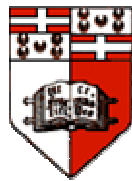


A Meta Model for Multiple Transaction Models

Final Year Project



University of Malta

Board of Studies for Information Technology

Department of Computer Science and Artificial Intelligence

Justin Spiteri

B.Sc. I.T. (Hons.) Year 4

May 2006

Declaration

Plagiarism is defined as "the unacknowledged use, as one's own work, of work of another person, whether or not such work has been published" (Regulations Governing Conduct at Examinations, 1997, Regulation 1 (viii), University of Malta).

I / We*, the undersigned, declare that the [assignment / Assigned Practical Task report / Final Year Project report] submitted is my / our* work, except where acknowledged and referenced.

I / We* understand that the penalties for making a false declaration may include, but are not limited to, loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected, and will be given zero marks.

* Delete as appropriate.

(N.B. If the assignment is meant to be submitted anonymously, please sign this form and submit it to the Departmental Officer separately from the assignment).

Student Name

Signature

Course Code

Title of work submitted

Date

Abstract

The main aim of this report is that of providing a solution which caters for a particular area of the field of transaction management; long running transactions. The inspiration for this thesis was mainly the fact that after an in depth research was carried out, a series of shortcomings in current solutions was noticed.

The main issue found is that current software solutions mainly adopt one particular transaction model, and thus a software solution is tied down to providing only one type of transaction management service. This greatly reduces the range of applications which may make use of the software solution, and is considered to be the main issue to be addressed in this thesis: A solution which may provide a transaction management service using multiple transaction models.

A secondary issue which has been identified and tackled includes the high complexity involved in building software which uses current transaction management solutions. An effort has been made to create a simple solution which is easy to understand and integrate into the needed applications. Techniques used in order to achieve this include the use of the globally renowned XML language, and the introduction of open source concepts into the proposed solution.

The artifact accompanying this dissertation includes the Implementation of the Transit Model Solution, which is an open source transaction management system designed with the intent of solving the issues identified in the current solutions. Ample example applications are also included on the disk.

While the implementation is fully functional, its main purpose is that of a working prototype, which proves the novel concepts proposed in this thesis.

Acknowledgements

Firstly, I would like to thank my supervisor, Mr. Patrick Abela, who gave me inspiration to choose Long Lived Transactions as my main research area, together with all the necessary professional and technical support in a real world working environment to make the project a successful one.

Secondly, I would like to thank Dr. Marek Prochazka (Charles University, Czech Republic) and Mr. Mark Little (Hewlett Packard/Arjuna Technologies), for remotely assisting me and exhaustively answering all the queries which I posed to them.

Finally, I would like to thank Mr. Matthew Rizzo, Mr. Mark Herrera, Mr. Jeremy Ellul, Mr. Christian Tabone, Ms. Maria Aquilina, Mr. Michael Usatchev, and all my family and friends, who gave me invaluable assistance and support throughout the whole Research & Development Process of the project.

Table of Contents

Chapter 1: Introduction.....	12
1.1 Overview	12
1.2 Aims	12
1.3 Non Aims	13
1.4 Document Structure.....	13
Chapter 2: Literature Review	15
2.1 Transaction Theory.....	15
2.1.1 What is a transaction?	15
2.1.2 Types of Transactions	15
2.1.3 Transaction Modeling	17
2.1.4 Transaction Models	18
2.1.5 Common Properties of Transaction Models.....	30
2.1.5.1 Transaction Modularization.....	31
2.1.5.2 RollBack and Recovery Concepts	32
2.1.5.3 Transaction States.....	33
2.1.5.4 Transaction Contexts	35
2.1.5.5 Transaction Inter Dependencies	35
2.2 Existing Transaction Management Solutions.....	37
2.3 Drawbacks of Current Solutions	39
2.3.1 Non Technical Issues	40
2.3.1.1 Choosing a Solution.....	40
2.3.2 Technical Issues	41
2.3.2.1 Separating Transaction Models from Transaction Management Systems	41
2.3.2.2 Standards in Long Running Transactions.....	41
2.3.2.3 ACID Principles in Long Running Transactions	42
2.4 Real Life Scenarios	43
2.4.1 E-Top Up Facility.....	43
2.4.2 The Travel Agent Facility (Referenced from JSR 95)	43
2.4.3 Case Study Analysis – Interdependencies & Workflow.....	44
2.4.4 ACID is good – take it in short doses!	46

2.5	The Problem	48
2.6	Motivation	49
Chapter 3: Requirements & Specification		50
3.1	Introduction	50
3.2	General Overview	50
3.3	Project Requirements.....	51
3.4	Transit Model Solution Specification	52
3.4.1	Semantics	52
3.4.2	Identification of Project Modules	52
3.4.3	Transit Scripting Language Specification	54
3.4.4	Transit Model API	55
Chapter 4: Architectural Concepts		57
4.1	Introduction	57
4.2	General Architecture	57
4.3	Architectural Concepts	58
4.3.1	Transaction Modularization	58
4.3.2	Activity/LLT Transaction States	59
4.3.3	Transaction Contexts – Definition & Propagation	60
4.3.4	Transaction Inter Dependencies.....	61
4.3.5	Transaction Workflow Generation.....	62
4.3.6	Suspend/Resume Enabled Pluggable Model Architecture	62
4.4	Transaction Handling	63
Chapter 5: The Transit Scripting Language		65
5.1	Introduction	65
5.2	Script Structure Considerations	65
5.3	Language Structure	65
5.4	Script Constructs	68
5.4.1	Script Constructs: The Model Tag.....	69
5.4.2	Script Constructs: Name Tag	70
5.4.3	Script Constructs: Global/Local Declaration Tag.....	71
5.4.4	Script Constructs: ActivityList Tag	71
5.4.5	Script Constructs: Counter Tag	72
5.4.6	Script Constructs: WorkFlow Tag.....	73
5.4.7	Script Constructs: Segment Tag	74
5.4.8	Script Constructs: Begin Tag.....	75
5.4.9	Script Constructs: For Do Tag	75
5.4.10	Script Constructs: If Then and Else If Tags	76
5.4.11	Script Constructs: Execute Tag	78
5.4.12	Script Constructs: Goto Tag	79

5.4.13	Script Constructs: CMD Tag	80
5.4.14	Script Constructs: Main Tag	81
5.5	Examples	81

Chapter 6: The TransitModel API84

6.1	General Architecture	84
6.2	TransitModel.Structure.....	84
6.2.1	TransitModel.Structure – Use Case	85
6.2.2	TransitModel.Structure – Class Diagram.....	86
6.2.3	TransitModel.Structure.Activity.....	87
6.2.4	TransitModel.Structure.ActivityInfo	89
6.2.5	TransitModel.Structure.LLT	90
6.2.6	TransitModel.Structure.Info	90
6.3	TransitModel.TManager.....	91
6.3.1	TransitModel.TManager.Logic	92
6.3.1.1	TransitModel.TManager.Logic.Mgr	92
6.3.1.2	TransitModel.TManager.Logic.TransitControlPanel	93
6.3.1.3	TransitModel.TManager.Logic.Coordinator	93
6.3.2	TransitModel.TManager.LanguageBlocks.....	94
6.3.2.1	TransitModel.TManager.LanguageBlocks.IBlock	94
6.3.2.2	TransitModel.TManager.LanguageBlocks.Structs	95
6.3.2.3	TransitModel.TManager.LanguageBlocks.Main	95
6.3.2.4	TransitModel.TManager.LanguageBlocks.Execute.....	95
6.3.3	Concepts - Language Parsing & Workflow Generation	96
6.3.3.1	Language Blocks	96
6.3.3.2	Flow Lists	99
6.3.3.3	Parameter Passing.....	99
6.3.3.4	Workflow Generation Logic	100
6.3.3.5	The Model Object.....	103
6.3.4	Concepts - Execution.....	104
6.3.4.1	Parameter Passing Revisited.....	104
6.3.4.2	Accessing Variables Revisited	104
6.3.4.3	Variable Expression Evaluation	107
6.3.4.4	State Switching	107
6.3.4.5	Suspension and Resumption of an LLT	108
6.3.4.6	Suspension of an LLT – State Tracking.....	109
6.3.4.7	Resumption of an LLT – Activity Execution Simulation	110

Chapter 7: Conclusion113

7.1	Producing an Integrated Solution.....	113
7.2	LLT Enabling a Typical Application using Transit	113
7.3	Transit Model Solution as an Open Source Project.....	114
7.4	Transit Model Solution Assumptions and Limitations.....	116

7.5	Evaluation	117
7.6	Future Work.....	118
7.7	Final Remarks	118
Appendix A: Glossary		120
Appendix B: Class Diagram Listing		124
Appendix C: Transit Script Examples		127
A.)	JSR 95 LLT Transaction Model (Ixaris Implementation)	127
B.)	A Custom SAGA Model	128
Appendix D: Transit API Usage Instructions.....		131
Appendix E: Example - Transit Enabled version of Skype .		132
A.)	Introduction	132
B.)	Application Design & Implementation.....	132
C.)	Transaction Management.....	137
D.)	Choosing a Model	141
E.)	The Result	141
Appendix F: Example - Transit Enabled Holiday Planner...		146
Appendix G: Transit API Testing		148
A.)	White Box Testing	148
	Test 1: Code Walkthrough: Model Object Creation.....	148
	Test 2: Code Walkthrough: Model Object Execution: Normal	149
	Test 3: Code Walkthrough: Model Object Execution: Suspend	150
	Test 4: Code Walkthrough: Model Object Execution: Resume	152
B.)	Black Box Testing	153
	Test List: Exhaustive Testing.....	153
Appendix H: Application Requirements & CD Contents.....		155
A.)	Application Requirements.....	155
B.)	CD-ROM Contents.....	156
Appendix I: SourceForge Details		157
A.)	The SourceForge Application Form:	157

B.)	The SourceForge Approval E-Mail:	158
C.)	The Transit SourceForge Web Site:	160

Appendix J: Bibliography & References.....161

A.)	Bibliography & References.....	161
B.)	Correspondence	166

Appendix K: Correspondence.....167

A.)	Michael Usatchev – Moscow Computer Science Academy.....	167
B.)	Mark Little – Arjuna Technologies	168
C.)	Marek Prochazka – Charles University Czech Republic	172

Figures and Tables

Figure 2.1.4.1 Two Phase Commit.....	18
Figure 2.1.5.1.1 Long Running Transactions and Activities	32
Figure 2.1.5.2.1 Pseudocode For a Generic Workflow Example	32
Figure 2.1.5.2.1 Two Phase Commit State Changes.....	34
Figure 2.1.5.2.2 Transaction States.....	34
Figure 2.1.5.5.1 Nested Model	36
Figure 2.1.5.5.1 Split/Join Model	36
Figure 2.1.5.5.3 Hybrid Model.....	36
Figure 2.4.3.1 Skype E-Topup System	44
Figure 2.4.3.2 Flight Booking System (Ref JSR95)	45
Figure 2.4.4.1 SAGA Transaction Model Descriptor	47
Figure 3.4.2.1 Use Case Diagram – Transit Model Solution Integrated Example	53
Figure 4.2.1 General Architecture.....	57
Figure 4.3.2.1 State Transition Diagram	60
Figure 4.3.2.2 Activity States Example.....	60
Figure 4.3.3.1 Activity Logic Pseudocode	61
Figure 4.3.6.1 FlowChart : The Nested Model.....	63
Figure 5.3.1 Transit Script Template Preview	66
Figure 5.3.2 "N" Based Expression Example	67
Figure 5.3.3 Custom XML Based Constructs	68
Figure 5.5.1 Nested Model Transit Script	83
Figure 6.1.1 Package Diagram – The Transit Model	84
Figure 6.2.1.1 Use Case for TransitModel.Structure	85
Figure 6.2.2.1 Class Diagram for TransitModel.Structure.....	86
Figure 6.2.3.1 Pseudocode for an Activity Workflow	87
Figure 6.2.3.2 Pseudocode For an Activity Method.....	88
Figure 6.3.1 Package Diagram for TransitModel.TManager	91
Figure 6.3.2 Component Diagram for TransitModel.TManager	92
Figure 6.3.2.1.1 Class Diagram for the IBlock Component.....	94
Figure 6.3.2.3.1 Class Diagram for the Main Language Block.....	95
Figure 6.3.2.4.1 Class Diagram for the Execute Language Block.....	96
Figure 6.3.3.1.1 Structure of the Workflow Language Block	97
Figure 6.3.3.1.2 Table for Transit Script Tag Classification	98
Figure 6.3.3.1.3 Sequence Diagram for Workflow Generation.....	98
Figure 6.3.3.4.1 FlowChart : Workflow Generation Part 1.....	100
Figure 6.3.3.4.2 FlowChart : Workflow Generation Part 2.....	101

Figure 6.3.3.4.3 FlowChart : Workflow Generation Part 3.....	102
Figure 6.3.3.5.1 Resulting Model Object – Workflow Tree Structure.....	103
Figure 6.3.4.2.1 Parameter Passing in the Transit Model Solution	105
Figure 6.3.4.2.2 Pseudocode for getVariables() and setVariables() methods...	107
Figure 6.3.4.6.1 Class Diagram for StateHolder Class.....	109
Figure 6.3.4.6.2 FlowChart : Suspension of an LLT	110
Figure 6.3.4.7.1 FlowChart : Resumption of an LLT	111
Figure 6.3.4.7.2 Screen Shot for the Transit Control Panel Resume GUI.....	112
Figure B1 TransitModel.Structure Class Diagram	124
Figures B2 & B3 TransitModel.TManager Class Diagrams	126
Figure CA1 JSR 95 LLT Transaction Model	128
Figure CB1 Custom Try Catch Saga	130
Figure EA1 Use Case: Transit Enabled Skype E - Top Up.....	132
Figure EB1 Transit Enabled Skype Top Up Architecture.....	133
Figure EB2 Architectural Changes to an Application	134
Figure EB3 CheckAllowedTopup Extending from the Activity Class	135
Figure EB4 CheckAllowedTopup Run Method	135
Figure EB5 Adding the TransitModel References.....	136
Figure EB6 Creating the Long Running Transaction	137
Figure EC1 Running the Transaction.....	138
Figure EC2 Instantiating the Transit Resume GUI.....	139
Figure EC3 The Transit Model Resume GUI Instance	140
Figure ED1 Choosing a Script for the Skype Topup Application	141
Figure EE1 Transit Enabled Skype Top Up Main Form	142
Figure EE2 Skype Top Up Manager Form - Idle	142
Figure EE3 Skype Top Up Form – New Transaction Started.....	143
Figure EE4 Skype Top Up Form – Suspended + Transit Resume GUI	143
Figure EE5 Skype Top Up Form – Resumed/Running	144
Figure EE6 Skype Top Up Form – Resumed/Running	144
Figure EE7 Skype Top Up Form – LLT Committed.....	145
Figure EE8 Skype Top Up Form – LLT Compensating.....	145
Figure F1 Holiday Planner Form – LLT Model	146
Figure F2 Holiday Planner Form 2 – LLT Model.....	147
Figure GA1 Connection Error Simulation	151
Figure IA1 The SourceForge Application Form.....	157
Figure IB1 The Transit Project’s Sourceforge Site	160
Figure IB2 The Transit Project’s Sourceforge Utilities.....	160

Chapter 1: Introduction

This chapter provides a general overview to the thesis, by introducing the reader to the main research area, and defining project aims and non aims. A detailed review of the organisation of the rest of the document is also provided.

1.1 Overview

This project can be considered as consisting of research and development in the field of transaction management, where the main area which is tackled is that of long running transactions.

There is are a number of supporting theories, concepts, models and frameworks that exist for developers to create transaction enabled applications. If however a developer needs to create a solution which handles complex transactions including multiple parties, possibly spanning over a long period of time, various difficulties may arise. The developer must have sound knowledge of advanced transaction theory in order to deal with such a situation and learning transaction theory is a very time consuming process, thus not being feasible.

This problem has led to the research and development of middleware solutions which abstract transaction management issues from developers. Various transaction management systems are currently available, each with distinct features, advantages and disadvantages. This thesis analyses a selection of systems from both a theoretical and a practical point of view in an effort to produce a solution which eliminates the shortcomings of the current systems.

1.2 Aims

The main aim of this thesis is that of providing an innovative solution which aids developers, in the creation of transaction enabled software systems through the concepts of flexibility and ease of development. The process to achieve this aim is fourfold, namely including:

- An in depth research in the field transaction theory, concentrating particularly on the area of long running transactions.
- Provision of a detailed analysis of existing solutions, or solution specifications. These may include both commercial and academic solutions.
- Proposal of novel concepts and ideas which enhance development of transaction enabled applications.
- Full design and prototypical implementation of the Transit Model Solution, which is a novel open source transaction management system, housing concepts which are either novel or derived from the research carried out. The main aim for the Transit Model Solution is that of providing a simple solution which allows developers to create transaction enabled applications, without the need of having expert knowledge in transaction management.

1.3 Non Aims

This thesis has a purely educational nature, and thus it is not aimed at any form of market, nor does it have a commercial purpose. Besides, while typical commercial transaction handling systems are usually represented as physically distributed middleware, it has been felt that physical distribution in this case is out of scope. Effort on creating a distributed system would offset the main focus of this thesis, leaving less time for research and development efforts on the project's actual aims and targets.

1.4 Document Structure

This document is divided into eight main chapters which offer complete coverage of the project, from a brief overview to a detailed theoretical insight, to implementation information and future work. The chapters are classified as follows:

This chapter introduced the reader to the subject, giving very high level information about the main area of research, and the aims of the project.

Chapter two caters for the theoretical aspect of the project. It has been assumed that the reader possesses only basic knowledge about transactions in

general, thus an ample literature review has been included in Chapter two. This ensures coverage of basic transactional concepts, different types of transactions present, transaction models and modeling techniques and associated technologies currently in use on the market. This research also includes an ongoing problem definition as the research builds up, culminating in a motivation section which describes the issues present in current transaction theory and solutions and proposes a solution, in an effort to solve these issues.

On the other hand, chapters three to six describe the conceptualization, development and deployment of the proposed solution in a considerable amount of detail, providing enough information for the developer reading this document to use the solution in his own projects. Ample reference to a particular example where the solution may be typically applied is also made.

Chapter seven includes a representation of the completed solution, together with instructions on how to apply the integrated solution to a project. Future work proposals of the project are also present, together with a project conclusion in Chapter 8.

A series of Appendices is present at the end of the document, which are referenced throughout the project. These include various types of material, ranging from a set of glossary terms, which further explain the terminology used in the project, to detailed class diagrams which further explains implementation of the solution. A practical example application which makes use of the Transit Model Solution is also present in the Appendices, together with testing procedures, results, and research correspondence material which had been archived.

Chapter 2: Literature Review

This chapter provides both an in depth review of transaction theory and an analysis of a selection of proposed transaction management system specifications and implemented solutions. The direct result of this analysis is the definition of the requirements of the Transit Model Solution.

2.1 Transaction Theory

2.1.1 What is a transaction?

A transaction in general, may be defined as a dedicated business oriented interaction between two or more parties, in which all stakeholders involved will be affected in some way. A more technical definition of a transaction can be found on by Microsoft's MSDN web site, which claims the following:

"A transaction is a set of one or more related tasks that either succeed or fail as a unit. In transaction processing terminology, the transaction commits or aborts." [32]

Microsoft's definition brings us closer to the realm of electronic transaction processing. Taking a practical example, if one tries to carry out a money transfer operation from one bank account to another, the whole transaction is made up from a series of smaller operations, which must all succeed in order for the transaction in general to take place. More detail and real life scenario descriptions are provided in the following sections.

2.1.2 Types of Transactions

As mentioned in Microsoft's definition, transactions are "sets of one or more related tasks". A transaction is thus an action containing multiple tasks which are coordinated to commit or to rollback any changes made to a body of data. In a traditional transaction, this is done within the context of ACID properties. In a long-lived transaction, although desirable, it may be difficult to maintain such

ACID properties over a long period of time. From a business management point of view, transactions are typically classified as being either Business to Consumer (B2C) transactions, or Business to Business (B2B) transactions, where both B2B and B2C transactions may be either atomic or long running.

- **Atomic Transactions**

As Mike Chapple says in his article entitled "Your Guide to databases" [36]; *"The concept of atomic transactions is based on one of the oldest but still relevant concepts of database theory, that is, the idea of ACID properties."* Acid properties give transactions atomicity, thus creating distinctions between one transaction and another, consistency, as in, results are either a complete success or complete failure of the transaction; there are no middle ways, isolation, which makes sure that in case of multiple transactions taking place at the same time do not impact each other's operation, and durability, which refers to the fact that any transaction which has been committed cannot be rolled back. In order to adhere to these properties, any resources which are shared between multiple transactions must be protected, and thus locked when in use by one user. A typical Atomic transaction, being ACID based, takes a short amount of time to complete, and is usually based on a "commit or reject" philosophy.

- **Long Running Transactions**

Long running transactions have a higher degree of complexity than Atomic transactions, due to the fact that a single long running transaction can be made up of several stake holders, potentially lasting hours or even days. This length of time makes the resource locking manifested in acid based transactions inappropriate, since situations can arise where all resources are blocked, with no conclusive transactions, as they all wait for each other to free resources in a massive inter-networked deadlock. Besides, in a long running transaction, partial roll back of a part of the transaction may be needed, thus invalidating the concept of the scoping mechanism present in ACID based transactions, which provides the "all or nothing" semantics. These differences present various implications when trying to build software models which handle these types of transactions.

2.1.3 Transaction Modeling

Transaction modeling is the process in which, a real world transaction or set of transactions are modeled into a business workflow and applied to a particular transaction context (see following section for definition of context). The result of this modeling process is the transaction model, which represents a formalized way in which an atomic or long running transaction can be carried out.

Most transaction management systems present today are based on the traditional two phase commit Transaction Model (see 2.1.4 for specification), which caters for Atomic Transactions. These systems are often completely ACID oriented, however, as Mark Little, from HP-Arjuna Technologies states in one of his online articles;

"The structuring mechanisms available within traditional atomic transaction systems are sequential and concurrent composition of transactions. These mechanisms are sufficient if an application function can be represented as an atomic, short lived transaction." (Acid is good – Take it in short Doses) [5]

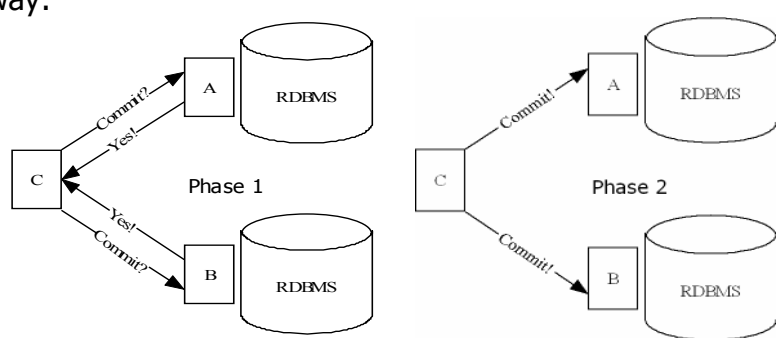
In simple terms, this statement refers to the fact that the transaction handling facilities present as at date are able to cater adequately only for transactions of an atomic nature. Consider a typical flight booking system as an example. The client issues a ticket request to the airline company, and the company either commits or denies the client's request. However when it comes to long running transactions, this mechanism based strictly on ACID properties is just not adequate.

A practical example of the inadequacy of strict ACID based long running transactions would be if one expands the flight booking system example into a full travel agent system, where one may book a flight, train, taxi service, or even hotel. If a client initiates a long running transaction where he wants to book both a hotel room and an air ticket, the intricate dependencies between the two transactions involved possess a much higher level of complexity than that of two separate transactions, where a client first books an air ticket; and then a hotel room, in two separate processes. What happens if the air ticket is committed, but the hotel booking rolls back? Should a compensating mechanism be introduced? How should a long running transaction recover? It can be seen that long running transactions must have a specialized mechanism which caters specifically for them, since a strict ACID based model such as the two phase commit does not have the necessary logic to handle such transactions. Thus the need for more advanced transaction models which is capable of handling these issues is felt.

2.1.4 Transaction Models

Transaction modeling can be considered to share its roots with database theory since the two phase commit model, which is one of the oldest models, is directly based on database ACID concepts. There is a vast amount of transaction models present today, some extending ACID models and some which have been redesigned from the ground up. These can be categorized into various sets, according to their different nature and properties. Below is a comprehensive list of the standard, most commonly known transaction models categorized into their various sets. Most of these models are currently available as specifications on www.omg.org.

○ Traditional Transaction Models

Model	Two Phase Commit (2PC/Atomic Model)
Orientation	ACID/Atomic Transactions
Released	1980
Description	<p>Being one of the oldest transaction models around, it is still the most powerful ACID based model, upon which various other models are built, including transaction models which cater for Long Lived Transactions. It provides the typical ACID "all or nothing" semantics, together with isolation in the case of compound parallel transactions. Variants of the 2PC model include basic rollback and recovery facilities, in case compound sets of atomic transactions fail. A typical two phase commit model could be graphically represented in the following way:</p>  <p>Figure 2.1.4.1 Two Phase Commit</p> <p>In the first phase, process 'C' sends a commit request to 'A' and 'B'. At this point in time, both 'A' and 'B' are</p>

	locked, and no other transactional process can use them. In the original 2PC algorithm, there is no time restriction about time delay between the execution of the first and second phases, and this may result in resources being locked for too long, creating a starvation situation. While this model ensures atomicity between multiple participants, this same atomicity is applied too strongly in the model, thus making it impractical for long lived transactions. (Reference: www.webservices.org article: The Smogasboard of Web Services Transactions – Mark Little)[3]
Pros	<ul style="list-style-type: none"> • Effective for Short Lived Transactions • Offers “all or nothing” semantics (ACID Properties)
Cons	<ul style="list-style-type: none"> • Inadequate for Long Lived Transactions • Is prone to excessive resource locking, thus making it not scaleable, this is due to its ACID-ity being too strong.

○ Advanced Transaction Models

The ACTA model specification defines advanced transaction models in the following way:

"An advanced transaction consists of either a set of operations on objects that execute atomically in a predefined order, or a set of extended transactions with an explicitly given control related to the notions of visibility, consistency, recovery, and permanence." [19]

Thus, advanced transaction models are typically made up from multiple traditional model based Units of Work. Following is a listing of the models which belong to this category.

Model	Nested Transaction Model
Orientation	Long Running Transactions – Extension of ACID
Released	1980's – Used in the ANSA project in 1993
Description	Nested transaction models are an extension of the traditional ACID based model which are capable of handling long lived transactions. A nested transaction consists of a tree of atomic transactions, starting with a root which has child transactions. Child transactions can in turn be parents to other sub-transactions. From an external point of view, the tree can be considered as one atomic transaction which follows strict ACID properties.

	The root transaction encapsulates all the tree structure, making this view possible. On the other hand, children still manifest atomic properties and are still isolated from each other, each having an individual outcome; commit or abort. However, a dependency on each child's parent is present since if the child transaction commits, the result is inherited by the parent, which is in turn inherited by the root or by other children if needed. If a parent with multiple children aborts, its children which had committed will also be aborted, breaking the ACID consistency rule. Finally, all results will recurse to the root, and a final commit or abort will be issued.
Pros	<ul style="list-style-type: none"> • Provides efficient long lived transaction handling • Promotes modularity, and concurrent execution of transactions.
Cons	<ul style="list-style-type: none"> • Does not cater very efficiently for durability.

Model	Saga Transaction Model
Orientation	Long Running Transactions – ACID model variant
Released	1980's – Used in the ANSA Project 1993
Description	SAGA based models are designed to specifically cater for long running transactions. This is done by breaking down a complex transaction scenario into various units of work which consist of recoverable and compensating actions. A typical SAGA based model uses backward compensation semantics, where if a Unit of Work fails, the system is returned to the state before executing that particular unit of work. Since SAGAS may recursively consist of multiple SAGAS, if one saga aborts, all its committed children must roll back and take recovery actions in reverse order of execution. Thus the nature of SAGA's is that of providing atomicity for long lived transactions at the cost of the loss of isolation and consistency. The majority of transaction models currently available are based on SAGA semantics, or variants of it. The lowest level Units of work of a SAGA model may essentially be a strict ACID based transaction.
Pros	<ul style="list-style-type: none"> • Provides long lived transaction handling. • Approximates atomicity for long lived transactions.
Cons	<ul style="list-style-type: none"> • Does not cater for isolation between Units of Work. • Difficult to conceptualize.

Model	Split Join Transaction Model
Orientation	Long Running Transactions (Originally intended for CAD)
Released	1980's – Used in the ANSA Project 1993
Description	The split join transaction model was originally intended for CAD/CAM purposes. It literally allows a transaction to split into multiple sub transactions, which may or may not be strictly ACID based. Split join models also allow separate transactions to merge into one parent transaction, thus giving it the possibility to be treated as one Atomic transaction. In fact, a typical split join model based activity starts with one atomic transaction which spawns off multiple child transactions, each committing or aborting. These then merge back into their parents until just one atomic transaction remains, containing the final result. The child processes may be either serial, where their execution is sequential, or independent, where they can be executed completely separately from each other.
Pros	N/A (Due to CAD/CAM Nature)
Cons	N/A (Due to CAD/CAM Nature)

○ True Advanced Transaction Models

This category includes transaction models which make use of established advanced transaction models with the addition of various enhancements and variations, such as Unit of Work interdependency determination parameters. This however means that the core concepts of operations of operation are similar in context to that of advanced transaction models.

Model	ACTA Model/Framework
Orientation	Short/Long Running Transactions
Released	1990 By Chrysanthis and Ramamritham
Description	<p>The ACTA model is based on the unification of the split join, nested and cooperative transaction models. In their specification paper, Chrysanthis and Ramamritham define ACTA as being a framework which extends the functionality of the amalgamation of these models. This allows solutions to include hybrid custom models which manifest unique behaviour rather than a simply new transaction model. What ACTA does is mainly :</p> <p><i>"allow the definition of structure, and behavior of transactions", and provides;</i></p>

	<p><i>"reasoning for the concurrency and recovery semantics of the transactions".</i></p> <p>It can be considered more of a framework of models, rather than just another model. The core of the ACTA model semantics concentrates on the effects of a Transaction on either another Transaction or an Object. These effects may include interdependencies between transactions, conflicts, and delegation of information from one transaction to the other.</p>
Pros	<ul style="list-style-type: none"> • Much more extensible than a conventional single model system, since it allows hybrid solutions to the models it contains.
Cons	<ul style="list-style-type: none"> • Complex to visualize and implement.

Model	BTP Atom Model
Orientation	Atomic Transactions (2PC Model Variant)
Released	2001 - Oasis Business Transaction Protocol Project
Description	<p>The atom model is a heavily customized version of the 2PC model, in which ACID semantics are still evident, however with a much less strong presence. It also consists of two phases, which are the 'prepare' phase, in which each participant reports the current state of availability of resources, and the 'confirm' phase, in which a transaction is either confirmed or cancelled. ACID has been abstracted from this model by not defining implementation details of the 'prepare', 'commit', and 'cancel' operations. These may be implemented at a higher level business logic.</p> <p>Another major change from the 2PC model is the fact that business logic decisions have been inserted in the transition between the two phases. This allows the user to have complete control over transaction timing from the application level, as opposed to the original 2PC algorithm. Nevertheless the ACID nature of the atom model is still shown in its output which is guaranteed to be consistent, where all stakeholders in a compound transaction within the atom model will have one common outcome, success or failure. The Atom model is a subset of the Cohesion Model, and is used for strictly transactional Units of Work. (Reference:</p>

	www.webservices.org article: The Smogasboard of Web Services Transactions – Mark Little) [3]
Pros	<ul style="list-style-type: none"> • Aids the Cohesion Model to cater for LLT.
Cons	<ul style="list-style-type: none"> • Does not cater for isolation between UOW.

Model	BTP Cohesion Model
Orientation	Atomic & Long Running Transactions
Released	2001 – Oasis Business Transaction Protocol Project
Description	<p>The cohesion model has been introduced in order to cater for long running transactions. Its composition is made entirely from a collection of one or more atom model based Units of Work, which explains the origin of the name cohesion. While each individual atom based unit follows ACID traits, the cohesion model combines the units together in such a way so as to relax atomicity, and allow a business logic level implementation of the 'prepare', 'confirm', and 'cancel' operations which each Atom Unit will use. This allows the cohesion model as a whole to move away from 100% consistency, since its member Atom Units may each have different outcomes, breaking the ACID rule of complete success or failure. This allows the efficient execution of long running transactions, where activity services are represented as atoms/Units of Work each of which is registered to the cohesion. These units may undergo a series of confirm, cancel operations as the business process moves on, finally reaching a global confirm state which had been pre-set at business logic level. The global confirm state is set by defining a set of states for a subset of units in the cohesion. When global confirm is reached;</p> <p><i>"the cohesion collapses down to being an atom, all members of the confirm set see the same outcome" – (Mark Little, Arjuna Technologies)[3]</i></p>
Pros	<ul style="list-style-type: none"> • Caters for Long Lived Transactions. • Highly generic model to suit a wide variety of applications
Cons	<ul style="list-style-type: none"> • Does not cater for isolation between Units of Work. • Highly Complex to implement

Model	WS-AT (Web Services – Atomic Transaction)
Orientation	Atomic Transactions – Web Services

Released	August 2002 – Microsoft, IBM, BEA WS-C/T Project
Description	<p>WS-AT is an extension of the 2PC Model, however still with the same purpose of servicing Short Lived Transactions. The logic behind WS-AT is very similar to 2PC and BTP's Atom Model. Transaction-aware system resources such as databases are registered by the activity service as members of the transaction process. This allows the resource to be updated constantly with the transaction's outcome.</p> <p>WS-AT also addresses the issue of Durability by providing a synchronization protocol which caters for communication of the business process with a persistent system resource such as a database, thus making it possible to dump the business processes' memory cache onto the database. (Reference: www.webservices.org article: The Smogasboard of Web Services Transactions – Mark Little)[3]</p>
Pros	<ul style="list-style-type: none"> • Caters for Durability Issues. • Provides Interoperability – Web Service Oriented
Cons	<ul style="list-style-type: none"> • Is not suitable for long running transactions

Model	WS-BA (Web Services – Business Activity)
Orientation	Long Running Transactions – Web Services (Forward Compensation Based)
Released	August 2002 – Microsoft, IBM, BEA WS-C/T Project
Description	<p>The WS-BA model is dedicated towards handling long lived transactions. The first noticeable feature in this model is that it does not manifest full ACID behaviour. This is achieved by not allowing any resource locking to occur. This makes it possible for transactions to span over any necessary amount of time to complete, without causing deadlocks. One ACID property still evident in this model is consistency of results, which is kept throughout a business activity with the aid of forward compensation actions which are catered for at service level.</p> <p>In this model, the Business Activity may be partitioned into tasks which act as parents to groups of Units of Work. This gives the activity control over which units UOW's to commit or rollback by sending execute commands to the respective units. A reporting system which allows the individual units to inform the parent task</p>

	whether it is possible to rollback an activity or not is also present. This Parent-Child behaviour between the Business Activity and Unit of Work enables the Business Activity as a whole to proceed, not bothering about each and every Unit of Work's failure; this is taken care of by each parent task.
Pros	<ul style="list-style-type: none"> • Caters for Long Running Actions • Has been designed with Inter Operability in mind.
Cons	<ul style="list-style-type: none"> • Has been designed to run on a system which is closed source.

Model	TX-ACID (WS – ACID)
Orientation	Atomic Transactions – Web Services
Released	August 2003 - Sun, Oracle, Arjuna WS-CAF Project
Description	The TX-ACID model can be considered to be nearly a mirror image of the WS-AT protocol described above, with scarce differences which have no significant relevance to the method of operation of the models. One would ask, why have two nearly identical models been made? This situation is merely a solution to a political complication between closed source and open source beliefs, since the TX-ACID has been designed by companies who promote open source development, while WS-AT has been developed by companies with a closed source philosophy.
Pros	<ul style="list-style-type: none"> • Caters for Durability Issues. • Provides Interoperability – Web Service Oriented • Has been designed with open source as a background concept.
Cons	<ul style="list-style-type: none"> • Does not cater for long running transactions

Model	TX-LRA (WS-LRA)
Orientation	Long Running Transactions – Web Services (Forward Compensation Based)
Released	August 2003 – Sun, Oracle, Arjuna WS-CAF Project
Description	Again having origins due to political issues, the TX-LRA coupled with the TX-BP defined below is a direct competitor of the WS-BA Model. TX-LRA is dedicated towards handling Long Running transactions. An LRA based business process may be split up into various sub tasks, each of which in itself is an LRA Unit of Work thus having a Parent-Child relationship similar to the BTP Cohesion model's subdivision of tasks. LRA also caters for

	triggering the necessary compensation actions using forward compensation semantics; however implementation details of these actions are not catered for by this model.
Pros	<ul style="list-style-type: none"> • Caters for Long Running Transactions • Provides Interoperability – Web Service Oriented • Has been designed with open source as a background concept.
Cons	<ul style="list-style-type: none"> • Complex when compared to the BA Model

Model	TX-BP (WS-BP)
Orientation	Long Running Transactions – Web Services
Released	August 2003 – Sun , Oracle, Arjuna WS-CAF Project
Description	TX-BP is the top level of a system of three models, TX-ACID, TX-LRA and itself, which together provide an enterprise solution for distributed long running transaction handling. TX-BP is not exactly a model in itself, but rather a container which may consist of various transaction models, amalgamated together in a typically distributed manner. The BP Model constitutes a recursive hierarchy, where one business process is split into business tasks, which are essentially transactional Units of Work. Each task is part of a business domain, which is a top level entity comprising of other business sub-domains, business processes, or individual tasks/units of work. In essence, a business domain constitutes a transaction model.
Pros	<ul style="list-style-type: none"> • Caters for Distributed LLT's using possibly multiple models.
Cons	<ul style="list-style-type: none"> • Highly complex to implement (contains over 40 messages in its protocol)

Model	conTract Model
Orientation	Long Running Transactions (CAD/CAM)
Released	By Andreas Reuter – 1989
Description	<p>The contract model was one of the early attempts at handling long lived transactions. It moves away from the idea of ACID transactions, and makes use of the concept of forward compensation.</p> <p>The structure of the contract model revolves around sequences of steps and scripts, where steps represent</p>

	<p>simple Units of work, and scripts represent descriptors which cater for the coordination of each unit of work. Coordination is defined using constructs such as transaction interdependencies, recovery parameters, etc. The main idea in the contract model is that of abstracting workflow issues completely to the application programmer, since the script takes care of this. The official definition of a contract model, as defined by Andreas Reuter is the following:</p> <p><i>"Contract is a consistent and fault tolerant execution of an arbitrary sequence of predefined actions (steps) according to an explicitly specified control flow description (script)"</i> – Andreas Reuter [35]</p>
Pros	<ul style="list-style-type: none"> • Caters for long running transactions. • Separates Units of work from coordination.
Cons	<ul style="list-style-type: none"> • Is limited to forward recovery. • Very complex to implement.

