also new. We provide a richer analysis of the variety of resource and predicate types, and of the ways to ensure that predicates remain true over an extended period. We also support a variety of possible implementation mechanisms, each tailored to the needs of specific ways of viewing and accessing resources.

In previous work [54], one of the co-authors of this work developed a transaction model for spatial data which was based on explicit constraints that could be set and unset to limit concurrent modification of properties of the data. Our current paper extends this to a world of autonomous services; as well we now offer an analysis of predicate types, and a better mechanism to structure the operations by providing atomicity between aspects of a single step of the promise exchange.

Recently Dieter Gawlick and other members of the Grid Computing community have suggested the ‘Option’ protocol [24] for reserving access to resources. This has similarities to Promises but our work deals with a wider class of conditions including those on anonymous resources and property-based views of resources, and supports a wider choice of implementation mechanisms. Also, our use of atomicity allows us to unify concepts such as securing, modifying, confirming, and dropping which are represented as separate message types in [24]. The “options” approach has been implemented inside an Oracle database management system, using “data cartridges” to define data types with appropriate indexing and triggers. Zhao at el [97] use the WS-* standards to coordinate the message exchanges in reservation handling. These papers do not consider how to implement various reservations.

There are interesting parallels between promises and the IMS/VS Fast Path mechanism [25]. In Fast Path, each operation is structured as a predicate check and a transformation on the data. The predicate is checked when the operation is submitted, and then at commit-time, the check is repeated, and the transformation is performed (provided the check succeeded). We can consider the operation submission as like a promise request, and commit as like the operation done under promise protection; however, in Fast Path, other operations do not worry about outstanding predicates, and so the commit check might fail because of concurrent activity.

Our Promises pattern unifies and abstracts over many possible implementation mechanisms, including those that are based on previous work mentioned above. The Promises approach offers a common way for clients to work without knowledge of the implementation technique used inside a service that can maintain some property between the time it is checked and a later time when the client relies on the property.

6.8 Conclusion

In this chapter we propose a unified approach to describing the interactions between a client and a service where the client can make sure that some condition over resources will hold at a later time, despite concurrent activities that occur between the check and the use of the condition.

We have analysed the variety of resource types and conditions on those types, identifying an important distinction between resources which are accessed anonymously (where the key property is just whether a given amount or volume is available), resources which are accessed by name, and a wider class where access is based on values for some subset of a collection of properties.

We have identified important cases where several promise-related activities need to be combined into an atomic unit in order to support valuable use-cases such as processing multiple predicates within a single promise request, consuming/releasing promises, and upgrading or weakening a previously obtained promise.

Our proposed Promises allows clients to ask a service to guarantee that a supplied predicate will remain true for some specified time into the future. The usefulness of this proposal depends on the existence of mechanisms which will allow the provider to guarantee that they can honour these promises despite any other actions that may take place against the same set of resources. We explored several well-known techniques that could be used in the implementation of promises for different resource types.

We summarise the structure and content of the promise protocol elements as they would be used in a SOAP-based implementation of the Promise Pattern. Clients and resource managers exchange promise-related information using <promise> and <environment> message header elements. <promise> elements are used in the creation and release of promises. <Environment> elements are used to establish a promise context for the SOAP requests carried in the associated message body.

In the next chapter we discuss the issues involved in implementing this Promise concept in a service provision framework. This will involve developing further details of the implementation for checking predicates against resources.

Chapter 7

Design Principles in Supporting Promises

One of the many problems facing the designer of complex multi-participant Web services-based applications is dealing with the consequences of the lack of suitable isolation mechanisms. This deficiency means that concurrent applications can interfere with each other, resulting in race conditions and lost updates which become one of the many cause service-based systems to produce consistent outcomes.

In the previous chapter, we proposed a unified approach called 'Promises' which can provide an isolation mechanism for service-based applications by describing the interactions between a client and a service where the client can make sure that some condition over resources (predicates) will hold at a later time, despite concurrent activities that occur between the check and the use of the condition.

In this chapter, we discuss some of the implementation issues that need to be resolved in promise-based systems and discuss how we built a proof-of-concept prototype of a Promise Manager that supported promise-based isolation without requiring changes to existing applications and resources. The major challenge in the implementation is to ensure Promise Manager takes overall responsibility and coordinates the activities to maintain the validity of non-expired promises; that is, resources must be available to satisfy every predicate that the Promise Manager is committed to maintain.

7.1 Design Issues and Constraints of Promises

The primary motivations behind the work reported in this chapter were to demonstrate the viability of the promise model by constructing a working prototype, and to observe what this prototype could teach us about building more general and higher-performing implementations. This limited goal meant that we could ignore some of the optimizations and sophistications that would be necessary if we were building general-purpose infrastructure, and concentrate instead on some of the important issues that would underlie all implementations of Promise-based system or infrastructure.

Some of the key design issues that have to be addressed in the implementation of any promise making system are: compatibility with existing applications and infrastructure; the representation of Promises; the relationship between Promises, resource schemas and the promise checking code; ensuring that the promise checking code itself works correctly when there can be many threads concurrently changing the state of promises and resources; and the construction and use of dynamically constructed sets of Promises.

7.1.1 Compatibility

The main constraint we placed on this prototype was it should provide Promise-based isolation support for existing applications, without requiring changes to applications, resource managers or the schemas of the resources being managed. This allows us to reuse existing applications and resource managers thus increasing the productivity of development of isolation support via promise manager as a proof of concept to demonstrate our research concept of “promises”.

7.1.2 Representing Promises

The Promise Manager needs to keep a persistent record of all promises that are currently in effect. Promises are added to the set of current promises as a result of a successful promise request, and are deleted when they are explicitly released by clients.

Promises also only have a limited duration, that is they are valid only for a limited time and then expire. Promise managers need to implement this attribute of Promises and remove them from the set of active promises when they expire.

7.1.3 Promises and Schemas

Promises are basically predicate expressions over the availability of conceptual resources, such as ‘hotel room’ or ‘bank balance’. These resources are defined and controlled by resource managers. Some mechanism has to be provided that will allow the availability of these resources to be queried during the promise checking process.

A general implementation of the Promises mechanism requires some way of automatically mapping between the resource identifiers used in predicate expressions and the corresponding database columns or pre-defined query expressions. This close-coupling between predicate expressions and schemas leads naturally to Promise implementations where the responsibility for promise checking is shared between a Promise Manager and the relevant resource managers. Alternatively, a Promise Manager could retrieve resource schemas from a resource manager and use this information to generate direct SQL query expressions that determine the availability of resources.

This degree of sophistication and complexity is unnecessary for a proof-of-concept prototype where we can restrict the nature and type of our predicate expressions and can write predicate evaluation code specific to the example data we are using.

7.1.4 Isolation and Concurrency

Information about promises and resource availability are stored in different places and controlled by different managers, but they are both accessed as part of promise operations. For example, performing an action which releases a promise requires changing the state of the resource manager (through action code), examining the promise table (to carry out promise checking), and then modifying the promise table (to remove the promise being released). Granting a promise request involves examining the state of RM resources and the promise table, as well as inserting the new promise into the promise table. Without taking special care when engineering a Promise system, we could be vulnerable to race conditions and other isolation failures resulting from concurrent promise operations.

For example, suppose that a request to create a new promise to keep a balance of at least $100 in Alice's bank account operation is running concurrently with an action that withdraws $60 from the account, and the balance is $150. If these two operations run without proper consideration of the potential impacts of concurrency, the check for whether the withdrawal violates any promises might use a list of promises that does not include the new promise, and the promise granting check might use a balance which has not yet been decremented. Both of these operations could succeed, resulting in an inconsistent outcome with a promise being granted that cannot be satisfied by the current state of the resource.

We use traditional transactions to prevent these situations. Note that we are not guilty of circular reasoning; promises are intended to offer isolation support between long running activities, while we exploit the isolation support that transactions provide between individual promise operations.

7.1.5 Dynamic Promise List

Promise checking is at the heart of the Promise Making system. It is the mechanism that allows us to honour the guarantees that have been given to promise clients. Promise checking is conceptually simple: it must make sure that every unexpired promise can be met using available resources at all times.

Promise checking works on a dynamically constructed list of relevant promises rather than on the complete set of promises which have been granted. The main reason for this is that promise checking often needs to be undertaken on proposed sets of promises rather than the complete set of already-granted promises. For example, promise checking during the granting process is done over a proposed new set of promises, including the promises being requested as well as any relevant existing promises.

Using a dynamically constructed list of promises, extracted from the already-granted promises and modified according to circumstances, simplifies the promise checking process. By moving the determination of which promises are relevant out to the Promise Manager, the promise checker is left with the simpler task of checking for consistency within a set of promises and against resource availability.

One advantage of this approach is that we can sometimes reduce the number of promises that must be checked by using semantic knowledge of the promises and the resources they cover. For example, when a promise request for a named or anonymous resource is being considered, the promise checker does not need to check any promises that do not refer to the same resources as the new request. The reduction in checking depends on the type of promise operation, so it is better placed outside the promise checker which executes the same code no matter which promise operation is being performed.

7.2 Structure

The major contribution of this chapter is to demonstrate the feasibility of the Promises concept by building a proof of concept system that provides isolation support for existing applications and resources. This section explains the system design and Section 7.3 discusses the design choices we made when building this prototype.

Figure 22 shows the architecture of our Promise-based prototype system containing different types of messages being exchanged and three major components, namely Promise Manager (PM), Application Server (App), and Resource Manager (RM), that handle different types of messages. The following discussion covers more details of these major messages and components of the system.

