A LINEAR LOGIC APPROACH TO RESTFUL WEB SERVICE MODELLING AND COMPOSITION

by

Xia Zhao

A thesis submitted to the University of Bedfordshire in partial fulfilment of the requirements for the degree of Doctor of Philosophy

March 2013

Abstract

RESTful Web Services are gaining increasing attention from both the service and the Web communities. The rising number of services being implemented and made available on the Web is creating a demand for modelling techniques that can abstract REST design from the implementation in order better to specify, analyse and implement large-scale RESTful Web systems. It can also help by providing suitable RESTful Web Service composition methods which can reduce costs by efficiently re-using the large number of services that are already available and by exploiting existing services for complex business purposes.

This research considers RESTful Web Services as state transition systems and proposes a novel Linear Logic based approach, the first of its kind, for both the modelling and the composition of RESTful Web Services. The thesis demonstrates the capabilities of resource-sensitive Linear Logic for modelling five key REST constraints and proposes a two-stage approach to service composition involving Linear Logic theorem proving and proof-as-process based on the π-calculus.

Whereas previous approaches have focused on each aspect of the composition of RESTful Web Services individually (e.g. execution or high-level modelling), this work bridges the gap between abstract formal modelling and application-level execution in an efficient and effective way. The approach not only ensures the completeness and correctness of the resulting composed services but also produces their process models naturally, providing the possibility to translate them into executable business languages.

Furthermore, the research encodes the proposed modelling and composition method into the Coq proof assistant, which enables both the Linear Logic theorem proving and the π-calculus extraction to be conducted semi-automatically. The feasibility and versatility studies performed in two disparate user scenarios (shopping and biomedical service composition) show that the proposed method provides a good level of scalability when the numbers of services and resources grow.

Contents

Abstract **i**

Contents **ii**

Acknowledgements **ix**

Declaration **xi**

List of Figures **xiii**

List of Tables **xv**

List of Publications **xvii**

Acronyms **xix**

Key Concepts **xxi**

1 Introduction **1**

1.1 Motivation 2

1.1.1 RESTful Web Service Modelling 2

1.1.2 RESTful Web Service Composition 3

1.2 Aims and Objectives 4

1.3 Research Questions & Key Assumptions 5

1.4 Proposed Solution 8

1.5 Contributions 9

1.6 Thesis Outline 10

2 Background and Related Work **11**

2.1 REST Architecture Style 11

2.2 Web Services 14

2.3 RESTful Web Services 16
2.4 Modelling RESTful Web Systems 18
2.4.1 UML 18
2.4.2 Formal methods 19
2.4.3 Semantic Web 22
2.4.4 Evaluation of the Modelling Methods 23
2.5 Composing RESTful Web Services 26
2.5.1 Workflow-based Approaches 26
2.5.2 Model-driven Approaches 27
2.5.3 AI Planning Approaches 28
2.5.4 Comparison of Composition Methods 29
2.6 Summary 31

3 Modelling RESTful Web Services In Linear Logic 33
3.1 Linear Logic 33
3.2 Linear Logic for Modelling 37
3.3 Modelling Key Elements of RESTful Web Services 40
3.4 Modelling Corresponding to the REST Constraints 45
3.5 An Example Modelling 49
3.6 Summary 50

4 Composing RESTful Web Services using Linear Logic Theorem Proving 53
4.1 Two-stage Composition 54
4.2 Stage 1: Resource-level Composition 57
4.3 Stage 2: Operation-level Composition 60
4.3.1 Linear Logic with the π-calculus 61
4.3.2 Proof-as-process for RESTful Web Service Composition . 66
4.4 Backward Reasoning in Theorem Proving 69
4.5 Summary 71