Model	Bourgogne Model
Orientation	Long Running Transaction Handling
Released	2000 – Marek Prochazka (PHD Project)
Description	<p>The Bourgogne model has been specifically designed in order to cater for long running component based transaction systems. It has been developed by Marek Prochazka as a PHD project at Charles University, Czech Republic. Bourgogne Transactions can be considered an extension of the ACTA framework. This project is mainly targeted at providing an extensible model which allows a degree of control over subtle attributes such as transaction interdependency, resource sharing, and delegation concepts. In fact in his thesis, Prochazka describes Bourgogne Transactions in the following way:</p> <p><i>"Bourgogne Transactions stem from the transformation of ACTA building blocks"</i> – Marek Prochazka[29]</p> <p>and</p> <p><i>"Bourgogne Transactions comprise:</i></p> <ul style="list-style-type: none"> - <i>A new transaction model, which specifies significant events, operations executed upon data objects, the lifecycle of a transaction, and the environment of a</i>

	<p><i>transaction.</i></p> <p><i>- A new transactional API that allows the management of transaction demarcation. The API is used by clients and also by containers, which use it for managing container-demarcated transactions.</i></p> <p><i>- Support for employing new techniques in container-interposed transaction settings. Namely, Bourgogne Transactions introduce a set of transaction attributes for specifying transaction propagation policy, including declarative transactions in the component interface.” - Marek Prochazka [29]</i></p>
Pros	N/A
Cons	N/A

Model	JSR 95 Model
Orientation	Long Lived Transaction Handling
Released	No Commercial Implementation as Yet
Description	<p>The LLT Model has the purpose of providing a framework which handles Long Running Transactions in a distributed industrial environment. It houses concepts based on the ACTA and SAGA models, where a Long running transaction is defined by a series of activities.</p> <p>Activities are executed in sequence, and each activity may commit, or rollback. In this model, the success of a long running transaction is determined by the commission of all its child activities. The following set of rules applies for this model:</p> <ul style="list-style-type: none"> • If a child activity rolls back, all the previous child activities must roll back, thus failing the long running transaction. • If rollback of an LLT is initiated, a compensating action must be carried out for any activity which has already committed, since this cannot roll back. <p>An interesting feature introduced in the LLT model is that it has been structured in such a way, to enable transaction management systems using this model to</p>

	<p>suspend or resume transactions. If during execution of an activity, a connection to a third party server fails, the activity does not abort, but rather goes into a suspended mode, which can be later resumed. This novel idea greatly increases the notion of robustness in long running transactions.</p> <p>Please note that the terminology used for long running transactions in the LLT model is the term "Long Lived Transactions".</p>
Pros	<ul style="list-style-type: none"> • Based on well established standards (ACTA & SAGA) • Supports Long Running Transactions • Caters for rollback and recovery concepts • Introduces the idea of suspension and resumption of activities • An effort is being made to release the model to the open source community, which is desirable in the research area.
Cons	<ul style="list-style-type: none"> • The design of the model does not allow flexible definition of a workflow but rather, it executes activities in an LLT sequentially. • Model must be hard coded into a transaction management system implementation.

Model	Cova Transaction Model
Orientation	Long Running Transaction Handling
Released	2002 – Bell Labs Research: Lucent Technologies
Description	<p>The Cova Transaction Model is an effort by bell labs to extend the idea of modeling long running transaction using scripts. While the structure is similar to Ixaris' LLT Model, where a Long Running Transaction is made up by a tuple of activities or units of work, the main difference is that workflow is represented using a scripting language, rather than hard coded into a sequential process. This presents a significant advantage over other transaction models, since it allows flexible definition of transaction inter dependencies through the script, in a workflow style.</p> <p>The main drawback of this model is that the script consists of specially designed transaction oriented syntax, which still requires solid knowledge about transaction</p>

	theory in order to understand it completely. This still does not completely abstract a top level developer from transactional issues. Another drawback is the fact that the script based model is tightly bound to one particular transaction manager implementation designed by bell labs, rather than being generic and usable by all. In fact, the Cova model specification paper also includes the transaction management engine's specification. This goes against desirable open source based concepts for research projects.
Pros	<ul style="list-style-type: none">• ConTract based scripting approach to transaction management• Script allows flexible definition of transaction Models, through a workflow styled interdependency description.
Cons	<ul style="list-style-type: none">• Code not released to the open source community• Script contains highly specialized transaction oriented syntax (eg. "set dependency" commands, "compensate forward" and "compensate backward" commands).

Note: There are several other models in existence which have not been referenced here, due to the fact that there are too many. Such models include the DOM Model, Flex Transactions, the CORD model, Cooperative Transaction Hierarchy Models, and H-Models amongst others. An accurate review of each of these models can be found both Marek Prochazka's PHD thesis entitled "Advanced Transactions in Component Based Software Architectures.", and in Eman Anwar's thesis entitled, "An Extensible Approach to realizing extended transaction models". These models are based on the models described in the "Advanced transaction models" section above, and do not introduce new concepts, but can be considered as hybrids of the above described models.

2.1.5 Common Properties of Transaction Models

When one sums up the generic properties of these models, a set of concepts which are common to all models may be identified. These include the following notions:

- Transaction Modularization
- Rollback and Recovery Handling
- Activity and LLT State Handling
- Transaction Context Definition and Propagation
- Transaction Inter Dependencies

2.1.5.1 Transaction Modularization

The first thing which must be carried out in the when designing any transaction model is that of assigning a proper structural transaction hierarchy. How are transactions structured? What granularity is there with regards to transaction complexity? It has been noticed that all the advanced and true advanced transaction models share the following Transactional hierarchy;

- **Long Running Transactions Revisited**

The richest, or highest level transaction is the Long Running Transaction, which has a compound nature and may extend over a long period of time to complete. A long running transaction can be made up of several units of work, which may or may not be transactional, and may or may not be atomic. Various synonyms have been given to long running transactions in the researched models, ranging from "long lived transaction" in the LLT model, and "Compound Transaction" in SAGA and ACTA, to "Long Running Transaction" in the WS-CAF project. In the Transit Model Solution accompanying this thesis, Long Running Transactions will be referred to as "Long Lived Transactions", in order to avoid confusion in terminology. Long lived transactions are typically physically be defined by programmatic classes, which contain array lists of activities which form an execution workflow.

- **Activities (Units of work)**

Activities, also referred to as Units of Work in various model specifications, are the base component of a Long Lived transaction and may or may not have a transactional nature. They also may or may not be ACID compliant. The most important concept to grasp is that an activity in itself consists of a physical structure, possibly a programmatically defined class, which contains a series of methods and remote calls to external servers. This series of methods creates a workflow which constitutes an execution step in a long running transaction. The following diagram illustrates the generic hierarchy of long running transactions and activities which is shared by all advanced and true advanced transaction models researched:

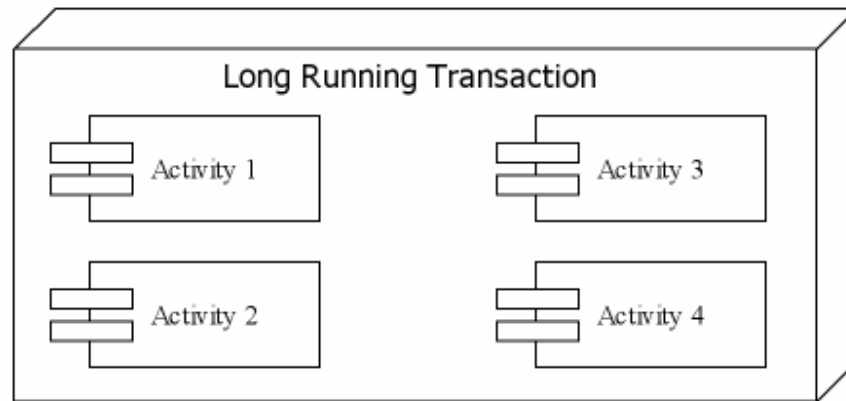


Figure 2.1.5.1.1 Long Running Transactions and Activities

2.1.5.2 RollBack and Recovery Concepts

When designing a model which is aimed at handling long running transactions, rollback and recovery concepts must always be considered. These concepts allow elegant recovery of a long running transaction, in case one of its activities is unsuccessful. Consider the long running transaction in figure 2.1.5.1.1. Assuming that this transaction is plugged into a model which executes its activities sequentially, the workflow in this case will consist of the following:

```
public void Workflow()
{
    //Execute Activity 1
    //If successful Execute Activity 2
    //If Successful Execute Activity 3
    //If Successful Execute Activity 4
    //If successful set Long Running Transaction to successful
    //Else set Long Running Transaction to unsuccessful
}
```

Figure 2.1.5.2.1 Pseudocode For a Generic Workflow Example

As execution progresses, activities are confirmed sequentially. If all activities are successful and no problem arises, the long running transaction is successful, and the system returns to an idle state. However, what happens if activity four fails? When executing a long running transaction such as this one the following rules apply in case of failure of the transaction:

- A “cleanup” process must be carried out in order to release resources which have been occupied by the transaction and each of its sub activities.
- An activity which fails can be reversed (rolled back) in order to clear out any resources which it is currently using.
- An activity which is confirmed (committed) cannot be reversed (rolled back).
- If a failed long running transaction contains activities which have been confirmed, they cannot be rolled back. However some form of recovery (compensating) action must be taken for each activity.

Thus if transaction four fails, it is rolled back, while transactions three, two and one must be recovered in some way since they have been committed. Rollback on a committed activity is prohibited for a simple reason. Let us assume that activity two is actually a flight booking process. Once a ticket is confirmed, in normal circumstances, it cannot be cancelled and re-funded. Thus a compensating action must be taken, where another flight ticket is found for the customer in question, while moving the old ticket to a “last minute offers” section. If no action is taken, the ticket would not be used, resulting in an empty seat on the flight. While this is just one context, it explains why rollback on a committed activity is impossible. It also explains why rollback and recovery concepts are necessary in long running transactions. If no compensating action is taken on committed activities, or no rollback occurs on failed activities, a great deal of resources would go wasted when a Long Running Transaction Fails.

With the introduction of rollback and recovery, the states in which a transaction may be are not limited any more to “committed” and aborted”, as was the case in two phase commit. This results in a rise in complexity when handling transaction states, also due to the fact that different terminology is used to describe similar concepts. Recovery is also called compensation, or Forward Compensation in some models, while rollback also has the “Backward compensation” synonym amongst others. The variance in terminology, together with the expansion from two base states to multiple states may result in confusion and thus the need for proper handling of transaction states is felt.

2.1.5.3 Transaction States

A transaction state is an indicator which refers to the current status in which a particular atomic or long running transaction currently is. Typically, a transaction starts off in an idle state and changes state as the transaction workflow

executes. Consider the previous example of Two Phase Commit model description:

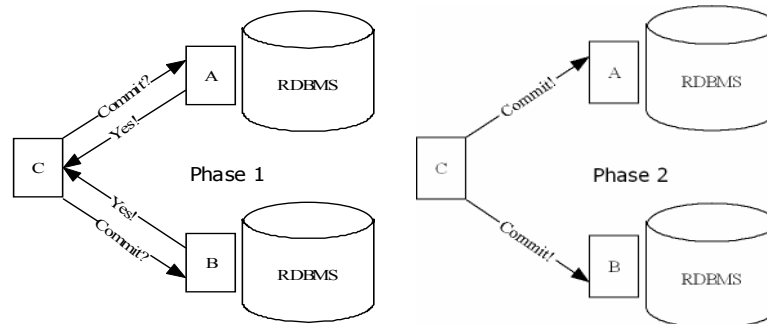


Figure 2.1.5.2.1 Two Phase Commit State Changes

The state changes for transaction "C" are from "idle" to "committed", of from "idle" to "aborted" when implemented using two phase commit; thus the 2PC model can be said to have three transaction states in all, idle, committed, and aborted. This set of states provides a standard interface which enables transaction management systems which implement 2PC to monitor the progress of a particular transaction. It would be very difficult for the Transaction Management software implementing the model, to later interpret the outcome of the transaction execution if some form of standard state handling is not introduced, since there would be no way of monitoring transaction progress. For this reason, a series of transaction states are used in each model. Consider the following models:

Model	State List
2PC	Transaction Idle Transaction Committed Transaction Aborted
JSR 95	Transaction Idle Transaction Committed Transaction Rolled Back Transaction Compensated Transaction Suspended
Cova TM	Transaction Idle Transaction Currently Running Transaction Completed Transaction Aborted Transaction Failed

Figure 2.1.5.2.2 Transaction States

The concept of transaction states is present in all researched models; however as shown in the table above, each model utilizes a different set of states, which have varying properties and terminologies. The state of a transaction may also be represented by a subset of states of multiple units of work or activities. For example, while in the two phase commit, the state of transaction C would be represented by an one state value, x , in the Cova and JSR 95 transaction models, the state of a transaction C would be represented by an n tuple (x_1, x_2, \dots, x_n) where each x_i represents the value of the state of a unit of work or activity which makes up the long running transaction.

While structure, properties, and terminology of transaction states may differ from one model to another, they share a common scope, which is that of providing a standard gauge mechanism, or interface which allows a transaction management system to track a transaction's progress.

2.1.5.4 Transaction Contexts

Transaction contexts represent a particular scenario in which a Long Transaction is executed. This directly affects the manner and sequence in which the Activities which make up the Long Lived Transaction are executed. In fact, all activities which make up a long running transaction share the same transactional context through which the various activities are coordinated to switch from one state to another according to conditions posed by the model or at runtime. In current solutions, transaction contexts have been described in various ways, ranging from hard coded programming structures, to scripting languages, as in the case of Contract Models, and the Cova Transaction Models. Transaction context definition is closely related to the definition of transaction inter dependencies.

2.1.5.5 Transaction Inter Dependencies

Inter dependencies in transactions are synonymous with the concepts of rollback, recovery, and compensation. Dependencies between activities may be used to primarily define execution sequence, and activity nesting. Consider a set of four Atomic units of work, named A, B, C, and D respectively. Each of these units is a two phase commit transaction, however if they are orchestrated into a workflow according to the nested transaction model, the result would be the following:

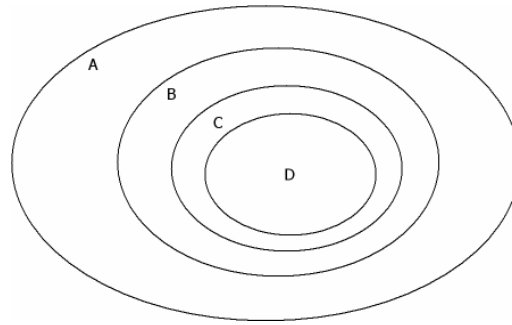


Figure 2.1.5.5.1 Nested Model

A hierarchy is formed where A is at the top level, and D is dependant on C, B, and A respectively. On the other hand, if they are orchestrated in a way which complies to the split join model, one possible scenario is the following:

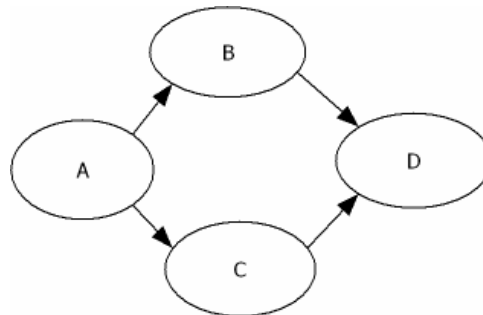


Figure 2.1.5.5.1 Split/Join Model

In this representation, unit A splits into the parallel execution of unit B and C, which in turn merge into unit D. Now consider having eight atomic units of work, A to H, and using them to orchestrate a compound unit of work using both nested and split/join concepts. The result would be the creation of a hybrid model which enables encapsulation of Units into each other, and Parallel or sequential execution.

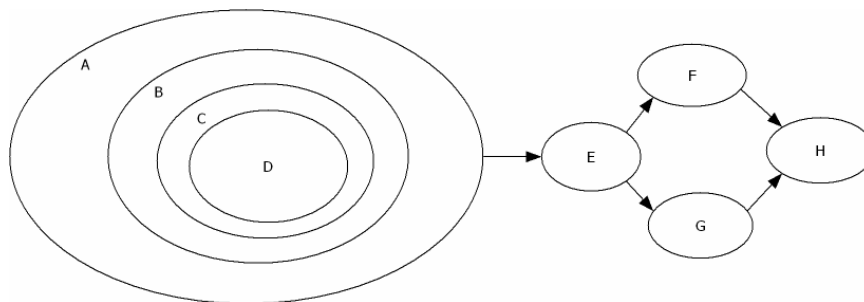


Figure 2.1.5.5.3 Hybrid Model

Now consider adding a third and fourth model, and generating combinations of multiple hybrids of each of them. This conjunction of models would create an extremely flexible and powerful way of expressing transaction workflows, where the solution would not be dependent on a particular model, but on the Meta-Model which is used to express the workflow. Thus ideally, a list of the most common constructs should be supported by the scripting language, thus solving the problem of the narrow applicability of a transaction handling solution. The most common transaction primitives include the following:

- Nesting of Units of Work
- Sequential Execution
- Parallel Execution
- Recovery & Compensation Concepts
- Rollback Concepts

These concepts all point towards the creation of the Scripting Language which models the execution of a transaction through a workflow style interdependency definition as the best possible solution for Long Running Transactions.

2.2 Existing Transaction Management Solutions

There are various transaction management systems which implement one of the models which we have researched. Currently, most systems are available as a specification, and are not yet implemented. This is due to the fact that while well defined standards are present for atomic transaction handling, this is not the case for long running transaction management; the area is still very fluid, with several proposed but few officially defined and approved standards.

This resulted in a pool of solutions, and specifications which define various transaction management systems each with their own standards and models, applicable in different scenarios. These solutions can be classified into three main categories:

- SOAP/XML Web Service based transaction frameworks.

Projects under this category include the various efforts made by IBM, Microsoft, BEA and Arjuna technologies amongst others in the Web Services Composite Application Framework (WS-CAF) and WS-C/T Projects, which are basically identical to each other however the first being oriented towards open source technologies and the latter being privately administered. The main architecture of these projects is that

of providing orchestration and choreography technologies to transactional objects in a distributed environment through Soap/XML messages. WS-CAF is divided into three main layers, each of which progressively supports distributed long lived transaction management:

- | | |
|---------|--|
| WS-CTX: | This is the initial lightweight framework which supports simple transactions. |
| WS-CF: | Handles message passing and transaction context management. |
| WS-TXM: | Comprises a full scale framework supporting three main transaction models, (two phase commit, long running actions, and transaction workflows). Over distributed transaction coordinator entities. |

They are typically used in conjunction with workflow control languages such as BPEL or WSCI.

- Solutions based on the Object Management Group's OTS Specification.

OMG.org's OTS specification extends the specification of CORBA in order to support transactions across multiple objects. One particular framework specification which falls under this category is Sun's JSR 95 Activity Service Specification Project. This specification aims at providing an extension of Sun's currently available Java Transaction Service/API, in order to make it support long running transactions. It introduces various concepts, such as the idea of having a framework into which various transaction models may be plugged, having transactional or non transactional batches of work forming a compound transaction (Units of Work), the concept of a High Level Service, and various other concepts. The main advantage of this framework is that it is highly flexible in terms of transaction models; however it relies strictly on CORBA/IIOP communication.

- Script Based Transaction Workflow Representation Solutions

The two most prominent specifications here are Contract Model based specifications, and variants which use control flow description methods. One particularly robust specification which is classified as such is the CovaTM control flow framework, funded by Bell Laboratories, USA. This framework contains detailed specifications of a script-like language which is used to control transaction workflow. Various features are present in this framework, which make it a

desirable one to use, ranging from adequate transaction context handling, flexibility, and abstraction of transaction model details from developers using the system. The main drawback of this solution is the complexity of the scripting language.

Another solution which, while not script based, is also based on a fixed sequential workflow is the implementation based on the JSR 95 LLT model developed by Ixaris (Malta) Ltd. The engine developed caters for Long Lived Transaction Handling (variant terminology for Long Running Transaction), while offering several novel features such as the suspension and resumption of a transaction. This includes persistence of the transaction to disk, thus allowing restart of the middleware and the server, without losing the transaction's information.

While the solutions described above are the most widely known, various other solutions are available, such as ebXML, BPEL4WS, etc. It is not possible to cover all available solutions, however the three categories described above include most features that typical transaction handling frameworks possess, giving the reader an idea of the way a typical transaction management system should operate, and an idea of the critical components which it should contain.

The choice of a transaction management framework strictly depends on the application which will be developed. Given the current solutions and transaction models, the developer must analyze each possible scenario in the system he is developing, and subsequently select the best fitting transaction model. The developer may either opt to implement the model himself, or choose a middleware software solution such as the ones described above, which manages transactions for him.

2.3 Drawbacks of Current Solutions

During the literature review, a series of drawbacks present in current model specifications and solutions can be identified. These include not only technical and design issues or each individual model, but also abstract problems present in the transaction management area as a whole. These include:

- Non Technical Issues
 - Developer's choice of an appropriate model and solution

- Technical Issues
 - Separation of Transaction Models from Management Systems
 - No apparent Long Running Transaction Standards
 - ACID principles in Long Running Transactions

2.3.1 Non Technical Issues

The problems presented in this section do not relate to the design, development and deployment of a transaction model or transaction management solution, but rather regards generic issues in the area of transaction management which developers may encounter when creating a transaction enabled application.

2.3.1.1 Choosing a Solution

The most obvious problem which results from the research is the difficulty of choice which a developer must face when selecting a way in which to transaction-enable his application. As it has been seen in the literature review, there is a wide range of varying solutions which cover various possibilities of a solution, some of which overlap. Rather than having the advantage of being spoilt for choice, developers thus encounter a problem, namely the dilemma of which model best suits their application, and which solution should they chose.

In certain cases, there can also be a situation where no readily available model fits the application needed by the developer, and thus a custom model must be designed. Since no solution would exist which caters for the custom model, the developer also has to go through the trouble of developing a transaction management system for his custom model.

The situation as it is requires developers to have medium to expert knowledge in the transaction management field, since this is needed both to choose an appropriate readily implemented transaction model solution and to develop a completely customized transaction model and management system. This is not the ideal solution, since transaction modeling techniques, and transaction system design and implementation are not a trivial task and require a great amount of time consuming effort and resources which could be better invested for the top level application's main scope. If for example a developer wants to create an e-Topup System, or a Travel Agent's booking system, if no ready made template system is suitable, he has to detour and develop a transaction management system for the e-Topup or Travel agent, probably using more time in its

development than in the development of the top level solution. Thus a solution should ideally be found for this issue.

2.3.2 Technical Issues

The problems presented in this section regard the technical issues encountered during the design and development of transaction models and solutions which cater for long running transactions. Special importance is given to the problem of the application of ACID principles to long running transaction models.

2.3.2.1 Separating Transaction Models from Transaction Management Systems

An issue which is common to various solutions is the close coupling between a particular transaction model and a transaction management solution. This is the case of any Transaction Management System which is implemented with the sole purpose of exploiting one particular transaction model in mind, as is the case of CovaTM. In CovaTM, the transaction management system has been designed to operate solely with its accompanying model, which is hard coded in it. The same situation is present in the LLT solution proposed by Ixaris, where the runtime engine was based solely on the LLT model. The notion of hard coding a model into an engine, and having the engine cater solely for that model reduces the range of applications which may use that solution to those which perfectly fit that particular model, thus possibly making the solution impractical. Ideally, a transaction management system should not be bound to a particular model and a model should ideally not be hard coded into the system but rather defined separately from the software. This concept is examined in Sun's Activity Service Specification (JSR95) where a system with a pluggable model architecture is proposed. This is a desirable feature in transaction management systems since it enables them to operate using different transaction models, making them suitable for a wider range of applications. Besides, if the concept takes off, a standard method of defining methods may evolve, resulting in the possibility of inter sharing transaction models between transaction management systems. For this to happen, a transaction model would ideally be defined in a physically separate file from the system, possibly in a standardized format.

2.3.2.2 Standards in Long Running Transactions

While well known and officially approved standards exist for short lived transactions, namely ACID principles, the relaxation of these principles in the case of long running transactions has caused various research efforts from

individuals and major software houses, resulting in each of these entities defining their own standards, principles and methods, and using them to cater for long running transactions. Thus while previously a globally approved set of principles (ACID) was available for anyone dealing with transactions, we now have a large pool of ways in which long running transactions may be handled, none of which is approved by a centralized authority. This does not necessarily mean that the methods are ineffective, however as previously explained, havoc is caused when a developer comes to choose a method for his application, probably resulting in him implementing his own custom solution. Ideally, a standardized way is defined to at least allow a way in which transaction models may be defined across all existing systems.

2.3.2.3 ACID Principles in Long Running Transactions

The most important technical problem which must be considered is the fact that while ACID principles work well in short lived transaction models such as two phase commit, in the case of long running transactions, they do not seem to fit. This is mainly due to the major change in logic from the concept of one atomic transaction which commits or aborts, to that of a compound transaction made up from various sub transactions, each of which may be atomic, or consist of compound transactions. This change in logic, coupled with the introduction of rollback and recovery concepts, give clear indications that ACID principles, while still partly valid, are not completely applicable to models which cater for long lived transactions. Thus the main problem presented here is the fact that ACID principles are not always adequate for long lived transactions. Thus the following questions arise:

- Is ACID good or not?
- How should this problem tackled?
- What is the solution to this problem?

The best method to resolve these queries is by looking at real life scenarios which have transaction processing potential and analyzing them, taking into account whether the ACID model based solution would be efficient in each scenario. There is an infinite amount of possible scenarios to which transactions can be applied, ranging from purely atomic transaction services to highly complex compound long running transactions, however for this particular case, two scenarios will be considered, a account e-Topup facility, and a typical travel agent scenario. A brief overview of each scenario, together with an analysis of the choice of a model and management system for each scenario is provided on the next page:

2.4 Real Life Scenarios

Below is a brief description of the logical mode of operation of each scenario from a high level perspective, followed by a generic confrontation of the case studies with acid model implementation scenarios:

2.4.1 E-Top Up Facility

Consider a software application which requires topping up of an account, such as a pre-paid mobile web-top up, or a Voice over IP phone software such as Skype. The main requirement of this application is that members who purchase the software are allowed to top up their Skype accounts through the software, using third party financial services such as Visa, MasterCard, PayPal, or Entropay.

When a user tops up his account, a skype topup transaction is executed. This transaction consists of three main processes:

- Checking if topup is allowed with a remote skype server
- Making a fund transfer request to a third party server such as VISA.
- Updating the user's account information on the remote skype server

The task of the developer in this case is to develop the voice over IP phone application, including this top up process. The way in which the processes are coordinated will be determined by the transaction model which the developer chooses to implement, or by the middleware solution which the user decides to exploit.

2.4.2 The Travel Agent Facility (Referenced from JSR 95)

Consider a typical travel agent, where various services which enable customers to plan their holidays are offered. In this case study, the travel agent facility considered here also presents a rather complex system, since it comprises of a multitude of these services which may have various interdependencies on each other when reservations are made. These include:

- A flight ticket booking service
- A hotel room booking service
- A train ticket booking service

The main idea here is that an end user selects a series of parameters through a GUI, which determine the services he wants for his or her holiday, and initiates a "Book Holiday" transaction. This transaction executes all the processes needed in order to book any service requested by the client for his holiday. It is being assumed that parallel transactions are possible, as in, if a client requires a compound transaction which comprises of a flight reservation and a train reservation, these both are two isolated transactions with their own resources, and thus they can be carried out in parallel without hindering each other's end results.

However, the results of each Unit of Work of the transaction do have to depend on each other. For example, if a client requests a flight and train reservation for a particular date, it may occur that the train booking is successful, while the flight booking fails, thus making the flight booking of course useless. For this case study, this situation is also catered for according to the choice of the transaction model or management system which the developer carries out.

2.4.3 Case Study Analysis – Interdependencies & Workflow

The first step in the determination of which transaction model or solution to select is that of defining the workflow which takes place in the application in question. In the Skype E-Topup scenario, a logical sequential execution can be identified:

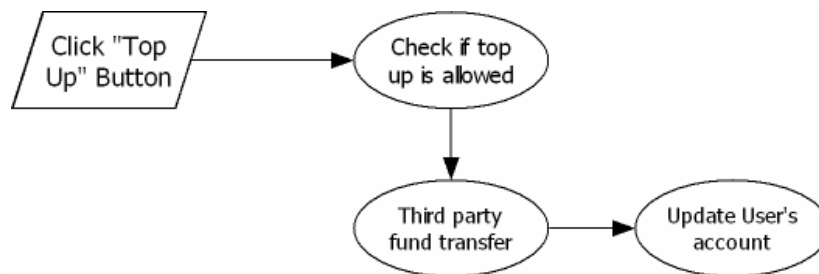


Figure 2.4.3.1 Skype E-Topup System

Initially the system must check if topup is allowed, then make a fund transfer request, and finally, if the transfer succeeds, the user's account is updated.

It can be noticed straight away, that a strict ACID based model such as two phase commit would not be adequate in this case, due to the compound nature of the topup transaction. This particular workflow does not require parallel execution, thus one may also rule out the choice of any transaction models or solution based on parallel execution such as the Split Join Model. Since this

application will always consists of the same workflow, with no possible variants, a transaction model such as the nested model may be applied. However, consider again two phase commit. What if a hybrid model is created, where each activity executes in an ACID environment, returning a "commit" or "abort" state? The long running transaction's result would thus depend on a series of two phase commit based activities. This, in essence is similar to the LLT model, or rather, a hybrid model which extends the two phase commit. It can be seen that while the user is, as previously stated "spoilt for choice", the element of confusion can still arise, on whether ACID is good or not, and which model and transaction management solution would best suit the application.

In the travel agent's case, the definition the workflow is different. Activities may be defined as "a set of one or more Units of Work both short and long lived, transactional or non transactional, possibly running in parallel, which together make up one compound long running transaction". The next diagram provides a graphical representation of a typical activity which could occur in this context:

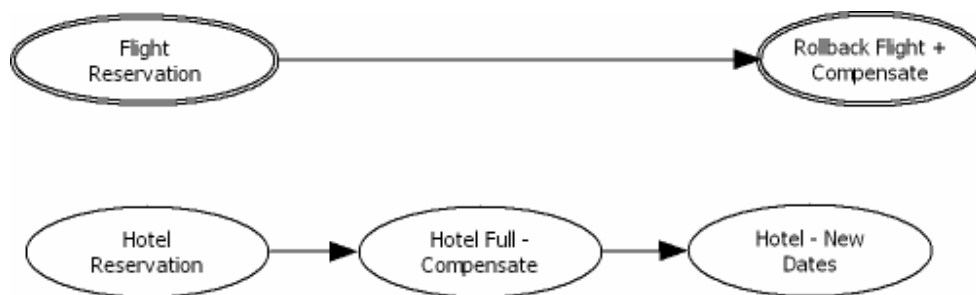


Figure 2.4.3.2 Flight Booking System (Ref JSR95)

In this scenario the process consists of parallel running transactions, where each atomic transaction has an impact on any other transaction running in parallel to it, thus full rollback and recovery capabilities are a must. Since the classic nested transaction consists of the parallel execution of all its sub activities, it may be considered as the ideal way to go in this case. However, since each activity in the nested model must wait for each of its children to commit, before it actually commits, in this case, the result of applying the nested transaction model would be that of having several resources waiting for each other over an extended period of time, which is unacceptable in the case of an airline, train company, or hotel. As opposed to the E-Topup example, in this case the workflow may vary according the parameters set by the end users, thus a solution which handles multiple workflows would be ideal. This eliminates all transaction models and management systems which are tightly bound to one particular model which defines strictly one workflow, leaving us with choices of script based models such as conTract and Cova TM. When considering two phase commit, it can be seen

that this ACID based solution definitely cannot handle such a transaction in its classic form, without any modification. This also rules two phase commit out.

However at this point, the general idea that ACID is not good, but still should not be completely scrapped may be visualized.

2.4.4 ACID is good – take it in short doses!

Transaction management systems which are based on strict ACID principles blindly follow the rules of atomicity, consistency, isolation and durability. However when looking at these scenarios, it is noticed that in both cases it is impossible to strictly follow all four ACID properties at all times. This is mainly due to the fact that a transaction is not seen as an atomic element, but as a compound element (previously described change in logic). This implies that a transaction is made from a series of sub transactions, which may also be of type long running, thus containing further sub transactions. However if one iterates throughout all the hierarchy of transactions and reaches the activities with the lowest level of granularity, the activities found at this level may be considered as atomic, isolated, and durable. This rule is consistent in any possible transaction management scenario.

This means that rather than scrapping ACID concepts, they can possibly be used as the base rules for handling activities which represent the lowest granularity level. The result is a hybrid model which includes a series of activities as its foundations which “more or less” conform to ACID principles, and coordinates them by encapsulating them in higher level long running activities and transactions. If one applies the nested model to the Skype E-Topup scenario, the long running transaction “Topup” would still consist of the three same activities, which however are now considered as atomic, durable, and isolated transactions. Technically speaking, the activities would still not be completely ACID conformant, since the consistency property is partly lost with the introduction of rollback and recovery concepts; however they can still be considered very similar to classic ACID based transactions. Mark Little identifies this event as “relaxation of ACID rules” in his article “ACID is good, take it in short doses”. [5]

This leads us to the conclusion that ACID principles are inadequate if viewed from the classic perspective of having a strict ACID based model handling a whole transaction, from beginning to end. However if a new perspective is taken, where ACID principles are used as the foundations for an extended model, ACID still works effectively. The great thing about this novel concept is that it can be applied to any existing transaction model, thus not requiring the further introduction of new transaction models. The only change needed is in the way of defining the transaction hierarchy, by considering the lowest level as a

transaction which is loosely based on a relaxed form of ACID principles. An example of a transaction model which readily exploits these concepts is that of the SAGA transaction model.

SAGA's notion is that of loosening the rigidity of strict ACID properties, however not completely scrapping them. In fact a typical SAGA:

"approximates atomicity over a long period of time, however not providing the isolation property". (Acid is good, take it in short doses – Mark Little) [5]

This is done by breaking down the whole activity of a scenario into sub-activities, and further into transactional or non transactional Units of Work or activities, which are loosely based on ACID principles. Taking the travel agent's scenario into consideration, if it has to be modeled around a SAGA based model, each Unit of Work (ellipse in the diagram above) would be an atomic transaction:

Transaction Method	Type
Book_flight();	ACID based Transaction
Book_hotel();	ACID based Transaction
/*At this point the flight fails but hotel succeeds*/	
Rollback_Flight();	ACID based Transaction
Compensate_Flight();	ACID based Transaction
/*If same date flight is found*/	
Ready();	
/*Else different date*/	
Rollback_Hotel();	ACID based Transaction
Compensate_Hotel();	ACID based Transaction

Figure 2.4.4.1 SAGA Transaction Model Descriptor

Thus in SAGA terms, the above table would be one long running activity, consisting of non isolated loosely coupled atomic transactions. Thus the saga model gives proof that ACID principles in long running transaction models are still applicable, however in a more relaxed manner than the way they are applied in short running transactions. While this analysis solves the dilemmas presented about the adequacy of ACID principles for long lived transactions, it still does not address the rest of the drawbacks discussed in section 2.3.

2.5 The Problem

It can be concluded that while the advanced transaction models and frameworks researched have been shown to cater in general for the resolution of the strict ACID problems; a series of issue are still present, whose solution serves as the main motivation for this thesis.

The largest problem which ties most classic advanced transaction models together is their lack of flexibility when catering for varying applications. As previously explained, the chances of having a wide range of applications perfectly fit a predefined model are scarce, if not impossible. The solution to this would mean restricting the ways in which developers can define transactions in their applications, which is of course an undesirable feature in a transaction model.

On the other hand, when one looks at the transaction framework specifications available, namely OMG.org's Object Transaction Service, upon which CORBA and Java's Activity Service Specification was modeled and Arjuna's WS-CAF Framework described in this chapter, the concept of long running transaction handling has always been synonymous to complex, difficult to understand, mainly container based specifications. Implementations based on these specifications include Java's Enterprise Beans, COM+, and Microsoft's latest effort in the area; the System.Transaction library. However;

"most of these standards employ simple, ad-hoc solutions without addressing key issues of transactional components." – Marek Prochazka (Jironde) [34]

Prochazka refers to the fact that even though various transaction framework specifications are present, there is no complete solution, applicable in all cases of long lived transactions which aids developers in handling long running transactions in a simple but complete manner. While framework specifications like IBM's JSR95 do offer multi model coverage, they tend to require a high degree of transaction modeling knowledge from the developer's side in order to be effective. These issues thus serve as motivation for this thesis, where an alternative solution which resolves these issues will be sought.