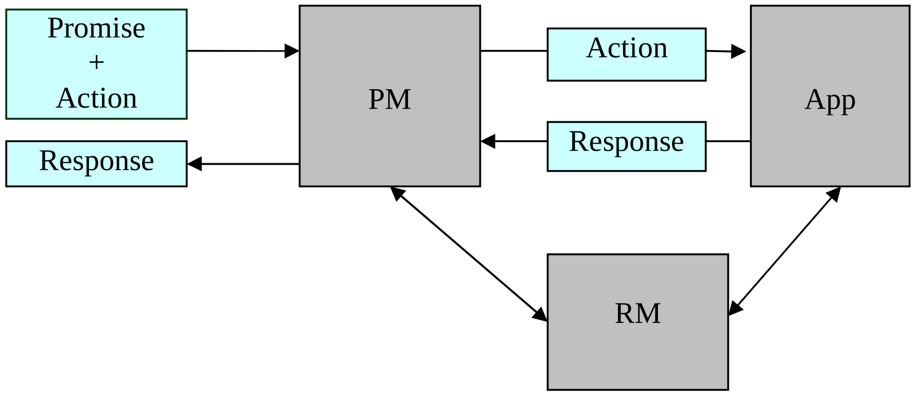


Figure 22 Structure of Promise System

7.2.1 Messages

The messages which arrive at the Promise Manager can contain two parts: an optional Promises part and an optional Action part.

- The Promises part contains the information relevant to promises. For example, it can contain a promise request asking the creation of a new promise that will ensure a given list of predicates will be valid at some later time. It may also contain a promise environment that indicates how the action relates to existing promises (for example, by releasing them if the action is successful).
- The Action part defines the application operation to be performed, with appropriate parameters. For example, the action part may indicate that the client wants to invoke the BookRoom operation and specify the room and date for the reservation. This part of the message is not changed by the addition of promises and will be processed by existing application code. The only difference is that the Promise Manager can now cause otherwise successful operations to fail if the changes they made to resource availability are prevented by currently active promises.

This message structure fits very naturally into SOAP and Web Services standards. All Promises parts can be transferred as elements within the SOAP message header while the Action parts messages are carried within the SOAP message body.

The Promises model makes each part of these request messages optional. However, a typical usage would be that the promise client sends a message that contains a promise request (Promise part only) to request creating a promise. Once the promise has been made, another message is sent with both Promise and Action parts, to perform a state-dependent action with an associated promise environment which indicates a promise that ensures success of the action, and is to be released in connection with performing the action.

The design in Chapter 6 is symmetric, so the Promise Maker can also act as Promise client; thus a single message might contain both promise requests and also responses to requests in the other direction. In this chapter we focus on the design of the Promise Maker, and so we do not discuss how to process any promise material related to the service's activities as a client.

7.2.2 Components

There are three different components shown in Figure 22. The Promise Manager is best seen as an interception layer or an intermediary. The client adds Promises header messages to its normal service requests and sends them to the Promise Manager for processing. The Promise manager then does its work and passes the request on to the application. The roles of each component of the Promise system are explained in the following.

7.2.2.1 Promise Manager (PM)

PM takes overall responsibility and coordinates the activities throughout the promise system. The key data structure kept in the PM is Promise Table recording all currently active promises.

The Promise Manager receives each message as it arrives from a promise client and breaks it up into its Promise and Action component pieces. If a message contains a Promise part, this is split into its promise requests and promises environments and any new promise requests are checked for consistency against the existing promises and resource availability (more details in the Section 7.2.3). After this step, any Action is passed on to the associated application and the Promise Manager waits for a response. If the Action succeeded, the Promise Manager then uses the promise environment to update the set of applicable promises and checks once again that all relevant promises are consistent with the resource availability information held by the RM. This step is what allows the Promise Manager to guarantee that promises will be honoured, regardless of what state changes have occurred as a result of executing the Action. If all promises can still be honoured, the Promise Manager passes back the response it received from the application back to the client. If the result of the action was that promises were violated, the promise manager will roll back the changes made by the Action and return a failure message to the client. The process of the Promise Manager is depicted in the flow chart in Figure 23.

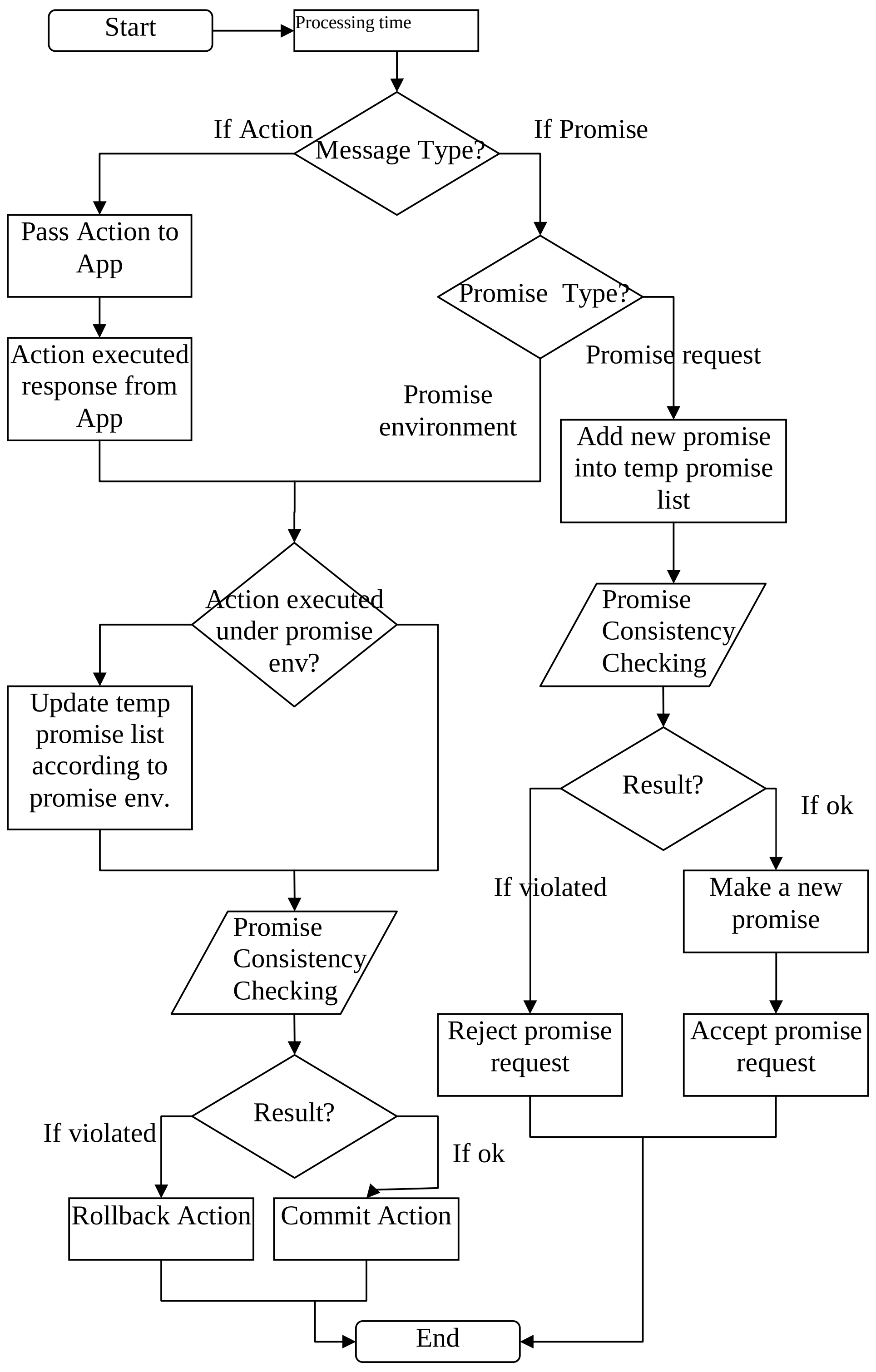


Figure 23 Promise Manager Flow Chart

In our current implementation, an ACID transaction is used for the complete processing of each promise and each action, and this allows us to either commit or rollback any changes made by the application. Note that the transaction covers short-term activity entirely within the components of the Promise Making System. Thus we do not suffer from drawbacks to autonomy that ruled out using a transaction across several client-service interactions.

7.2.2.2 Application

The responsibility of the application is to process the action request passed from the Promise Manager. The application in our design is unchanged and is exactly the same code as would have been executed previously. For example, the application for a hotel booking service must be able to process CheckAvailability and BookRoom operations and these are what are passed untouched from the client to the application via the Promise Manager. We assume that application uses a Resource Manager to keep the state which is shared between operations. After the action has completed, the application sends a response message back to the Promise Manager.

7.2.2.3 Resource Manager (RM)

The role of RM is also unchanged and its responsibility is to store the state of the domain, and to process queries and updates on this data requested by the application and the Promise Manager. For example, the RM for a hotel booking service will keep information about the hotel rooms, their prices, and which rooms have been booked for each day. In Section 7.3.3, we discuss the extent to which the schema of the resource state information must be made explicit to the Promise Manager. The design is able to handle applications which spread their data across several RMs, as long as they support distributed transactions.