5 Tool Supported Composition Validation **73**

5.1 Linear Logic Theorem Provers 73

5.2 Key Coq Syntax 76

5.2.1 Inductive Definitions 76

5.2.2 Assumptions 76

5.2.3 Assertions and Proofs 77

5.2.4 Tactic-sytle Proving 77

5.3 Encoding the Logic-based Approach in Coq 78

5.3.1 Encoding the Intuitionistic Linear Logic (ILL) Formulas 79

5.3.2 Encoding the ILL Sequent Inference Rules 79

5.3.3 Encoding the π-calculus Syntax 82

5.3.4 Encoding the ILL with the π-calculus Sequent Inference Rules 83

5.3.5 Encoding the business constraints and the composition requirements 85

5.4 Summary 87

6 Case Studies **89**

6.1 The E-shopping Scenario 89

6.1.1 Case I: Core Categories of Service 92

6.1.2 Case II: Multiple Services in One Category 106

6.1.3 Case III: Extended Categories of Service 113

6.2 The Biomedical Service Composition Scenario 121

6.3 Summary 129

7 Evaluation **131**

7.1 Performance Evaluation 131

7.2 Comparison with Other Methods 134

7.2.1 Comparison with Other Modelling Methods 134

7.2.2 Comparison with Other Composition Methods 137

7.3 Answers to Research Questions 139

7.4 Limitations 144

7.5 Summary 145

8 Conclusion and Future Work 147

8.1 Thesis Summary and Contributions 147

8.2 Future Work 150

A Encodings in Coq 153

A.1 Encoding Intuitionistic Linear Logic in Coq 153

A.2 Encoding Intuitionistic Linear Logic and the π-caluclus in Coq . 156

B Use case implementations in Coq 161

B.1 E-shopping Scenario - Case I Resource Level Implementation in Coq 161

B.2 E-shopping Scenario - Case I Service Method Level Implementation in Coq 163

B.3 E-shopping Scenario - Case II Resource Level Implementation in Coq 168

B.4 E-shopping Scenario - Case II Service Method Level Implementation in Coq 173

B.5 E-shopping Scenario - Case III Resource Level Implementation in Coq 189

B.6 E-shopping Scenario - Case III Service Method Level Implementation in Coq 192

B.7 Biomedical Scenario - Resource Level Implementation in Coq . . 201

B.8 Biomedical Scenario - Service Method Level Implementation in Coq 206

C Proof Trees 212

C.1 E-shopping Scenario Case I at the Second Stage 212

C.1.1 212

C.2 E-shopping Scenario Case II at the First Stage 214

C.3 E-shopping Scenario Case II at the Second Stage 214

C.4 E-shopping Scenario Case III at the First Stage 216

C.5 E-shopping Scenario Case III at the Second Stage 217

C.6 Biomedical Scenario at the First Stage - Surface Decimation Applied Once 220

C.7 Biomedical Scenario at the First Stage - Surface Decimation Applied Twice 221

C.8 Biomedical Scenario at the Second Stage 223

References **226**

Acknowledgements

First of all, I would like to express my deep and sincere gratitude to my supervisor Dr Enjie Liu for her encouragement, discussions and fantastic supervision throughout my PhD research work. Her continuous guidance enabled me to complete my research work successfully.

My special thanks go to my co-supervisor Prof Gordon Clapworthy who accompanied in my everyday work and proof-read substantial parts of this thesis. Without his kind and invaluable support, the work presented in this thesis would not have been possible.

I wish to thank all researchers in the Centre for Computer Graphics and Visualisation (CCGV) for the valuable support and pleasant working environment. Additional thanks go to my colleagues within the Institute for Research in Applicable Computing and the Department of Computer Science and Technology who have supported me in many ways.

I would like to express my deepest emotional thanks to my parents and brother for their constant support and love throughout my studies.

Above all, I thank my husband, Hongqing, for his continuing emotional and practical support over the recent years. Last but not least, I thank our son, Maxwell, for all the joy he brought to our life.

Declaration

I declare that this thesis is my own unaided work. It is being submitted for the degree of Doctor of Philosophy at the University of Bedfordshire.

It has not been submitted before for any degree or examination in any other University.