2.6 Motivation

From the research carried out it can be clearly seen that even though there has been a massive effort in the issue of long lived transaction handling since the early 1980's, concrete solution attempts have never been fully satisfactory. The main problem lies at the heart of the subject; the transaction models. A vast amount of transaction models have been proposed since 1980, varying from simple Atomic Transaction Handling Models to very complex Compensation Based models, the problem being that its impossible to have one transaction model which caters for all possible transactional scenarios. Each proposed model fits an application, or a range of applications, and thus is most effective when a developer uses it for the relevant range of applications. In certain cases, an application may need a completely custom model, made from Advanced Transaction Model Primitives, but not conforming completely to any of them, nor to any of the true advanced transaction models currently available. This would require the developer to conceptualize and implement a transaction model for the application from scratch each time a different model variant is needed, thus of course creates a problem, since it results in inefficiency in time and resources.

The motivation of this Thesis is that of providing an intermediate solution to the issues mentioned in the previous section. This can be done with the creation of a meta-model which allows the developer to either build a custom model for a transactional application under development, or use a pre-implemented template, in both cases abstracting him from the core implications of transaction handling. This would make it possible for a developer to implement the separate Units of Work in a conventional manner, without having to cater for nesting, transaction dependencies, delegation, and all issues related to transactions. The transactional behavior of each Unit of Work would then be expressed separately, possibly with the help of a specialized descriptor or scripting language. This solution in essence would be similar to the structure presented in conTract models, however offering a framework, which houses similar concepts to the ACTA, and the Ixaris LLT frameworks. Such a meta-model would allow developers to have no restrictions on the manner of operation of the transactions required by the application under development; since a possible open – source approach could possibly be taken to enhance extensibility of the meta-model itself. Extensibility may also be applied to transactional behavior, by converting the Unit of Work behavior script into an extensible one. Further development may include a graphical application which allows the developer to graphically represent Units of work, together with the transactional behavior needed for the system in question, thus reducing the learning curve for the developer.

Chapter 3: Requirements & Specification

3.1 Introduction

This chapter will provide a specification of the main modules needed for a Meta-Model based solution, as described in the previous chapter. The goal of this chapter is a detailed specification of the components, enough to permit a solid design, to the proposed solution.

3.2 General Overview

The idea is that of providing a solution which extends on the features provided by traditional transaction models, eliminating their complexities and issues at the same time. This leads us to the proposal of the Transit Model, an open source academic project which defines a meta model for Long Lived Transaction processing. Keeping in mind the theoretical information previously analyzed, together with the existing models and solutions, we shall now draw a set of requirements which would ideally be present and operational the in Transit Model solution;

The main issues with current systems are the following:

- Use restricted to a narrow range of applications.
- Those which cater for a wide range of applications are very complex to implement.
- Each solution/proposal has completely different designs, and thus there is no way of having commonly defined standards.
- Developer must have extensive knowledge of Transaction Models.
- Very few systems are completely open source.

The target to achieve during the design phase of the Transit Model will be that of conceiving a new way of handling Long Running Transactions which eliminates these issues.

3.3 Project Requirements

The main requirement of this project is that of creating a solution which allows developers to easily create long lived transaction enabled applications, without having the necessity to learn transaction modeling in depth. Thus, the solution to this project should include a series of tools and structures which make rapid development of transaction enabled applications possible. When analyzing the research material, a set of critical features which would be desired in the ideal Long Lived Transaction handling solution may be identified. Besides the resolution of the predefined issues in current systems, the Transit Model solution should ideally include or be able to manifest the following features:

- **Abstract Transactional details from developer**

As previously mentioned, in most cases developers lack expert knowledge about advanced transaction management, thus making transaction enabled development a lengthier process. Ideally, the Transit solution would encompass a complete package, which the user just adds to his solution, taking care of any transactions which take place. This would mean the total separation of transaction logic from top level application logic, where the Transit Model Solution caters for all transaction coordination and execution and the developer creates an application without transactional implications.

- **Model Plugin Concept**

One feature found lacking in many existing solutions is the possibility of changing the transaction model upon which a system executes. Current systems are tightly bound to a particular model, and thus do not support multiple models. This feature would be considered as one of the foundations of the Transit model. It has been proposed in the JSR95 specification.

- **Open Source Nature**

One key feature of this project is that should have an open source nature. The overall mind set which makes this project successful would be that of providing an initial solution, hence the artifact accompanying this thesis, and then posting it to the open source community for comments, feedback, revisions, and suggestions.

3.4 Transit Model Solution Specification

In this section a more detailed look at the solution and its components is provided. When one takes into account the previous requirements, the initial architecture of the system can be outlined. The various components are identified and described in the following text.

3.4.1 Semantics

From a theoretical point of view, the Transit Model Solution will provide an innovative solution for easy transaction enabled system development, which handles long lived transactions, including concepts such as transaction commit, abort, rollback techniques, and compensation techniques. The main theoretical frame on which the Transit Model is based is the following: The Solution consists of a scripting language which defines transaction context, into which Long Lived Transactions are plugged, thus being executed according to the context. A Long Lived Transaction may span over a substantial length of time, and consists of sub modules called activities. Activities, which may synonymously be identified as Units of work, represent the lowest possible component in transaction modeling granularity, possibly consisting of a basic workflow, or atomic transaction. Success of a transaction is greatly determined by the custom model used, and cannot be rigidly defined. With the Transit Meta-Model, it is possible to define classic models such as SAGA or NESTED models, or completely custom models.

3.4.2 Identification of Project Modules

The main components for the Transit Model Solution in order to comply with the mentioned requirements can be identified in diagram 3.4.2.1. These include:

- **The scripting Language**, which is the actual meta-model implementation;
- **The transaction manager** presented as an API, which includes a runtime engine;
- **The LLT**, which represents a long lived transaction implemented by the developer, consisting of a set of activities, and;

- **The Activities**, which may be defined as a transaction workflow which may or may not be atomic;

Essentially, the actual LLT and activities are not part of the Transit solution, but rather an implementation of the developer, which extends from an interface, present in the Transit Model API. Consider the following use case diagram:

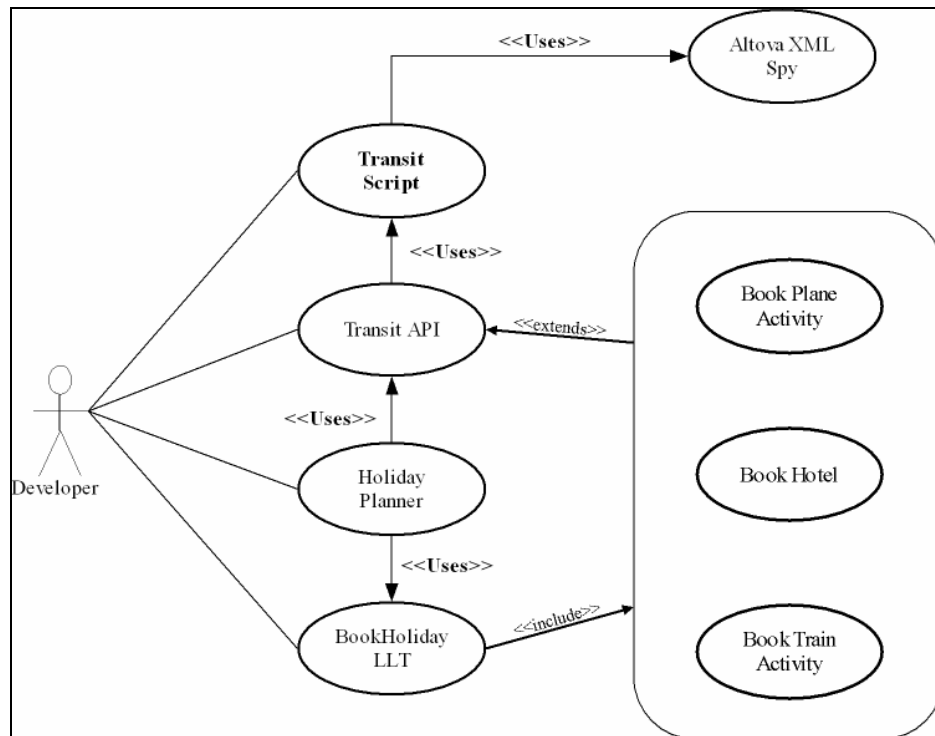


Figure 3.4.2.1 Use Case Diagram – Transit Model Solution Integrated Example

This use case diagram illustrates a typical case in which the developer makes use of the Transit Model Solution to create a Long Lived Transaction enabled holiday planning application. There are a series of simple steps which the actor has to follow before actually designing and implementing the holiday planner system, in order to make it transaction enabled. These include:

- Creating or obtaining a Transit Script which defines a transaction model
- Plugging the Script/Model into the Transaction Manager (Transit API)
- Adding the Transit API to the Holiday Planning development project
- Creating a series of Activities (BookPlane, BookTrain, Book Hotel)

- Create an LLT from these Activities (BookHoliday)

Once these steps have been carried out, development of the holiday planner application may continue normally, quite as if no transactions were being handled. This architecture fulfills the most important requirement of the Transit Model Solution, that is, the promotion of simplicity for the developer.

3.4.3 Transit Scripting Language Specification

The Transit script presented in the use case diagram, defines a custom or standard transaction model, either defined by the user, or downloaded as a template from a separate source. The script itself is not part of the transit model solution, but rather a product of it. The transit model solution thus has to include the definition of the scripting language, used to define the models. This scripting language should have the following properties:

- **Definition of a complete transaction context**

The main scope of the scripting language is that of defining a transaction context in which, a long lived transaction may execute. This includes, an actual transaction model, and transaction inter dependency definitions, as in, which activity executes before which, what happens if an activity fails, etc. The most relevant pieces of information researched in this case include the indirectly related Web services orchestration techniques, and CovaTM's definition of transaction context, a project by Bell Research Labs. (CovaTM a transaction model for cooperative Transactions)

- **An easily learnable yet complete syntax.**

Since one of the main requirements is that of simplifying transaction enabled development, a complex language definition would kill the purpose of this project, thus, the choice of XML based syntax has been made. XML is a standard language, well known, and easy to use if not known. The true power of XML lies in its simplicity, extensibility, but yet full functionality.

- **The Scripting Language should be easily extensible**

This is especially important since the project is oriented towards an open source environment. This problem is again addressed through the use of XML. Using XML, one can define virtually any language construct,

exploiting already existing technologies such as XPath, Node Traversing, and Schema Validation to parse it in a much simpler and quicker manner than fully custom language syntax.

3.4.4 Transit Model API

The Transit Model solution must also include a transaction manager which runs models defined with the transit scripting language. This engine thus enables the combination of a long lived transaction defined by a developer in a top level application such as the example Holiday Planner into the context defined by the Transit Script. The combined product is then executed using a simple interpreter in the API. The main features that the Transaction Manager should include are defined below:

- **Activity/LLT Descriptor Interfaces**

This is the first section which is to be included in TManager. It should provide an interface for developers in order to create standard Activities and Long Lived Transactions, which can be successfully interpreted by the core engine of TManager.

- **Transaction Workflow Interpreter**

This would include a simple parser, and interpreter, which translates the XML syntax of the code into Visual C#.NET code, and caters for the process of fusing the Long Lived Transaction forwarded by the developer to the transaction model, defined using the Transit Script. The interpreter should thus cater for the execution of the resulting workflow, resulting in the commission or abortion of the Transaction.

- **State Capture Structures (Suspend / Resume)**

The transaction Manager should also cater for having a system of keeping the current execution state, in order to make it possible to freeze execution flow, persist it to disk, and load it and resume it at a later time, possibly after a system restart. A possible GUI may be introduced in order to assist the suspend/Resume Process.

- **Easy Integration**

In order to allow easy integration of the Transit Model technology into other projects, it would be best to develop the Transaction Manager as an API.

The Transit Model API should also include a set of abstract or interface classes which allow developers to create a standardized form of activity, and long lived transaction, thus allowing the transaction context interpreter to easily run and monitor the execution progress of a long lived transaction.

At this point, the Transit Solution can be seen as a series of structures and tools categorized into two main modules;

The Scripting Language, which provides transaction context definition, and;

The Transaction manager, which provides an interface with which LLT's can be cast into multiple contexts and executed, suspended or resumed.

Chapter 4: Architectural Concepts

4.1 Introduction

This chapter will provide introduction to the solution's architecture, serving as basis for a detailed design of the Transit Model Solution. We will start where the specifications phase left, and provide enough detail to allow the actual systems design to be carried out. While no particular design methodology has been adopted, UML has been selected as the primary tool to illustrate the system's design.

4.2 General Architecture

Let us reconsider the requirements for the Transit Model Solution. The main idea is that of defining a Transaction Model using Scripting Language constructs similar to the idea of the ConTract Model, however at the same time allowing developers to define the context of a particular scenario through the use of a standardized structure of Activities and Long Lived Transactions based on the JSR95 LLT Model. This structure is provided as an abstract class in the Transit Model API, and plugged into the model and executed through a specialized transaction manager which is also contained in the Transit Model API:

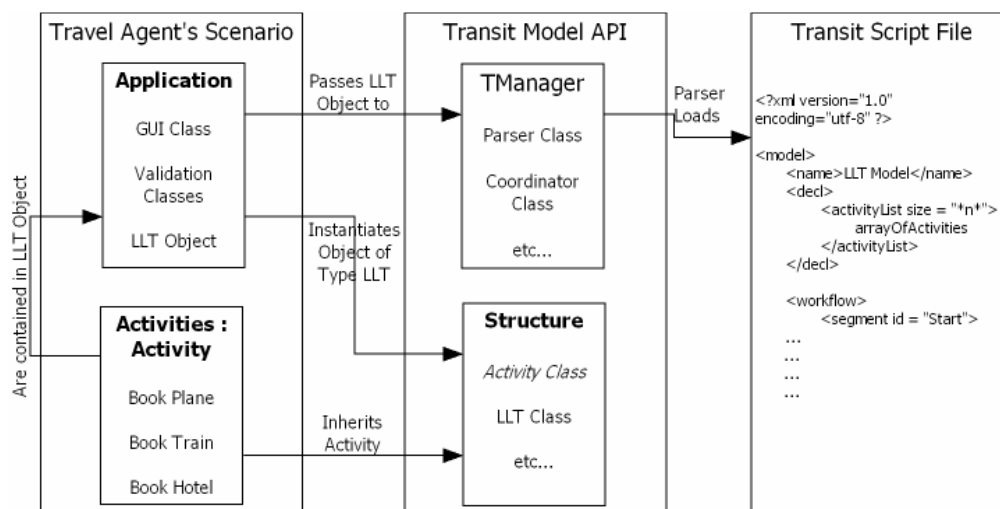


Figure 4.2.1 General Architecture

4.3 Architectural Concepts

Let us now reconsider the building blocks which make up a transaction model discussed in section 2.1.5, and apply them for the Transit Model Solution. The properties discussed are:

- Transaction Modularization
- Activity and LLT State Handling
- Transaction Context Definition and Propagation
- Transaction Inter Dependencies

4.3.1 Transaction Modularization

The first thing which must be carried out in the Transit Model Solution design is that of assigning a proper structural hierarchy to the concept of transactions. In the case of the Transit Model Solution, the best structural hierarchy deemed fit is one similar to that used in the Long Lived Transactions Model by JSR 95/Ixaris and also in the SAGA model, that is, the notion of having a long lived transaction being represented by sub activities as a top level, and transactions loosely based on ACID principles at the lowest level of granularity;

- **Long Lived Transactions (LLT's)**

As in any transaction model, the Long Lived Transaction in the Transit Model Solution will represent the highest level activity, which has a compound nature and may extend over a long period of time to complete. A transit long lived transaction can also be made up of several sub activities, themselves being long lived or atomic transactions, loosely based on ACID principles.

- **Activities (Units of work)**

As previously explained, activities are the base component of a Long Lived transaction, and in the Transit Model Solution's case will consist of a class containing a series of methods and data structures which represent the activity. Remote connections to third party entities are very likely to be established in an activity. The main idea is that the Transit Model Solution provides a base class from which the developer extends, and develops his own custom activities. While there is no restriction on whether an activity should be restricted to being atomic or not, it would be good practice to keep the granularity of an activity as low as possible, that is, it would be

better for a developer to organize the activities according to the transactional contexts they require, where activities with the same context are possibly grouped or merged. While this feature increases robustness in the solution, it is not controlled directly by the Transit Model Solution, and is purely a responsibility of the developer who is using the Transit Model Solution in his projects. This is due to the fact that while a base class for activities and LLT's will be included in the solution, the actual context definition and remote query handling must be carried out by the developer himself in a custom class, which extends from the Transit Activity descriptor Abstract Class.

4.3.2 Activity/LLT Transaction States

While it has been agreed that activities consist of a transactional workflow which will be defined by top level developers, it would be very difficult for the Transaction Manager engine to later interpret the outcome of the code execution, if some form of standard state handling is not introduced. For this reason, in the case of the Transit Model, the following Transaction Structure considerations have been assumed:

- An LLT may have two outcomes, commit, or abort, where commit indicates success, and abort indicates failure.
- An activity may have five main outcomes:
 - **Idle** - The starting state
 - **Completed** - Activity ready but result not confirmed
 - **Committed** - Activity ready and result confirmed
 - **Rolled Back** - Activity failed and result rolled back
 - **Compensated** - Activity had committed, but needs to rollback

The interdependencies between these states are depicted in the following state transition diagram:

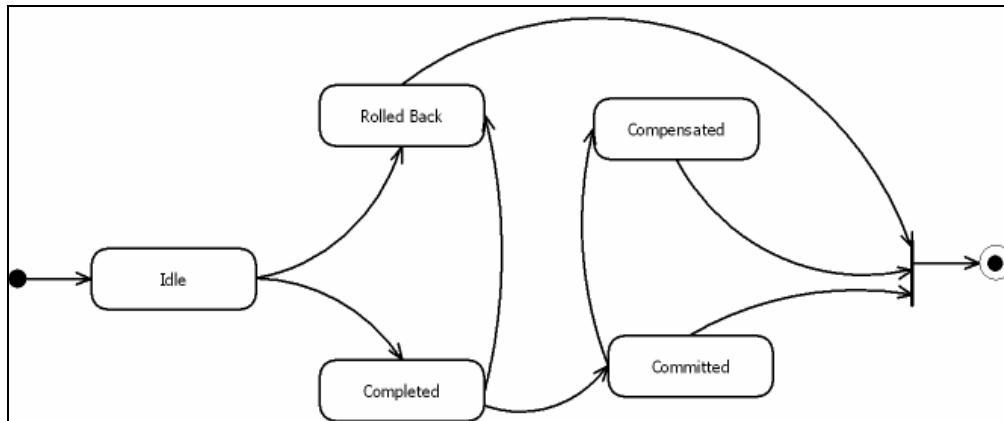


Figure 4.3.2.1 State Transition Diagram

While the “completed” and “committed” states indicate positive outcomes, the “rolled back” and “compensated” states indicate negative stances in the execution process. When an activity executes, and reaches the completed state, it has not yet committed the transaction, and can still roll back, however once the activity reaches the committed state, it cannot move to the rollback state again. Thus, reconsidering the travel agent, if a “BookPlane” activity is committed, it can not roll back, but rather compensate. In a practical context these states would represent the following example, for the BookPlane activity:

Idle	Activity not Started
Completed	Check if there is a free seat on a flight to Heathrow, next week
Rolled Back	Clear any resources and send termination message to Heathrow Server.
Committed	Re-Check if seat is still free, and confirm it.
Compensated	Try to cancel booking, if not permitted, find new customer for committed ticket.

Figure 4.3.2.2 Activity States Example

4.3.3 Transaction Contexts – Definition & Propagation

As stated in section 2.1.5.4, transaction contexts represent a particular scenario in which a Long Lived Transaction is executed. The definition of a context is shared throughout all the activities participating in a long lived transaction, thus having an impact on the interdependencies of these activities. A context is thus one execution case of a long running transaction.

In our case, defining a transaction context will be partly the task of the transaction manager, and partly the task of the script. While the Transaction Manager will cater for providing a form of standard Activity Abstract structure from which developers extend, the Script and its Interpreter will cater for the interpretation and execution of these activities which have been implemented by the developer. Thus, while the transaction context is actually defined inside an Activity, the scripting language will coordinate execution of that particular transaction context. If one observes a typical activity from a holiday booking application, contents similar to the following would be observed:

```
Begin
SQL Transaction BookPlane
SQL QUERY - Check Flight to Heathrow, Tuesday, 6 pm British Jet
IF Seat Found
SQL COMMIT BookPlane
Else
SQL ROLLBACK BookPlane
End
```

Figure 4.3.3.1 Activity Logic Pseudocode

The pseudocode above represents the transaction context definition inside an activity identified as "BookPlane", where a flight, with a particular destination, thus a particular connection to a server, is being booked. This constitutes a transaction context. Context Propagation on the other hand refers to sharing the context with all the elements in a Long Lived Transaction execution process. This will be catered for by the script, through the definition of transaction inter dependencies.

4.3.4 Transaction Inter Dependencies

As stated in the literature review, the most common transaction primitives which are used to define transaction models include the following:

- Nesting of Units of Work
- Sequential Execution
- Parallel Execution
- Recovery & Compensation Concepts
- Rollback Concepts

These concepts are the purpose of the creation of the Transit Scripting Language, which provides a series of language constructs and facilities, which allow quick and easy modeling of these concepts, into workflows and models.

4.3.5 Transaction Workflow Generation

The script's job is that of modeling a transaction workflow in a standard language syntax which may be parsed and interpreted by the appropriate classes present in the Transit Model API. The logic behind workflow generation through the script is that of having a tree structure where each language construct represents a physically defined node, and each node contains a flowlist of child constructs. Thus the best way to represent each node would be through a programmatic class, which contains an array list of child nodes. During execution, the top level node is executed, thus triggering the sequential execution of each of its child nodes. This process iterates till the bottom level nodes, which contain actual commands instead of further children. Thus commands are executed in a structured way. Further details are provided in chapter six.

4.3.6 Suspend/Resume Enabled Pluggable Model Architecture

Let us now summarise all these concepts into one concise example which skims over the general architecture. Consider our Travel Agent's Scenario, which as previously stated, will in this case be plugged into a nested transaction model. The Transit model Components needed to carry this operation out include the following:

- A nested Transaction Model Definition using the Transit Script.
- A Long Lived Transaction named "BookHoliday"
- A Series of Activities named "BookHotel", "BookTrain", "BookPlane".
- The Transaction Manager API

The nested model defines void placeholders into which activities may be plugged:

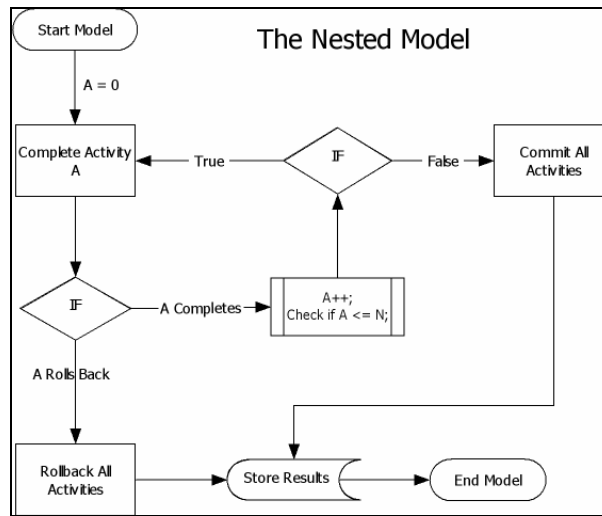


Figure 4.3.6.1 FlowChart : The Nested Model

This flowchart diagram is actually the conceptual design workflow which the Transit Scripting Language must be able to express. Besides creating or obtaining the Nested Model Script, the developer must also create a list of standardized activities, by actually using interface classes present in the Transaction Manager API, store them in an array list structure, and cast them into an LLT, which will also have an interface class defined in the Transaction Manager. This LLT is passed onto the Transit Manager, which caters for plugging the LLT into the script, thus generating a context, and then and executing it. The Transit Manager will also cater for suspend, resume capabilities, however more details about this architecture is given in the separate design sections.

4.4 Transaction Handling

Being able to handle multiple models, the Transit Model Solution does not implement a completely stand alone transaction handling mechanism. The Solution will offer a series of method calls which cater for the coordination of transaction committing, rolling back and compensation, however it will be at the discretion of the developer when to call these methods to construct the transaction context. The way in which the developer calls these methods should be in sync with the transaction model used in the application under development. While this architecture requires that the developer must have basic knowledge of what transaction commit, rollback and compensation are, it ensures the applicability of multiple transaction models.

Contexts for this project have been assumed to only contain a set of transaction states. While the Transit Model Solution will only cater for context from a transaction state switching point of view, full context propagation and handling may be achieved through a third party component such as Microsoft's System.Transaction library, or, in case of a Java implementation, Java's JTA service. These two services offer various transaction handling services, including full transaction context management, amongst others. Alternatively the developer may decide to use any other tool, if needed, however the use of these tools is preferred, since it would not make sense to re-code already existing standard tools which have been proved to be robust. Consider the travel agent's scenario previously illustrated; the pseudocode in figure 4.3.3.1 can be a JTA or System.Transaction Object, if deemed necessary by the developer.

Chapter 5: The Transit Scripting Language

5.1 Introduction

This chapter provides the Transit Scripting language specification, structure, and syntax keywords, which constitutes the Transit Meta Model. Examples of transaction models defined using this meta model are also presented in the last section.

5.2 Script Structure Considerations

In order to fulfill all the preliminary and architectural requirements, various scripting language structures have been proposed, scrapped, and redesigned from scratch. However the finally evolved and resulted in an XML based scripting language, with specially designed syntax.

5.3 Language Structure

The nature of the Transit Scripting language is that of an XML based classic imperative procedural language, whose sole aim is that of permitting the definition of transaction model templates, which will then be used in the Transit Transaction Manager. The aim is that of having a language which defines a generic workflow, possibly using expressions with "n" variables, thus making them usable for multiple applications. The language possesses the following features:

- **Classic Imperative language constructs** (For Do, If Then, etc)

A detailed definition and application of each construct is available in the following "Script Constructs" section.

- **XML based language syntax**

As previously explained, there are various advantages in the choice of having XML as the foundation syntax of the language. Constructs such as

elements and attributes suit perfectly for the creation of the Transit language constructs, while the strict hierarchical and modular structure of schema validated XML files proves ideal for reducing the chances of having a buggy transaction model created by a developer. Finally Techniques such as schema validation, node traversing, XPath, etc are all aids for the parsing and analysis process. Thus, a typical syntax of the Transit Scripting language would look similar to the following:

```
<model>
  <name>A Model Template</name>

  <decl>
    <activityList size = "*n*">
      arrayOfActivities
    </activityList>
  </decl>

  <workflow>
    ...
  </workflow>

  <main>
    ...
  </main>
</model>
```

Figure 5.3.1 Transit Script Template Preview

Where every open tag has a closing tag, and a well defined nesting can be seen. Each and every model defined with the Transit Script should have the four top level tags defined in 5.3.1; **name**, **global declaration**, which holds the a generic mapping of an n sized array of activities, **workflow tag**, which contains the actual model, and **main tag**, which sets the initial segment to be called in the workflow.

- **Named Methods**

As in any procedural language, the notion of methods, or segments, has also been introduced into the transit scripting language, thus allowing more complex transaction workflows to be defined. These methods also include the ability to pass parameters by value, thus enabling easier transaction propagation.

- **Variable Declaration and Parameter Passing Constructs**

While parameter passing by value has been engineered mainly for enabling parameter passing from one segment to the other, the Transit

Solution also includes structures for both global and local variable declaration handling. While the language handles the syntax aspect of these constructs, the back end parser and interpreter engine include state handling structures which enable parameters and declarations to be evaluated and passed. More details about these structures is provided in the "TManager Design" section.

- **Generic "n" expression handling**

This is one of the most important features included in the Transit Model Solution, since it allows the definition of generic n based models, which are suitable for Long Lived Transactions of different sizes. Using n based expressions, the following logic (defined in pseudocode) may be used to define a transaction model;

```
Let a Long Lived Transaction X have size *n*;  
For increment counter c = 1 to *n*, execute each activity till the complete state.  
If counter is equal to *n* then  
For decrement counter c = *n* to 1, execute each activity till the commit state.
```

Figure 5.3.2 "N" Based Expression Example

This is the pseudocode of a basic nested model, which still doesn't cater for transaction failure, similar to the one defined in the previous flowchart, however its main scope is demonstrating that such a model would cater for Long Lived Transactions of any size, ranging from one to *n*. This eliminates one of the crucial problems in traditional transaction modeling techniques.

- **The ability to define custom constructs through XML**

Transit's design based on XML syntax, coupled with the previously defined segmentation technique allows developers to create custom language constructs, embed them in segments, and simply call them whenever needed. These segments may also be saved as templates and used across various models. While the hard coded language syntax consists of classic imperative language constructs, these provide all the functionality needed to model any possible workflow, and subsequently create any construct. Taking a practical example, let's say that a developer wishes to

create a try catch statement; the following pseudo code defines how Transit Script may be used to define a try catch statement:

```
<segment name = "Try">  
  For increment counter c = 1 to *n*, execute each activity till the complete state.  
  If after each loop, Activity[c] has state rolled back, go to the catch segment and  
  break the loop.  
</segment>  
<segment id = "Catch">  
  For decrement counter c2 = c (passed from try), execute each activity's rollback  
  statement.  
</segment>
```

Figure 5.3.3 Custom XML Based Constructs

- **Parallel Versus Sequential Execution**

The idea of parallel execution, has been considered several times during the Architectural design of the Transit Model, however while still being considered as a desirable asset, it has been marked off as future work material, mainly due to the fact that parallel execution introduces complexities which make suspension & resumption of transactions practically unstable even if still semantically possible. Thus preference was given to the suspend/resume facility over parallel execution. Sequential language constructs are still able to model any type of transaction model in the market at present date, since none of these uses direct parallel execution.

5.4 Script Constructs

The list on the next page includes a full definition of the transit scripting language syntax, together with a case example for each construct. These constructs define the language which fulfills all the requirements discussed until now.

5.4.1 Script Constructs: The Model Tag

Syntax	<code><model></model></code>
Definition	<p>This section introduces the basic script template upon which every model should be built. As previously stated, a script should contain a model XML tag, into which a name tag, a global declaration tag, a workflow tag, and a main tag are embedded. These tags are tackled and explained individually in the following text.</p>
Rules	<ul style="list-style-type: none">• The script must always be embedded in a <code><model></code> tag.• One <code><name></code> tag containing the script's name should always be present as a first child node to the model tag.• One global declaration tag should always be included, and it should always contain one <code><activityList></code> tag which has a size XML attribute, amongst other tags of other types, if necessary.• One <code><workflow></code> tag should always be present in the script, positioned after the <code><name></code> and <code><declaration></code> tags, containing the actual script workflow.• One <code><main></code> tag should always be present as the last child node of the <code><model></code> tag.• Uppercase or lowercase format may be used for tag definitions, there is no difference in operation since the engine will convert all the tags to lowercase during the parsing process. However parameters, and variable declarations are case sensitive, thus attention should be paid when assigning them.• Standard XML format rules apply (see www.w3schools.com for details), and are enforced by the system. XML script files which are non conformant to W3C rules will not be processed by a Transit based Transaction processing system.

Sample Script	<pre> <?xml version="1.0" encoding="utf-8" ?> <model> <name><!-- Put Model Name Here --></name> <decl> <activityList size = "*"n*"> <!--arrayOfActivities --> </activityList> </decl> <workflow> ... <!--Actual Workflow, segments & imperative language constructs. --> ... </workflow> <main> <!--A goto statement indicating which segment to execute first.--> </main> </model> </pre>
----------------------	---

5.4.2 Script Constructs: Name Tag

Syntax	<name>Alpha Numeric Value</name>
Definition	The name tag simply serves as a data holder for the current name of the Transaction Model which is being defined in the script in question.
Rules	<ul style="list-style-type: none"> A script should have one instance of the <name> tag. It should be included as the first child node inside a <model> tag. The name tag has no XML attributes, and takes an alphanumeric inner text value, which represents the Model's Name.
Sample Script	<pre> <?xml version="1.0" encoding="utf-8" ?> </pre>

	<pre> <model> <name><!--Put Model Name Here --></name> ... </pre>
--	---

5.4.3 Script Constructs: Global/Local Declaration Tag

Syntax	<code><decl>...</decl></code>
Definition	The <decl> tag's main purpose is that of providing an indicator for a global or local variable declaration present in the script.
Rules	<ul style="list-style-type: none"> A script should always have one instance of the <decl> tag included as the second child node inside a <model> tag, right after the <name> tag. This should include an <activityList> tag, amongst other global declarations of type <counter>. Local declarations can also be present in script segments. These are also constituted by a <decl> tag, present in a segment tag, before the actual workflow code. A local <decl> tag possesses no attributes, and can have one or more children of type <counter>.
Sample Script	<pre> <model> <name><!--Put Model Name Here --></name> <decl> <activityList size = "*"n"> arrayOfActivities </activityList> ... <!--Variables of type <counter> --> ... </decl> ... </pre>

5.4.4 Script Constructs: ActivityList Tag

Syntax	<code><activityList>Name of LLT</activityList></code>
Definition	The activityList tag has the sole purpose of defining an abstract list of activities, which will be used in order to create the model.

	The activityList tag contains an XML attribute, named "size" which defines the size of the list. In essence the activityList has properties of an ArrayList where each position in the list signifies an activity.
Rules	<ul style="list-style-type: none"> • An <activityList> tag, should always be declared globally in a script definition. Only one instance of this tag is allowed per script. • The size attribute of this tag may be alphanumeric, since the value can either be definite, as in integer values, or indefinite, as in *n*, where *n* refers to the size of the list.
Sample Script	<pre> <model> <name><!--Put Model Name Here --></name> <decl> <activityList size = "*n*"> arrayOfActivities </activityList> ... <!--Variables of type <counter> --> ... </decl> ... </pre>

5.4.5 Script Constructs: Counter Tag

Syntax	<pre> <counter value = "V">Name of Variable</activityList> </pre> <p>where V is a Natural Number</p>
Definition	The counter tag is used in the <decl> tag in order to declare a local or global variable of type Integer. The Inner Text of this tag is considered to represent the variable name, while the value is stored inside a value attribute.
Rules	<ul style="list-style-type: none"> • While any amount of declarations is allowed, a <counter> tag may be used only inside a <decl> tag. • The value attribute of this tag must always be of type natural number, since the value can only be of definite

	<p>type.</p> <ul style="list-style-type: none"> Counter tags can be assigned a value externally by <goto> statements. This is done if a <goto> statement has a parameter attribute which has the same name as a local variable in the segment it is calling. If this is the case, the local variable, takes the parameter's value.
Sample Script	<pre> <?xml version="1.0" encoding="utf-8" ?> <model> <name>Put Model Name Here</name> <decl> <activityList size = "*n*"> arrayOfActivities </activityList> <counter value = "0">globalk</counter> </decl> <workflow> <segment id = "A Segment"> <decl> <counter value = "0">k</counter> </decl> ... </segment> </workflow> </pre>

5.4.6 Script Constructs: WorkFlow Tag

Syntax	<code><workflow>...</workflow></code>
Definition	The scope of the workflow tag is that of containing the actual workflow definition of the model, described using classic imperative language constructs.
Rules	<ul style="list-style-type: none"> Every script should contain one workflow tag, placed after the global declarations. Workflow tags do not possess XML attributes. The workflow tag must contain one or more child notes of type <segment>.
Sample	<code><workflow></code>

Script	<pre> <segment id = "A Segment"> <decl> <counter value = "0">k</counter> </decl> ... </pre>
---------------	---

5.4.7 Script Constructs: Segment Tag

Syntax	<pre><segment id = "X">...</segment></pre> <p>Where id is Alphanumeric</p>
Definition	<p>The segment tag is responsible for containing the core part of the Transit Script, where the actual workflow resides. The segment tag has an XML attribute named "id" whose value represents the name of the segment. This name is used by <goto> statements in order to call the segment.</p>
Rules	<ul style="list-style-type: none"> Segment tags should always be contained in a workflow tag. Multiple segment tags are allowed, however each one must have a unique value in the "id" attribute. Parameters may be passed to segments from <goto> statements. This is done by declaring a local variable inside the segment tag, which has the same name as a parameter which is being passed. The TManager engine will then automatically cater for value mapping. Please note that recursion is not permitted in the Transit Script. The child structure of a segment should include, primarily any variable declarations, and then a <begin> tag.
Sample Script	<pre> <workflow> <segment id = "A Segment"> <decl> <counter value = "0">k</counter> </decl> <begin> ... </segment> </pre>

5.4.8 Script Constructs: Begin Tag

Syntax	<code><begin>...</begin></code>
Definition	The begin tag is the first tag which servers as an indicator point for the parser that the actual workflow definition has begun. From this point onwards, the script takes a more "Procedural 3 rd Generation Language" look.
Rules	<ul style="list-style-type: none"> There are no strict rules for the content of the begin tag, as long as it contains one of the following tags: <code><fordo></code>, <code><ifthen></code>, <code><elseif></code>, <code><execute></code>, <code><goto></code>, or <code><cmd></code>. A begin tag should always be used inside a <code><segment></code> tag, and should follow and <code><decl></code> tags which define local variables. Only one begin tag is allowed per segment.
Sample Script	<pre> <workflow> <segment id = "A Segment"> <decl> <counter value = "0">k</counter> </decl> <begin> </begin> </segment> </pre>

5.4.9 Script Constructs: For Do Tag

Syntax	<code><fordo begin = "A" end = "B" counter = "C" step = "D">...</fordo></code> Where: A,B and C are *n* based expressions Where D is either ++ or --
Definition	The <code><fordo></code> tag is similar to the for loop in the C# and Java languages. It contains four attributes in all; the "begin" and "end" attributes indicating the starting and ending value through which to loop, the "counter" indicating the variable used to keep the current value, and the "step" attribute indicating whether the loop is ascending or descending step values.