7.2.3 Promise Consistency Checking

The crucial responsibility of a Promise system is maintaining the validity of non-expired promises; that is, resources must be available to satisfy every predicate that the Promise Manager is committed to maintain. Ensuring this is made difficult as promises are maintained and understood by the Promise Manager while information about the availability of resources is maintained by RM which has no awareness of promises. To ensure that granted promises are not violated, the Promise system must have a mechanism in place where it can evaluate a set of promises against the current state of resources. We call this mechanism promise consistency checking. The complexity of the promise checking process depends on the particular predicates which have been guaranteed in promises.

For the case of a named resource, promise checking is relatively simple. We must ensure that one of the following situations holds: there are no duplicate promises for the resource identified by the same unique identifiers; or the resource must be recorded as available in the RM, and there is at most one unexpired promise over that resource.

For an anonymous resource where there is a pool of equivalent items, the promise checking sums the quantities of the specified resource required by all unexpired promises, and this must be at least as large as the amount recorded in the RM as being available for this item.

When we have a type of resource which could be relevant to several different predicates, the check is much more complicated. Our proof-of-concept implementation does not deal with this. We would need to consider a bipartite graph containing all the predicates from unexpired promises and all the available resources, with an edge from a predicate to every resource that could satisfy the predicate. A set of promises is consistent with the state of resources provided that a matching edge can be found in this graph.

7.2.4 Promise Operations

Promise checking is then used in several places during message processing, with various sets of promises:

- *Making New Promises*: A Promise request can be sent by a promise client to a promise server in order to create a new promise. Granting the new promise must consider the mutual satisfiability of all existing unexpired promises, together with the requested promise, using currently available resources as known by the RM. For example, when Alice requests a new promise that a specific room 202 in ‘Sydney Hilton’ is available for the date 30 December 2006, this promise request must be rejected if the room is not available (already booked) for that date, or if there is already a promise for the same room on that date.
- *Executing Actions*: The Application executes actions that were coded without knowledge of the PM or its promises. The actions might change the state of resources, for example by updating the account balance upon receiving payment or modifying the availability of rooms when customers make a booking. In a well-designed system, actions would make no state changes except those guaranteed by any covering promises. However the Promise Making System cannot rely on coding of the Application, and so the promise checking must be performed by the PM once an action has been executed to ensure that the state changes made in RM (by the App code) have not violated any existing promises (except for the promises that are being released atomically with the action).
- *Updating Existing Promises*: Promise clients can request to update existing promises. The request can be either to strengthen the existing

promise or to weaken it. Updating existing promises can be seen as the combination of two operations: removing the previous promise and creating the new promise. These two changes must be done atomically. Thus a check must be performed to check the consistency of the resource state against the newly requested promise as well as the set of all unexpired promises except the one to be removed. For example, if Alice wishes to upgrade an existing promise of at least 5 pink widgets, to now guarantee at least 10 pink widgets, and Bob has already been promised 6 pink widgets, then we must ensure that the number of available pink widgets is at least 16.

7.3 Reflecting on our design

This section we discuss our responses to the key design issues we discussed in Section 7.1. These design decisions reflect the needs of the prototype implementation only, and different decisions, and more complex implementations, would be justifiable, and probably necessary, for production-quality Promise-based infrastructure components.

7.3.1 Compatibility

The compatibility constraint required us to engineer the Promises prototype so that we could provide Promises-based isolation support without requiring any changes to existing server applications, resource managers or schemas.

Our solution to this constraint was to implement our Promises prototype as a layer that wrapped existing application systems and ensured that promises could be both granted and honoured. Client applications had to be changed to request promises and associate actions with promise environments, but no changes were required to applications or resource managers. The Promise Manager takes action requests from clients and passes them along, unchanged, to existing applications. These applications process these requests in the normal way and pass back their responses to the Promise Manager which checks for promise violations before committing and returning the response to the client.

7.3.2 Representing Promises

The Promise Manager needs to keep a persistent record of all promises that are currently in effect. Each promise is represented by an object that is persisted by storing it as a row in an SQL database table. Each promise has attributes of Promise Identifier, promise request correlation, predicate and expiration. The set of all currently-effective promises make up the Promise Table.

The database Promise Table is reflected in an in-memory table that is protected by locks as necessary. Changes to this table are committed and persisted by storing them into the database version of the table.

Every promise has a fixed duration, represented by its expiration attribute. These expiry times are used by the Promise Manager is constructing lists of promises for checking and expired promises are deleted at appropriate times.

7.3.3 Promises and Schemas

The compatibility constraint discussed above meant that the prototype had to assume that the application, RM and schemas are given and were developed without knowledge or understanding of promises. Our design does not require changes to the application or schemas but the promise checker does need to access the RM in order to check resource availability. This means that the Promise Manager must understand something of the schema of the RM so that it can generate the appropriate queries. We would like to limit the coupling, however, so that a Promise Manager can be coded in a fairly generic way.

We have assumed that the Promise Manager is able to query the RM to find out the availability of each named resource. Coding the Promise Manager involves finding out the schema that describes the resources. At least we need to know how to express a primary key for the resource (for example, in the hotel booking service the primary key might be a composite of the columns hotel_name, city, room_number, date) and how to find out whether the resource is available. Similarly, the Promise Manager needs to know how to identify a pool of equivalent anonymous resources, and how the RM stores the available quantity for the pool. Finally, for general predicates, the Promise Manager must be able to determine which resources meet a given predicate; this requires matching attributes mentioned in the predicate with columns stored in the RM.

All of this can be solved with a reflection mechanism in which the Promise Manager dynamically-generates the appropriate SQL code to access the RM, supposing that the RM publishes the list of resources it manages, and the relevant schema. This degree of sophistication and complexity were not needed for a proof-of-concept prototype, and instead we have hard-coded the Promise Manager using knowledge of the schema for the limited resources we are managing.

7.3.4 Isolation and Concurrency

The solution we adopted to prevent problems arising from concurrent access to the promises table and shared resources is to wrap each promise operation in a transaction. This transaction is started when we begin processing each client request and committed or rolled back just before the result of the request is returned to the client. This transaction covers all of the action code executed inside the application as well as the subsequent promise checking (and possible modifications of the promise table if the action has a promise environment that releases previous promises). This means that all accesses to the RM's tables, as well as the accesses to the promise table are transactional which gives us the required level of isolation.

This design makes coding the Promise Manager very easy but does risk creating a performance bottleneck under very high load since there are times where we will want to scan (and so lock) the entire Promise Table, and this could block concurrent insertions or deletions.

We also have considered more sophisticated implementations, where insertions in the promises table are speculatively done in a separate transaction from the promise checking and deletions are done in a separate transaction after the promise checking has completed. However we decided to use the straightforward design based on using a transaction through entire promise process since our purpose was to demonstrate the feasibility of implementing Promises using existing technologies rather than building high-performance infrastructure.

7.3.5 Dynamic Promise List

The design we adopted for the prototype has the Promise manager first searching the promise table and extracting the relevant promises to be checked. These promises are placed in a local data structure which is then adjusted by adding or deleting promises to create a proposed set of promises which is passed to the promise checking code. Consider, for example, where we are calling promise checking after performing an action whose promise environment indicates that a promise will be released if the action succeeds. In this case we construct of list of all relevant promises and construct a proposed set of promises by removing the promise that is about to be released. This proposed set of promises is then checked for consistency before making the same changes to the real Promise Table.

Alternative designs to dynamically constructing sets of proposed promises were considered. We initially intended promise checking to take no arguments, but rather to find the list of promises directly by looking up the Promise Table. This would require that each promise operation would modify the global Promise Table and then call promise checking to verify that the table was consistent. For the example just given above, we have to speculatively remove the promise from the table before calling promise checking, since the action just completed is very likely to have introduced an inconsistency between the resources and the about-to-be-removed promise. Speculative modification of the Promise Table is not unreasonable when under a transaction covering the whole promise operation, but this approach is less flexible and limits our ability to introduce greater concurrency between promise operations.

7.4 Implementation

In this section, we discuss the details of a proof of concept implementation we have built embodying the design decisions mentioned above. Our prototype uses the .NET platform with C# as programming language, and it extends a simple App and RM which provide the services typical of a hotel booking service.

We first explain how Promise Consistency Checking interface has been coded. We also show the different implementation mechanisms which need to be applied in checking the resource availability, for promises that refer to different types of resources. Due to time limitations, the current prototype system can deal with named resources and anonymous resources only; resources mentioned via properties will be implemented in the future. We then show the coding of the main Promise Operations calling, each of which includes a call to Promise Consistency Checking.

7.4.1 Overview of Promise Consistency Checking Interface

The promise checking evaluates consistency between a promises list, which is provided by PM, and information about the resource availability which is maintained by RM.

In our system, the promise checking is implemented as a method within PM that takes a list of promises as an input and returns a Boolean value indicating whether the promise system will be able to satisfy the list of promises passed in with the current resources that are available. The input parameter is a list that is dynamically constructed by the PM from its Promises table, reflecting those unexpired promises that might need to be checked, and also being modified to reflect the potential changes in the promise operation for which the check is done. The following code excerpt illustrates the promise checking interface in our prototype system;

```
public bool promise_consistency_checking
(ArrayList promises)
{
    ….
}
```

The real complexities are inside the checking algorithm. In the next two subsections, we discuss in detail how we check the availability of different types of resources. Currently we have two different types of promises objects; one object type represents promises that concern named resources. The other object type represents promises that use anonymous resources.

7.4.1.1 Named Resource Promise Consistency Checking

It is straightforward to check the consistency of the promises over Named Resources. Since each resource is distinguished by its unique key, it is easy to see whether each promised resource is available in the RM, and also to avoid the situation where a resource is doubly promised.