Rules	<ul style="list-style-type: none"> • The fordo must always be contained inside a begin statement. • Nesting is allowed, thus a <fordo> can contain another <fordo> • The begin and end attributes may contain variable names which have been locally or globally declared instead of literal values. These are then converted into a natural number the parent segment is called through a <goto> statement, which passes variable values. • The counter attribute's value must be alphanumeric, and must match the name of a locally or globally declared variable. This variable will hold the value of the current loop count. • The step attribute must always contain either ++ for step up, or – for step down loops. • A for do statement can contain the same tags as a <begin> statement.
Sample Script	<pre> <workflow> <segment id = "Start"> <decl> <counter value = "0">k</counter> </decl> <begin> <fordo begin = "paramone" end = "paramtwo" counter = "k" step = "++"> </fordo> ... </pre>

5.4.10 Script Constructs: If Then and Else If Tags

Syntax	<pre> <ifthen type = "normal" index = "A" result = "B" >...</ifthen> </pre> <p>Where: A is an *n* based expression Where B is "completed/committed/rolledback/compensated"</p>
---------------	--

	<p style="text-align: center;">OR</p> <p style="text-align: center;"><ifthen type = "expression" expression1 = "A" operator = "B" expression2 = "C"></p> <p style="text-align: center;">Where A and C are *n* based expressions including + or – Where B is one of the operators (<, >, <=, >=, ==)</p>
Definition	<p>The <ifthen> tag is also similar to the if then else loop in the C# and Java languages. However the use of if then statements in the transit model is restricted to two types; those which check the outcome of the execution of an activity, and those which evaluate expressions, as seen in the syntax formats above. The "type" attribute present in the tag has two values, "normal", which indicates that the statement is an expression outcome evaluator, or "expression" which indicates that the statement is an expression evaluator.</p> <p>In the "normal" statement, the "index" attribute indicates the position of the Activity in the "activityList", which is under question, while the result indicates the expected outcome.</p> <p>In the "expression" statement, the attributes "expression1" and "expression2" may contain alphanumeric expressions with operators + or -, while the operator attribute may contain a selection of Boolean operators.</p>
Rules	<ul style="list-style-type: none"> • The <ifthen> must always be contained inside a begin statement. • Nesting is allowed, thus an <ifthen> can contain another <ifthen> • The <ifthen> tag can contain any structure which the <begin> tag or the <fordo> tags contain. (<fordo>, <ifthen>, <elseif>, <cmd>, etc...) • The index attribute in the normal <ifthen>, and the expression attributes in the expression valuator <ifthen> may contain *n* based expressions, or natural numbers. • When an <ifthen> tag closes, it may be immediately followed by an <elseif> tag, which possesses the same attribute properties of the <ifthen> tag, or an <else> tag with no statements, which simply executes the child notes

	inside if if the <ifthen> or <elseif> statements preceding it fail.
Sample Script	<pre> <workflow> <segment id = "Start"> <decl> <counter value = "0">k</counter> </decl> <begin> <ifthen index = "k" result = "rolledback" type ="normal"> ... </ifthen> <elseif type = "expression" expression1 = "k" operator = "<" expression2 = "*n*> ... </elseif> <else> ... </else> ... </begin> ... </pre>

5.4.11 Script Constructs: Execute Tag

Syntax	<p><execute position = "A" type = "B">LLT Name</execute></p> <p>Where A is an *n* based expression Where B is "complete/commit/rollback/compensate"</p>
Definition	<p>This construct is the most important construct in the script, since it maps an activity from the LLT provided by the developer, and executes it according to the parameters defined in this statement. The execute statement has two attributes, the position, which indicates the actual position of the activity to process in the activityList, and the type, which defines till what level should the execution proceed.</p>
Rules	<ul style="list-style-type: none"> Execute statements can only be used inside a begin tag, inside a segment.

	<ul style="list-style-type: none"> • An Activity may be executed several times, progressively, starting from type complete, and moving on to type commit, to type compensate. The same state cannot be executed twice, as this would cause not make sense in a transactional context. The previously explained rules apply, where if an activity commits, it cannot be rolled back, but has to be compensated. • The position of the activity to execute may be expressed either by a natural number, or by an <i>*n*</i> based expression. • An execute statement does not contain child notes, but its inner Text represents the name of the activityList from which Activities are being processed.
Sample Script	<pre> <begin> <execute position = "k" type = "commit"> arrayOfActivities </execute> </pre>

5.4.12 Script Constructs: Goto Tag

Syntax	<pre> <goto paramone = "A" paramtwo = "B">Segment Name</goto> </pre> <p>Where A and B are <i>*n*</i> type expressions</p>
Definition	<p>The <goto> statement has the main task of issuing calls to segments, either from the main program, or from within a segment itself. Unlike the classic "goto" statement in assembly language, this goto does not promote spaghetti code, since it can only issue segment calls, similar to a method call in C# or Java. The <goto> statement can have an indefinite number of elements, which act as parameters in order to pass values to global or local variables. The engine matches the name of the attribute (for example: paramone), to a the name of a variable inside a segment, or a global variable, and propagates the parameter value to it. Thus parmeters may be passed between segments through the <goto> statement.</p>

Rules	<ul style="list-style-type: none"> • The <goto> statement can only be used inside a <begin> tag, where multiple instances are allowed, or inside the <main> tag, where only one instance is allowed. • The parameter names should match already existent variables which have been globally or locally declared. • The inner text of the command should match a segment which is listed inside a <workflow> tag inside the same script file.
Sample Script	<pre> <ifthen index = "k" result = "rolledback" type ="normal"> <goto paramone = "k-1" paramtwo = "0">CompensateAll</goto> <cmd>exitscript</cmd> </ifthen> </pre>

5.4.13 Script Constructs: CMD Tag

Syntax	<code><cmd>exitscript</cmd></code>
Definition	This is a simple command which is part of the workflow, and at present contains only one command, which is the "exitscript" command. As soon as this tag is found, its inner text is analysed, and the corresponding command is executed. Plans to extend this tag are classified as future work.
Rules	<ul style="list-style-type: none"> • <cmd> statements can only be used inside a begin tag, inside a segment. • Since at present, <cmd> has only the "exitscript" command, it can be stated that <cmd> is solely used to exit the script in case a transaction fails, however this may be extended in future versions.
Sample Script	<pre> <ifthen index = "k" result = "rolledback" type ="normal"> <goto paramone = "k-1" paramtwo = "0">CompensateAll</goto> </pre>

	<pre> <cmd>exitscript</cmd> </ifthen> ... </pre>
--	--

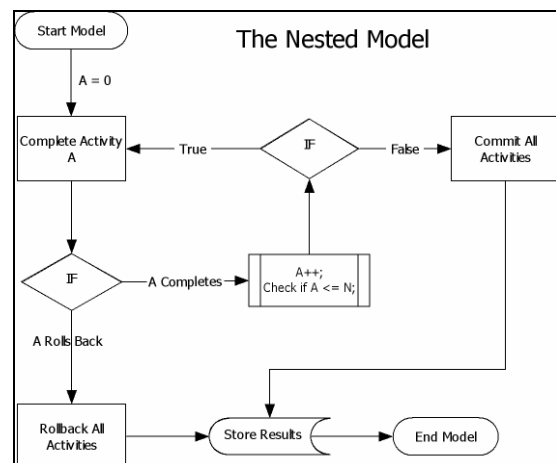
5.4.14 Script Constructs: Main Tag

Syntax	<code><main>...</main></code>
Definition	The main tag has the simple scope of containing one <code><goto></code> statement, which indicates the first segment which must be called upon initial execution.
Rules	<ul style="list-style-type: none"> The <code><main></code> can only be used once in a script, and it should be placed as the final child of the model tag, after the <code><workflow></code> tag. The <code><main></code> tag is only allowed to have one child of type <code><goto></code> which indicates the starting segment, and passes initialization parameters.
Sample Script	<pre> <main> <goto paramone = "0" paramtwo = "*n*">Start</goto> </main> </pre>

5.5 Examples

Let us reconsider the previous flowchart, where a generic nested model was described.

This model, as described before, may now be translated into a Transit script, which caters for all requirements, and includes the “placeholders” for activities to be plugged in. These are the `<execute>` tags, which define interdependencies, through execution conditions, and `*n*` based expressions. The resulting Transit XML based script will look similar to the following example:



```
<?xml version="1.0" encoding="utf-8" ?>

<model>
  <name>Nested Model</name>

  <decl>
    <activityList size = "*"n">
      arrayOfActivities
    </activityList>
  </decl>

  <workflow>
    <segment id = "Start">
      <decl>
        <counter value = "0">k</counter>
      </decl>
      <begin>
        <fordo begin = "paramone"
          end = "paramtwo"
          counter = "k"
          step = "++">

          <execute position = "k" type = "complete">
            arrayOfActivities
          </execute>

          <ifthen index = "k" result = "rolledback" type = "normal">
            <goto paramone = "k-1"
              paramtwo = "0">

              RollbackAll
            </goto>
            <cmd>exitscript</cmd>
          </ifthen>
        </fordo>
        <ifthen type = "expression"
          expression1 = "k"
          operator = "=="
          expression2 = "paramtwo">

          <goto paramone = "paramone"
            paramtwo = "paramtwo">

            CommitAll
          </goto>
        </ifthen>
      </begin>
    </segment>

    <segment id = "RollbackAll">
      <decl>
        <counter value = "0">k</counter>
      </decl>
      <begin>
        <fordo begin = "paramone"
          end = "paramtwo"
```

```
        counter = "k"
        step = "--">

        <execute position = "k" type = "rollback">
            arrayOfActivities
        </execute>
    </for>
</begin>
</segment>

<segment id = "CommitAll">
    <decl>
        <counter value = "0">k</counter>
    </decl>
    <begin>
        <for> begin = "paramone"
            end = "paramtwo"
            counter = "k"
            step = "++">
                <execute position = "k" type = "commit">
                    arrayOfActivities
                </execute>
            </for>
        </begin>
    </segment>
</workflow>
<main>
    <goto paramone = "0" paramtwo = "n">Start</goto>
</main>
</model>
```

Figure 5.5.1 Nested Model Transit Script

This sample script proves that transaction models can be defined using an XML based meta-model scripting language, in a quite trivial manner. This also proves the effectiveness of the concepts and ideas introduced to developers through the Transit Model Solution, particularly the idea of having exploiting the scripting language's simple constructs to define the workflows. In addition, the script can be very easily developed using any XML editor such as Altova's XML Spy, since it strictly conforms to XML standards.

In the following section, the design of the TransitModel is defines. The TransitModel API has various complex sub architectures and algorithms which allow its operation. These include the flowlist based execution architecture, state handling architecture, which allows suspension and resumption of transactions, and the parameter passing framework amongst others. While this section included pure theory and design of language syntax, the next section includes conventional application design.

Chapter 6: The TransitModel API

6.1 General Architecture

As previously stated, the Transit model API has a twofold task; that of providing a structure descriptor interface, which the developer may extend in order to create a standardized LLT from an array of Activities; and that of providing an execution engine, which allows suspension and resumption of Long Lived Transactions. Thus the best way of proceeding with the design process is as described in the following package diagram:

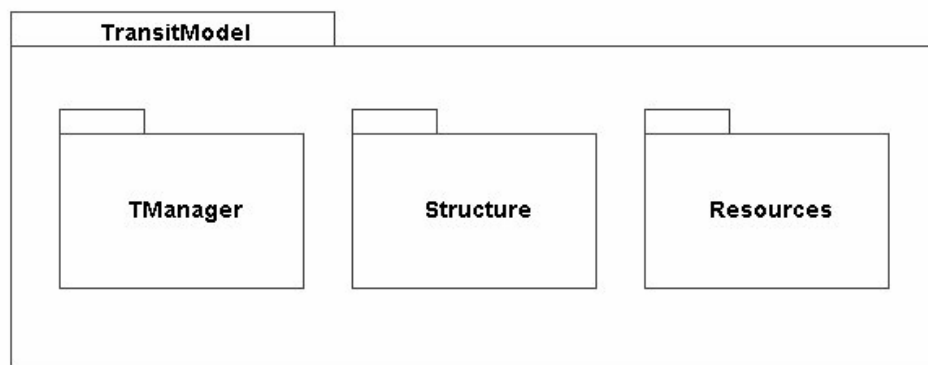


Figure 6.1.1 Package Diagram – The Transit Model

The Transit Model API is split into two main namespaces, the `TransitModel.Structure` namespace which provides the structure needed by developers; and the `TransitModel.TManager` namespace, which caters for the execution, suspension, and resumption of Long Lived Transactions. The resources package considered of secondary importance, as it will simply contain shared embedded resources, such as icons, button images, etc... these namespaces cater for the creation of an appropriate transaction handling environment.

6.2 TransitModel.Structure

This namespace represents the "base abstract class", previously described in Chapter 4 (Section 4.2.1). At this point it has been expanded into a complete

namespace, rather than a simple class. The main task of this namespace is to provide precise abstract definitions of Activities and LLT's from which the developer can extend in order to create a fully fledged Long Lived Transaction in his application. This allows a developer to exploit this namespace's facilities to create an LLT instance, which he then passes to TransitModel.TManager in order to be executed. Thus all the structural definition and state handling methods must be present in this namespace, in order to allow the developer to create an LLT which is interpretable by the TManager component. This leads to the creation of two main classes; an Activity Class of type abstract, and an LLT class, both classes handling Activity/LLT structure and states respectively.

6.2.1 TransitModel.Structure – Use Case

The following diagram describes the case in which a developer is making use of the TransitModel API's Structure namespace, in order to create a Long Lived Transaction to book a holiday. Note that the representation of the TransitModel.TManager module in this diagram is just indicative, since the TransitModel.TManager design has not been illustrated yet.

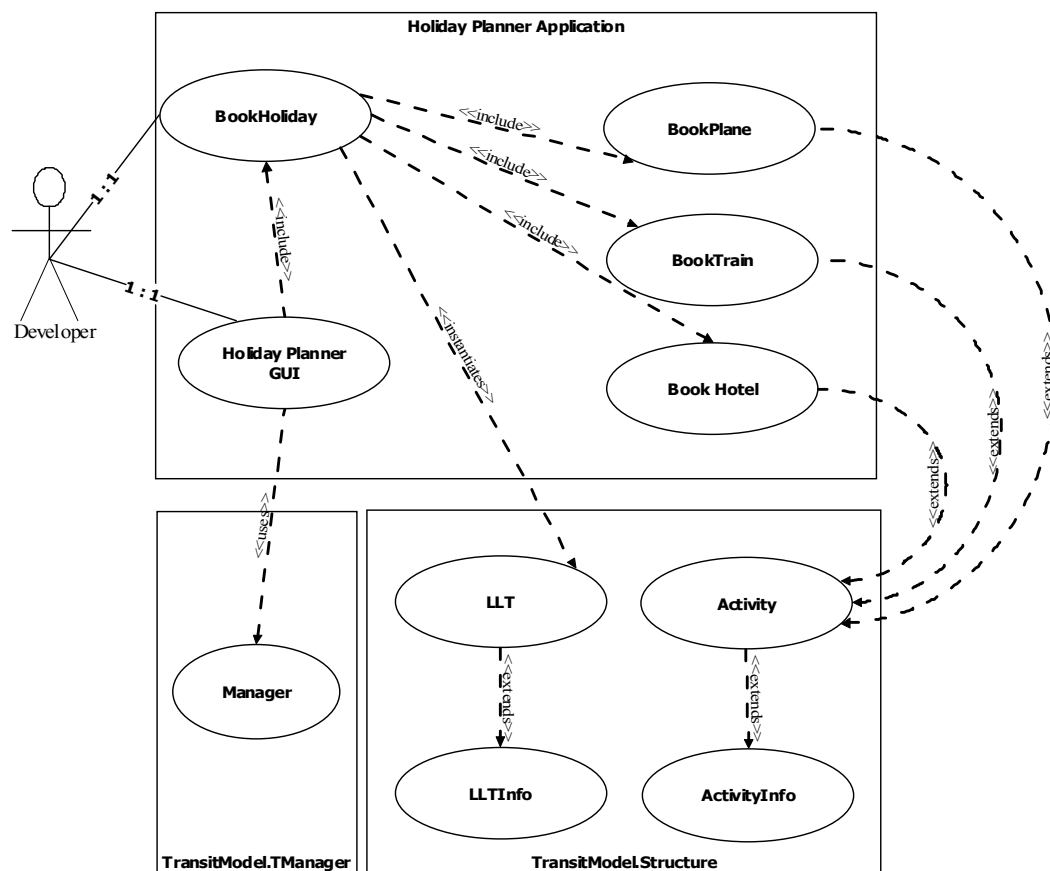


Figure 6.2.1.1 Use Case for TransitModel.Structure

The developer creates one long lived transaction, hence the 1 : 1 relation with bookHoliday. This is done by creating an object instance of the TransitModel.Structure.LLT class. In this case we are assuming that the developer handles this in the GUI class, which is implemented by him. The BookPlane, BookTrain, and BookHotel Classes, also implemented by the developer, extend TransitModel.Structure.Activity, and implement specialized methods which condition the mode of execution of an activity. Each method is defined in the TransitModel.Structure.Activity class as a stub. Finally, an instance of each of the activity classes is created, and is added to the BookHoliday LLT object, which in turn is passed to TransitModel.TManager. The TManager then handles execution, suspension and resumption of the Long Lived Transaction.

6.2.2 TransitModel.Structure – Class Diagram

The diagram presented on the next page represents a class diagram which contains the structure and inter relations of the classes present in the TransitModel.Structure namespace. While this diagram gives full details about the classes and their methods, each class is analysed in depth separately, in the sections which follow;

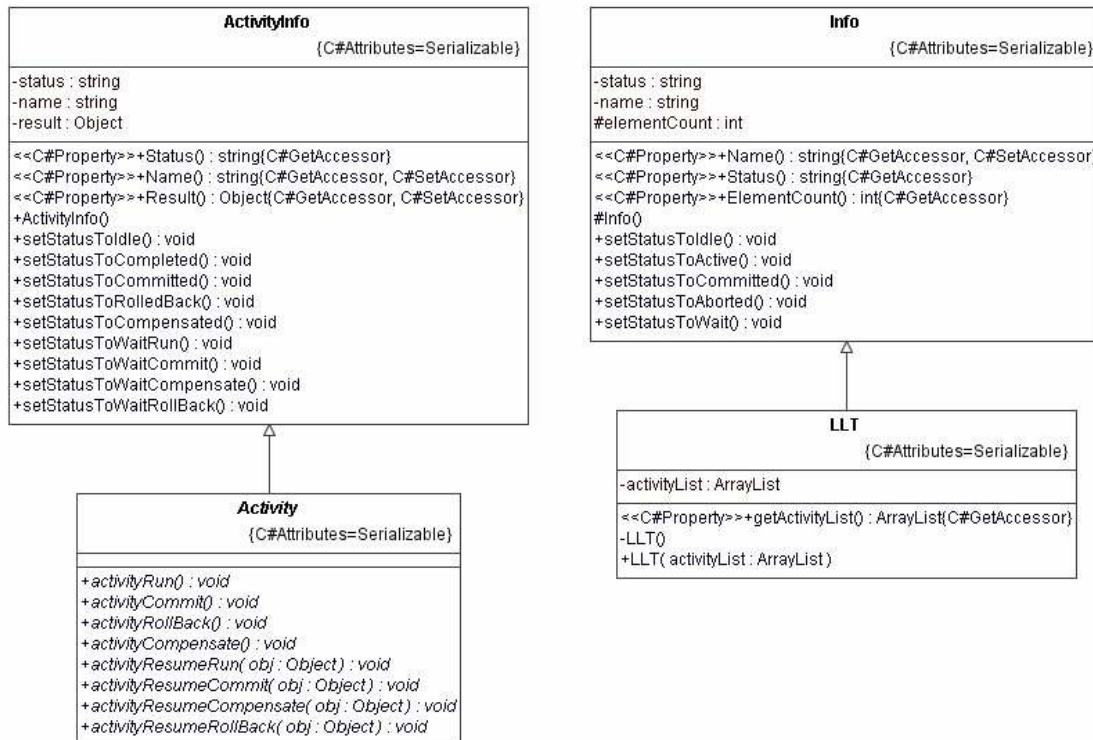


Figure 6.2.2.1 Class Diagram for TransitModel.Structure

While the LLT class contains the `activityList` array list to store a number of activity objects, the activity class contains the previously mentioned stub definitions which allow a user to run, commit, rollback or compensate an activity, while suspending or resuming the LLT at any stage. These method descriptions must be overridden and implemented by the developer, when extending the activity class.

Two state managing classes have been also been added, the `ActivityInformation`, and the `LLTInformation` classes. The job of these classes is to contain the current state of an activity, and provide custom get and set methods which allow state changes. Thus the `Activity` class and `LLT` class must inherit from these two classes respectively, as shown in the diagram. Please note that due to the suspension/resumption logic, each of these classes must be marked as serializable, since they may be persisted to disk. More details on the suspension mechanism are provided in the `TransitModel`.

6.2.3 TransitModel.Structure.Activity

The series of specialized methods whose stubs are defined in the `Activity` abstract class are considered as distinct steps in an activity, since they directly affect the state of the activity, thus leading to a state change. Actually, the methods are directly related to the possible states of an activity. These methods include:

- **`activityRun();`** - Which caters for running an activity.
- **`activityCommit();`** - Which caters for committing an activity.
- **`activityRollBack();`** - Which caters for Roll backing an activity.
- **`activityCompensate();`** - Which caters for Compensating an activity.

If we reconsider the requirements for an activity discussed in chapter 4, it can be seen that what is being done is simply segmenting the flow of work in an activity in a structured manner, to allow interpretation by the `TManager`;

```
Begin
SQL Transaction BookPlane
SQL QUERY - Check Flight to Heathrow, Tuesday, 6 pm British Jet
IF Seat Found
SQL COMMIT BookPlane
Else
SQL ROLLBACK BookPlane
End
```

Figure 6.2.3.1 Pseudocode for an Activity Workflow

Each SQL statement in the original Activity pseudocode will now be segmented into the four methods, where each method indicates a state change. The main aim of this, besides better interpretation by the TManager, is to allow better management of states of an activity, and better suspension or resumption handling of an activity at any of these four stages. A typical method implementation carried out by the developer will be similar to the following:

```
public override void activityRun()
{
    try
    {
        //Carry out remote server request
        //If response is positive
        this.setStatusToCompleted(); //Transaction Successful

        //Else if response is negative
        this.setStatusToRolledBack(); //Transaction Failed
    }
    catch
    {
        // If Server connection has been lost
        this.setStatusToWaitRun();
    }
}
```

Figure 6.2.3.2 Pseudocode For an Activity Method

The activityRun() method represents the method which caters for running the activity. While the Activity class defines four methods, the ActivityInfo class is responsible for keeping the current Activity's status by providing status management methods such as setStatusToCompleted(). This means that the activity class must inherit ActivityInfo's properties and methods. For a full list of methods see the TransitModel.Structure.ActivityInfo section.

The second important issue which, even though handled by the TManager, has its framework defined in the Structure namespace, is the concept of suspension and resumption of activities. One may notice the try catch statement in the previous code example. This stipulates that, if a server connection error Exception is thrown, the Activity Status is set to Waiting; which means that the activity has been suspended. A suspended activity may be resumed, by calling a set of resume methods, which mirror the Activity's four standard methods. These methods are also defined in the abstract class, and must also be implemented by the developer. The resume methods include:

- **activityResumeRun(Object o);**

Which caters for resuming an activity which switched to suspended state in the activityRun() method.

- **activityResumeCommit(Object o);**

Which caters for resuming an activity which switched to suspended state in the activityCommit() method.

- **activityResumeRollBack(Object o);**

Which caters for resuming an activity which switched to suspended state in the activityRollBack() method.

- **activityResumeCompensate(Object o);**

Which caters for resuming an activity which switched to suspended state in the activityCompensate() method.

These methods allow the developer to resume an activity, according the step in which it had switched to suspended state. In fact, the generic object parameter has the sole purpose of allowing the user of the final system to manually input any necessary information that an activity may need in order to resume; input which would have been sent automatically by the server, had the connection not failed. The manual input process will occur through Resume handler GUI provided by the TManager namespace.

6.2.4 TransitModel.Structure.ActivityInfo

As explained, this class keeps the current state of an activity which inherits it, and provides the following public methods which aid state handling by the developer;

- **setStatusToIdle();**
- **setStatusToCompleted();**
- **setStatusToCommitted();**
- **setStatusToRolledBack();**
- **setStatusToCompensated();**

While these methods indicate the actual status of an activity, a set or mirroring methods indicate the suspended state of an activity, according to the standard method in which it switched to suspended state:

- **setStatusToWaitRun();**
- **setStatusToWaitCommit();**
- **setStatusToWaitRollBack();**

- **setStatusToWaitCompensate();**

The status system basically works by having a global variable named "status" in the activityInfo class, and having the listed methods change it accordingly.

6.2.5 TransitModel.Structure.LLT

The LLT class is a conventional class, containing the previously discussed public array list structure, into which Activities may be inserted. While originally intended to be abstract with the inclusion of execution methods, it has been preferred to move control handling completely to the TransitModel.TManager namespace and use this namespace purely for structural purposes. Having a similar nature, the only difference between the LLT class and the Activity Class is that the Activity class is abstract and contains stub methods, thus being intended to be extended, while the LLT class is conventional, and is simply instantiated.

Once the developer has created a system of activities which extend from the Activity Class and implemented each of its compulsory methods, each class is considered to be a complete activity. It can then be simply instantiated from a controller class such as the Holiday Planner GUI in the previous use case diagram, and added to an LLT Object. This LLT Object is then passed on to the TManager for processing and execution.

6.2.6 TransitModel.Structure.Info

This class has a very similar, yet simpler purpose to the ActivityInfo class. It also contains state handling getter and setter methods, which may be used by the developer to switch the state of the overall LLT:

- **setStatusToIdle();**
- **setStatusToActive();**
- **setStatusToCommitted();**
- **setStatusToAborted();**
- **setStatusToWait();**

These methods function with the same current status global variable string concept, however adhering to the statement made in the specifications section which states that the outcome of an LLT is either committing or rolling back. In this case the active state indicates that the LLT is currently in processing, and the wait state indicates that the one of the LLT's activities have been suspended.

6.3 TransitModel.TManager

The TManager namespace contains the interpreter and runtime engine, which caters for execution of the LLT and handles suspension and resumption issues. The Resume GUI mentioned in the specifications section has also been catered for. TManager has the responsibility of abstracting the developer completely from transactional issues, except for the LLT structure definition through the TransitModel.Structure namespace, which is quite simple in itself. The TManager namespace is further subdivided into two main namespaces, as displayed in the following package diagram:

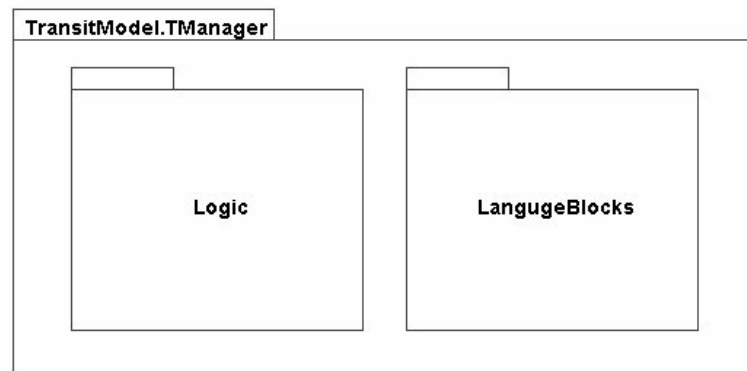


Figure 6.3.1 Package Diagram for TransitModel.TManager

These two sub name spaces contain a series of classes which handle the following set of tasks:

- Language Parsing.
- Generation of a workflow.
- Execution of the workflow.
- State switching according to workflow.
- Suspension and Persistance to disk, and resumption of an LLT.

Since these tasks are closely related, there is no clear cut logical distinction between the handling of each task in the Solution. While logical design of the system has overlapping tasks, physical design still consists of distinct classes, each with a separate task. In fact the system design includes an architecture of closely related modules each of which contribute to the system by carrying out their distinct tasks. These include a parser, an execution engine, language descriptor classes, and abstract classes which provide transaction structure. The component structure of the TManager includes the following items:

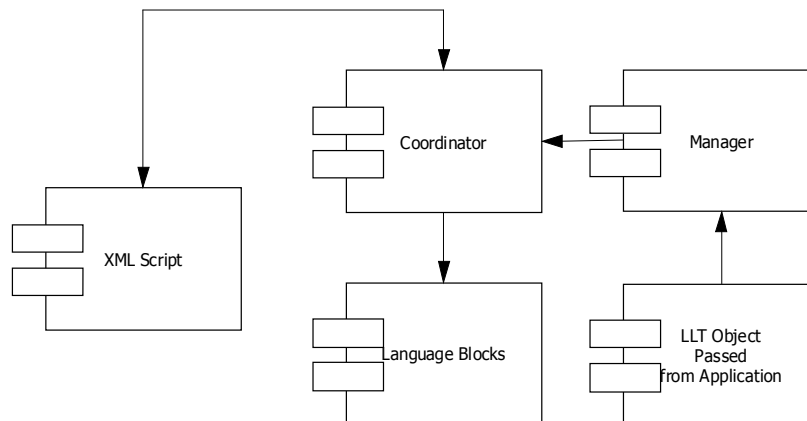


Figure 6.3.2 Component Diagram for TransitModel.TManager

The solution operates in the following manner; a reference to an instance of the Manager component is obtained by the developer in his application, to which he passes the LLT which has been constructed using the TransitModel.Structure namespace. Upon instantiation of the Manager component, the Manager component requests a reference to an instance of the coordinator component, forwarding the obtained LLT to it in the process. At this point, the actual parsing and workflow generation process commences.

6.3.1 TransitModel.TManager.Logic

This namespace is responsible for providing both a programmatic and a graphical user interface, with which the developer or end user may access the Transit Model solution externally, in order to either process transaction results, or in order to integrate it into another top level solution. It contains three main classes, the Manager, the TransitControlPanel, and the Coordinator.

6.3.1.1 TransitModel.TManager.Logic.Mgr

The Manager class is one of the two classes in the Transit Model Solution whose methods are public, thus completely viewable and evocable by external users, when keeping in mind that the final produce will consist of an API. It is of type singleton, thus having only one possible instance of it at all times. This has been done in order to avoid putting the external developer in confusing situations where multiple instances of the manager are present. The manager contains the methods necessary for general operation of the solution. These include:

- **Get Instance();** - This method returns a method instance, and automatically retrieves a coordinator instance.
- **BrandNewLLT();** - This method creates calls the necessary functions, passing the LLT Object which the user provided, in order to return a parsed Model Object, which is ready to be executed.
- **LoadLLT();** - Loads a suspended LLT Object from disk, which is ready to be executed.
- **AbortLLT();** - Clears from memory any information about the currently loaded LLT.
- **runLLT();** - Actually runs the Long Lived Transaction by calling the executeNode() method in the Main object found within the Model object.

It is the manager which amalgamates all the logic contained in the Transit Model Solution, in order to process a transaction. It accepts an LLT from the user as a parameter; parses the script and fuses the Activities in the LLT using the coordinator class, and executes the transaction by calling the appropriate execute methods. For execution details, please see the following sections.

6.3.1.2 TransitModel.TManager.Logic.TransitControlPanel

The TransitControlPanel class serves as a simple Graphical User Interface which the developer may access externally by invoking it, in case a manual input needs to be used for Transaction resumption purposes. Through the GUI, the end user may decide to suspend or abort a suspended transaction through a series of buttons. A simple transaction logger is also provided. The GUI is also based on the singleton model, allowing only one instance of the class at all times. Screenshots are provided in the suspension and resumption concepts section.

6.3.1.3 TransitModel.TManager.Logic.Coordinator

The coordinator class caters for the generation of a complex architecture, obtained through parsing and workflow generation, which enables easy execution of the Long Lived transaction provided by the user, according to a Transit Script which is embedded in the system. The coordinator loads, parses, and creates this architecture, or workflow, which is then executed by the manager. A method in order to parse each language construct is present, resulting in the formation of Language Blocks. For the method listing of this

class, please see the Appendix Section. For an explanation of the language blocks concept, please see the following sections

6.3.2 TransitModel.TManager.LanguageBlocks

The language blocks namespace can be considered to contain a series of classes which map the XML's constructs into c# objects, while constructing a complete architecture which renders execution easy. Thus, taking an example, for a <fordo> xml tag, TManager.LanguageBlocks contains a fordo class, which caters for all conditions and issues of the fordo class. The most important classes in the LanguageBlocks namespace are highlighted below.

6.3.2.1 TransitModel.TManager.LanguageBlocks.IBlock

This is an abstract class, with various utilities which allow advanced parameter handling and variable value assignments, in classes which inherit from it. All the classes which directly make part of the workflow architecture inherit from IBlock, and thus have to implement the stubs defined in the IBlock class. Following is a class diagram representing the IBlock abstract class, which plays a very important part in the execution process, especially variable and parameter handling:

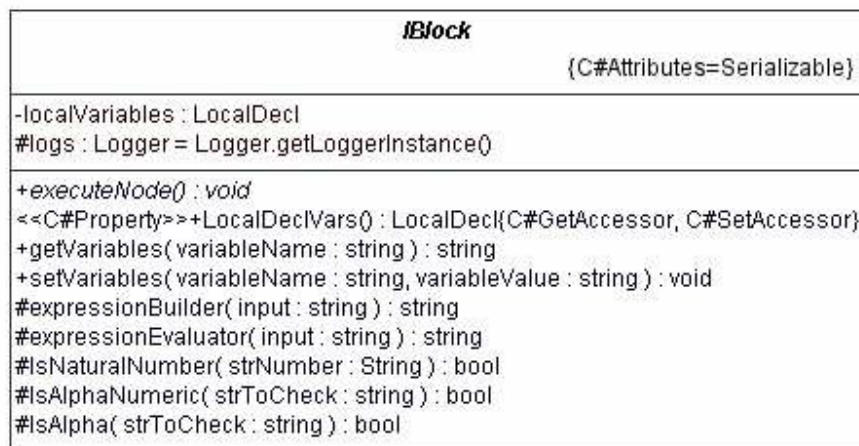


Figure 6.3.2.1.1 Class Diagram for the IBlock Component

The various relevant methods and properties present in this class are further explained in the following sections. Please note that the localDecl, variable refers to a list of local variables, which is present in each Language Block, while the executeNode stub refers to the method which must be implemented by each language block, which makes part of the resulting workflow. Taking an example,

a ForDo Object inherits from IBlock, and thus must implement the method `executeNode()`, which in turn executes all the children in the fordo loop, for the number of times specified by conditions parsed from the script. The `getVariables` and `setVariables` methods are explained in detailed in the following sections.

6.3.2.2 TransitModel.TManager.LanguageBlocks.Structs

The structs namespace contains any necessary structures or dataholder classes which are needed by any class in the TransitModel.TManager namespace. The most important class in this namespace is the StateHolder class, which is used extensively for suspension/resumption purposes. More details are provided in the “concepts” sections.

6.3.2.3 TransitModel.TManager.LanguageBlocks.Main

The main class is a class of type LanguageBlock, and inherits from the IBlock abstract class. The reason for highlighting this class is that it serves as a starting point for execution of the LLT. In fact, the structure of the main language block can be identified in the following class diagram:

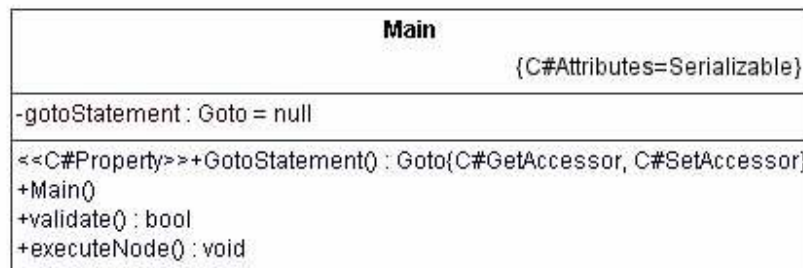


Figure 6.3.2.3.1 Class Diagram for the Main Language Block

The `goto` statement present in the main contains a reference to the starting segment from which execution should commence. Execution is triggered off by the manager class, which calls the `executeNode()` method of this Main object.

6.3.2.4 TransitModel.TManager.LanguageBlocks.Execute

The `execute` class, also being in the LanguageBlock namespace, and also inheriting from IBlock, is one of the most crucial classes in the Transit Model Solution. This is due to the fact that it is the class responsible for catering for the execution of activities which have been passed by the end user. Various structures are present in this class, which enable the possibility of suspension

and resumption of Long lived transactions. The structure of the class is displayed in the diagram below. One can identify a series of relevant methods, including the executeNode method, which caters for execution process of an Activity, and four separate execution nodes, handleCompleteExecute, handleCommitExecute, handleRollbackExecute, and handleCompensateExecute, which cater for the different stages of the execution process of an activity.

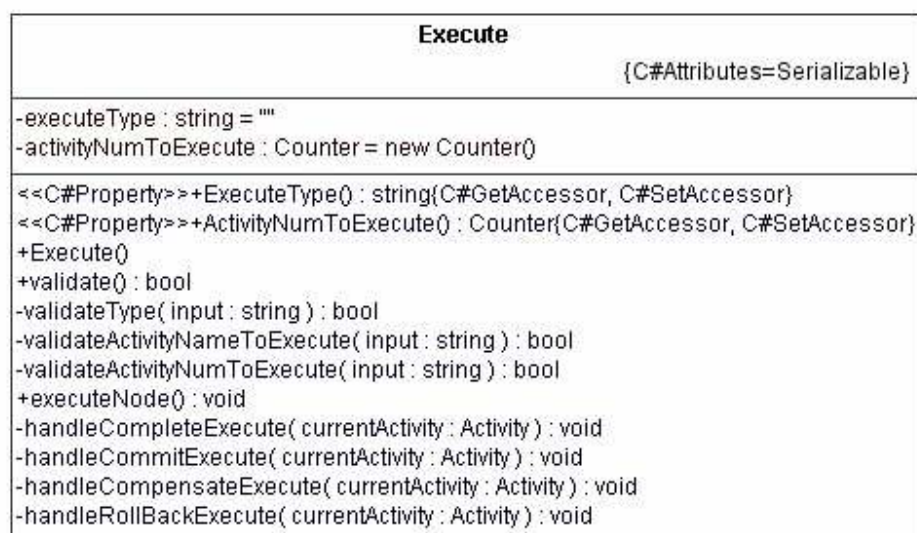


Figure 6.3.2.4.1 Class Diagram for the Execute Language Block

6.3.3 Concepts - Language Parsing & Workflow Generation

The parsing process is handled by the coordinator component, where the xml script resident in a configuration folder named "Model" is loaded, and parsed in the coordinator class using a specific logic, as displayed in the flowchart in section 6.3.3.4.