We illustrate the promise checking for named resources with code where the resources are hotel rooms described as follows.

- RM maintains a list of resources in a table [rooms]. Each room is uniquely identified by the composite key (hotel_name, room_id, available_date). The “availability” field indicates whether the room is still available or has been booked.
- Each promise object over a hotel room contains a promise_id, a combination of (hotel_name, room_id, date) as a unique resource key, and a field expiry.

Here is the promise checking code for these named resources.

```
public bool promise_consistency_checking (
    ArrayList promises)
{
    bool consistency = true;

    // check first if there are duplicate promises
    // concerning the unique key (hotel_name,room_id,
    // available_date)
    IEnumerator ie1 = promises.GetEnumerator();

    // We use a hashtable to check for duplicates of the
    // resource’s unique key
    Hashtable ht = new Hashtable();

    while (ie1.MoveNext())
    {
        Promise p = (Promise)ie1.Current;

        string key = p.hotel_name + p.room_id +
            p.available_date;

        try
        {
            // check the expiry of promises
            DateTime today = DateTime.Today;
            if (p.expiry_date >= today)ht.Add(key, p);
        }
        catch (Exception e)
        {
            consistency = false;
        }
        if (consistency == false) break;
    }

    // Now check if resources used by promises are
    // available: we does by returning an arbituary value
    // 1. SQL server returns 1 if matching record defined
    // by a compsite key is found. Otherwise it returns
    // empty
    foreach (DictionaryEntry de in balances)
    {
        Promise p = (Promise)de.Value;

        // finding a matching record from [rooms] table
        try
        {
            string sql = " SELECT 1 FROM rooms " +
            " WHERE hotel = '" + p.hotel + "'" +
            "   AND room_id = '" + p.room_id  + "'" +
            "   AND date = '" + p.date + "'" +
            "   AND availability = 1";

            SqlCommand cmd = new SqlCommand(sql, conn);
            SqlDataReader reader = cmd.ExecuteReader();

             // if no matching record is found, there is
              // no available resources used by the
              // promise, therefore consistency fails.
              if (!reader.Read())consistency = false;
        }
        catch (Exception e)
        {
            consistency = false;
            if (consistency == false) break;
        }
        return consistency;
    }
}
```

7.4.1.2 Anonymous Resource Promise Consistency Checking

Checking the consistency of promises over Anonymous Resources is more complex, compared to promises about Named Resources, as we need to compare the quantity on hand to the total amount needed to satisfy all the promises concerning this pool of resources.

We show our implementation for Anonymous Resources that use bank account balances. We first explain the data structure.

- RM maintains amount of funds available for customers in a table [fund]. For each customer, his/her balance is stored in a record with fields including (customer_id, funds_available)
- Promise objects contain promises for keeping certain amounts of funds for registered customers. Each promise object contains a promise_id, a customer_id, amount which indicates the amount of funds the system has promised to keep available in the given customer's balance, and expiry.

In this scenario, the consistency requirement is that the total amount in promises for the same customer does not exceed the balance held by the customer as recorded in the resource maintained by RM.

We first process the promises list to combine promises over the same customer's balance.

```
public bool promise_consistency_checking (
    ArrayList promises)
{
    IEnumerator ie = promises.GetEnumerator();

     // a new list that contains the total unexpired
     // funds promised for each customer
    Hashtable balances = new Hashtable();

    while (ie.MoveNext())
    {
        Promise p = (Promise)ie.Current;

        // first check expiry of promises
        DateTime today = DateTime.Today;

        // if promises are not expired, accumulate all
        // funds for the same customer
        if (p.expiry_date >= today)
        {
            if (balances.Contains (p.customer_id))
            {
                decimal b = (decimal)balances
                    [p.customer_id];
                b += p.amount;
                balances[p.customer_id] = b;
            }
           // insert each customer with their total funds
           // in the list
           else balances.Add (p.customer_id, p.amount);
        }
    }
```

After all promises for each customer have been totalled, then we check whether this total exceeds his/her balance recorded in RM.

```
    IEnumerator ie1 = balances.GetEnumerator();

    bool consistency = false;

   // look up the list which contains customers with
   // their total funds
   foreach (DictionaryEntry de in balances)
   {
       try
       {
           // get funds available maintained by RM for
           // each customer
           string sql = "SELECT funds_available" +
           " FROM funds " +
           " WHERE cust_id= '"+(string)de.Key+"' ";

           SqlCommand cmd = new SqlCommand(sql, conn);
           SqlDataReader reader = cmd.ExecuteReader();

           decimal funds_avail = 0;
           if (reader.Read())
               funds_avail = reader.GetDecimal(0);

           // if total amount promised for this customer
           //exceeds the funds available, consistency has
           //been violated.
           if (funds_avail - (decimal)de.Value < 0)
                consistency = false;
        }
        catch (Exception e)// eg database problem
        {
            consistency = false; // for safety
        }

        if (consistency == false) break;
     }
     return consistency;
}
```

7.4.2 Implementation of Promise Operations

In this section, we discuss how Promises Operations are implemented in our prototype system, using the promise checking method as described in the previous subsection.

Depending on the nature of each Promise Operation, it is essential to find an appropriate set of promises to check for consistency with one another and with the state recorded in RM; if consistency is shown, then we update the PM's promises table. As described earlier, we use .NET transactions to provide isolation between interleaving Promise Operations.

7.4.2.1 Making New Promises

The important consideration is to grant a promise only if we can satisfy it (and all previous promises) with resources that are available. To achieve this, the operation for making new promises runs as a transaction. It takes a snapshot of the relevant entries from the promises table stored persistently by PM and makes a temporary promise list using the snapshot. It then includes the new (requested) promise into the temporary promises list, and passes this for the promise consistency checking. If granting the new promise would not violate consistency, PM now inserts the requested promise record into the persistent promise table in the persistent storage and commits the transaction. If promise checking returns *false*, granting the request would violate the consistency of promises, so PM aborts the transaction. These message sequences are illustrated in Figure 24.

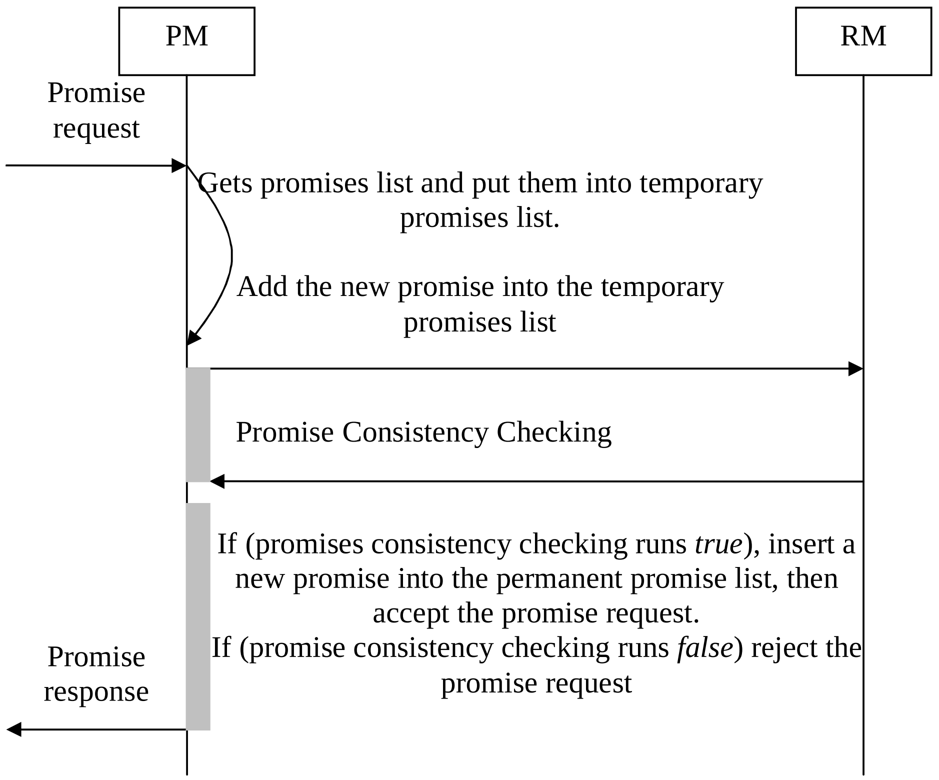

PM
RM
Promise request
Gets promises list and put them into temporary promises list.
Add the new promise into the temporary promises list
Promise Consistency Checking
If (promises consistency checking runs *true*), insert a new promise into the permanent promise list, then accept the promise request.
If (promise consistency checking runs *false*) reject the promise request
Promise response

Figure 24 Message Sequence on Making New Promises

We show code examples that illustrate the scenario described in the message sequence in the following.

```
// method that handles the making of new promises
public void making_new_promises(Promise p)
{
     // Create an explict transaction instance to run this
     // method as a transaction. Transaction begins here.
    CommittableTransaction tx =
        new CommittableTransaction();

    // Form Get the current promises PM kept in its
    // Promise Table and make a temporary promises list
    ArrayList promises = get_promises();

    // Add the requested promise into the temporary
    // promises list
    promises.Add(p);

     // run the promise consistency checking passing the
     // temporary promises list
    bool pcc = promise_consistency_checking(promises);

    // if all promises in the list can be satisfied
    if (pcc == true)
    {
        // grant the promise request and insert into the
        // Promise table
        create_promise(p);
        tx.Commit();
    }
    // If all promises are violated reject the promise
    else
    {
        tx.Rollback();
    }
}
```