Initially, the root node is selected, and XPath is used to determine whether each of its children is present. While all children must be present for a successful parse, the most important child tags are the workflow tag, the main tag, and the global declaration tags. For tag details please refer to the previous scripting language chapter. The parsing and workflow generation processes involve two main novel concepts; the idea of having language blocks, and the idea of having flow lists. Another issue which is related to workflow generation is the structure used for parameter passing and handling.

6.3.3.1 Language Blocks

The concept of language blocks involves two main processes:

- The absolute mapping of the Transit Script info Object form.
- The Plugging of Activities from the developer defined LLT into the script's placeholders.

In order to obtain mapping from the XML script into object form, it has been considered ideal to create a class which represents each tag in the script, hence the languageblock namespace in the solution. Thus, an object of type tagname can be created in the coordinator, and any results or child nodes may be stored in it. Child nodes may be other language blocks. Consider a practical example; a workflow language block object may contain three segment language block objects, which in turn may contain fordo, or execute languageblock objects;

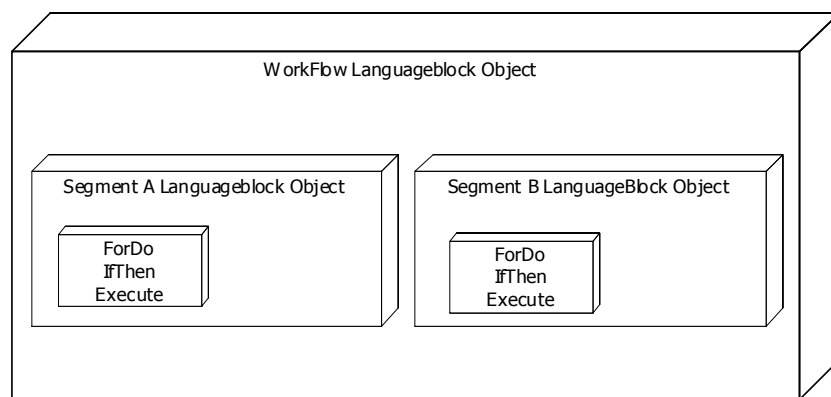


Figure 6.3.3.1.1 Structure of the Workflow Language Block

The main goal is that the parsing process returns an object of type Model, which is actually a hierarchy of objects representing a script instance with the particular activities for this case plugged in. Thus the object of type model would actually represent a transaction context.

For this reason, tags have been categorized into two main forms, those which execute, and those which don't. Executable tags include tags whose sole purpose is not that of storing information, but also of executing a command, such as <execute> or a series of commands, such as <fordo> or <ifthen>, while those which do not execute have purely a structural or storage nature such as <counter>. The following table defines all the tags and their categories:

Tag name	Executes	Has Children
<model>	No	Yes (Name, Decl, Workflow, Main)
<decl>	No	Yes (activityList, counter)

<activityList>	No	No
<counter>	No	No
<workflow>	No	Yes (Array of segments)
<segment>	No	Yes (Begin Node)
<begin>	Yes	Yes (Flow list)
<fordo>	Yes	Yes (Flow list)
<ifthen> <elseif> <else>	Yes	Yes (Flow list)
<execute>	Yes	No
<goto>	Yes	No
<cmd>	Yes	No
<main>	Yes	Yes (Goto)

Figure 6.3.3.1.2 Table for Transit Script Tag Classification

The tags, (or object mappings) which are marked as execute, will extend from the IBlock interface, which defines parameter passing structures (discussed in the following sections) and an execute method stub. Thus, as previously stated, each language block which executes, has an execute method.

A parent language block may execute a child language block by calling its execute method. For example, if a <begin> languageblock contains an <ifthen> languageblock, the execute statement of <begin> must call the execute statement of the <ifthen> which in turn calls the execute statement of any of its child nodes. A workflow is thus constructed, with its starting point being the execute method contained in the <main> languageblock, which the TManager calls in order to initiate transaction execution. In the case of a language block, having multiple children, the concept of flow lists is implemented. The following State transition diagram illustrates this workflow example:

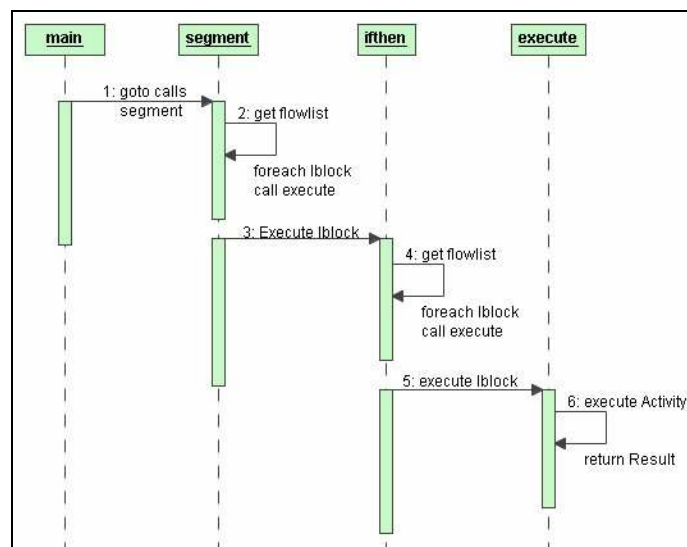


Figure 6.3.3.1.3 Sequence Diagram for Workflow Generation

6.3.3.2 Flow Lists

Each language block object which executes may either execute a single command, as in the case of <goto>, or execute a number of child nodes which it contains, sequentially. These child nodes are typically stored in an Array List in the language block object, and each contain an execute method. A workflow structure is basically this ArrayList of structures which is present in the constructs indicated in the table in the previous section. Constructs which contain a flowlist typically have an execution method which iterates through the flow list, executing each child node. Flow lists are major contributors towards the creation of a workflow.

6.3.3.3 Parameter Passing

While actual parameter evaluation has been designed to occur during the execution phase of the Long Lived Transaction, it is imperative that the structure used for parameter passing is defined in the parsing process, and is directly related to variable declaration. Global variables and local variables are handled in separate manners, as it will be seen in the flowcharts in the following section.

- **Global variables**

During the parsing process, as soon as the <model> tag is parsed, the presence of <decl> tags is checked. If present, a Language Block of type GlobalDecl is created, and this is available to all the child nodes of the <model> tag, since it is the second highest Language Block in the Model Object hierarchy, second only to the Model Tag (see "model object").

- **Local Variables**

On the other hand, local variable declarations have been handled by providing each Language Block element with a table, which contains the local variables assigned to that segment and its children. During the parsing process, as soon as a new segment tag is found, a private global arraylist in the coordinator, named currentLocalDecls (current local declarations) is initialized. Parsing of the segment's children continues, and if a <decl> tag is encountered, the array list is filled with the names, and initial values of the variables. When parsing of the particular segment has completed, the array list, or table, is propagated to all the Language Blocks present in the segment, by assigning each individual Language Block's local variable table declaration, a pointer to the arraylist in memory. Each segment has its own arraylist in memory, to which its children point. This ensures that each child node of a segment has access to the segment's local variables. For details see the following flowcharts.

6.3.3.4 Workflow Generation Logic

The following set of flowcharts defines the complete parsing and workflow generation process which occurs in the coordinator class in an adequate amount of detail:

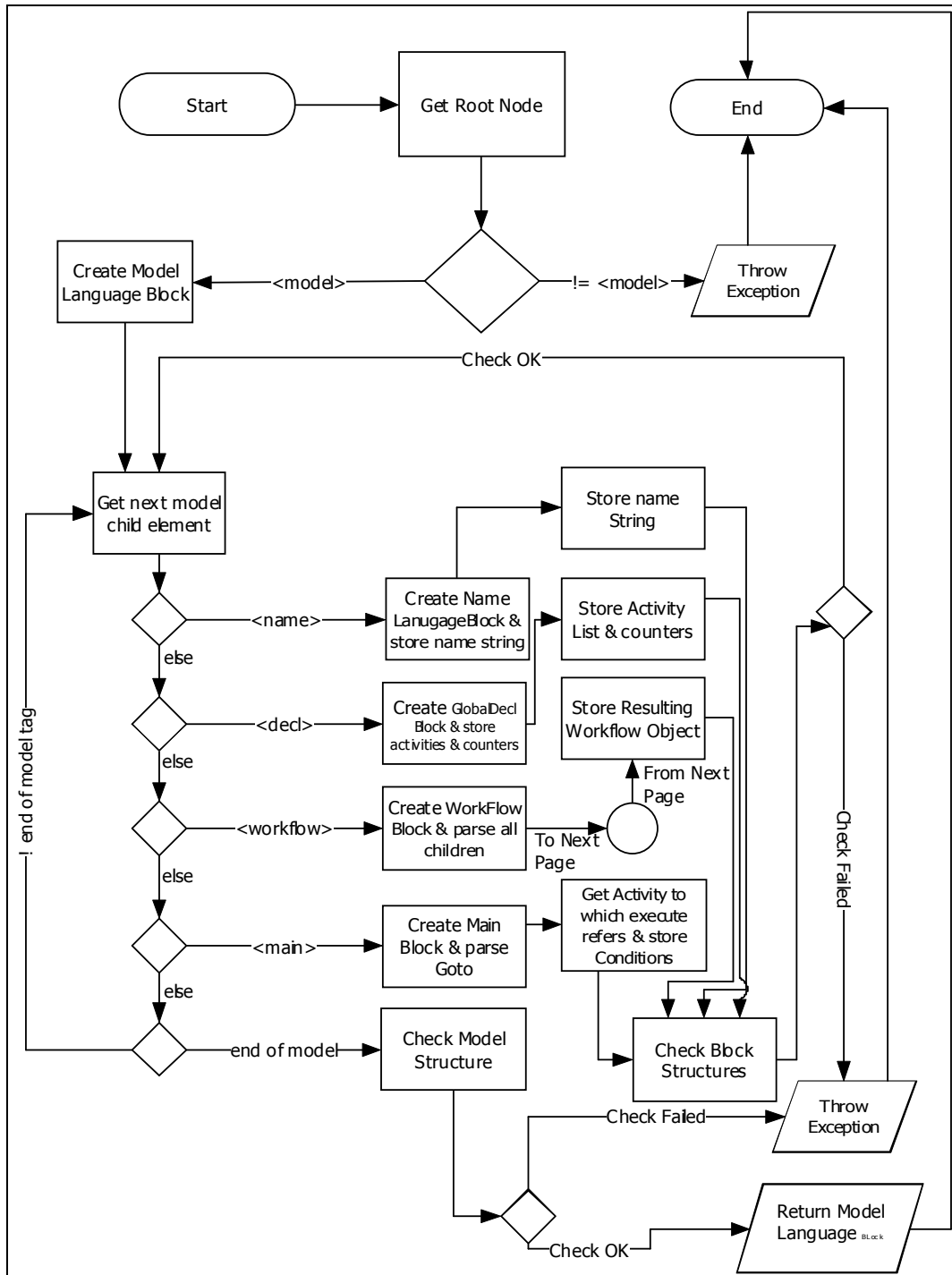


Figure 6.3.3.4.1 FlowChart : Workflow Generation Part 1

which actually contains a mix of classic imperative language constructs, and custom transaction specific constructs:

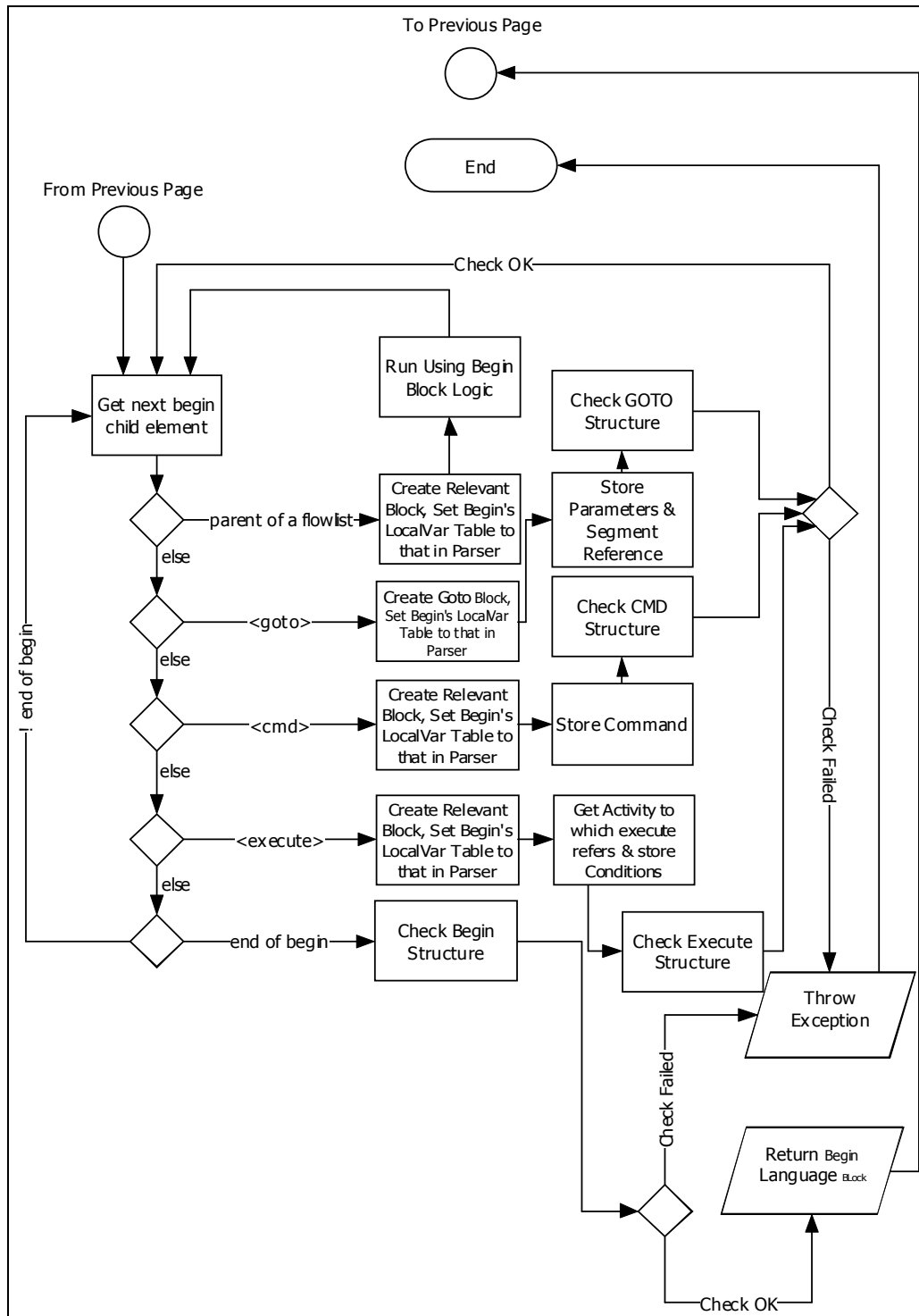


Figure 6.3.3.4.3 FlowChart : Workflow Generation Part 3

Please note that in the flowchart on this, all tags which include a series of children, that is, <fordo>, <ifthen>, <elseif>, and <else> have all been amalgamated into one generic process. While this greatly reduces flowchart complexity, it does not have a negative impact on the representation, since their operational logic is identical to that of the <begin> tag, which has been described in full.

6.3.3.5 The Model Object

The result of this parsing algorithm is an object of type Model, which contains all the structures necessary to carry out an execution procedure. This includes variable declarations and parameter passing constructs, together with an appropriate workflow. Taking the classic Holiday planner example, the structure of a typical model object would in this case look similar to the following:

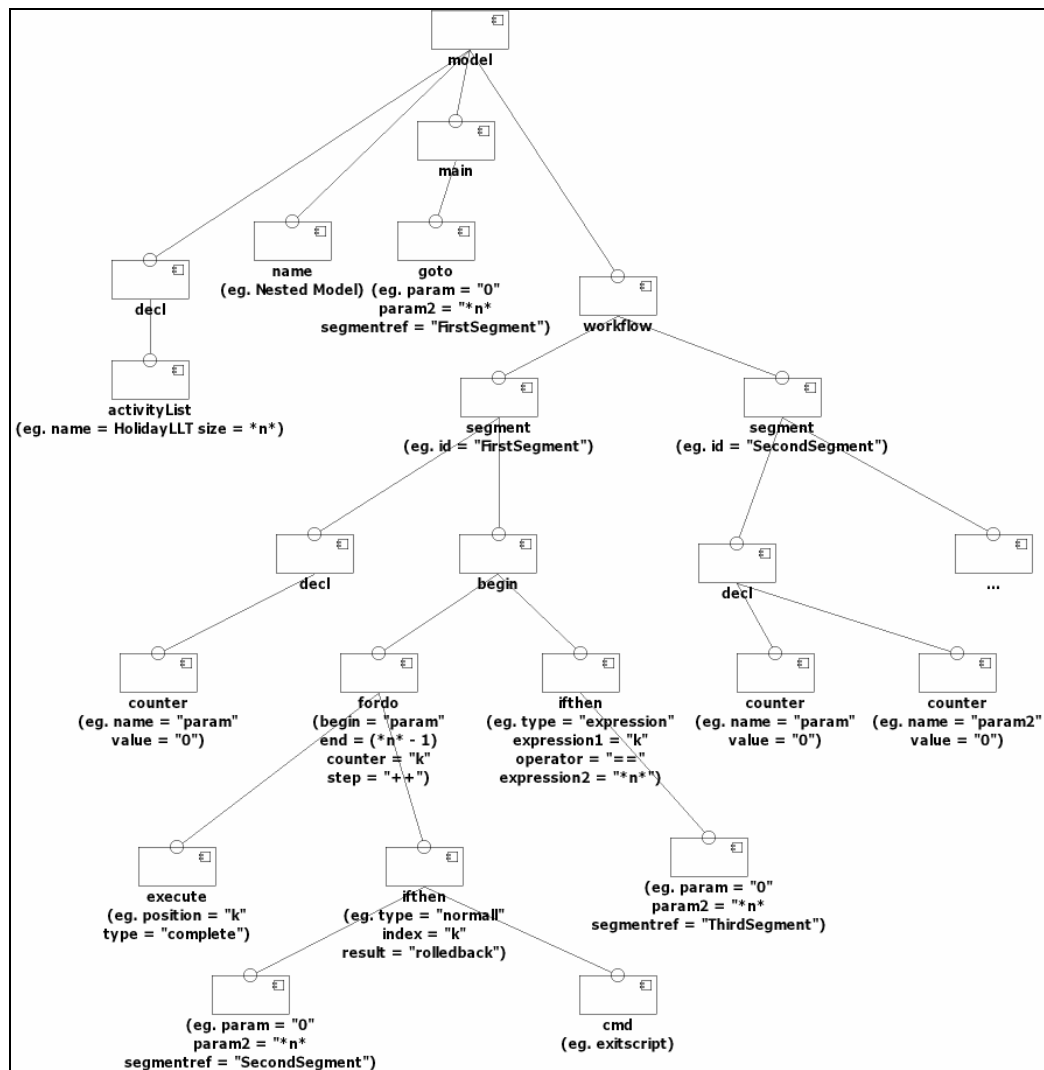


Figure 6.3.3.5.1 Resulting Model Object – Workflow Tree Structure

6.3.4 Concepts - Execution

At this point, execution is a rather a trivial matter. The execute method of the main component is called, in the following manner:

Model.main.executeNode();

Due to the architecture developed, a chain reaction process starts, where the main execute method runs the execute method of the first goto object, which in turn runs the execute method of the begin object inside the referred segment object, and so on. The process either throws an exception, or returns the processed Model Object, which the developer can use to extract results from. The non trivial processes which occur during the execution process are two; parameter passing and variable handling, and state switching of the activities which are executed. Let us now revisit parameter passing, this time from an executive point of view, and then analyze state switching.

6.3.4.1 Parameter Passing Revisited

In the transit script, as explained, the concept of parameter passing involves assigning a value to a local or global variable through a goto statement, by matching the attribute name which represents a parameter to the name of a variable inside a segment. Technically, this occurs by reading the parameter attribute in the goto tag, and searching a match for it in the segment's local declarations arraylist. If a match is found, the value in the arraylist is updated, and thus made available to all the children in the segment.

6.3.4.2 Accessing Variables Revisited

Nodes get or set variable values by two special methods which are present in the IBlock abstract class, named "getvariables()" and "setVariables()". The child node does not need to cater for locating the position of a variable, as in local or global, but simply needs to provide a variable name to one of these two methods, and if the variable exists locally or globally, the value is updated, or returned. These two methods simply initially traverse the local variable table; if unsuccessful, they traverse the global variable table, in order to find the requested variable. If still unsuccessful, an exception is thrown. The following diagrams depict the concept of parameter passing, by updating the segment's arraylist values;

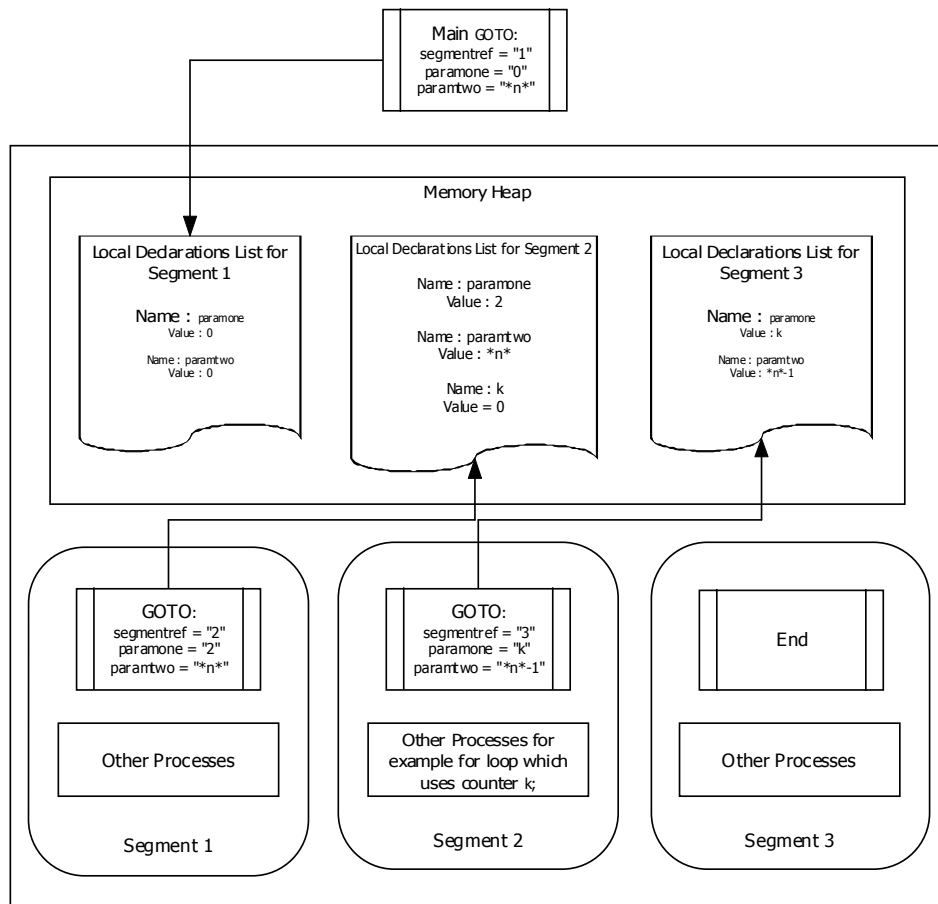


Figure 6.3.4.2.1 Parameter Passing in the Transit Model Solution

Initially, all the values in the tables in the heap are set to 0. However, as execution starts, the first goto assigns parameter values to the first table. Each goto further propagates parameter values, assigning them with the "name matching" technique previously mentioned. Thus, parameter passing is made possible. On the other hand, the following is the pseudocode for the two crucial methods present in the IBlock abstract class which allow the getting and setting of variable names;

```
public string getVariables(string variableName)
{
    try
    {
        foreach (Variable c in this.localVariables)
        {
            if (c.Name == variableName)
            {
                return c.CounterValue;
            }
        }
    }
}
```

```
        }
    }

    foreach(Variable in Model.Decl.GlobalVariables)
    {
        if(c.Name == variableName)
        {
            return c.CounterValue;
        }
    }

    throw new Exception("Variable not found");
}

catch
{
    throw new Exception("Variable not found");
}

}
```

```
public void setVariables(string variableName, string variableValue)
{
    try
    {
        foreach(Variable c in this.localVariables)
        {
            if(c.Name == variableName)
            {
                c.CounterValue = variableValue;
                return;
            }
        }

        foreach(Variable c in Model.Decl.GlobalVariables)
        {
            if(c.Name == variableName)
            {
                c.CounterValue = variableValue;
                return;
            }
        }
    }
}
```

```
        throw new Exception("Variable not found");
    }
    catch
    {
        throw new Exception("Variable not found");
    }
}
```

Figure 6.3.4.2.2 Pseudocode for getVariables() and setVariables() methods

It can be observed that in both cases, the local table is initially traversed to look for the requested variable, if not found, the global table is traversed. If the requested variable is still not found, an exception is thrown. While these methods handle getting and setting of variables, variable values are initially in expression format, which must be evaluated in order to be significant. Variable transformation from expression format to literal format is catered for by special evaluator methods, also present in the IBlock abstract class.

6.3.4.3 Variable Expression Evaluation

In order to be able to handle n based modeling, expression evaluation is a very important factor in this project. While expression syntax has been explained in the Transit Scripting Language chapter, let us now define the detailed process which leads to evaluation of an expression. Initially, *n* values are handled during the parsing process, and are immediately substituted throughout the Model Object with the integer value of the size of the Array List containing the Activities which make up the LLT.

Secondly, two specialized methods, also present in the IBlock abstract class, are used to handle expression building and evaluation; the expressionBuilder() which takes n based expressions such as "*n* + k + 1" and transforms them into literal valued expressions, thus "3 + 2 + 1", and the expressionEvaluator() method, which takes a literal valued expression, such as "3 + 2 + 1" and evaluates it, thus, considering our example, resulting in "6". This method permits n based expressions to be built and evaluated, thus permitting n based modeling, which is one of the requirements defined in the initial chapters.

6.3.4.4 State Switching

While the implementation of state switching is completely handled by the developer, there are certain cases where the change in state directly effects execution in a manner, outside of the normal workflow process. While the

"completed", "committed", "compensated", and "rolledback" states are handled by the user, in the TransitModel.Structure namespace, and conditioned by the script, the "wait" states occur due to erroneous runtime conditions, such as a third party server connection loss during the execution of the transaction. This induces us to add a set of structures to development, which handle the cases in which the execution of an activity returns a wait state.

6.3.4.5 Suspension and Resumption of an LLT

This is a feature present in the Transit Model Solution which caters for the case in which a "wait" state is returned from an execute statement. As stated in the requirements and specifications, in this case, ideally the transaction goes into suspended mode, and may be resumed at a later time, possibly after an application restart.

One rather complex concept must be implemented, in order to make suspension and resumption of a long lived transaction possible; the concept of state tracking. The main idea is that of having an array list structure which keeps a list of all the state changes that each of the activities underwent during execution, so that when an activity is suspended, and then resumed, the part of the workflow which has already executed may be "simulated", without re-executing the actual activities.

In order to cater for suspension/resumption, various minor changes have to be made to the conventional architecture of the system. These include the following:

- The addition of a resume Boolean value to the coordinator class. This indicates whether the loaded model is "brand new", or "resumed", thus possibly having already half executed.
- The conversion of all the language blocks and related models into [Serializable] objects, thus allowing persistence.
- The addition of a specialized structure, which keeps a record of each state change which happens to all the activities in a long lived transaction.
- Changes must be done to the executeNode() method of the <execute> languageblock, since this is the only language block which is able to execute an LLT, thus having direct control for suspension and resumption of an activity.

6.3.4.6 Suspension of an LLT – State Tracking

The main architectural change which must be introduced is the creation of a specialized structure which caters for keeping a record for each state change which occurs to each activity in an LLT, as execution proceeds. This procedure has the sole purpose of determining which activity has already executed, and till which point, in order to allow a stable resumption, in case of suspension. A special class, the stateHolder class, caters for the implementation of this State Tracking structure. The stateHolder class possesses the following structure:

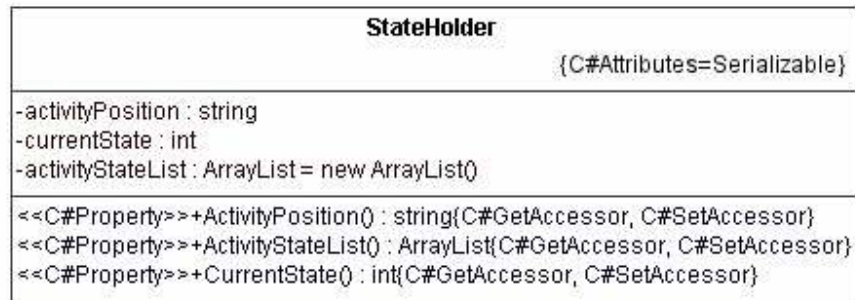


Figure 6.3.4.6.1 Class Diagram for StateHolder Class

An array list of objects of type StateHolder is present in the <workflow> language block. The amount of objects present in the arraylist equals the number of activities present in the LLT. Each object instantiated from this class will keep track of each state change which occurs to a particular activity, every time the <execute> language block's executeNode() method calls one of its methods. Upon each execution, a new entry with the state change is added to the activityStateList array list, and the current state is updated with the current size of the activityStateList.

Upon the switching of the state of an activity to one of the waiting states, the following procedure takes place:

- The Resume Boolean variable in the <workflow> object is set to true.
- The Waiting state is added to the activityStateList of the particular activity which failed. This activityStateList is contained in the array list of state holders in the <workflow> object. The waiting state may be any one of the four wait states discussed in the previous TransitModel.Structure section.
- Since all the classes directly related to the Model object have been set as serializable, the object is simply serialized to a binary encrypted file on disk.

In order to make this process possible, the <execute> language block's executeNode() method's logic has been altered, to conform to the one in the following flowchart:

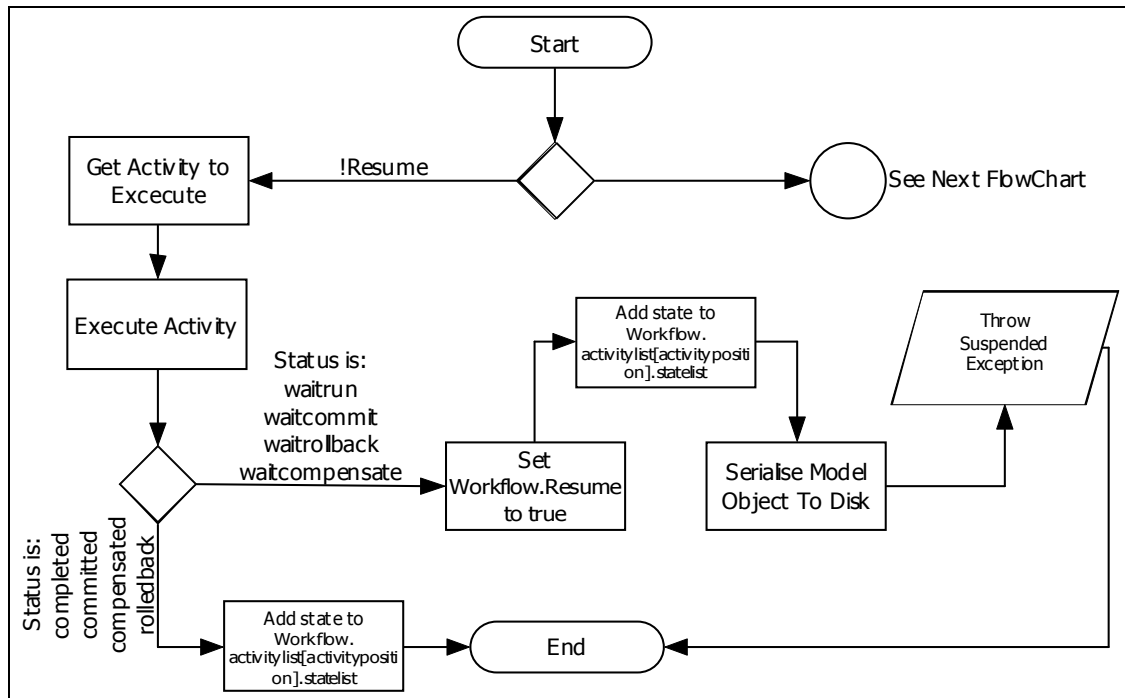


Figure 6.3.4.6.2 FlowChart : Suspension of an LLT

The suspension process has thus been tackled in a rather simple way, however, let us now consider the issues of resuming a half executed, suspended LLT.

6.3.4.7 Resumption of an LLT – Activity Execution Simulation

The resumption process, requires a further architectural addition to the <execute> language block's executeNode() method, which allows the "simulation" of activity executions which have already occurred, without actually re-executing the activity. Thus, when resuming an activity, the script will execute from the start. However, when arriving at an execute statement, if the resume mode in the workflow is set to "true", the next value on the activitystateList of the state tracking structure is retrieved, thus simulating a state change without executing the activity. The following flowchart represents the addition to the method's logic which was made:

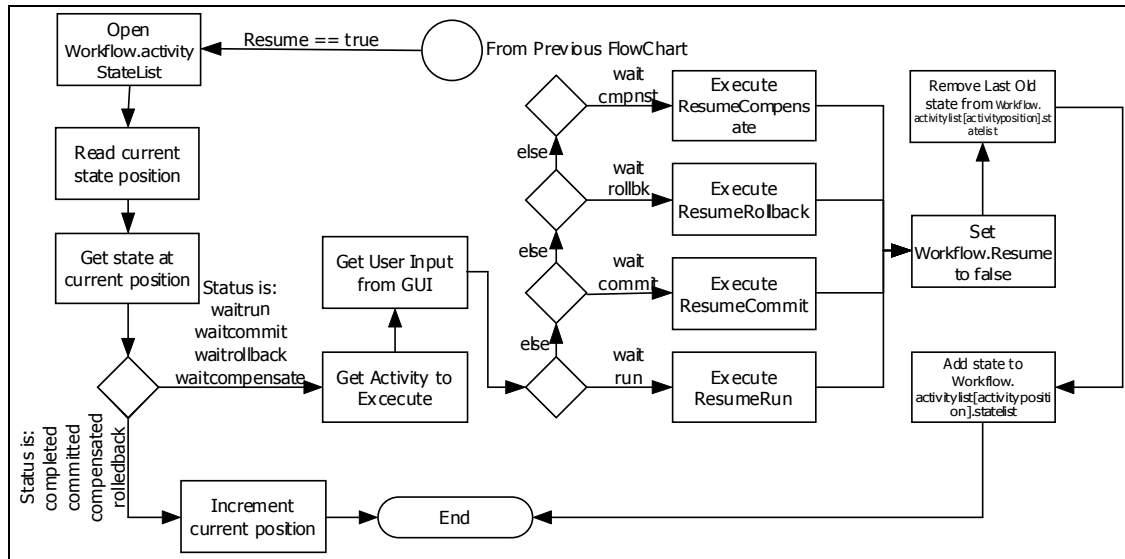


Figure 6.3.4.7.1 FlowChart : Resumption of an LLT

While simulated execution proceeds, a point arrives when a switch between simulated execution and actual execution must take place. This point is indicated when the next state in the activityStateList for a current Activity equals a wait state. This signifies that the point where the activity had been suspended has been reached.

At this point, one of four resume methods is called, according to the particular stage in which the activity had suspended. If the activity had returned a "wait" status as a result of an attempted execution of its "activityRun()" method, the resumeRun() method is now called. The same logic applies to the other three methods, resumeCommit(), resumeRollback(), and resumeCompensate(). The point at which the activity had halted is determined by the type of wait message, "waitrun" indicates a suspension in the run method, "waitcommit" indicates a failure in the commit method, and so on. The resume methods necessitate the instantiation of a specialized GUI, which enable the user of the application to input the necessary information needed by the resume method, in order to continue execution. As explained, had no suspension occurred, this information would have been automatically received by the Transit API, without the need of user intervention.

The GUI consists of a simple input interface, into which the user inputs raw bytes of information, which are stored in a generic System.Object. The main reason for storing the information in a generic object, is that since different activities may need different input object types, the Transit Model Solution must cater for allowing manual input of multiple object types, according to the currently suspended activity. GUI input into a System.Object is ideal to cater for this

solution, since any input can be handled, and then type cast by the activity's resume method, into the desired object type;

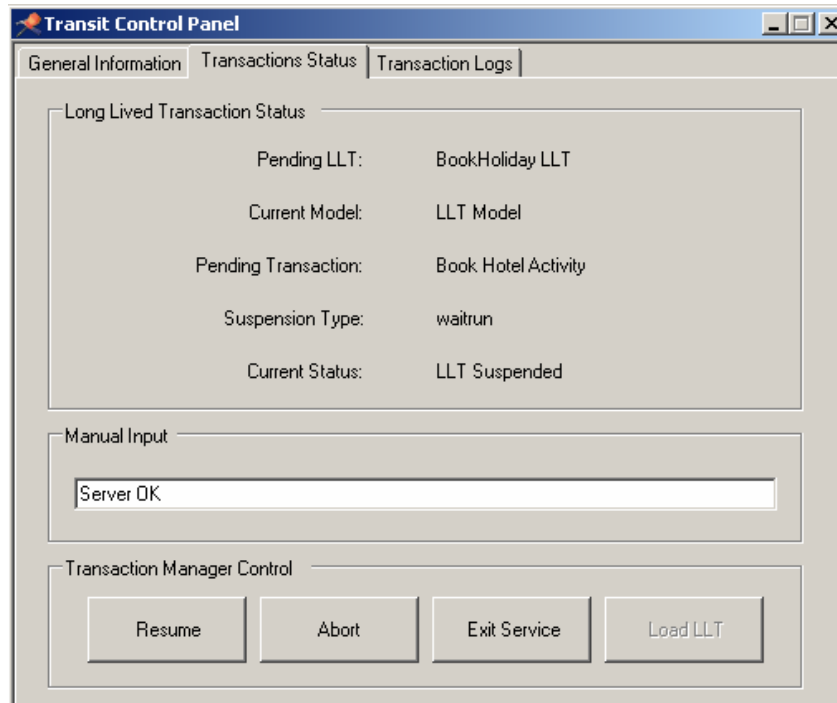


Figure 6.3.4.7.2 Screen Shot for the Transit Control Panel Resume GUI

In this example, the System.Object will contain a manually input "Server OK" message, which the method may cast into a string, or an array of characters, as needed. While this implementation serves the purpose of proving a concept, in a full scale application, advanced object input support may be added to a commercial system.

At this point, normal execution must continue. This is indicated by setting the resume mode Boolean value to false, thus, since the <execute>'s executeNode() method checks the resume mode variable's value each time that it is called, upon the next execute language block call, a normal execution will occur. Finally, the Long lived transaction may either complete, abort, or get suspended for a second time.

Chapter 7: Conclusion

7.1 Producing an Integrated Solution

The result obtained when amalgamating the Transit Scripting Language, the Structure namespace, and the TManager namespace is one compact visual c#.NET 1.1 based Dynamic Link Library, which may be integrated into any .NET based application requiring support for long lived transaction processing. In this particular implementation, three model template scripts have been provided, the classical Nested Model, a custom Saga based model, and an LLT model conceptualized in the JSR 95 specification. However any model may be implemented using the transit scripting language constructs. The TransitModel API is extremely simple to use, since the amount of function calls needed to construct a long lived transaction, and process it are relatively small. In the following section, a practical example of the integration, and use, of the transit scripting model, in the typical Holiday Booking example is provided.

7.2 LLT Enabling a Typical Application using Transit

In this example it has been assumed that the developer is creating a simple Travel Agent's Booking system, which requires long lived transaction support. Let us assume that the developer is using .NET version 1.1, Visual C#, and Visual Studio 2003 Enterprise Architect. The developer must follow the next steps:

- Add a reference of the TransitModel Library to the current Project.
- Create a set of Activity Classes, which inherit from TransitModel.Structure.Activity, and implement the necessary methods as shown in the previous chapters. In the holiday booking example's case, three classes should be implemented, BookPlane, BookTrain, and BookHotel, each with their own implementations and connections to remote servers.
- Create an arrayList object, and add these classes to it, in the order the developer wishes them to be processed by the particular model chosen.

- Create an instance of `TransitModel.Structure.LLT`, and copy the `arrayList` object of activities to it.
- Call the `TransitModel.TManager.Logic.Mgr.GetInstance()` method.
- Call the `Mgr's brandNewLLT()` method, passing the freshly constructed `LLT` object as parameter.
- Call the `runLLT()` method.
- Process the results and display them on screen.

In the top level application, the developer must also handle for calling an instance of the `TransitModel.TManager.Logic.TransitControlPanel` GUI, which may be invoked by the developer in order to handle resumption of suspended `LLT's`. Since the GUI is also based on the singleton model, its `getInstance()` method must be called. For a complete coverage of an example application, please view the appendices section.

7.3 Transit Model Solution as an Open Source Project

One of the initial goals of this project has been that of posting the resulting research and code to the open source community, with the intention of creating a stream of feedback from experts competent in this area from this community. This feedback would allow the project to be improved from a series of aspects, including inclusion of previously unthought-of features and the discovery and fixing of any bugs gone undetected amongst others. Open source projects must also conform to a set of features such as formal versioning of file releases, patches and bug fixes amongst others, which could be of benefit both to the project and to the developers who download and use it.

It has been felt that the best way to transform the Transit Model Solution into an open source conformant project is by posting it to the <http://www.sourceforge.net> open source community, through a project submission application for provision of space on their servers. This also served as an exercise to gauge the quality of the project, since project submission to source forge are reviewed by a series of technical staff, before being approved. The submission to source forge brought about the following changes/feature additions to the Transit Model Solution:

- **Web Site:** Open source practice (as stated by sourceforge) includes the creation of a web site which presents the project to the open source community. This should include a brief project description, together with links to the appropriate documentation, source code, and binary files.

Links to the various open source communication tools, in this case offered by sourceforge should also be provided. The site for the Transit Model Project has been implemented and uploaded to the space provided on: <http://transitmodel.sourceforge.net>. For a screenshot of the site, see the appendices section, appendix H.

- **Source & Binaries:** For a project to be classified as open source, both its source code and its binary files must be posted to the open source community. In this case, source forge provides project subscribers with a standard file release system to which project administrators can post both source and release files. Source and release files for the Transit Model Solution have been made available on : <http://sourceforge.net/projects/transitmodel/>
- **File Versioning:** When submitting source, binary files, or bug fixes, they must be appropriately versioned, typically using an incremental numbering system. Source forge provides a standard versioning structure through its specially implemented file release system. There is currently only one version release for the Transit Model Solution, that is, version 1.0, available publicly in the downloads section of the source forge site.
- **Bug Reporting & Patch Manager:** This is the first of a series of communication tools which enables members in the open source community to report bugs to the project administrator. In our case, it is available through the source forge site for the Transit Project, together with a patch manager, which hosts similar properties to the file release system, however catering solely for project patches.
- **Feature Requests:** This utility allows community members to post suggestions to the project administrator, specifically, desirable features which the project does not possess, and which would significantly improve the project. Feature requests are also offered through the Transit Project's source forge site.
- **Screenshot Manager:** The screenshot manager allows the postage of project screenshots in a standardized image with preset dimensions and file format, ensuring view ability by all the community's members. A series of screenshots of the Transit Suspend/Resume GUI together with Sample Application Screenshots is present on the Transit Sourceforge site.
- **Forums, Mailing Lists & News:** These tools further enhance communication between the community and the project administrator, thus allowing the overall improvement and expansion of the project.

7.4 Transit Model Solution Assumptions and Limitations

The following list contains all the assumptions which have been made throughout the development of this project, together with project limitations which are present in the current implementation:

- An important goal of this project was that of proving an academic concept, rather than developing an industrial solution.
- While development has been made in .NET technologies, there are no low level system calls which tie down the logic of the Transit Model Solution down to .NET. Therefore re-coding in any desired language is possible. It is considered an advantage if any alternate language in which the project may be re-coded is based on OOP concepts.
- While this project proves that the application of a workflow control language in order to handle Long Lived Transactions is simple and practical, the implementation presented has certain limitations, such as restricted expression handling (expression handling is restricted only to + and – operators), or the inability for a segment to operate in a recursive manner. While all the basic functionality is operational, the scripting language may be extended to infinity, since there is no limit to the amount of functionality that may be added.
- The main limitation in this solution is that transaction contexts were not completely handled. In fact, a lower level accompanying layer such as Microsoft's System.Transactions Library, or JTA in case of a Java engine would be needed for complete transaction context handling and propagation. In this solution, a transaction context has been assumed to be the set of states of the transactions composing an LLT. While in this case, the context is shared, a specific value can't be propagated.
- It has been assumed that the reader possesses development knowledge, and that anyone who applies this system in a project is familiar to XML Scripting techniques.
- The main goal of this project was to prove the scripting language and workflows concepts' applicability to LLT processing, thus the main effort was concentrated on developing a centralized robust language and processor without catering for a distributed environment.

7.5 Evaluation

If one takes into account the Transit Model Solution as it has been implemented in this thesis, the following main features may be outlined:

- It offers a powerful and simple scripting language which allows the definition of multiple transaction models.
- The main power in the language lies in its highly standardized XML conformant syntax, and in its simplicity. When the developer is faced with the choice of a model as discussed in the previous chapters, there is no need of having expert knowledge in the transaction management field, as the script consists of a rather simple workflow description which may be understood by anyone with basic imperative language programming knowledge.
- Since the scripting language is standardized to conform to W3C XML standards, if developers adopt the language to define existing and custom models as a default choice, the language itself may be a good candidate for becoming a common well known standard, possibly used in all transaction management systems to define the models.
- Being open source, as developers create more and more models using the scripting language, there will be a pool of readily available transaction models, which may be simply downloaded and plugged into the Transit Model Solution, or custom solutions which use the same Transit Script Standards.
- Should the case arise where a developer must code a custom transaction model, there is still no need of expert transactional knowledge, but just the understanding of simple workflow concepts. The language in itself has been specially designed using conventional language constructs to make it learnable literally within minutes.
- The solution thus successfully abstracts the developer from excess transactional details which may consume time and resources.
- The accompanying transaction model API also offers various advanced transaction handling techniques such as suspension or resumption of a long lived transaction, while persisting it to disk. Thus while this implementation simply serves to prove the meta-model concept, it also explores advanced transaction handling features.

It can be concluded that these features are more than enough to resolve the initially discussed problems present in current solutions, since they fulfill all the desirable features which a typical transaction model and transaction management system should contain. In essence, all problems, current system drawbacks, and desirable features discussed in the first three chapters of this thesis have been fulfilled by the Transit Model Solution.

7.6 Future Work

While all the core logic for a stable solution has been included in this dissertation, and the accompanying implementation, there are various desirable features which may be considered for inclusion in future versions of the implementation. These include the following:

- Distribution of business logic into a fully fledged middleware Transaction Processing system. This would include the addition of the TransitModel API to a middleware server, rather than a top level application. Typically, objects would be passed over XML or soap to the server, processed, and returned to the requesting application.
- Full expression evaluation features, including all operators, not just + and - operators.
- Re-coding into a multi platform programming Language.
- Extension of the programming language to include better constructs, possibly not by extending the coordinator and language blocks engine, but by creating new constructs based on XML itself. An initial attempt has been made, with the implementation of try ... catch, commitAll, and compensateAll constructs, which has been successful. Further extension of the language is desirable, as long as functionality is increased.
- Analyse feedback from the open source community, and improve the system according to that feedback. As previously stated, the project has successfully been posted to www.sourceforge.net, thus giving it a major exposure to the open source community.

7.7 Final Remarks

From the testing carried out (see appendices), it can be concluded that the Transit Model API provides a stable and easily integratable solution for handling various types of Long Running Transactions, through the introduction of a series

of both novel concepts and concepts which have been based on the positive features of current transaction model theory, and solutions. The main sources of inspiration for this thesis included ConTract Models, which introduce the notion of scripts for transaction processing, Arjuna's WS-CAF project, which has workflow based architecture, and Bell Lab's Cova TM, which is also a script based transaction management system. The main inspiration for theoretical research has been Marek Prochazka's PHD thesis on Long lived transactions, which contained invaluable information regarding currently available transaction model specifications.

The original scope of this thesis was that of the creation of a Meta Model, allowing developers with scarce transactional knowledge to easily express their own transaction models, or use ready made model templates. This scope has been reached with the development of the Transit scripting language, and the accompanying API, which proves the practical nature of the Language. At present, the project has been reviewed by source forge technical staff, and successfully approved for registration as a www.sourceforge.net open source project under an academic free licence. This effort has been undertaken in order to expose the project to the open source community, and get relevant feedback regarding ways in which the solution can be improved. The source forge application and approval forms may be viewed in the appendices section. The project material is available for viewing on the source forge site, at the URL's:

<http://www.sourceforge.net/projects/transitmodel>

or

<http://transitmodel.sourceforge.net/>.

While the features mentioned in the future work may be desirable, they are considered as extras, which only enhance the functionality of the core logic which has already been defined in this thesis. Various references have been used, since the nature of this project had a strict emphasis on heavy research, and logical design, rather than development capabilities. A list of these may also be found in the appendices section.

The appendices section also contains information about correspondence carried out throughout the project, practical examples, and a full listing of class diagrams which represent the classes contained in the TransitModel namespace.

Appendix A: Glossary

When one comes to contact with the world of transactions, a set of dedicated technical terms is noticed, some of which are fundamental for the comprehension of the concepts of transactions itself. This syntax is particularly used in the definition of software solutions for transactions, and thus it is mandatory that the reader has good knowledge of it, in order to fully understand this project. In aid of this cause, the most commonly used technical terms which are needed to grasp transaction concepts are listed and explained below:

[A] ACID Properties

When a transaction manifests ACID properties, it means that it is atomic in nature where ACID stands for Atomicity, Consistency, Isolation and Durability.

[B] Activities

An activity may be considered as building block, at the lowest level of granularity in the Transit Model Solution, a series of which makes up a Long Lived Transaction. From a theoretical point of view, activities may range from a strict ACID transaction, to a more complex transaction, based on the developer's choice.

[C] Atomic & Compound (aka Long Lived, Long Running) Transactions

This term is explained in detail above in the previous pages, however in essence, atomic transactions are those which have a compact, "yes or no" nature, while Compound transactions include complexities such as compensation and rollback over extended time. Typical a long lived transaction is made up of multiple ACID transactions.

[D] B2B & B2C Transactions

B2B refers to any transactions occurring between businesses, while B2C refer to transactions which typically occur between a business and a Consumer. The latter is typically of an atomic nature.

[E] Commit & Abort Concepts

The commit and abort concepts in a transaction are simply the occurrences of a successful transaction in the case of commit, which means the transaction would have been confirmed, and the occurrence of a transaction failure in case of an abort.

[F] Long Lived Transactions (LLT's)

Long lived transactions, as previously stated in the generic description, are compound transactions made up from smaller building blocks, also known as activities. This is the context applied in the solution of this thesis. A Long Lived transaction may commit, abort, or get suspended and resumed.

[G] Rollback & Compensation/Recovery Concepts

Rollback and compensation are two concepts which are present only in long running transactions, where rollback refers to the point in time where one Unit of Work of the transaction fails, and thus the system has to reverse any actions which had been taken. Taking a bank transfer as an example, money would have to be sent back to the original account. Compensation on the other hand refers to the activity of finding an alternative solution to the failed Unit of Work. Referring again to the bank transfer situation, compensation in this case may be finding an alternative account of the same destination customer, and transferring the cash to it.

[H] Suspension/Resumption

Suspension and Resumption are to concepts present in the Transit Model Solution which allow the halting of the processing of a transaction mid-way, switching off of the application, re-switching on at a later time, and resumption of the halted transaction process.

[I] Transaction Model

A transaction model is a template to which the behavior of a particular transaction can be compared, thus determining whether the transaction in question falls under the model's category. There exists no standard set of models, especially in the case of long lived transactions, as one model is typically good for just one or a small range of similar transactional applications, thus anyone may define a new transaction model at any time. This may have negative effects by creating confusion and an environment of non standardization in the transaction processing realm, where nobody knows which model is best for what particular practical application.

[J] Transaction Processing System/Framework/Service

These terms simply refer to the existent software applications which in one way or another handle transactions of any type and process them. In many cases, a specification of a service/framework/system is found to be available, without the actual system having been yet implemented. These systems are usually based on one or more transaction model.

[K] Transaction States

Transaction states are a series of conditions to which an activity in a long lived transaction conforms, after a particular execution procedure. For example, if an activity, executes, it will typically switch from an "idle" state, into a "committed" state.

[L] Transit Model API

The Transit model API is the part of the Transit Model Solution which has been developed using Microsoft's Visual C#, and .NET 1.1 Technology, with the sole purpose of providing a parser, interpreter and execution framework for Long Lived Transactions

[M] Transit Model Solution

The term "Transit Model Solution" refers to nothing less than the software solution proposed in this dissertation. This includes an XML based scripting language with specially developed syntax, an API which parses

and processes workflows containing long lived transactions, and allows for the persistence of these workflows to disk.

[N] Transit Scripting Language

This term refers to the XML based custom scripting language which has been conceptualized and specially developed in order to handle Long Lived Transactions. The language is considered as one of the core parts of this thesis, and has the main scope of providing a meta-model with which transaction models may be defined in an easy and concise manner. The script has been based on workflows constructed using classic imperative programming language constructs.

[O] Unit of Work

A Unit of Work represents a single business process, which may be both transactional or non transactional, which typically makes part of a long lived transaction. If a transactional Unit of Work fails, a rollback operation takes place. This may be followed by a compensation operation, depending on the transaction model which is being used.

Appendix B: Class Diagram Listing

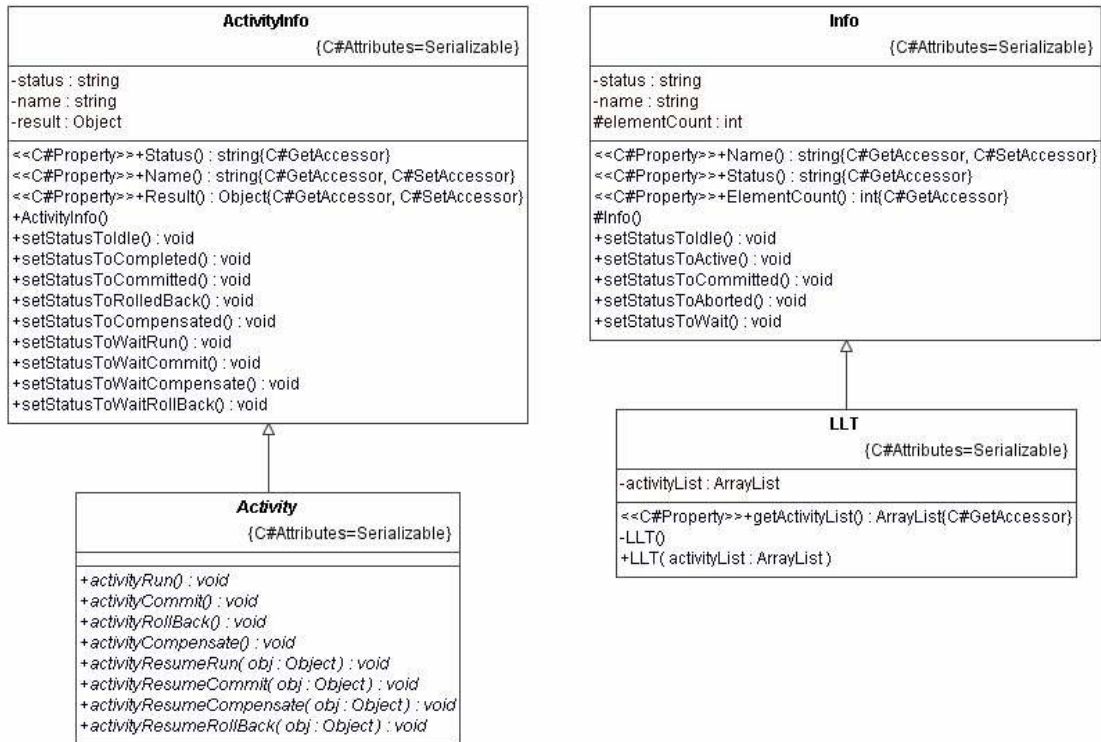
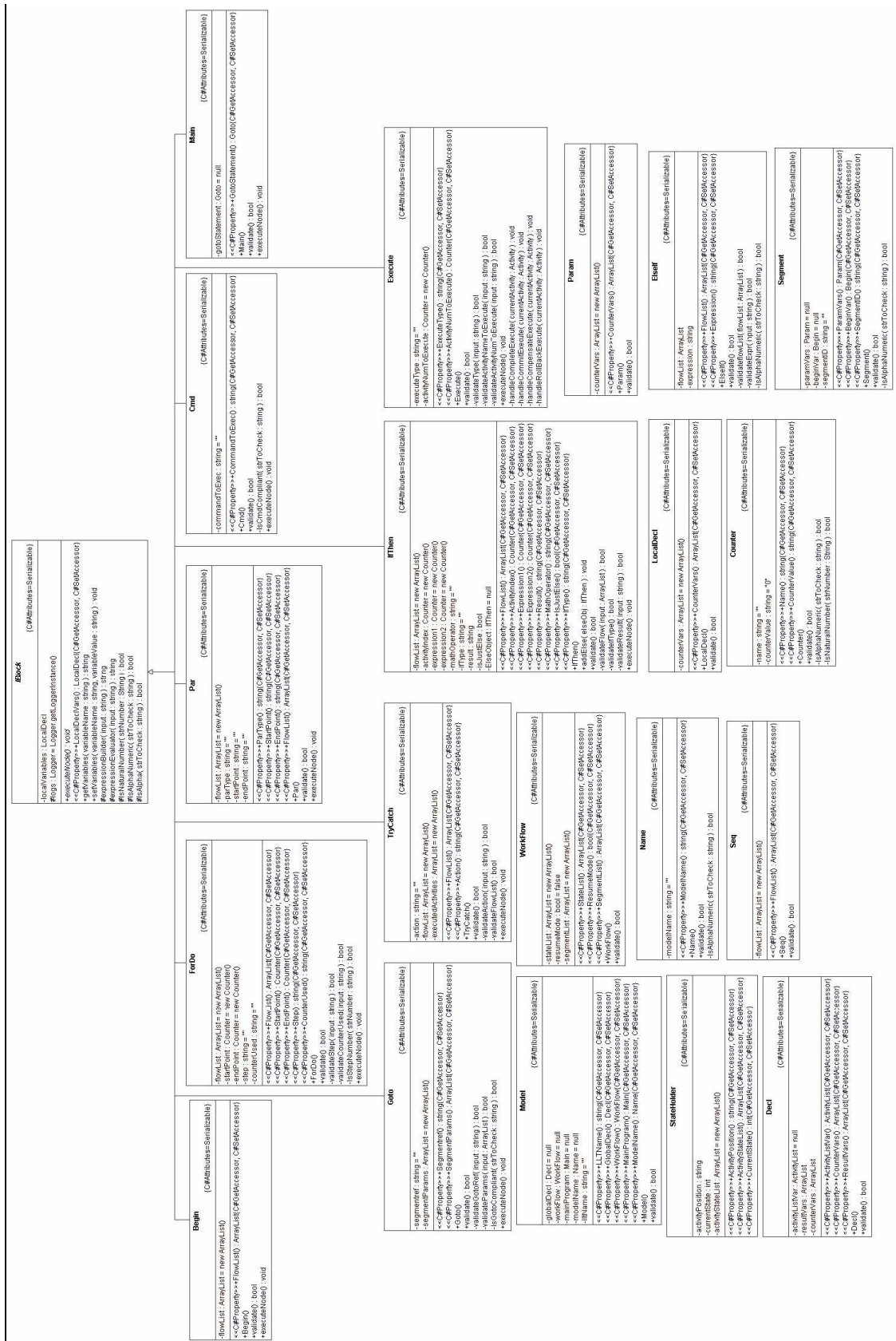
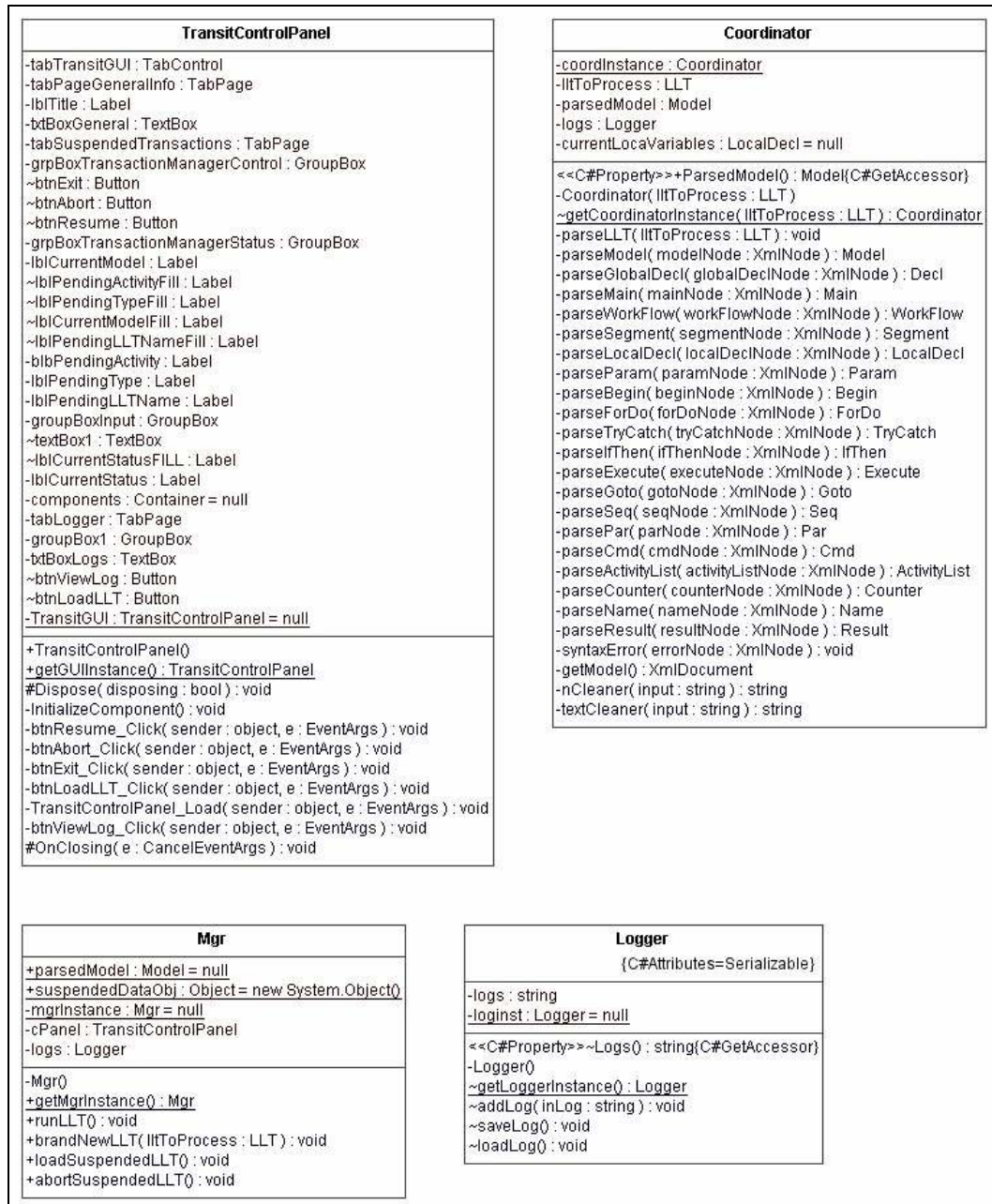


Figure B1 TransitModel.Structure Class Diagram

The above diagrams represent the TransitModel.Structure namespace, which provides the activity class, from which a developer can extend, and the LLT class, which the developer can instantiate, in order to create an LLT object, composed from Activity objects, which is then processed by the TManager component. For more information read dissertation's chapter 6.

On the other hand, the diagram on the next page represents the architecture mentioned in the dissertation, which enables the parsing and execution of the Long lived transaction passed by an end user, according to a loaded Transit Script. One can see the language blocks for each statement, some of which extend from the IBlock interface, which provides execution method definitions and parameter passing and variable declaration utilities.





Figures B2 & B3 TransitModel.TManager Class Diagrams

Finally, this third batch of diagrams represents the TManager namespace, which contains the Coordinator class, the Manager class, the Transit Resume GUI, and a simple logger, which is used by the GUI. This set of classes amalgamates the whole solution, since the developer may call only methods from the MGR class, and may only get an instance reference of the GUI, for resumption purposes. All classes are singleton, since only one instance of each class is needed per application.

Appendix C: Transit Script Examples

The following xml based scripts represent two of the three transaction models present in the implementation on the accompanying disk. The Nested model has been already included in this dissertation in chapter five. While section A contains the Transit version of the JSR 95 LLT model as implemented by Ixaris, a custom SAGA based model is presented in section B. This makes use of custom try catch constructs defined in XML syntax, together with extensive use of N based expressions. These scripts were build using a typical XML editor application, such as Altova's XML Spy, which allows on the fly XML validation.

A.) JSR 95 LLT Transaction Model (Ixaris Implementation)

```
<?xml version="1.0" encoding="utf-8" ?>
<model>
  <name>LLT Model</name>

  <decl>
    <activityList size = "*n*">
      arrayOfActivities
    </activityList>
  </decl>

  <workflow>
    <segment id = "Start">
      <decl>
        <counter value = "0">k</counter>
      </decl>
      <begin>
        <fordo begin = "paramone"
          end = "paramtwo"
          counter = "k"
          step = "++">

          <execute position = "k" type = "commit">
            arrayOfActivities
          </execute>

          <ifthen index = "k" result = "rolledback" type = "normal">
```



```

                                <goto paramone = "k-1" paramtwo = "0">
                                    CompensateAll
                                </goto>
                                <cmd>exitscript</cmd>
                            </ifthen>
                        </for>
                    </begin>
                </segment>

                <segment id = "CompensateAll">
                    <decl>
                        <counter value = "0">k</counter>
                    </decl>
                    <begin>
                        <for> begin = "paramone"
                            end = "paramtwo"
                            counter = "k"
                            step = "--">

                                <execute position = "k" type = "compensate">
                                    arrayOfActivities
                                </execute>
                            </for>
                        </begin>
                    </segment>
                </workflow>

                <main>
                    <goto paramone = "0" paramtwo = "*n">Start</goto>
                </main>
            </model>

```

Figure CA1 JSR 95 LLT Transaction Model

B.) A Custom SAGA Model

```

<?xml version="1.0" encoding="utf-8" ?>
<model>
    <name>TryCatch Saga</name>

    <decl>
        <activityList size = "*n">
            arrayOfActivities
        </activityList>
    </decl>

```



```
<workflow>
  <segment id = "Try">
    <decl>
      <counter value = "0">k</counter>
    </decl>
    <begin>
      <for do begin = "*n*-*n*"
        end = "*n*-(n*-1)"
        counter = "k"
        step = "++">

        <execute position = "k" type = "complete">
          arrayOfActivities
        </execute>

        <if then index = "k" result = "rolledback" type = "normal">
          <goto param1 = "k-1" param2 = "*n*-*n*">
            Catch
          </goto>
          <cmd>exitscript</cmd>
        </if then>
      </for do>
      <if then type = "expression"
        expression1 = "k"
        operator = "=="
        expression2 = "*n*-(n*-1)">

        <for do begin = "*n*-*n*"
          end = "*n*"
          counter = "k"
          step = "++">

          <execute position = "k" type = "commit">
            arrayOfActivities
          </execute>
          <if then index = "k"
            result = "rolledback"
            type = "normal">

            <goto param1 = "k-1"
              param2 = "*n*-*n*">

            Finally
          </goto>

          <cmd>exitscript</cmd>
        </if then>
      </for do>
    </if then>
  </begin>
</segment>

<segment id = "Catch">
```

```
<decl>
    <counter value = "0">k</counter>
</decl>
<begin>
    <for do begin = "param1"
        end = "param2"
        counter = "k"
        step = "--">

        <execute position = "k" type = "rollback">
            arrayOfActivities
        </execute>
    </for do>
</begin>
</segment>

<segment id = "Finally">
    <decl>
        <counter value = "0">k</counter>
    </decl>
    <begin>
        <for do begin = "param1"
            end = "param2"
            counter = "k"
            step = "--">

            <execute position = "k" type = "compensate">
                arrayOfActivities
            </execute>
        </for do>
    </begin>
</segment>
</workflow>

<main>
    <goto>Try</goto>
</main>
</model>
```

Figure CB1 Custom Try Catch Saga

Appendix D: Transit API Usage Instructions

This appendix provides a complete list of utilities externally available to the developer present in the Transit Model API:

- **Classes:**

TransitModel.Structure.Activity – Class which provides standard structuring for an activity. Developers must extend from it to create activities for their applications.

TransitModel.Structure.LLT – Class which provides structuring for a Long Running Transaction. Developers must instantiate it and add an array list of activities to it, thus having created an LLT Object.

- **Utilities:**

TransitModel.TManager.Logic.Mgr.getMgrInstance() – Method which returns a pointer to a singleton instance of the transaction manager.

TransitModel.TManager.Logic.Mgr.brandNewLLT(LLT object) – This method may be used to “post” the LLT object created in the application to the transaction manager.

TransitModel.TManager.Logic.Mgr.runLLT() – Method which executes the transaction which has been currently loaded, either brand new, or suspended.

TransitModel.TManager.Logic.Mgr.loadSuspendedLLT() – Method which loads an LLT Object from disk into the Transaction Manager. While this method is public, explicit use of it should be avoided as much as possible. LLT Resumption should be handled through the Transit Model Solution Resume GUI.

TransitModel.TManager.Logic.TransitControlPanel.getGUIInstance() – Method which returns a pointer to a singleton instance of the Transit Model Solution’s Resume GUI. This method call includes automated loading of suspended transactions from disk, without the developer’s intervention.

Appendix E: Example - Transit Enabled version of Skype

A.) Introduction

The following example assumes a particular use case of the popular Voice over IP software program, Skype. The following implementation extends the Skype application's top up process, making it long running. A long running transaction thus has to be designed, implemented, and passed onto the Transit Model API, which executes the transaction, returning a positive or negative result to the Skype Application. While the actual skype implementation is out of the scope of this thesis, this Appendix covers implementation details of how the developer should handle integration of the Transit Model API into his application, creation of the Long Running Transaction, execution, and interpretation of the results. Please note that only the parts relevant to this thesis have been implemented, and thus the actual implementation of skype and third party server communication have been simulated.

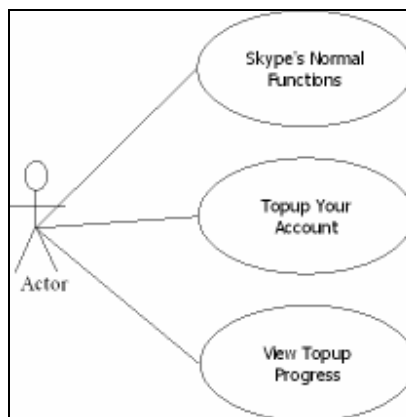


Figure EA1 Use Case: Transit Enabled Skype E - Top Up

B.) Application Design & Implementation

When one considers the Skype's utilities, other than the top up process, it is developed in the typical way any application is developed, consisting of a series of classes containing data structures and methods. There is no need of radical structural changes to the design process in order to transit enable an application.

Conforming to the explanation provided in this thesis, the general architecture of the Transit Enabled version of Skype consists of the following:

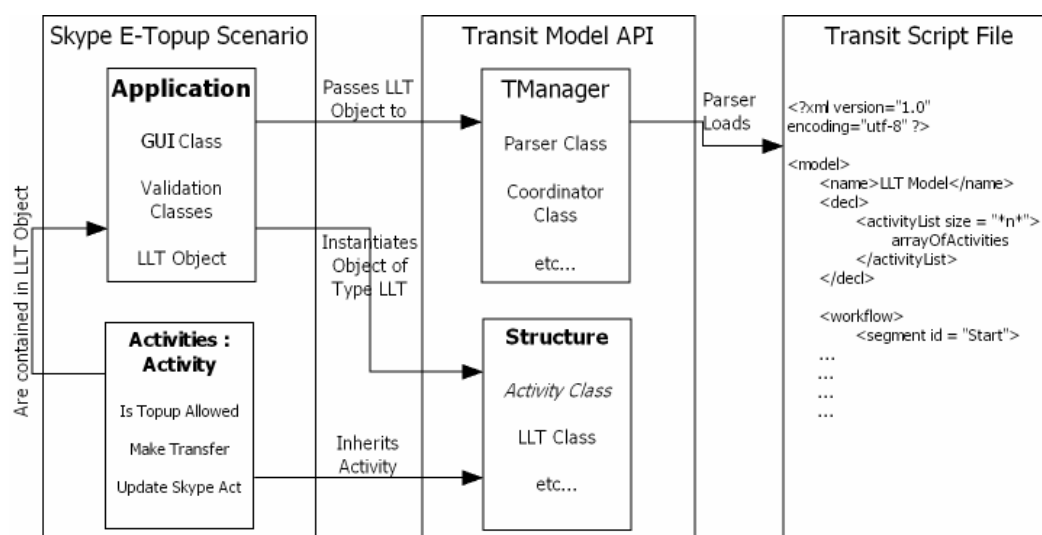


Figure EB1 Transit Enabled Skype Top Up Architecture

In this case we are assuming that the top up process for the skype application consists of three main processes:

- Checking if the top up process is allowed for the particular customer.
- Making a fund transfer request to a third party server
- Remotely updating the Skype Account.

Besides these processes, the application is normally implemented, as if it was not Transit enabled. The only changes in design needed include the re-coding of the transaction which tops up the Skype account into a long running transaction, thus posing minor alterations to the above three processes. This includes the following three steps:

- 1. The addition of the Transit Model API into the Project.**
- 2. The Creation of an "Activities" folder and Activity Classes**

This folder should contain three classes, one for each activity which is contained in the long lived transaction. Each of these classes should extend the Activity abstract class available in the Transit Model API, and over ride each of its virtual methods. As previously explained, these methods are crucial for the runtime engine to execute the final transaction.