For a request over the hotel room named resources we discussed earlier, getting the list of relevant unexpired promises is coded as follows.

```
// Gets the current promises kept in the PM's promise
// table
private ArrayList get_promises()
{
    // declare a list to contain the current promises
    ArrayList promises = new ArrayList();

   DateTime today = DateTime.Today;
```

```
    // Get all unexpired promises that is the promises
    // whose expiry_date is beyond the moment the query
    // is requested
    string sql =
        "SELECT promise_id, hotel_name, room_id,
            available_date, expiry_date " +
         " FROM promisesNR " +
         " WHERE expiry_date >=" + today;

     // Query is executed
     SqlCommand cmd = new SqlCommand(sql, conn);
     SqlDataReader reader = cmd.ExecuteReader();

     //Add each promise record to the promises list
     while (reader.Read())
     {
         Promise p = new Promise();
         p.promise_id = reader.GetInt32(0);
         p.hotel_name = reader.GetString (1);
         p.room_id = readr.GetString(2);
         p.available_date = reader.GetDateTime(3);
         p.expiry_date = reader.GetDateTime(4);

         promises.Add(p);
     }
     …

     return promises;
}
```

In our implementation, we remove expired promises whenever inserts or deletes are made to the persistent promises table. The following code except illustrate this.

```
private void create_promise(Promise p)
{
    …
    // inserting a new promise into promises table
    string sql =
        "INSERT INTO promisesNR" +
        "(promise_id, " +
        " hotel_name, " +
        " room_id, " +
        " available_date,” +
        " expiry_date " +
        "VALUES (" + p.promise_id + ",'" +
           p.hotel_name + "','" +
           p.room_id + "','" +
           p.available_date + "','" +
           p.expiry_date + "')";
    SqlCommand cmd = new SqlCommand(sql, conn);
    cmd.ExecuteNonQuery();

   // remove all expired promises from the promises
   // table
    DateTime today = DateTime.Today;

   string sql1 = "DELETE FROM promisesNR " +
       " WHERE expiry_date < " + today;

    SqlCommand cmd1 = new SqlCommand(sql1, conn);
    cmd1.ExecuteNonQuery();
    …
}
```

7.4.2.2 Executing Actions

The messages being exchanged among Components of a Promise Making System need careful coordination to allow the clean separation between executing actions and the checking for the consistency. When invocation of an action arrives from the promise client at PM, a transaction is created by PM, and PM then passes the action to App with the transaction context. App executes the action which may update/query the resources maintained by RM.

Once the action has been executed, PM checks if the action made any updates on resources so these are no longer consistent with the promises that must be maintained. For this check, PM first gets a snapshot of list the promises table, called the temporary promises list. Then, PM updates the temporary promises list to reflect any promise environment which was presented by the action. For example, if the action is in the context of a promise environment which releases existing promises, then the promises released by the action need to be removed from the promises list. The possibly modified promises list is used when running the promise checking. If promise checking returns *true*, the updates on resources made by the action have not violated promises consistency so the persistent promise table can be updated and the transaction is committed. On the contrary, if the promise checking returns *false*, the action has changed the resources so as to conflict with the promises which must still be maintained, so it needs to be rolled back. These message sequences are illustrated in Figure 25.

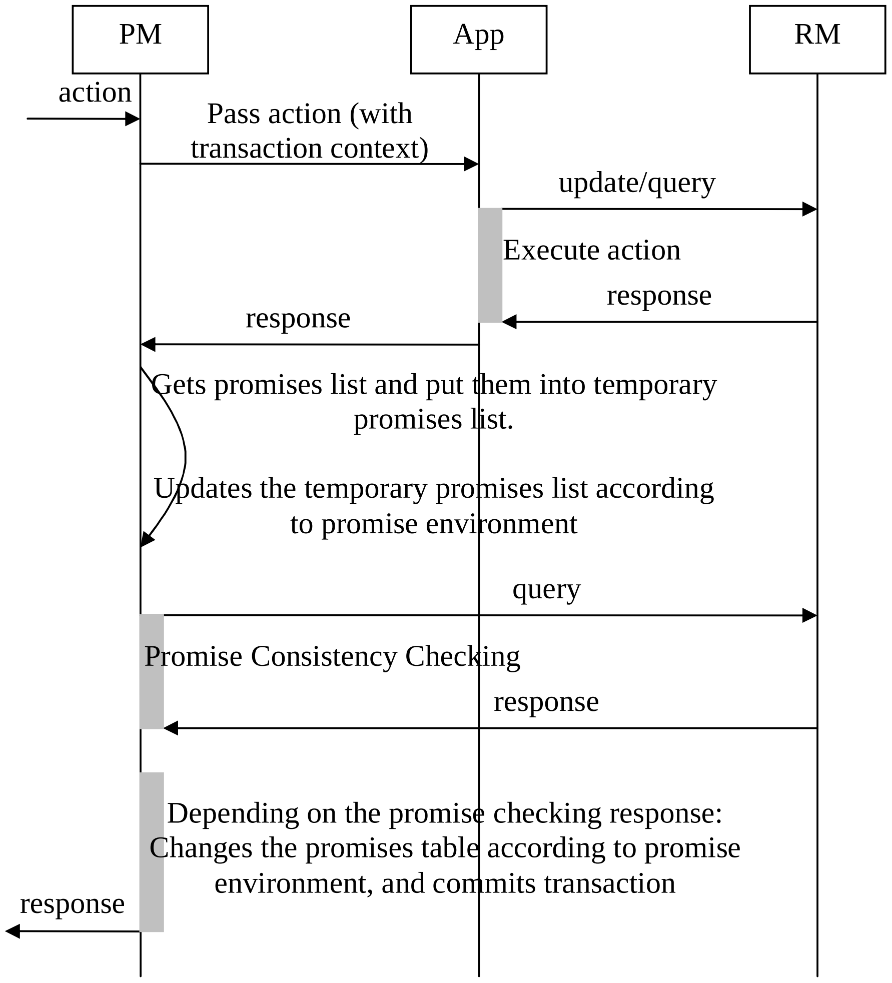

PM
App
RM
action
Pass action (with transaction context)
update/query
Execute action
response
response
Gets promises list and put them into temporary promises list.
Updates the temporary promises list according to promise environment
query
Promise Consistency Checking
response
Depending on the promise checking response: Changes the promises table according to promise environment, and commits transaction
response

Figure 25 Message Sequences on Executing Actions

We show the implementation of executing action in our system in the following code examples.

We first define a class that represents a promise environment which contains list of promise identifiers with options as whether to release the listed promises.

```
class PromiseEnvironment
{
  public int[] promise_id;
  public int releaseOption;
}
```

We also assume that there is an action executed by an application. The execution of an action is demonstrated by the method `making_payment()` in our implementation. If the action `making_payment()` is in the context of a promise environment to release the existing promise, for example a user is paying to take the hotel room that has been booked for the user, the action is successfully executed and the promises in the promise environment are released permanently from the promises kept by PM. However, if the action `making_payment()` is not in the context of a promise environment, for example a different user is paying for the hotel that has been booked for another user, if the consistency of promises list could be violated. The PM must roll back the action in this case to maintain the consistency of the promises list kept in the PM.

The following code excerpt illustrates how PM can maintain consistency when an action is executed with or without the presence of a promise environment.

```
// Method handles executing actions
public void executing_actions(PromiseEnvironment pe)
{

    // Create and begin a transaction to run while
    // checking validity of action being executed
    CommittableTransaction tx =
        new CommittableTransaction();

    // Let’s assume make_payment() action has been
    // executed by an application
    App app = new App();
    app.make_payment(tx);

    // Gets the current promises list kept in PM and make
    // it as a temporary list
    ArrayList promises = get_promises();

    // Create another promise list that is presented in
    // the promise environment. This list is used to
    // remove promises from the pemerant promises list
    // kept in PM
    ArrayList promises_affected = new ArrayList();

    // if a promise environment presents with the action
    if (pe != null && pe.releaseOption == RELEASE)
    {
        // remove all promises presented to release in
        // promise environment from the temporary list
        for (int i = 0; i < pe.promise_id.Length; i++)
        {
            int promise_id = pe.promise_id[i];
            Promise p = get_promise(promise_id);
            promises.Remove(p);

           // add promises in the promise environment
            promises_affected.Add(p);
        }
    }

    // Pass the updated temporary list for the
    // consistency checking
    bool pcc = promise_consistency_checking(promises);

    // If the consistency is still maintained with the
    // updated promise list, the action can be granted to
    // commit.
    if (pcc == true)
    {
        // Promises presented in the promise environment
        // are removed permanantly from the promises list
        // kept in PM
        IEnumerator ie =
            promises_affected.GetEnumerator();

        while (ie.MoveNext())
        {
           Promise p = (Promise)ie.Current;
           remove_promise(p);
        }

      tx.Commit();
    }
    // If consistency is violated with the updated
    // temporary list, the action must be roll back
    else
    {
        tx.Rollback();
    }
}
```

7.4.2.3 Updating Promises

The promise allows clients to request to update existing promises. The important consideration is to grant an update request only if the update doesn’t conflict with existing promises and resources for the update promise are available. Similar to other promise operations, the operation to update promises runs as a transaction. In the start of the transaction, the PM takes a snapshot of the current promises kept in the promise table and makes it a temporary list. As updating existing promises can be seen as the combination of removing the previous promise and creating the new promise, PM removes the previous promise and adds the new promise from/into the temporary list. The temporary list which has been updated according to the update request is passed for the consistency checking. If granting the update would not violate consistency, which is confirmed by consistency checking returning *true*, PM now deletes the previous promise and inserts the new promise record into the persistent promise and commits the transaction. If consistency checking returns *false*, PM aborts the transaction to avoid violation of the consistency of promises. These message sequences are illustrated in Figure 26.