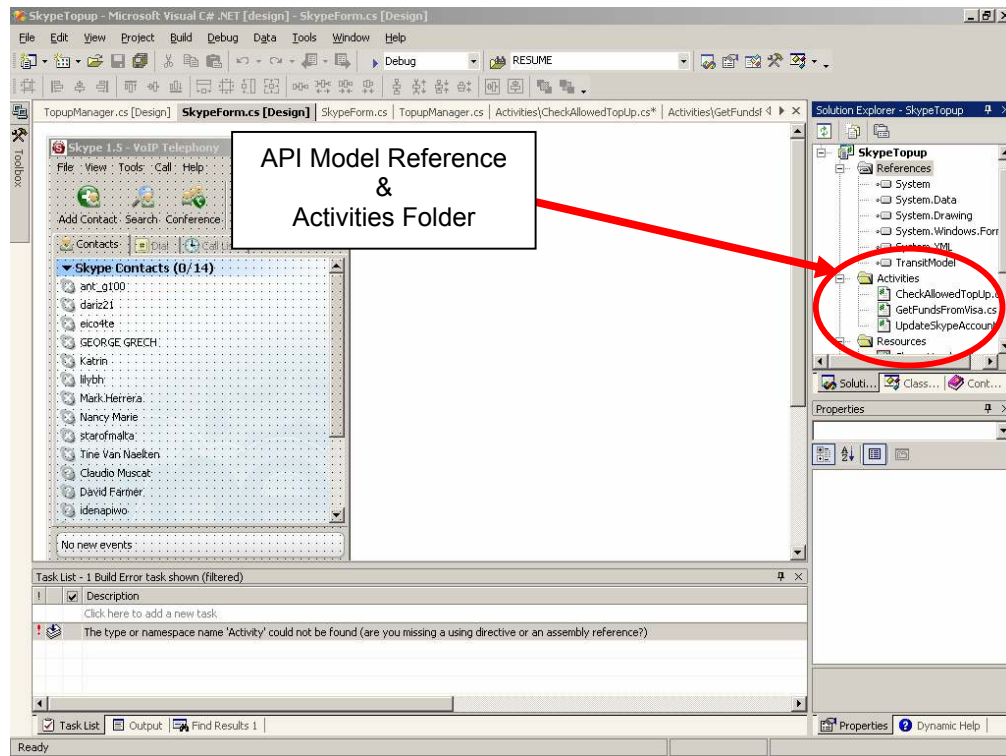


Figure EB2 Architectural Changes to an Application

The following code shows the structure of the CheckAllowedTopup Activity implementing the standard methods which have been overridden from the Activity class contained in TransitModel.Structure. While other custom methods may be implemented, it is imperative to implement these methods in the “try” “catch” format described in the second diagram below. Also note that Activity classes must be marked as Serializable, to enable persistence to disk.

```
using TransitModel.Structure;
namespace SkypeTopup.Activities
{
    [Serializable]
    public class CheckAllowedTopUp : Activity
    {
        public CheckAllowedTopUp()
        {
            //
            // TODO: Add constructor logic here
            //
        }
        public override void activityRun()
        {
            // TODO: Implement Here
        }
        public override void activityCommit()
        {
        }
    }
}
```

```
        // TODO: Implement Here
    }

    public override void activityRollBack()
    {
        // TODO: Implement Here
    }

    public override void activityCompensate()
    {
        // TODO: Implement Here
    }

    public override void activityResumeRun(object obj)
    {
        // TODO: Implement Here
    }

    public override void activityResumeCommit(object obj)
    {
        // TODO: Implement Here
    }

    public override void activityResumeRollBack(object obj)
    {
        // TODO: Implement Here
    }

    public override void activityResumeCompensate(object obj)
    {
        // TODO: Implement Here
    }
}
}
```

Figure EB3 CheckAllowedTopup Extending from the Activity Class

```
public override void activityRun()
{
    try
    {
        //Carry out remote server request
        //If response is positive
        this.setStatusToCompleted(); //Transaction Successful

        //Else if response is negative
        this.setStatusToRolledBack(); //Transaction Failed
    }
    catch
    {
        // If Server connection has been lost
        this.setStatusToWaitRun();
    }
}
```

Figure EB4 CheckAllowedTopup Run Method

3. The Instantiation of the Activities and the LLT Object

The next step includes the instantiation of each class in the Activities folder, and its inclusion into an LLT Object. In this case, this has been carried out in the main Windows Form, however there is no restriction on the developer on the code location, as long as the Activity Objects are created, and added to an LLT Object. The only requirement needed is the addition of the TransitModel.Structure namespace to the class in which the LLT Object is created:

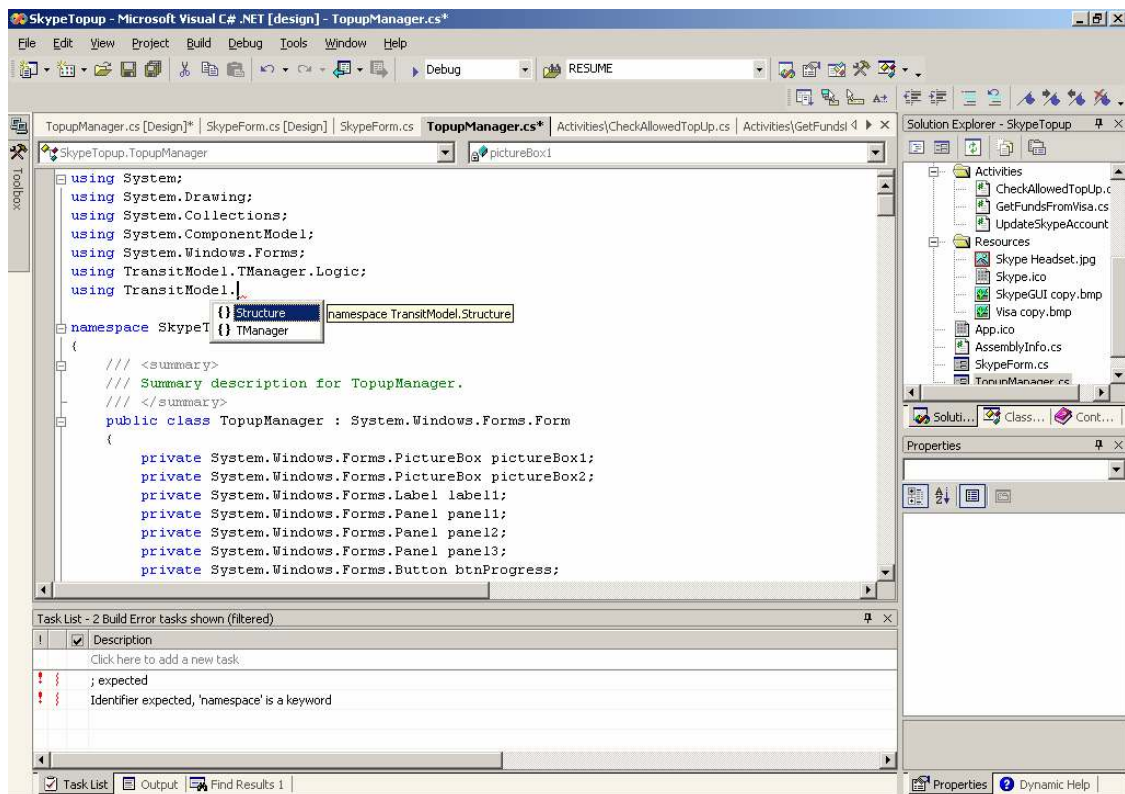


Figure EB5 Adding the TransitModel References

```
//Construction of the activity structure

Activity checkIfAllowedTopup = new Activities.CheckAllowedTopUp();
checkIfAllowedTopup.Name = "Is Topup Allowed";

Activity getFundsFromVisa = new Activities.GetFundsFromVisa();
getFundsFromVisa.Name = "Get Funds From Visa";

Activity updateSkypeAccount = new Activities.UpdateSkypeAccount();
updateSkypeAccount.Name = "Update Skype Account";

//Copy everything into an ArrayList
```



```
ArrayList activityList = new ArrayList();
activityList.Add(checkIfAllowedTopup);
activityList.Add(getFundsFromVisa);
activityList.Add(updateSkypeAccount);

//Create an LLT

LLT topupTransaction = new LLT(activityList);
```

Figure EB6 Creating the Long Running Transaction

At this point, all the alterations needed to the architecture of the application itself have been completed. We now have a new transaction which may be passed onto the transaction manager and executed. Let us now consider the previous use case diagram. In order to transit enable an application, two extra functions are typically added:

- A function to execute a new Transaction, in this case “Buy Credit”, represented as a clickable button in the Skype Top up Manager GUI.
- A function to check the progress of an already initiated Long Running Transaction, in this case represented by the “Progress” button in the Skype Top up Manager GUI.

Both these functions are handled using the transaction manager provided in the Transit Model Solution’s TManager Namespace.

C.) Transaction Management

The transaction manager provides a series of methods which enable transaction execution, suspension, resumption, and log viewing (see previous appendix for complete list of utilities of the Transit Model API). However, in order to obtain access to these utilities, the following simple steps are initially followed:

- The addition of a reference to the TransitModel.TManager namespace. (in this case, to the Skype Topup Manager class)
- The creation of a pointer to the Manager’s singleton instance:

```
Mgr tmanager = Mgr.getMgrInstance();
```

Once a reference to the transaction manager instance has been acquired, the developer has access to all its methods. In this case, we want to either post a brand new transaction to the transaction manager and execute it, or view the progress of a running transaction. Consider each possibility:

- **Running a new Transaction**

Due to the architectural changes previously carried out, it is now a trivial task to execute a new transaction. The process includes the following two steps:

- Call the Transaction manager's "brandNewLLT" method, in order to post the previously constructed LLT object to the manager as a new transaction.
- Call the Transaction manager's "runLLT" method, which initiates execution.

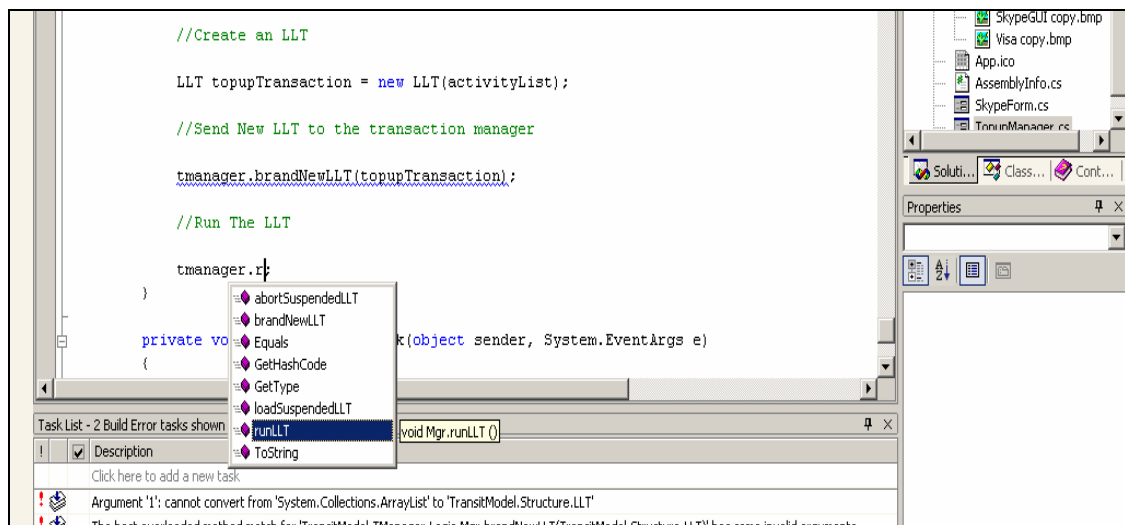


Figure EC1 Running the Transaction

Since this process may take a lengthy amount of time, it is ideally implemented on a separate thread. It results in a series of changes in states of the Activities contained in the LLT object, which may be then interpreted by the Skype Top Up Manager class in order to inform the end user of the current state of the transaction. If any of the Activities in the LLT object is set to a wait mode, the transaction is not yet completed, and may be resumed through the second function which enables to view the progress of a transaction, and resume it if needed.

In this case, we have chosen to graphically represent the progress of a transaction in the Skype application, by implementing a colour coded progress indicator which gives indications about the transaction's progress in real time. However this is not part of the transit model solution, but rather a real time graphical interpretation of the execution progress of the Transit Model Solution's transaction manager engine. Sample screenshots can be viewed in section E.

- **Viewing Transaction Progress/Resuming a Transaction.**

While actual transaction progress has been catered for through visual illustrations in the Skype Top Up Manager, resumption of a transaction must be handled through the specialized GUI provided in the Transit Model Solution. The skype application developer can handle the situation by simply creating an instance of the GUI, upon suspension of a transaction, or upon a user request through a button click. In this example, it has been decided to trigger off instantiation of the Transit GUI thorough the "Progress" button, which is present in the Skype Topup Manager. Upon click of the Progress button, the following process takes place:

- A Transit reference to the TransitControlPanel Singleton Instance is obtained.
- The show() method is called, in order to display the Transit GUI which then handles resumption.

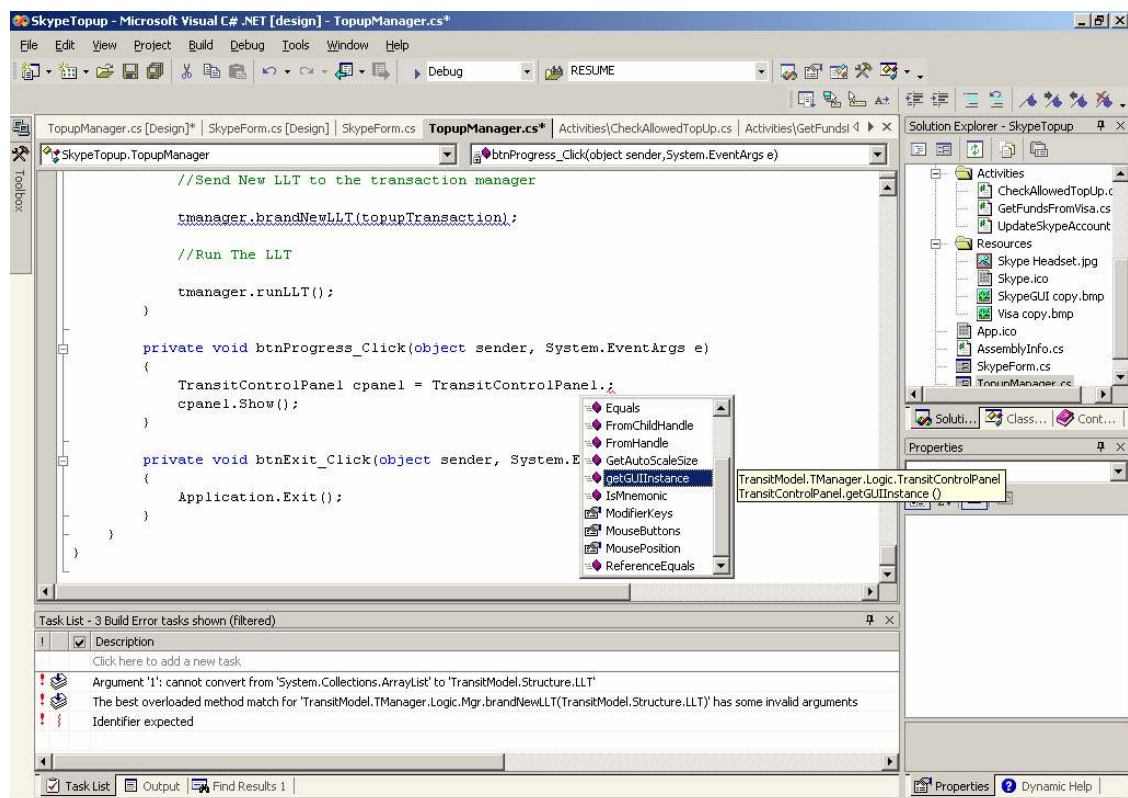


Figure EC2 Instantiating the Transit Resume GUI

This results in the instantiation of the transit control panel:

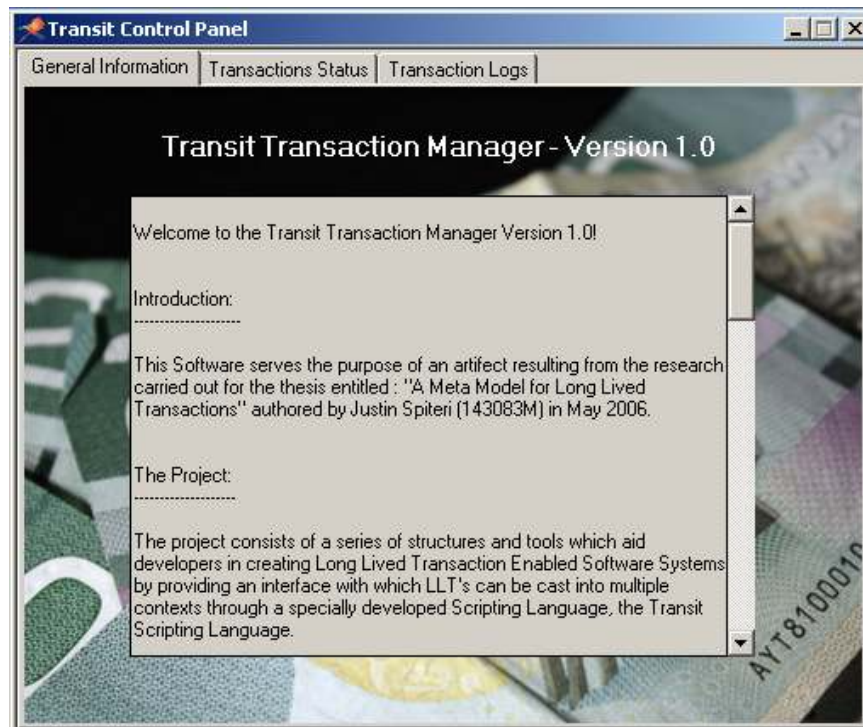


Figure EC3 The Transit Model Resume GUI Instance

At this point, the implementation per se is complete, when considering it from a transaction handling perspective. We thus have a working Transit Enabled Topup Process, which simulates Skype's Top Up process if it had to be run as a long running transaction.

One may notice that up till now, the developer has absolutely not catered for transaction coordination, inter dependencies, and execution sequences. He has merely put a series of activities into an Array List, and LLT Object, and passed them to the Transit Transaction Manager for handling. This outlines the success of the Transit Model solution in the "abstraction of transactional complexities" context.

The only drawback presented in this system is that the developer must pay attention to the sequence in which he constructs the array list of activities, since this directly effects the execution process. Transit Scripts are based on the idea of executing a workflow which defines an execution sequence based on the arraylist positions of a set of activities. As explained in the thesis, these arraylist positions act as "place holders" into which activities are plugged by the developer. Thus the developer must be sure to have plugged the correct activity in the right position, according to the script he is using; otherwise, erroneous execution would occur.

D.) Choosing a Model

The choice of a model largely depends on the choice of the developer. This particular application has been developed for demonstrative purposes, and has been made to be compatible with all three sample scripts provided with this thesis. In order to enable an application to run on a particular script, upon compilation, a script file must be copied to the application's /Logic/Model directory;

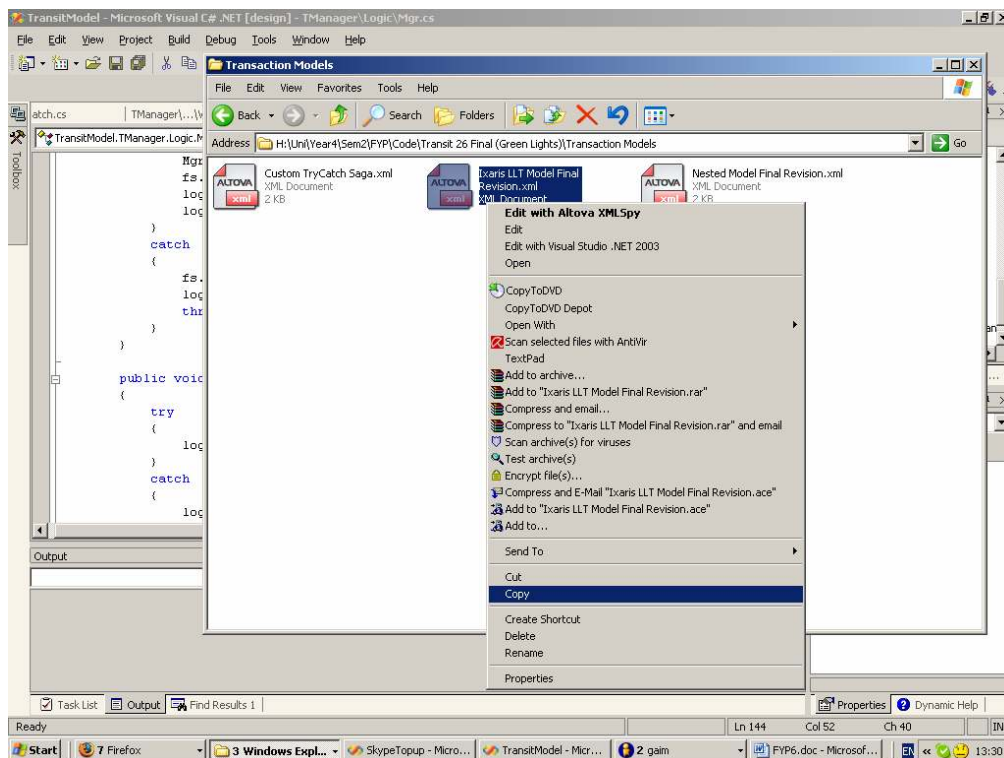


Figure ED1 Choosing a Script for the Skype Topup Application

If the script file is changed, the application (Skype) will successfully run on the new model provided it has compatible logic. This would however mean that the application needs to be re-started, and any pending transactions will be wiped out.

E.) The Result

The following series of screenshots provide a walkthrough of the execution of the Top Up Process for the Transit Enabled Skype Top Up Facility using the LLT Model proposed by Sun's JSR 95. The source code and executables of this example can be found on the compact disk accompanying this thesis:



Figure EE1 Transit Enabled Skype Top Up Main Form

This is the initial menu of the Skype program. Note the "Top Up Your SkypeOut Button". When this button is clicked, the following screen pops up:

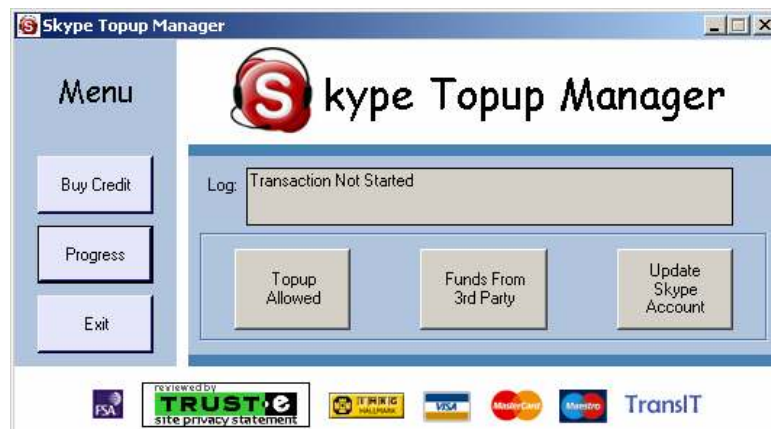


Figure EE2 Skype Top Up Manager Form - Idle

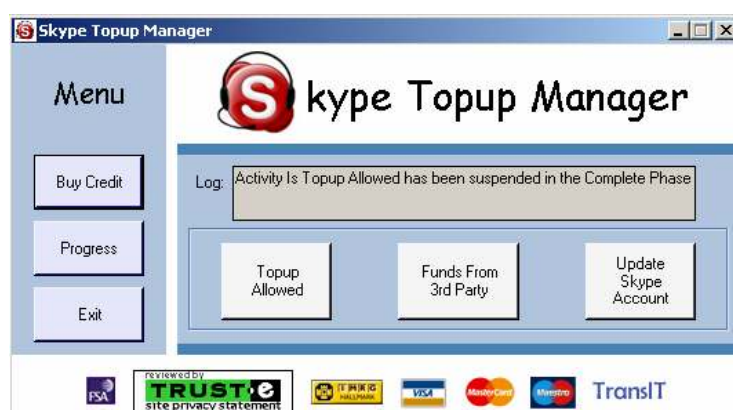


Figure EE3 Skype Top Up Form – New Transaction Started

In the above diagram, the Buy credit has been clicked, and the execution process starts. In the one below, the Transaction got suspended due to third party server connection loss. The user then clicked the Progress button, thus displaying the Transit Resume GUI:

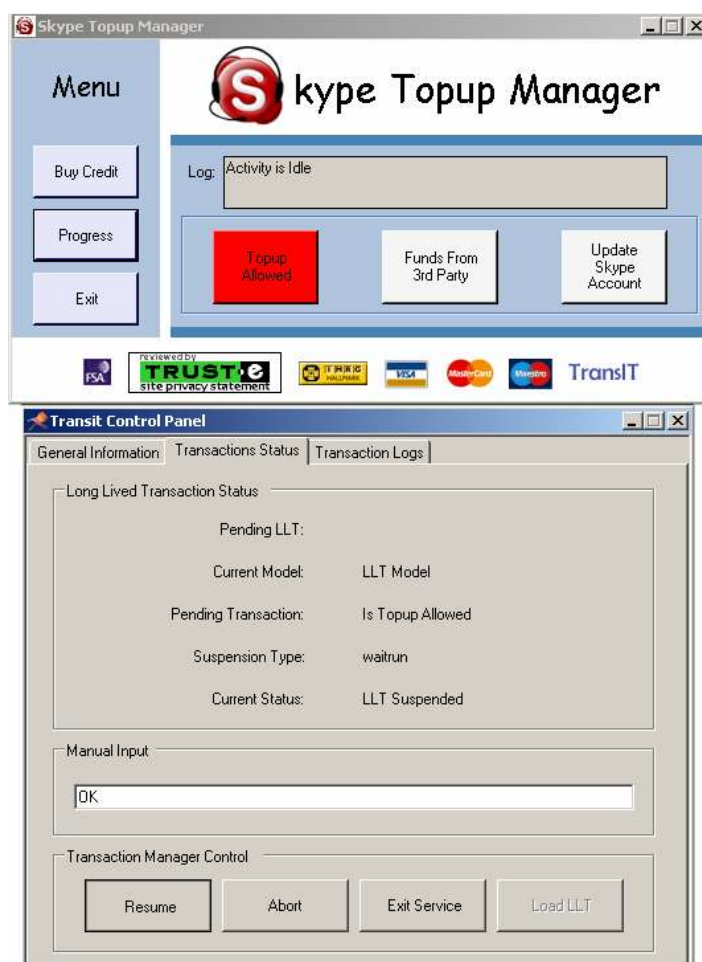


Figure EE4 Skype Top Up Form – Suspended + Transit Resume GUI

While the transaction is in wait mode, detailed execution logs may be viewed in the Transit Control Panel's Logging facility:

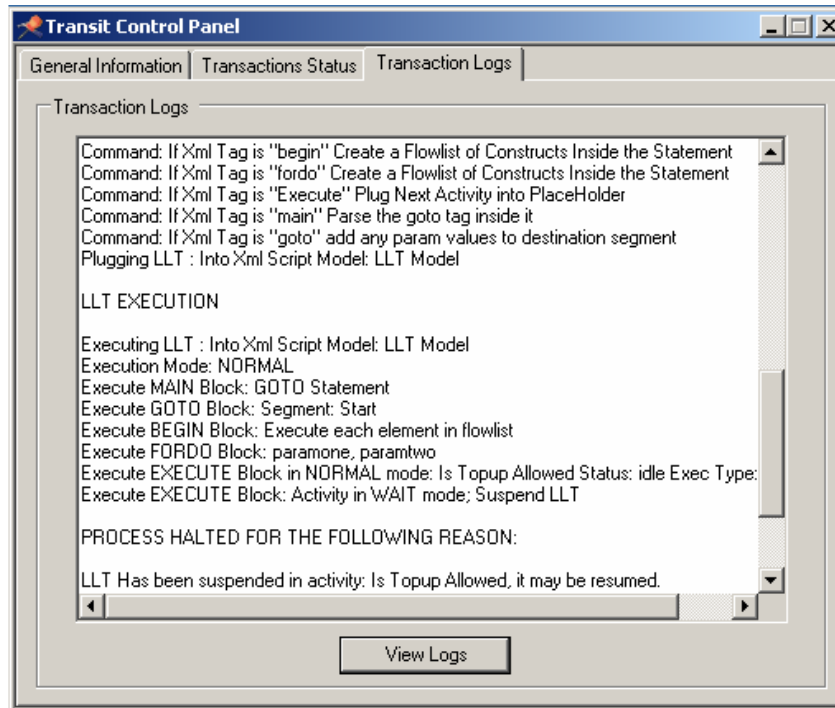


Figure EE5 Skype Top Up Form – Resumed/Running

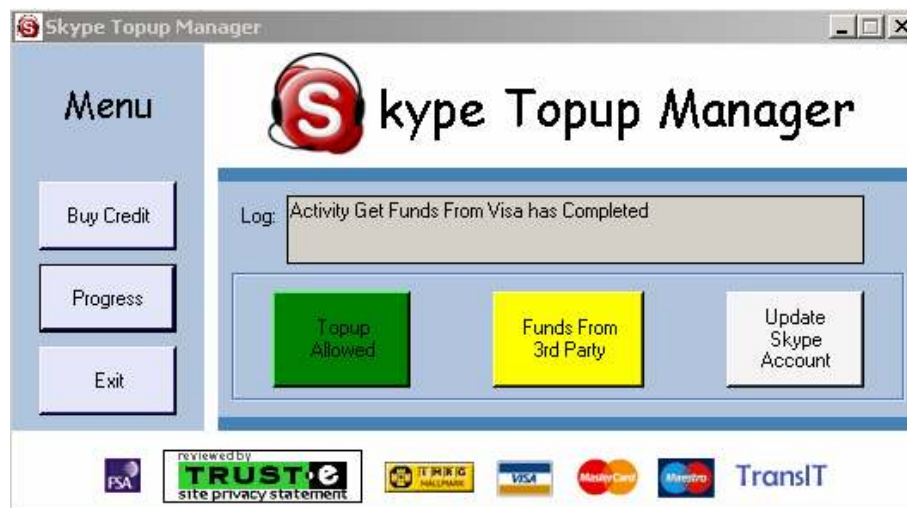


Figure EE6 Skype Top Up Form – Resumed/Running

At this point, the transaction has been resumed through the Transit GUI. The first activity has committed, while the second activity has completed.

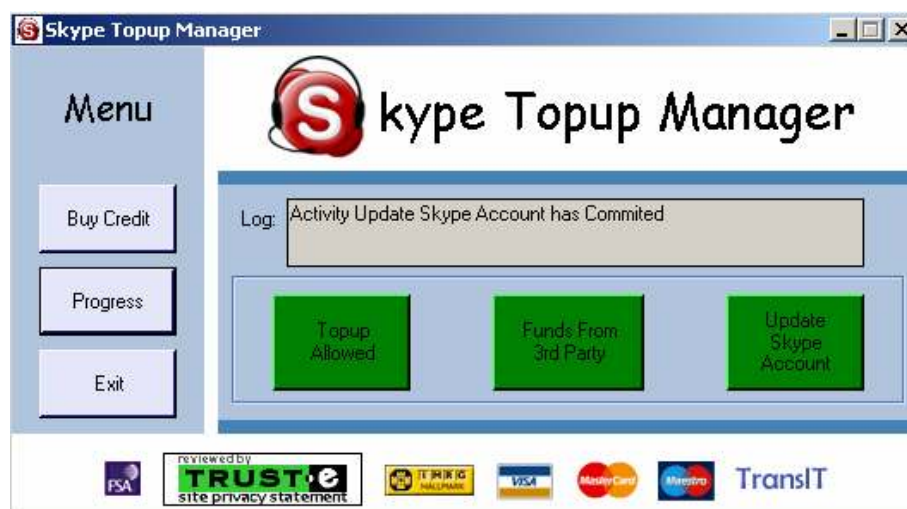


Figure EE7 Skype Top Up Form – LLT Committed

Finally, this diagram displays the final result after execution. In this case, all three activities have successfully committed. Had one activity rolled back, according to the JSR 95 LLT model which has been used in this case, all previous activities would have compensated, displaying a gray colouring on the respective Activity Icons:

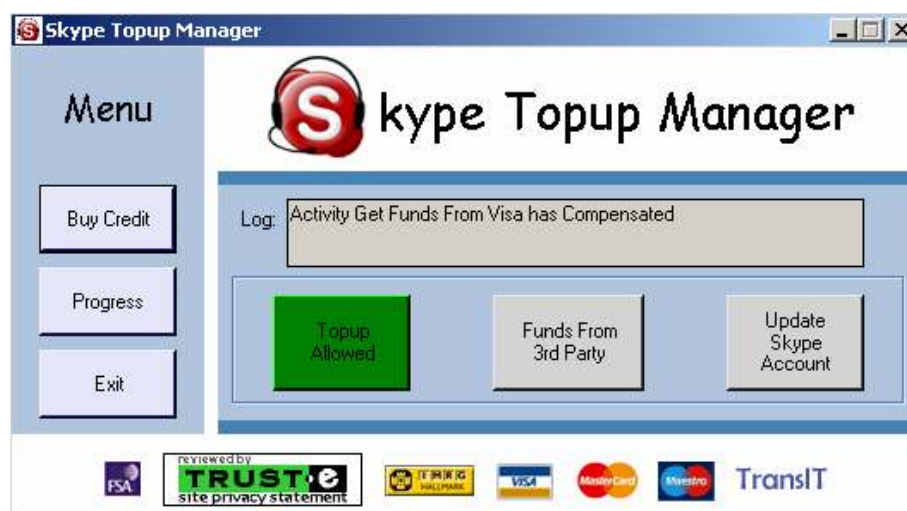


Figure EE8 Skype Top Up Form – LLT Compensating

Appendix F: Example - Transit Enabled Holiday Planner

The Holiday Planner Application represents a typical holiday booking system which allows customers to book flight tickets, hotel reservations, and train tickets for their holiday trips. The main goal of this application is to demonstrate the wide range of applications to which the Transit Model Solution may be applied. While the application architecture is unique, the transaction management architecture is identical to the Skype E-Topup Facility. It has thus been deemed non practical to repeat the illustration of the architecture, since once can refer to the previous appendix for technical details. The following screenshots illustrate the operation of the application:

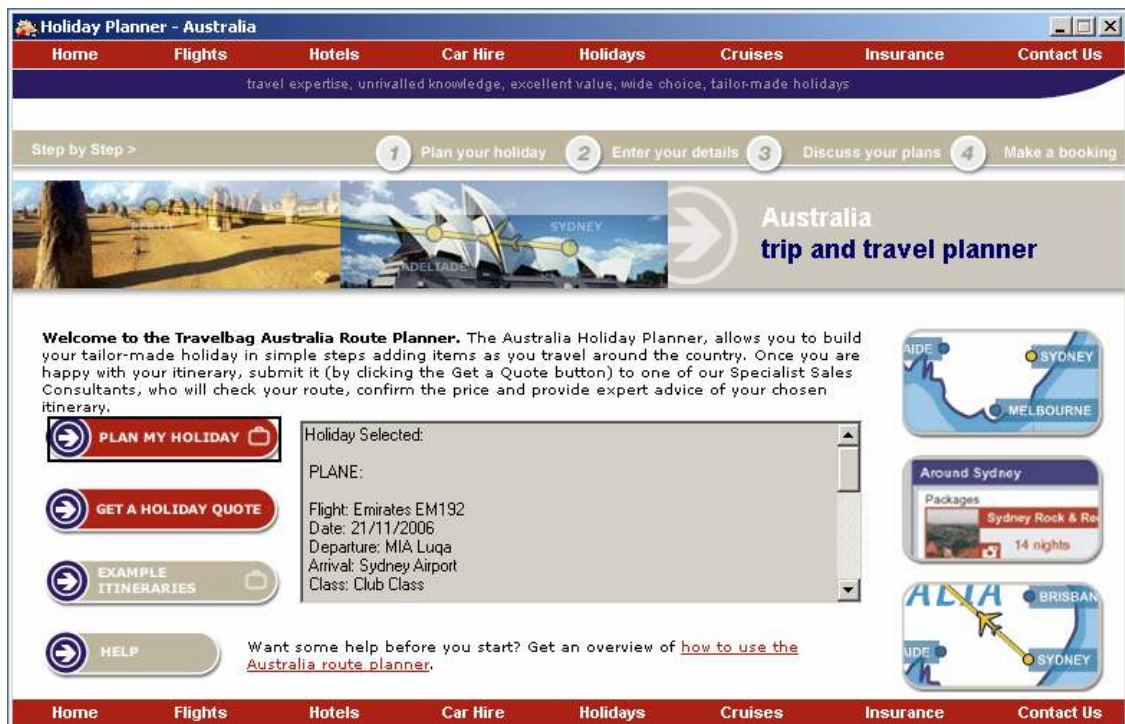


Figure F1 Holiday Planner Form – LLT Model



Figure F2 Holiday Planner Form 2 – LLT Model

Appendix G: Transit API Testing

The main idea of testing is to ensure that the quality of the software application is up to standard, and thus the this test plan has been set up in order to aid the developer to make sure that this goal is reached. The first type of testing to be carried out will be that of white box testing on the most important modules using the code walkthrough method.

Upon successful completion of white box testing, black box testing will be undertaken, this time through the use of an external application; the Transit Enabled Skype E - Top Up facility.

A.) White Box Testing

Test 1: Code Walkthrough: Model Object Creation	
Test Rig	The rig for this particular test consists of the code trace through the process of loading a transaction model from the script file into the transaction model, parsing using the coordinator class, and merging the loaded model to an example LLT in order to create a Model Object.
Test Data	<p>For this particular test, the script file containing the SAGA model (included on compact disk /Executables/Transaction Model Examples/TryCatchSaga.xml) will be used.</p> <p>With regards to the LLT Object, the structure used will be the one described in appendix E, consisting of the three activities making up the Skype Top Up long running transaction :</p> <ul style="list-style-type: none">• Checking if the top up process is allowed for the particular customer.• Making a fund transfer request to a third party server

	<ul style="list-style-type: none"> Remotely updating the Skype Account.
Expected Results	<p>Since the file has been tested through Altova's XML validation software, it is known that the XML file is syntactically correct.</p> <p>The LLT is also programmatically correct, and compatible with the model's logic.</p> <p>Since all the test data is correct, a positive result is expected, where the Model Object is successfully created.</p>
Actual Results	<p>The test was carried out using the test data described, and the result was the successful creation of the Model Object. No compromising bugs were detecting during the tracing of the code.</p>
Outcome	<p>This test proves that the the following classes or methods are operating as expected:</p> <ul style="list-style-type: none"> TransitModel.TManager.Logic.Coordinator TransitModel.TManager.Logic.Mgr.brandNewLLT(); TransitModel.TManager.Logic.Mgr.getMgrInstance(); TransitModel.TManager.Logic.Mgr.abortSuspendedLLT(); <p>It also proves that all data structures involved in this process, mainly the LanguageBlock Classes, are all operating correctly.</p>

Test 2: Code Walkthrough: Model Object Execution: Normal	
Test Rig	<p>The rig for this particular test consists of the code trace through the process of executing the Model Object in a case where all the activities commit. The data rig used is identical to the one in the previous test.</p>
Test Data	<p>For this particular test, the script file containing the SAGA model (included on compact disk /Executables/Transaction Model Examples/TryCatchSaga.xml) will also be used.</p> <p>With regards to the LLT Object, the structure used will also be the</p>

	<p>one described in appendix E, consisting of the three activities making up the Skype Top Up long running transaction :</p> <ul style="list-style-type: none"> • Checking if the top up process is allowed for the particular customer. • Making a fund transfer request to a third party server • Remotely updating the Skype Account.
Expected Results	<p>This test should result in the complete execution of the transaction, without any persistence to disk. The Boolean value indicating suspension of a transaction found in the execute(); method of the "Execute" language block should be set to false throughout the process.</p>
Actual Results	<p>Execution proceeded, and concluded successfully, with the commission of all activities. The execute language block did not switch to suspend mode, since the Boolean value indicator was set to false throughout the process.</p>
Outcome	<p>This test can be considered as successful. The execute() in the Execute Language Block can be considered to be functioning as expected, with regards to normal execution.</p>

Test 3: Code Walkthrough: Model Object Execution: Suspend	
Test Rig	<p>The rig for this particular test consists of the code trace through the process of executing the Model Object in a case where the long running transaction gets suspended in its last activity (update skype account). The data rig used is identical to the one in the previous test.</p>
Test Data	<p>For this particular test, the script file containing the SAGA model (included on compact disk /Executables/Transaction Model Examples/TryCatchSaga.xml) will also be used.</p> <p>With regards to the LLT Object, the structure used will also be the one described in appendix E, consisting of the three activities</p>

	making up the Skype Top Up long running .
Expected Results	<p>While the SAGA model is still correct, this time, a connection loss is simulated in the "update skype account" activity, which happens to be the last activity to be executed in the Skype Top Up Long Running Transaction. The connection loss is simulated by throwing an exception when the activity tries to commit:</p>
	<pre> public override void activityCommit() { try { //Simulate Work on Remote Server Thread.Sleep(1500); //Simulate Connection Loss throw new Exception(); ... } catch { this.setStatusToWaitCommit(); } } </pre>
	<p>Figure GA1 Connection Error Simulation</p> <p>This should trigger off a transaction suspension mechanism which switches the boolean value indicator in the Execute language block to true, thus indicating a suspension. The activity should also change to waitCommit. This should subsequently trigger off serialization of the entire Model object to disk, halting of execution, and wiping of the transaction information from memory.</p>
Actual Results	<p>Execution proceeded normally until the simulated connection loss. The activity switched state successfully, and the execute(); method in the Execute Language Block also executed serialization of the Model Object Successfully. The boolean indicator switched to true before serialization occurred, thus indicating the presence of a suspended transaction.</p>
Outcome	<p>This test can be considered successful, since the actual results equaled the expected results in a satisfactory manner. Serialization to disk works well, and all the necessary parameters</p>

	switched to suspended state.
--	------------------------------

Test 4: Code Walkthrough: Model Object Execution: Resume	
Test Rig	The rig for this particular test consists of the code trace through the process of executing the Model Object in a case where the long running transaction is resumed from disk, rather than posted by the end user. The class being tested in this case is the Execute Lanugage Block, which contains the entire resumption Logic. The data rig used is identical to the one in the previous test.
Test Data	<p>For this particular test, the script file containing the SAGA model (included on compact disk /Executables/Transaction Model Examples/TryCatchSaga.xml) will also be used.</p> <p>With regards to the LLT Object, the structure used will also be the one described in appendix E, consisting of the three activities making up the Skype Top Up long running .</p>
Expected Results	The most relevant results to be observed in this case are the correct de-serialization of the Model Object from disk, the correct simulation of execution of activities which have already been run, and the switching from simulation mode to normal execution, as soon as the point where the transaction was suspended is reached. The execution of the resume method in the activity is also to be observed.
Actual Results	<p>The code trace initially started with the successful de-serialization of the Model from disk and proceeded with the simulated execution of the already executed activities. As soon as the simulation arrived to the third activity's commit step, the wait state was identified, and the resume method was called.</p> <p>This executed successfully, and was followed by the switch from simulation resume mode to normal mode. This resulted in the third activity being committed, and the overall Transaction being committed.</p>

Outcome	The seamless transition between the simulated execution and the normal execution of the transaction makes this test a successful one.
----------------	---

B.) Black Box Testing

Test List: Exhaustive Testing			
Test Rig	<p>Black box testing on the Transit Model API was carried out in by exploiting the Skype Top Up Application in order to create a series of scenarios which exhaustively cover every possible execution outcome of a transaction, using different models transaction models. All the tests have been included on the compact disc accompanying this thesis, and can be found in the /Executables/Application Examples directory. The tests are listed below, together with the scenario they represent, and the outcome of each test.</p> <p>Please note that each test was repeated using a second application, a Travel Agent Facility, also found on the compact disc.</p>		
Test Name	Expected Result	Actual Result	Outcome
LLT 1	Transaction Commit	Match	Successful
LLT 2	Activity 1 Suspends, Then resumes & Transaction Commit	Match	Successful
LLT 3	Activity 3 Rolls Back, Activities 1 & 2 Compensate.	Match	Successful
LLT 4	Activity 3 Rolls Back, Activities 1 & 2 Compensate, suspension & resumption in compensate.	Match	Successful
SAGA 1	Transaction Commit	Match	Successful
SAGA 2	Activity 1 Suspends, Then resumes &	Match	Successful

	Transaction Commit		
SAGA 3	Activity 3 Rolls Back, Activities 1 & 2 Compensate.	Match	Successful
SAGA 4	Activity 3 Rolls Back, Activities 1 & 2 Compensate, suspension & resumption in compensate.	Match	Successful
Nested 1	Transaction Commit	Match	Successful
Nested 2	Activity 1 Suspends, Then resumes & Transaction Commit	Match	Successful

Appendix H: Application Requirements & CD Contents

A.) Application Requirements

The minimum hardware requirements for this software to run are tied down to any system which is capable of running the .NET Framework 1.1. Since the source accompanying this project has a prototypical nature, no network connections are required to run the applications present on disc. With regards to software requirements, below is a summary of the generic software requirements which the application needs.

- **General Framework:**

- .Net Framework 1.1 or higher
- The Test Applications need to have a Transit Script in their /Logic/Model directory.

- **Application Installation:**

- Windows based OS
- 50 Mb Hard Drive Space (Excluding Prerequisites)

Installation & Execution:

Installation of the software is very simple, it merely involves inserting the compact disc into the drive and copying the "Executables" Folder onto the desired location in the hard drive. The folder containing the desired test case may then be navigated to and opened. To run a test case, simply double click the executable file in the corresponding folder.

B.) CD-ROM Contents

- Deployment
 - Transit Model API Compiled Library
 - Transaction Model Examples (Transit Scripts)
 1. SAGA Model
 2. JSR 95 LLT Model (Based on Ixaris Implementation)
 3. Nested Model
- Documentation Files
 - Application Documentation
 - XML Documentation
 - In Line HTML based Documentation
- Prerequisites
 - .NET Framework
 - Adobe Reader
 - Textpad (for viewing Transaction Models)
- Source Code
 - Transit Model API Source Code
 - Skype E-Top Up Facility Source Code (various test scenarios)
 - Holiday Planner Facility Source Code (various test scenarios)
 - Transaction Model Examples (Transit Scripts)
 1. SAGA Model
 2. JSR 95 LLT Model (Based on Ixaris Implementation)
 3. Nested Model
- Tests
 - Skype E-Top Up Facility Source Code (various test scenarios)
 - Holiday Planner Facility Source Code (various test scenarios)
 - Transaction Model Examples (Transit Scripts)
 1. SAGA Model
 2. JSR 95 LLT Model (Based on Ixaris Implementation)
 3. Nested Model

Appendix I: SourceForge Details

A.) The SourceForge Application Form:

The screenshot shows the SourceForge website interface. At the top, there's a navigation bar with links like 'SF.net', 'Projects', 'My Page', and 'Help'. Below this is a sub-navigation bar with 'Summary', 'Projects', 'Tracker', 'Tasks', 'Donations', and 'Preferences'. The main content area is titled 'Project Registration Details Page'. It contains several sections: 'Project submission' with fields for 'Created', 'Last modified', 'Submitter', 'Project type', 'UNIX name', and 'Descriptive name'; 'Public description' with a text area and character count; 'Trove categorization' with a list of categories and an 'Edit' button; 'Registration description' with a text area and character count; 'Current status' showing 'Pending Review'; and 'Review comment' with a text area. At the bottom, there are three buttons: 'Save draft', 'Resubmit for review', and 'Cancel registration'.

Project submission

Created: 2006-04-24 05:02
Last modified: 2006-04-24 06:28
Submitter: Justin Spiteri ([jspiteri](#)) [True Identity: Justin J. Spiteri]
Project type: An Open Source Software Project
UNIX name: transitmodel
Descriptive name: Transit Model

Public description:
Transit
is an academic project which involves the creation of an XML based scripting language which allows developers to easily create custom models for long lived transactions. An API is also available, allowing
250 Characters in public description - max of 255 chars (Requires JavaScript)

Trove categorization:

- License :: OSI-Approved Open Source :: Academic Free License (AFL)
- Intended Audience :: by End-User Class :: Developers
- Development Status :: 3 - Alpha
- Topic :: Software Development :: Frameworks
- Programming Language :: C#
- Operating System :: Modern (Vendor-Supported) Desktop Operating Systems :: WinXP
- User Interface :: Graphical :: Win32 (MS Windows)
- Translations :: English

Registration description:
This project is the result of an extensive research carried out for an Academic Thesis on Long Lived transactions, mainly Sun's JSR 95 specification, Hp Arjuna's WS_CAF Project, and various other papers and solutions. includes the development of a simple scripting language based on XML, which simplifies the life of the developer who needs to develop complex transaction handling applications, and lacks heavy knowledge about official transaction models etc. The script has a predefined syntax, and an API which includes a parser and execution framework is also present, thus making integration into other projects possible. An extra feature of the API is that it contains a GUI which
1011 Characters in registration description - min 200 chars (Requires JavaScript)

Current status: Pending Review
Review comment:

Figure IA1 The SourceForge Application Form

B.) The SourceForge Approval E-Mail:

Approval Email

If this project were approved today, the following email would have been sent to the project administrator. (If this project was approved in the past, a different version of this text may have been provided; shown is the current version of this text, sent to newly approved projects.)

Subject: SourceForge.net Project Approved

Your project registration for SourceForge.net has been approved.

Project Information:

Project Descriptive Name:	Transit Model
Project Unix Name:	transitmodel
CVS Server:	cvs.sourceforge.net
Shell Server:	shell.sourceforge.net
Web Server:	transitmodel.sourceforge.net

Project Administration:

The Project Admin page for your project may be accessed at
https://sourceforge.net/project/admin/?group_id=166712
after logging-in.

Service Availability for New Projects:

The DNS for your project web site may take up to 24 hours to become active. Until DNS is active for your project, attempts to access your project web site will result in 404 errors. Once DNS is active, you will see an empty directory index when accessing your project web site, until you have placed content in your project web space (remember: project web space is provided solely for use in storing project-related information; see the Web section of the Project Admin page for additional details).

Your access to the project shell and CVS servers (including your new CVS repository, which has already been initialized and is ready for your first import) are typically available within four hours from the time when your project was approved. If after 6 hours your shell/CVS accounts still do not work, please submit a Support Request (on the "alexandria" project, see below), so as that we may look in to the problem.

Site Documentation and Support:

SourceForge.net maintains a large amount of documentation about the SourceForge.net site and services offered to hosted projects. This documentation may be accessed using the "Site Docs" link in the left navbar, or directly at: https://sourceforge.net/docman/?group_id=1

Should you need to contact the SourceForge.net team, we may be reached by submitting a Support Request at:
https://sourceforge.net/tracker/?func=add&group_id=1&atid=200001

Reminder: Acceptable Use and Project Licensing:

By using the SourceForge.net site, you agree to be bound by the terms and conditions of the SourceForge.net Terms of Use Agreement.

SourceForge.net provides hosting solely for Open Source software development projects; if your project is not being released under an Open Source license, or is not developing software, please contact the SourceForge.net team immediately for assistance. Questions regarding acceptable use of the SourceForge.net site and resources should be directed to the SourceForge.net team by submitting a Support Request (see above).

Donation System:

SourceForge.net provides a donation system that allows users and projects to accept donations on an opt-in basis.

You may opt-in your user account to receive donations at:
https://sourceforge.net/my/donate_manage.php

You may opt-in this project to receive donations at:
https://sourceforge.net/project/admin/donations.php?group_id=166712

Documentation on the donation system may be found at:
https://sourceforge.net/docman/display_doc.php?docid=20244&group_id=1

Getting Started:

A significant amount of project service information may be found on the Project Admin pages for your project, as seen at:
https://sourceforge.net/project/admin/?group_id=166712

The Project Admin page for your project is the best place to start. Please ensure that you have established a suitable Public Description for your project, and have categorized your project within the Trove; both of these operations may be performed using the "Public Info" section of your Project Admin pages.

Enjoy the system, and please tell others about SourceForge.net. Let us know if there is anything we can do to help you (we can always be reached

by submitting a Support Request on the "alexandria" project (see above)).

- the SourceForge.net crew

C.) The Transit SourceForge Web Site:

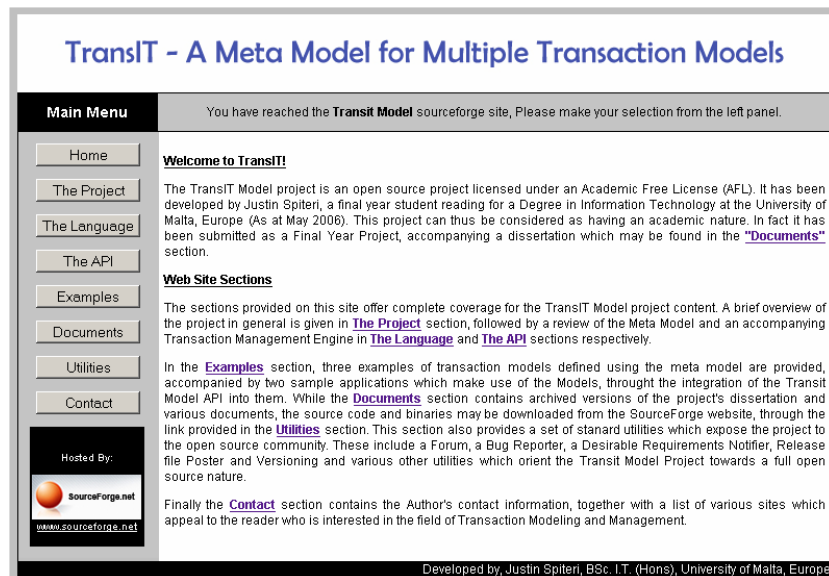


Figure IB1 The Transit Project's Sourceforge Site

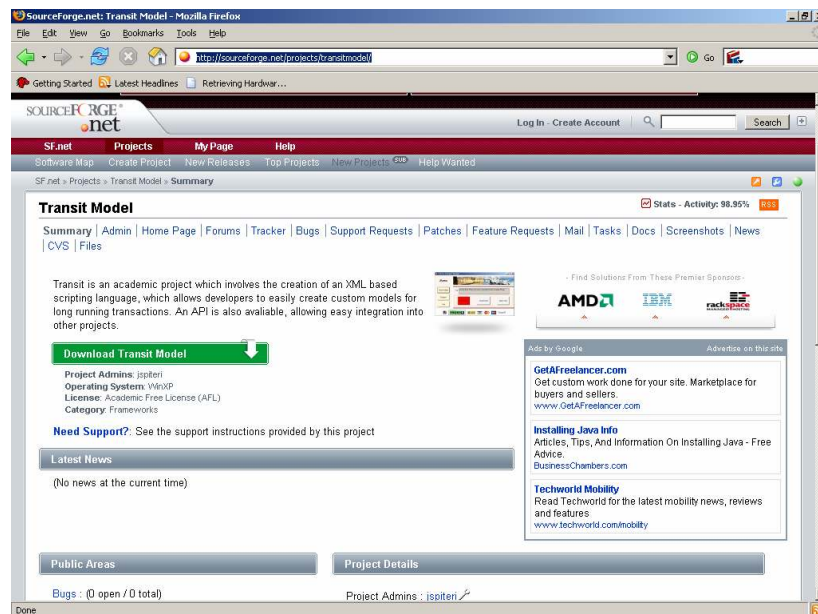


Figure IB2 The Transit Project's Sourceforge Utilities

Appendix J: Bibliography & References

A.) Bibliography & References

- [1] A news site about who released what protocol/standard/framework and when: <http://xml.coverpages.org/coordination.html#specs>
- [2] Mark Little's Personal Blog Site:
http://markclittle.blogspot.com/2004_11_01_markclittle_archive.html
- [3] Mark Little's Web Log on Webservices.org:
<http://www.webservices.org/ws/content/view/full/52229>
- [4] Current Standards used by Arjuna Technologies:
<http://www.arjuna.com/standards/>
- [5] Article: Acid is good – Take it in short doses:
<http://www.theserverside.com/articles/article.tss?l=AcidShortDoses>
- [6] Article: Business Transaction Protocols – Transactions for a new age:
<http://webservices.sys-con.com/read/39607.htm>
- [7] Article: An Overview of Support for Extended Transaction Models in J2EE:
<http://www.developer.com/java/ent/print.php/1136071>
- [8] A collection of articles and papers from Arjuna Technologies:
<http://www.arjuna.com/library/reading.html>

- [9] Article: Corba VS SOAP based Webservices:
http://searchwebservices.techtarget.com/ateQuestionNResponse/0,289625,sid26_gci930913_tax298966,00.html?bucket=ETA
- [10] Article: JTA and JTS:
<http://www.developer.com/java/ent/article.php/2224921>
- [11] Article: A comparison of Many Transaction Frameworks by Mark Little:
<http://www.webservices.org/index.php/ws/content/view/full/52213>
- [12] Framework Specification: JSR109 Web services for J2EE Documentation:
<http://jcp.org/en/jsr/detail?id=109>
- [13] Framework Specification: JSR95 Activity Service Specification:
<http://jcp.org/en/jsr/detail?id=095>
- [14] Framework Specification: Java API for XML Transactions:
<http://www.jcp.org/en/jsr/detail?id=156>
- [15] Framework Specification: WS-CAF :
<http://webservices.sys-con.com/read/39936.htm>
- [16] Framework Specification: WS-Coord :
<http://www-128.ibm.com/developerworks/library/specification/ws-tx/>
- [17] Framework Implementation: Novell Bank:
<http://www.novell.com/documentation/extendas50/jbroker/tm/examples/docs/resBank-1.htm>
- [18] Framework Implementation: WS-AT standards for IBM's Websphere:
<http://www.alphaworks.ibm.com/tech/wsat>
- [19] Framework Implementation: HP Arjuna JSR95 Transaction Service for JBOSS:

<http://www.arjuna.com/products/arjunats/>

- [20] Transaction Models: Two Phase Commit Model:
<http://www.jguru.com/faq/view.jsp?EID=20929>

- [21] Transaction Models: ACTA Model:
<http://swig.stanford.edu/pub/summaries/database/acta.html>

- [22] Transaction Models: Split/Join Model:
<http://www2.parc.com/csl/groups/sda/projects/reflection96/docs/barga/reflect/node10.html>

- [23] Open Source: Rules for open source development:
<http://www.advogato.org/article/395.html>

- [24] Open Source: The Free Software Definition:
<http://www.gnu.org/philosophy/free-sw.html>

- [25] Open Source: OSI:
<http://www.opensource.org/>

- [26] Xml Syntax Revision:
<http://www.w3schools.com/xml/default.asp>

- [27] OCCAM Syntax Revision:
<http://www.wotug.org/occam/>

- [28] The Code Project:
<http://www.codeproject.com/>

- [29] C# Corner:
<http://www.csharp-corner.com/>

- [30] Dot Net 247:
<http://www.dotnet247.com/>
- [31] Source Forge:
<http://www.sourceforge.net/>
- [32] Generic definition of Transactions (Microsoft - Msdn)
- [33] Container Interposed Transactions (Marek Prochazka & Frantisek Plasil)
- [34] Jironde: A Flexible Framework for Making Components Transactional (Marek Prochazka)
- [35] The Contract Model (Helmut Wachter & Andreas Reuter)
- [36] The ACID Model – www.about.com (Mike Chapple)
- [37] CovaTM: A Transaction Model for Cooperative Transactions (Jinlei Jiang, Guangxin Yang for Bell Labs)
- [38] Advanced Transactions in Enterprise Java Beans (Marek Prochazka)
- [39] A Practical and Modular Method to Implement Extended Transaction Models (Roger Barga & Calton Pu)
- [40] Recovery for extended transaction models (Roger Barga)
- [41] Using Constraints to manage long duration transactions in spatial information systems (Kerry Taylor & Dean Kuo)

- [42] Long Lived Transactions in a Loosely coupled Environment (John McDowall)
- [43] Recovery in Long Lived and Distributed Transaction Models (M.M. Gore)
- [44] An Open Standards approach to web services business transactions (Mark Little)
- [45] Web Services Transaction Management (Michael Felderer)
- [46] Towards a framework which captures the requirements of real workflows (Dean Kuo)
- [47] Acta: The Acta Framework Model (Chrysanthis & Ramamritham)
- [48] An Extensible Approach to realizing Extended Transaction Models (Eman Anwar)
- [49] Advanced Transactions in Component Based Software Architectures (PHD Project by Marek Prochazka)
- [50] Extendible Long Lived Transaction Processing on Distributed and Mobile Environments with recovery guarantees (PHD Project by MM Gore)
- [51] C# Core Language – Little Black Book (Bill Wagner)
- [52] Programming in C# (Jesse Liberty)
- [53] Software Engineering - A Programming Approach (Douglas Bell)
- [54] Testing Applications on the Web (Bob Johnson, Michael Hackett, Hung Q. Nguyen)

B.) Correspondence

Mr. Patrick Abela :

Project Supervisor, Developer of Long Lived Transaction Framework for Ixaris (Malta) Ltd.

Contact: patrick.abela@ixaris.com

Dr. Marek Prochazka :

Developer of Bourgogne Transactions, Charles University, Czech Republic.

Contact: marekproc@yahoo.co.uk

Mr. Mark Little – Arjuna Technologies

Chief Architect in the WS-CAF Long Lived Transaction Handling Specification

Contact: mark.little@arjuna.com

Mr. Michael Usatchev – Computer Science Academy of Moscow

Developer at the Department of Informatics of the Russian Federation - Moscow

Contact: misha@nw.mos.ru

Appendix K: Correspondence

A.) Michael Usatchev – Moscow Computer Science Academy

Subject: RE: Justin Spiteri - Dissertation
From: Michael Usatchev <misha@nw.mos.ru>
Date: Tue, 02 May 2006 18:40:45 +0200
To: Justin Spiteri <justins@waldonet.net.mt>

In the given work the author investigates processes that run in systems driven by transactions. (transaction enabled applications). General attention is paid to multi-step compound transactions in complex systems that involve objects with long(-time) life-cycles. In this work the author tries to prove propriety of application of some theoretical research and suggests a unique solution as a simple and flexible mechanism backed by modern theoretical principals and aimed to ease up complexity of transaction management.

In as much as the project does not make its aim to be a commercial solution, such parts as marketing research or investment return evaluation in this project are not presented.

In chapters 1, 2, which I can regard as the common part, the author immerses into the theory and covers general terms and issues concerning application of transaction management approaches. He also shows in detail all existing disadvantages of the traditional models and provides an illustrative real-world example as evidence. Such models as ACID, ACTA, BTP, WS, and many others are completely analyzed to detect drawbacks. After that in his conclusion the author insists on the only solution that is a combination of several models in one meta-model what will provide flexible manipulation and eliminate complicity in long live transaction management.

In the third chapter mode deeply describes the meta-model that he created. The material is elaborated from general ideas to concrete requirements and specifications. The text is well illustrated by charts and figures which successfully facilitate understanding of the ideas. The author does not avoid mentioning the Open Source technology in the requirements as a comfortable opportunity for system integration in the modern IT world. Alongside this, I found some more general business requirements to the meta-model implementation such as abstraction, simplicity of use, and plug-in architecture without which none of modern information systems is able to work.

In the fourth and fifths chapters, that can be considered as the special part, the author gives the description (specification?) of the system architecture and the XML based transaction management script language. He provides the whole specification of the script language supported by numerous illustrative examples.

In chapter six, the structure of API implementation of the meta-model is given that also demonstrates good knowledge of UML; however, I could only guess that the programming

language for those API was Java. If any other languages were supported, the reporter did not mention.

In this work the author has demonstrated deep awareness at long-live transaction management problems in the modern period and also independency in his conclusions and decisions. The basis was presented correctly and consecutively from the point of logic. Definitely, the reporter demonstrates modern approaches in his solutions and a fresh look at the problem.

The style the material is written is very easy to understand, I underline this particularly because English is not my native language. Successfully combine figurativeness of treatment of the material with laconism in some terms and definitions, the author skillfully accompanies the text with illustrations and quotes of authoritative specialists.

As the most important point I would like to note the usage XML as the script language and the way the author envisaged the architecture.

In general, this work inspires an interest for further scientific research.

There are a few weaknesses in the report, however:

- 1) The Abstract chapter is brought out of the content table and comes first.
- 2) The application sphere needs to be provided in chapter 1.
- 3) The programming language is not specified in the API requirements

Apart from these unessential drawbacks, I consider that the given dissertation is performed at rather a high level and deserves an appropriate grade.

Chief of program developing department,

*System analyst *

*Mikhail Usatchev *

* *

* *

B.) Mark Little – Arjuna Technologies

**From: "Mark Little" <mark.little@arjuna.com>
To: "Justin Spiteri" <justins@waldonet.net.mt>
Subject: Re: Justin Spiteri - Query
Date: 07 November 2005 13:41**

Justin Spiteri wrote:

> Hi Mr. Little,
> I'm Justin, the student who had asked information

> about WS-CAF. My thesis is under way, however i encountered some
> minor problems.
>
> 1.) There seems to be some confusion on the web regarding
> WS-ACID/LRA/BP, and TX-ACID/LRA.BP. I know theres the WS-AT/BA, and
> they're something different, but the reference i found, namely one
> draft of yours regarding WS-CAF's WS-ACID made me think twice. I
> think that they're exactly the same thing, however can you confirm
> please?. Thanks.

WS-TXM was the original name of the specification that contained
WS-ACID/LRA/BP. However, we then split them into separate
specifications, so WS-TXM no longer exists. The specifications are
WS-ACID, WS-LRA and WS-BP.

> 2.) Another small issue, would you kindly point out to me the
> latest version of theWS-CAF please?. I found the 1.0 (July 28, 2003)
> one online.

You need to look at the OASIS WS-CAF committee home page and download
the latest versions of WS-Context, WS-CF, WS-ACID, WS-LRA and WS-BP to
get a complete view of WS-CAF. There is no single download for WS-CAF
any longer.

Mark.

>
> Thanks and best regards,
>
> Justin Spiteri
> IT Year 4 Student
> University of Malta
> Europe

--

Mark Little
Chief Architect
Arjuna Technologies Ltd
www.arjuna.com

From: "Mark Little" <mark.little@arjuna.com>
To: "Justin Spiteri" <justins@waldonet.net.mt>
Subject: Re: Justin Spiteri - University of Malta
Date: 20 October 2005 12:22

Sure, that's not a problem.

Mark.

Justin Spiteri wrote:

> Hi Mark, thanks a lot for the info, you saved me hours of painstaking

> work reading through every possible option available. With the time
> saved, i'm aiming at putting effort in building it as an open source
> project. Just for the records, can i please add you as reference
> source in my thesis?. Thanks again.
>
> Regards,
> Justin
>
> Mark Little wrote:
>
>> Hi Justin.
>>
>> Justin Spiteri wrote:
>>
>>> Hi Mr. Little,
>>> I'm Justin Spiteri, a final year student at the
>>> University of Malta, currently reading for an honours degree in
>>> Information Technology. I have always been interested in
>>> transaction processing and I've chosen to carry out my research
>>> based thesis on this particular subject. During my research I
>>> couldn't help but noticing your name in most of the documents which
>>> I read, ranging from WS-CAF/AT/BA and Oasis BTP specifications, to
>>> your recent JSR156 specification request.
>>>
>>> Specifications apart, my main idea is that of
>>> following JSR95's Activity service specification, and creating an
>>> extended version of JBOSS, which caters for Long Running
>>> Transactions, however, eliminating the CORBA/IIOP based part of
>>> JSR-95, and replacing it with a form of web service based
>>> structure. The application may be similar to your recent
>>> development with ARJUNA, however being not CORBA based. The main
>>> idea is to eliminate corba in favour of a more commonly used web
>>> service based protocol such as XML/SOAP.
>>>
>>
>>
>>
>> You should look at WS-CAF in this case: the basic infrastructure
>> defined by WS-Context/WS-CF is essentially what you have described.
>>
>>>
>>> I would greatly appreciate any comments tips or
>>> suggestion by an experienced person like you, which may especially
>>> help me out in choosing which standards to use for this
>>> application. I've seen various standards, and just can't make out
>>> which one fits best. My main dilemma is the choice between WS-BA,
>>> WS-CAF, or BTP. Thanks in advance for your help, I admire your
>>> works, and I wish you good luck in all your present and future
>>> projects.
>>
>>
>>
>> Don't bother with BTP: it does not fit the requirements you've
>> mentioned. WS-BA does not either, although WS-Coordination is close.
>> However, I believe that the combination of WS-Context and WS-CF (both
>> from WS-CAF), much more closely matches what you've described. So,
>> I'd recommend you take a look at them.

>>
>> Mark.
>>
>>>
>>> Best Regards,
>>>
>>> Justin Spiteri
>>> BSc. IT Year 4 Student
>>> University of Malta
>>>
>>> Tel : +35699856894
>>>
>>> Address:
>>>
>>> 37, St. Augustine Str.
>>> Zejtun ZTN02
>>> Malta, Europe.
>>
>>
>>
>>
>
>

From: "Mark Little" <mark.little@arjuna.com>
To: "Justin Spiteri" <justins@waldonet.net.mt>
Subject: Re: Justin Spiteri - University of Malta
Date: 14 October 2005 11:50

Hi Justin.

Justin Spiteri wrote:

> Hi Mr. Little,
> I'm Justin Spiteri, a final year student at the
> University of Malta, currently reading for an honours degree in
> Information Technology. I have always been interested in transaction
> processing and I've chosen to carry out my research based thesis on
> this particular subject. During my research I couldn't help but
> noticing your name in most of the documents which I read, ranging from
> WS-CAF/AT/BA and Oasis BTP specifications, to your recent JSR156
> specification request.
>
> Specifications apart, my main idea is that of
> following JSR95's Activity service specification, and creating an
> extended version of JBOSS, which caters for Long Running Transactions,
> however, eliminating the CORBA/IOP based part of JSR-95, and
> replacing it with a form of web service based structure. The
> application may be similar to your recent development with ARJUNA,
> however being not CORBA based. The main idea is to eliminate corba in
> favour of a more commonly used web service based protocol such as
> XML/SOAP.

You should look at WS-CAF in this case: the basic infrastructure defined by WS-Context/WS-CF is essentially what you have described.

>
> I would greatly appreciate any comments tips or
> suggestion by an experienced person like you, which may especially
> help me out in choosing which standards to use for this application.
> I've seen various standards, and just can't make out which one fits
> best. My main dilemma is the choice between WS-BA, WS-CAF, or BTP.
> Thanks in advance for your help, I admire your works, and I wish you
> good luck in all your present and future projects.

Don't bother with BTP: it does not fit the requirements you've mentioned. WS-BA does not either, although WS-Coordination is close. However, I believe that the combination of WS-Context and WS-CF (both from WS-CAF), much more closely matches what you've described. So, I'd recommend you take a look at them.

Mark.

>
> Best Regards,
>
> Justin Spiteri
> BSc. IT Year 4 Student
> University of Malta
>
> Tel : +35699856894
>
> Address:
>
> 37, St. Augustine Str.
> Zejtun ZTN02
> Malta, Europe.

C.) Marek Prochazka – Charles University Czech Republic

Subject: Re: Thesis on Extending EJB for Long Lived Transactions - Student from Malta
From: Marek Prochazka <marekproc@yahoo.co.uk>
Date: Fri, 10 Mar 2006 10:08:00 +0000 (GMT)
To: Justin Spiteri <justins@waldonet.net.mt>

Justin,

I've seen pages 4-6 and especially your preliminary proposal, and quickly saw the rest of the document.

>> The question is this, i have researched and reviewed vast amounts of
>> papers, and i came to a personal conclusion that, (agreeing with your
>> ideals), transactional behaviour should not be catered for at
>> deployment

>> time, but rather before implementation of actual units of work. Thus
>> i
>> had in mind to develop a new concept, that of a transactional
>> behaviour
>> descriptor script, which is written by the developer prior to actual
>> coding, and which possibly generates a raw code framework (stubs) for
>>
>> the developer to "fill in". The concept in itself is very simple,
>> and
>> it actually eliminates completely the idea of having a container
>> handling the transactions at runtime (such as JBOSS), however i still
>>
>> have some issues so as to how can the developer possibly be
>> restricted
>> from executing Units of Work, according to their dependencies defined
>> in
>> the script, however this is an implementation issue. What i would
>> really like is your opinion about the general idea. I'm the only
>> student working on this subject at the University of Malta, and i
>> feel
>> quite lost with no guidance. According to you, is the idea doable?.
>>
>> I've arrived at a quite advanced definition stage of the script and
>> the
>> parser, which will be XML based.

I understand in principle where you're heading, but I don't understand any details. O.K., if you going to do something like ConTracts, what's data your UOWs manipulate? Do you have any sketchy example on how it should look like and what are the benefits? Any example of a model based on the metamodel? The metamodel? Benefits?

One more comment: I've seen you list of references. Do you know this:
http://jotm.objectweb.org/TP_related.html
It is now little bit out-dated, but still there is a lot of useful references...

If you explain me the idea of your work more precisely, I can try to evaluate it. But please keep in mind that I finished working on transactions in Summer 2003, and I suspect there has been a lot of progress since then.

Regards,
Marek

Subject: Re: Thesis on Extending EJB for Long Lived Transactions - Student from Malta
From: Marek Prochazka <marekproc@yahoo.co.uk>
Date: Wed, 16 Nov 2005 10:11:43 +0000 (GMT)
To: Justin Spiteri <justins@waldonet.net.mt>

Hello Justin,

>> I'm Justin Spiteri, a 22 year old student from Malta, Europe.
>> I'm currently in my final year for a BSc IT Degree at the
>> University of Malta, and i'm undertaking a thesis entitled
>> "Transaction Models for
>> J2EE". During my research, i came over your Phd Thesis about
>> Bourgogne Transactions, and since i'm looking for a good model
>> with which to
>> extend J2EE to support Long Lived Transactions, i found it
>> extremely
>> interesting. I am seriously considering of extending J2EE with
>> a version of the Bourgogne Transaction Model. My only fear is
>> that the concept may be too complex to implement, since i'm
>> still reading for
>> a BSc Degree, and not a Phd. Would you kindly answer any issues
>> which i may have, if i decide to undertake Bourgogne
>> implementation?. I'm more

it is certainly interesting that you want to implement Bourgogne Transaction for J2EE. BTW, what you mean by "a version" of BT?
As for answering your questions, I'm ready and would be happy to answer eventual questions, but I can't you promise to answer "any issues which you may have". It very much depends 1) on how many and how difficult questions you are going to have and also 2) on my working schedule. I'm employed and do not longer work on transaction nor Java (I used to work for JOTM/ObjectWeb). But as I said, in general I'm willing to answer any meaningful queries.

Apart from wondering which "version" of BT you have in mind, I wonder whether you know that BT were partially implemented for EJB, so that the implementation deals with XA resources/JDBC connections. So, in which sense your new implementation would differ, putting aside that my implementation was incomplete and dealt only with inter-transaction dependencies?

>> concerned about implementation complexities such as, must a
>> specific
>> Application Server such as JBoss be extended? Or would the
>> Bourgogne implementation be completely independent?.

Imagine you have a transaction manager which provides JTA. JBoss or JONAS can for example use JOTM and use exactly the API specified in JTA. Moreover, they provide some interface which is used by JOTM, for example XAResource or JDBC connection.

What you want to do, I guess, is to extend JTA to support long-lived transactions, so that your applications can use it. Let's call this API JTA/BT.

Now, what you have to consider is

1) XAResource or JDBC connections do not support BT. They can't delegate some resources from a transaction to another. They can't share data differently than in the standard ACID way with isolation levels defined. (Also standard relational databases such as Oracle or MySQL

can't do it).

2) The applicaion server will not use BT, as it is designed to use only JTA.

So, my answer is: yes, I think you'd have to modify an application server (or database) to support some of the BT features.

Another, more philosophical issue is as follows: I'm not sure it was actually a good idea to support so generic model for end users. Maybe BT should be used as a metamodel and a more specific model should be built on top of that. Just an idea.

Best regards,
Marek