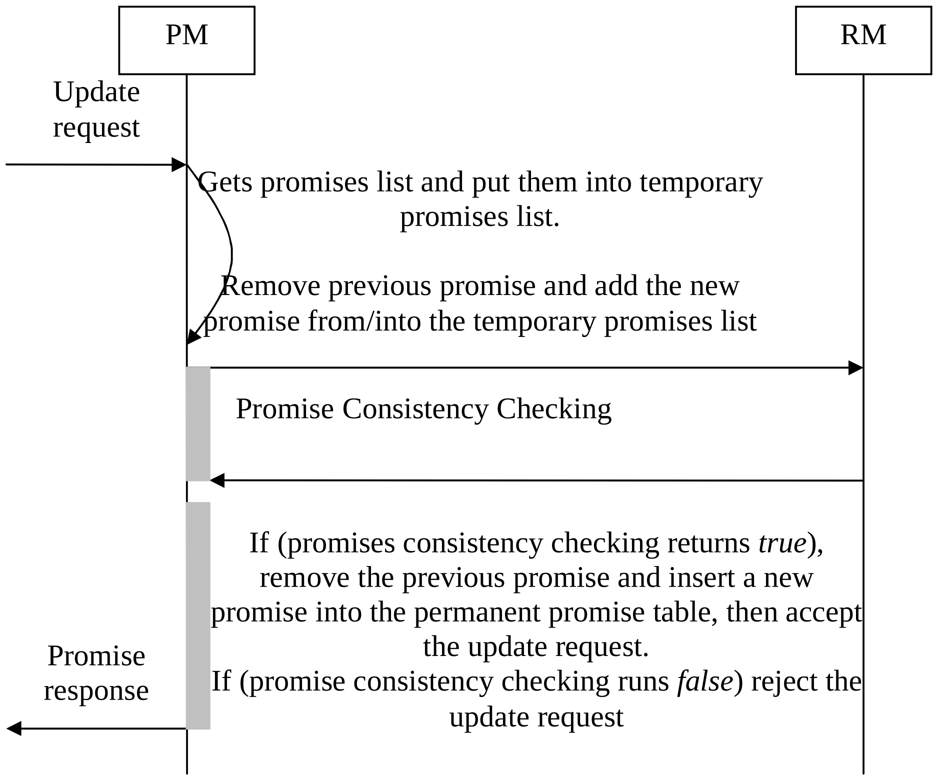

PM
RM
Update request
Gets promises list and put them into temporary promises list.
Remove previous promise and add the new promise from/into the temporary promises list
Promise Consistency Checking
If (promises consistency checking returns *true*), remove the previous promise and insert a new promise into the permanent promise table, then accept the update request.
If (promise consistency checking runs *false*) reject the update request
Promise response

Figure 26 Message Sequence on Updating Promises

The code showing the message sequence of updating promises is illustrated in the following example.

```
// the method handles updating promises
public void updating_promises(
    Promise old_p, Promise new_p)
{
    //the updating promise operation runs as a
transaction
    CommittableTransaction tx =
        new CommittableTransaction();
    // Get the current promises kept in PM and make it as
    // a temporary list. Remove the previous promise in
    // the update from the temporary list, and add the
    // new promise into the temporary list
    ArrayList promises = get_promises();
    promises.Add(new_p);
    promises.Remove(old_p);

    // Pass the udpated temporary list for the
    // consistency checking
    bool pcc = promise_consistency_checking(promises);

    // if consistency checking returns true
    if (pcc == true)
    {
        // delete the previous promise and insert the new
        // promise into the permanent promise list, then
        // commit the operation
        update_promise(old_p, new_p);
        tx.Commit();
    }
    // if consistency checking returns false, abort the
    // operation
    else
    {
        tx.Rollback();
    }
}
```

7.5 Other Alternatives

There are some similarities between the implementation mechanisms we introduce for maintaining a promise, and algorithms previously used in database locking such as escrow locking [73] and predicates [21]. However we can identify some clear differences. All our promises have limited duration, and thus none of our techniques violate autonomy by allowing a client to deny access to resources unduly. Also, our promise making techniques generally deal with problems by rejecting a promise request, rather than blocking as in traditional database concurrency control. The only blocking we allow is during the ACID transaction that checks if a promise request can be granted, or if an action has violated any promises; these are very quick steps, and can be coded with resource ordering to avoid deadlock, which is a very common error in conventional locking systems.

Promises are also analogous to integrity constraints, and many researchers have considered how to enforce integrity in database management systems. Techniques based on index data structures are commonly used for the simplest constraints such as primary and foreign keys. A more general approach involves modifying each query to incorporate the constraint [77]. In [73], it showed how compile-time checks could ensure that application code preserved constraints. Techniques like these might be useful in implementing a promise manager which needs to check each client action for compatibility with previously granted promises. However, there are important differences between integrity constraints and promises. Most significantly, each integrity constraint can be considered independently, while promises need to be satisfiable by disjoint resources. For example, two integrity constraints ‘balance>100’ and ‘balance>50’ are both met if the balance is 120, but two promises for ‘balance>100’ and ‘balance>50’ imply that the balance must be kept over 150. Any promises that may violate “keeping the balance over 150” will be rejected by the promise system. With property views, promise satisfiability can require a graph matching algorithm, whereas integrity satisfiability is just logical satisfiability. Also, our promises could ensure checking of dynamic constraints on the fly among real-time business process interactions.

7.6 Summary

We have presented a detailed design for how to engineer support for Promise Making in a web service, to provide support for isolation of long running activities. A prototype implementation has been done using .NET technologies, using two scenarios to cover the different handling mechanisms for named and anonymous resources.

Our prototype is designed to provide Promise-based isolation support for existing applications, without requiring changes to applications, resource managers or the schemas of the resources being managed. We implement our Promise Manager prototype as a layer that wrapped existing application systems and ensured that promises could be both granted and honoured. The Promise Manager takes action requests from clients and passes them along, unchanged, to existing applications. These applications process these requests in the normal way and pass back their responses to the Promise Manager which checks for promise violations before committing and returning the response to the client.

The crucial responsibility of a Promise system is maintaining the validity of non-expired promises. In other words, resources must be available to satisfy every predicate that the Promise Manager is committed to maintain. To ensure that granted promises are not violated, the Promise Manager implements a Promise Consistency Checking mechanism where it evaluates a set of promises against the current state of resources. We illustrate two Promise Consistency Checking mechanisms to cover named resources and anonymous resources. We also demonstrate the ways Promise Consistency Checking are used in various operations, such as making new promises, executing actions, and updating existing promises, which could violate the validity of promises.

In the next chapter we bring together the lessons we have learnt and the insight we have built up through this research.

Chapter 8

Conclusions

Web Services and service-oriented architectures are being promoted as the best way to build the next generation of Internet-scale distributed applications. These applications are made by gluing together opaque and autonomous services, possibly supplied by business partners and third party service providers, into loosely-coupled virtual applications that can span organisational boundaries and connect large-scale business processes.

Services are just applications that expose some of their functionality to other applications in a particularly simple and restricted way. Services are autonomous, opaque (and probably stateful) applications that communicate with each other solely by exchanging asynchronous messages. This services model is extremely simple but, unfortunately, this simplicity does not mean that large-scale service-based applications will prove to be easy to develop in practice or sufficiently reliable when they are deployed.

There are now a number of proposed standards for EAI and B2Bi solutions for building service-based systems. Through such tool support and standards, it is fairly easy to design and construct this kind of integrated system. Current technology does not, however, make it easy to design reliable and robust applications: ones that can deal with events that cause deviations from normal processing paths, such as failures and concurrent activities, while still maintaining overall, cross-organisational consistency. The main focus of our work has been providing programming models and protocols which make it easier to detect and avoid consistency faults in the service-based system.

In Chapter 3, in order to understand the nature of service-based systems, we defined a realistic e-procurement scenario and listed in detail the common problems faced by the developers which prevent them from building a reliable and robust system. Through the analysis of the common problems, we identified key principles the developers must consider to avoid producing the common problems.

In Chapter 4, based on the key principles we identified in the previous chapter, we proposed a new framework called GAT in the orchestration infrastructure. We discussed key innovative features of GAT, such as uniform processing between the normal activities and deviational events, accessing a wide range of state aspects, and many more. Using the example taken from part of the e-procurement case study, we illustrated how developers could use the GAT framework to design their business requirements. We also discussed how key features of the new framework help the developers to avoid producing consistency faults.

In Chapter 5, we defined the critical challenges that have to be addressed when designing a business process system to support the GAT model. These include implementing control flow based on the evaluation of guards, the management and distribution of events, and enforcing atomicity constraints across the evaluation of guards and the execution of the corresponding activity. We demonstrated that one can build a system following this approach and illustrated our proof-of-concept GAT prototype with code examples.

The GAT model still requires the developers to write code that handles deviations that arise from interference from concurrent activities. In Chapter 6, we provided a sophisticated unified isolation mechanism called Promises that is not only applicable to our GAT framework, but also to any applications that run in the service-based world. We discussed the concept, how it works, and how it defines a protocol.

In Chapter 7, we defined some of the implementation issues that need to be resolved in promise-based systems such as how to ensure Promise Manager takes overall responsibility and coordinates the activities in order to maintain the validity of non-expired promises. We provided a proof-of-concept prototype system showing that one can implement the promises mechanism.

For future work, we plan to build a fully general GAT engine which can take any business descriptions and turn them into executable code, as was briefly mentioned in Chapter 5. In this new version, rather than using a proprietary technology such as .NET, we plan to use Web service standards for the sending and receiving of external event messages to allow any underlying implementations to be interoperable.

We also plan to add a feature which the engine can check for various mistakes at compile time and also we wish to support a protocol which checks that consistency conditions are correct at termination time.

We are also considering building a graphical user interface to make it easier for business analysts to define a business description following the GAT model. We also plan to include optimisations in the engine to improve the performance of the compilation process itself and of the executable code it produces.

We will implement support for Promise interactions in several service-provision frameworks, including our own GAT engine and also some commercial approaches. This will involve developing further implementations for checking predicates against resources, as discussed in Chapter 7. This may need more understanding of semantics of resources involved. Some techniques studied in Ontology can perhaps be useful. As well, we plan to provide simple heuristics to choose an appropriate implementation technique for each class of resources. We also will integrate the processing of promises with other frameworks for service-oriented messaging, including the transaction support found in standards like WS-Transactions and WS-BusinessActivity.

Web 2.0 is the latest buzzword that has hit cyberspace. Though the exact definition of "what it is Web 2.0" seems to be still controversial depending on who interprets it, still it can be generally understood as referring to a next generation of internet and web-based communities which facilitate collaboration and sharing between users.

According to a white paper [79] published by the founder of Web 2.0 Tim O'Reilly, there are some key principles typically appear in the Web 2.0 applications such as web blogs, social bookmarking sites (e.g. del.licio.us), wikis, podcasts, RSS/Atom, internet forums, Web APIs, and many more. These applications use a web as a delivery platform, allowing users to use applications entirely through a browser. Architecture is designed to encourage user participations which add value to the application as they use it. They provide a rich, interactive, user-friendly interface based on Ajax or similar frameworks often providing social networking for communities of people who share interests and activities.

In [19], the author reports that the concept and technologies advocated by Web 2.0 to have many similarities with SOA computing. Both technologies encourage the autonomy of services that hides all the implementation complexity underneath and only provides open and simple access to users. Both embrace Web services and they advocate providing a new solution by the aggregation of existing functionality that crosses trust boundaries. The importance of making large, back-end database driven functionality is realized. And both Web 2.0 and SOA provide the building blocks for creating more user-centric processes where the end users can use the system without a steep learning curve.

Due to its lack of maturity with Web 2.0 being an early stage of the development, there still seems to be a lot of emphasis on connecting people and resources to form a social network among communities of people rather than using the web for business use. For this reason, the emphasis on the quality of service, such as robustness issues, seems to be often ignored from many discussions of the Web 2.0.

However, more recently, Enterprise 2.0 [19] has emerged to embrace the convergence of Web 2.0 technologies with web services and SOA to enable enterprises to deploy robust, reliable, and secure business applications over the Web. In this regard, it will be really interesting to investigate in the future how our work on GAT and Promises will find ways to provide robustness support for the Web 2.0 applications.

BIBLIOGRAPHY

[1]. Alonso, G., Abbadi, A. El., Kamath, M., Gunthor, R., Agrawal, D., Mohan, C. Failure handling in large scale workflow management systems. 1994, Technical Report IBM RJ9913.

[2]. Alonso, G., Agrawal, D., Abbadi, A. El., Kamath, M., Gunthor, R., Mohan, C. Advanced transaction models in workflow contexts. *IEEE Conference on Data Engineering*, pp 574-581, 1996.

[3]. Babara, D., Mehrotra, S., Rusinkiewicz, M. INCAs: Managing Dynamic Workflows in Distributed Environments, *Journal of Database Management, Special Issue on Multidatabases*, 7(1):5-15, 1996

[4]. BizTalk Server. http://www.microsoft.com/biztalk/

[5]. BEA Systems Inc. BEA tuxedo: The Programming Model, 1996. http://edocs.bea.com/wle/tuxedo/main/stref.htm

[6]. Birrell, A.D., Nelson, B. J. Implementing remote procedure calls. *ACM Transaction Computer Systems*, 2(1):39-59, February 1984

[7]. Borgida, A., Murata, T. Tolerating exceptions in workflows: a unified framework for data and process. *Proceedings of WACC'99*, 1999.

[8]. Business Process Execution Language for Web Services (BPEL), Version 1.0. http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/

[9]. Cabrera, L. F., Jones, M. B., Theimer, M. Herald. Achieving a Global Event Notification Service. *In Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, Elmau, Germany. IEEE Computer Society, May 2001

[10]. Carzaniga, A., Rosenblum, D.S., Wolf, A.L. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19(3):332-383, August 2001

[11]. Casati, F., Ceri, S., Parboschi, S., Pozzi, G. Specification and Implementation of Exceptions in Workflow Management Systems. *ACM Transactions on Database Systems*, vol. 24, no. 3, pp. 405-451, 1999.

[12]. Casati, F., Pozzi, G. Modeling Exceptional Behaviors in Workflow management Systems. *Proceedings of International Conference on Cooperative Information Systems (CoopIS'99)*, 1999.

[13]. Ceri, S., Grefen, P., Sanchez, G. WIDE – a distributed architecture for workflow management. *In Proceedings of RIDE'97*, pp 76-81, 1997.

[14]. Chiu, D. K. K., Li, Q. ADOME-WFMS: Towards cooperative handling of workflow exceptions. In Romaovsky, A., Dony, C.,

Knudsen, J. L., Tripathi, A. editors, *Advances in Exception Handling Technique*, pp 271-288. Springer-Verlag, LNCS-20022, 2001.

[15]. Chiu, D. K. W., Li Q., and Karlapalem, K. A Meta Modeling Approach for Workflow Management System Supporting Exception Handling. *Information Systems, vol. 24, no. 2*, pp. 159-184, May 1999.

[16]. Chiu, D. K. W., Li, Q., and Karlapalem, K. Facilitating Exception Handling with Recovery Techniques in ADOME Workflow Management System. *Journal of Applied Systems Studies, vol. 1, no. 3*, pp. 467-488, 2000.

[17]. Chiu, D.K.W., Li, Q., and Karlapalem, K. Web Interface-Driven Cooperative Exception Handling in ADOME Workflow Management System. *Information Systems, vol. 26, no. 2*, pp. 93-120, 2001

[18]. Dayal, U., Hsu, M., Ladin, R. Organizing Long-Running Activities with Triggers and Transactions. *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pp. 204-214, 1990.

[19]. Dion Hinchcliffe's Web 2.0 Blog. http://web2.socialcomputingmagazine.com/

[20]. Encina from Transarc. http://www-306.ibm.com/software/sw-atoz/

[21]. Eswaren, K., Gary, J., Lorie, R., Traiger, I. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624-633, 1976.

[22]. Eder, J., Liebhart, W. The Workflow Activity Model WAMO. *Proceedings of the International Conference on Cooperative Information Systems*, Vienna, Austria, 1995.

[23]. Fitzpatrick, G. Kaplan, S., Mansfield, T., Arnold, D., Segall, B. Supporting public availability and accessibility with Elvin: Experiences and reflections, Computer Supported Cooperative Work: *the Journal of Collaborative Computing*, 2002

[24]. Fletcher A. (ed). *GRID Transaction Management Research Group Report*. http://www.ggf.org/mail_archive/tm-rg/2006/06/doc00005.doc

[25]. Freemantle, P., Weerawarana, S. & Khalaf, R. Enterprise Services *Communications of the ACM* 45(10), 77–82. 2002

[26]. Fung, C., Hung, P. Distributed System Recovery through Dynamic Regeneration of Workflow Specification. *The 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, May 18-20, 2005

[27]. Gawlick D. and Kinkade D. Varieties of Concurrency Control in IMS/VS Fast path. *IEEE Data Engineering Bulletin*, 8(2):3-10, 1985.

[28]. Georgakopoulos, D., Hornick, M. A Framework for Enforceable Specification of Extended Transaction Models and Transactional Workflows, *Journal of Intelligent and Cooperative Information Systems*, 3(3):599-617, 1994.

[29]. Georgakopoulos, D., Hornick, M. F., Sheth, A. P. An overview of workflow management: From process modelling to workflow

automation infrastructure, *Distributed and Parallel Databases*, vol. 3, no. 2, pp. 119–153, 1995.

[30]. Geppert, A., Tombros, D., Dittrich, K. Defining the Semantics of Reactive Components in Event-Driven Workflow Execution with Event Histories. *Information Systems*. 23(3/4):235-252, 1998.

[31]. Goodenough, J.B. Exception Handling Issues and a Proposed Notation. *Communications of the ACM*, 18(12):683-696, 1975.

[32]. Gray, J. Reuter, A. Transaction Processing: Concepts and Techniques, Morgan Kaufmann Publishers, 1993.

[33]. Greenfield, P., Fekete, A., Kuo, D., Nepal, S. Consistency for Web Services Applications. *Very Large Database Conference (VLDB)*, 2005

[34]. Greenfield, P., Fekete, A., Jang, J., Kuo, D. Compensation is Not Enough. *In proceedings of the 7th IEEE International Enterprise Distributed Object Computing Conference (EDOC'03)*, pp. 232-239, Brisbane, Australia, September 2003.

[35]. Greenfield, P., Fekete, A., Jang, J., Kuo, D., Nepal, S. Isolation Support for Service-based Applications: *Third Biannual Conference on Innovative Database Systems Research (CIDR'07)*, Asilomar, USA, January 2007.

[36]. Garcia-Molina, H., Salem, K., "Sagas," *ACM International Conference on Management of Data (SIGMOD)*, pp. 249-259, 1987.

[37]. Hagen, C., Alonso, G. Exception Handling in Workflow Management Systems. *IEEE Transactions on Software Engineering*, 26(10):943-958, October 2000.

[38]. Hagen, C., Alonso, G. Flexible Exception Handling in the OPERA Process Support System. *IEEE International Conference on Distributed Computing Systems*, pp 526-533, 1998.

[39]. Hagen, C., Alonso, G. Beyond the Black Box: Event-Based Inter-process Communication in Process Support Systems. *IEEE International Conference on Distributed Computing Systems*, pp 450-457, 1999.

[40]. Helland, P. Data on the Outside versus Data on the Inside. *Conference on Innovative Database Systems Research (CIDR'07)*, pp 144-153, Asilomar, USA, January 2005.

[41]. High Performance Counter. http://support.microsoft.com/kb/q172338/

[42]. Horswill, J., Miller, S. Designing and Programming CICS Applications. O'Reilly & Associates, 2000.

[43]. Hwang, S. Y., Ho, S. F., and Tang, J. Mining exception instances to facilitate workflow exception handling. *Proceedings of the 6th International Conference on Database Systems for Advanced Applications*, pp. 45-52, 1999

[44]. IBM. WebSphere MQ Integrator Broker: Introduction and Planning, June 2002.

[45]. Jang, J., Fekete, A., Greenfield, P., Kuo, D. Expressiveness of Workflow Description Languages. *International Conference on Web Services(ICWS)*, Las Vegas, USA, June 2003
[46]. Jang, J., Fekete, A., Greenfield, P., Nepal, S. An Event-Driven Workflow Engine for Service-Based Business Systems. *Conference on the Enterprise Computing (EDOC)*, pp 233-242, Hong Kong, China, October 2006
[47]. Jang, J., Fekete A., Greenfield, P. Delivering Promises for Web Service Applications. Technical Report of University of Sydney School of Information Technologies, TR-605, December 2006.
[48]. Java Messaging Service. http://java.sun.com/products/jms/
[49]. Knolmayer, G., Endl, R., Pfahrer, M. Modeling Processes and Workflows by Business Rules. *In Business Process Management, ed W. van der Aalst,* LNCS 1806, pp 16-29, 2000.
[50]. Koschel, A., Kramer, R. Applying Configurable Event-triggered Services in Heterogeneous, Distributed Information Systems. *Engineering Federated Information Systems Workshop EFIS'99*, Kühlungsborn, Germany, pp 147-157, May 5-7, 1999.
[51]. Krishnamoorthy, V., Shan, M. Virtual transaction model for workflow applications. *International Symposium on Applied Computing (SAC)*, Como, Italy, March 2000.
[52]. Kuo, D., Fekete, A., Greenfield, P., Jang, J. Towards a Framework for Capturing Transactional Requirements of Real Workflows. *International Workshop on Cooperative Internet Computing,* pp. 113-122.
[53]. Kuo, D., Fekete, A., Greenfield, P., Jang, J. Just What Could Possibly Go Wrong In B2B Interaction? *International Computer Software and Applications Conference(COMSAC),* Dallas, USA, November 2003
[54]. Kuo D., Gaede V. and Taylor K. Using Constraints to Manage Long Duration Interactions in Spatial Databases. *CoopIS'98*, pp 383-395, 1998.
[55]. Leymann, F. Supporting Business Trnasactions via Partial Backward Recovery in Workflow Management Systems, *GI-Fachtagung Datenbanken in Buro Technik und Wissenchaft*, 1995.
[56]. Luo, Z., Sheth, A., Kochut, K., Miller, J. Exception Handling in Workflow Systems. *Applied Intelligence* 13(2):125-147, September 2000.
[57]. Mehrotra, S., Rastogi, R., Silberschatz, A., Korth, H. A Transaction Model for Multidatabase Systems. *International Conference on Distributed Computing Systems.*
[58]. Microsoft BizTalk. http://www.microsoft.com/biztalk/default.mspx
[59]. Microsoft CodeDOM. http://msdn2.microsoft.com/en-us/library/y2k85ax6.aspx
[60]. Microsoft Corporation. Message Queuing in Windows XP, 2001.
[61]. Model Driven Architecture (MDA). http://www.omg.org/mda/

[62]. Mourani, H., Antunus, P. Exception Handling Through a Workflow. *CoopIS 2004*, pp. 37-54, 2004.

[63]. MSDB .NET Framework Developer's Guide Using CodeDOM. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconUsingCodeDOM.asp

[64]. Muller, R., Greiner, U., Rahm, E. AGENTWORK: a workflow system supporting rule-based workflow adaptation. *Data and Knowledge Engineering*, 51(2):223-256, November 2004.

[65]. Naur, P. "Revised Report on the Algorithmic Language ALGOL 60.", *Communications of the ACM*, Vol. 3 No.5, pp. 299-314, May 1960

[66]. Ng, A., Chen, S., Greenfield, P. An evaluation of contemporary commercial SOAP implementations. *Workshop on Software and System Architectures*, pp64-71, June 2004

[67]. OASIS UDDI Specification. http://www.uddi.org/faqs.html

[68]. Object Management Group. BPMI.org http://www.bpmi.org/

[69]. Object Management Group CORBA Notification Server Specification 1.1 http://www.omg.org/technology/documents/formal/notification_service.htm

[70]. Object Management Group. Unified Modeling Language Specification (Version 1.3), June 1999.

[71]. Object Management Group. CORBAservices: Common Object Services Specfications, 1997.

[72]. The Open Group. Transaction Processing Titles. The Open Group. http://www.opengroup.org/products/publications/catalog/tp.htm

[73]. O'Neil P. The Escrow Transactional Methods. *ACM TODS*. 11(4):405-430, 1986

[74]. Petlz, C. http://devresource.hp.com/drc/technical_white_papers/WSOrch/WSOrchestration.pdf

[75]. Puustj¨arvi, J., Laine, H. WorkMan — A Transactional Workflow Prototype. *In Database and Expert Systems Applications*, pp 212–221. Springer, 2000.

[76]. Sheth, A., Georgakopoulos, D., Joosten, S., Rusinkiewicz, M., Scacchi, W., Wileden, J., Wolf, A. Report from the NSF *Workshop on Workflow and Process Automation in Information Systems*, Technical report, University of Georgia, UGA-CS-TR-96-003, 1996.

[77]. Stonebraker, M. Implementation of integrity constraints and views by query modification. *ACM SIGMOD Conference*, pp 65-78, 1975.

[78]. Sheard T. and Stemple D. Automatic Verification of Database Transaction Safety. *ACM TODS* 14(3):322-368, 1989.

[79]. Tim O'Reilly. What is Web 2.0 white paper. http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html

[80]. Tombros, D., Geppert, A. Building Extensible Workflow Systems using an Event-Based Infrastructure. *CAiSE'00*, pp 325-339

[81]. Urban, S. D., Kambhampati, S., Dietrich, S. W., Jin, Y., Sundermier, A. An Event Processing System for Rule-Based Component Integration, *International Conference on Enterprise Information Systems*, Portugal, pp.312-319, April, 2004,

[82]. W3C SOAP Specification. http://www.w3.org/TR/soap/

[83]. W3C WSDL Specification. http://www.w3.org/TR/wsdl

[84]. Wachter, H., Reuter, A. The ConTract Model, *in Database transaction Models for Advanced Applications (edited by A. Elmagarmid)*, pp. 219-263. Reprinted in 'Readings in Database Systems, 3rd edition' (edited by M. Stonebraker and J. Hellerstein), 1992.

[85]. WebMethods. MebMethods Enterprise Integrator: User's Guide, 2002.

[86]. WebSphere MQ Workflow. http://www-3.ibm.com/software/integration/wmqwf/

[87]. Web Service Choreography Interface (WSCI) 1.0 Specification. http://wwws.sun.com/software/xml/developers/wsci/

[88]. Weikum, G. Extending Transaction Management to capture more Consistency with better Performance, *French Database Conference*, pp. 27-30, 1993.

[89]. WWW WS-Events Version 2.0 http://devresource.hp.com/drc/specifications/wsmf/WS-Events.pdf

[90]. Widom, J., Ceri, S. Active Database Systems: Trigger and Rules For Advances Database Processing. Morgan Kaufmann, 1995

[91]. Wise. A. Little-JIL 1.0: Language reports. Technical Report UM-CS-1998-024, University of Massachussets, Amherst, MA, USA, 1998.

[92]. WS-AtomicTransaction Specification. ftp://www6.software.ibm.com/software/developer/library/WS-AtomicTransaction.pdf

[93]. WS-BusinessActivity Specification. ftp://www6.software.ibm.com/software/developer/library/WS-BusinessActivity.pdf

[94]. WS-Coordination Specification. ftp://www6.software.ibm.com/software/developer/library/WS-Coordination.pdf

[95]. WSFL specification. http://www4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf

[96]. XLANG specification. http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm

[97]. Zhao, W., Moser, L., Melliar-Smith, M. A Reservation-based Coordination Protocol for Web Services. *Proceedings of ICWS'05*, pp 49-56, 2005.