Name of candidate: Xia Zhao Signature:

Date: March 2013

List of Figures

1.1 Key research components 7

2.1 An example of RESTful Web System. 14

3.1 Inference rules of Intuitionistic Linear Logic. 36

3.2 An overview of RESTful Web Services. 41

3.3 Key components discussed in Chapter 3. 51

4.1 The first stage composition 58

4.2 The second-stage composition 61

4.3 Inference rules of ILL with the π-calculus attachments. 64

4.4 Key components discussed in Chapter 4. 71

5.1 Key components discussed in Chapter 5. 87

6.1 The flow of core categories of service. 93

6.2 Case I flow at the abstract resource level 94

6.3 Case I resource level Linear Logic proof 96

6.4 Case I flow at the service method level 99

6.5 Case II flow at the abstract resource level 107

6.6 Case II flow at the service method level 110

6.7 Case III flow at the abstract resource level 116

6.8 Case III flow at the service method level 118

6.9 The overall flow of the services in the biomedical scenario 122

6.10 The abstract resource flow in the biomedical scenario 124

6.11 Key components discussed in Chapter 6 129

7.1 Number of service resource groups vs. lines of Coq code. 132
7.2 Number of resource choices in one resource groups vs. lines of Coq code. 133

List of Tables

2.1 Pros and cons of the Representational State Transfer (REST) constraints in the Web system development. 12
2.2 Summary of RESTful Web system modelling approaches. 24
2.3 Summary of RESTful Web Service composition approaches. . . 29

3.1 Linear Logic connectives used in the thesis. 35
3.2 Amazon S3 Bucket RESTful Web Services. 49

5.1 The π-calculus syntax encoded in Coq. 82
5.2 Syntax of Linear Logic with the π-calculus attachment encoded in Coq. 84

6.1 Services and resources in core categories of service in the e-shopping scenario. 91
6.2 Multiple payment services and resources in Case II of the e-shopping scenario. 105
6.3 Extended services and resources in Case III of the e-shopping scenario. 114

7.1 Summary of the Linear Logic approach for RESTful Web system modelling. 135
7.2 Comparison of the proposed logical method with other RESTful Web Service composition approaches. 137

List of Publications

1. H. Q. Yu, X. Zhao, S. Reiff-Marganiec, and J. Domingue, Linked context: A linked data approach to personalised service provisioning, in Proc of the 19th International Conference on Web Services (ICWS), 2012.

2. X. Zhao, E. Liu, G. J. Clapworthy, M. Viceconti, and D. Testi. SOA-based digital library services and composition in biomedical applications. Computer Methods and Programs in Biomedicine, 106(3):219-233, 2012.

3. X. Zhao, E. Liu, G. J. Clapworthy, N. Ye, and Y. Lu. RESTful Web Service composition: Extracting a process model from Linear Logic theorem proving. In Proc of 7th International Conference on Next Generation Web Services Practices (NWeSP 2011), pages 398-403. IEEE, 2011.

4. X. Zhao, E. Liu, and G. J. Clapworthy. A two-stage RESTful Web Service composition method based on Linear Logic. In Proc of 9th IEEE European Conference on Web Services (ECOWS 2011), pages 39-46. IEEE, 2011.

5. H. Wei, E. Liu, X. Zhao, N. McFarlane, and G. Clapworthy, Web-based 3d visualisation for biomedical applications, in Proc of the 15th International Conference on Information Visualisation (IV), 2011, pages. 632-637.

6. N. Ye, Y. Zhao, X. Zhao, E. Liu, and G. Clapworthy, A linked data-driven solution to user modeling in a multi-application environment, in Proc of the IEEE 2nd International Conference on Software Engineering and Service Science (ICSESS), 2011, pages. 872-876.

7. X. Zhao, T. Wang, E. Liu, and G. Clapworthy, Web services in distributed information systems: Availability, performance and composition, International Journal of Distributed Systems and Technologies (IJDST), 1(1):1-16, 2010.

Acronyms

AI Artificial Intelligence.

Amazon S3 Amazon Simple Storage Service.

APIs Application Programming Interfaces.

BPEL Business Process Execution Language.

CORBA Common Object Request Broker Architecture.

CT Computed Tomography.

DICOM Digital Imaging and Communications in Medicine.

FSM finite-state machine.

HATEOAS hypermedia as the engine of application state.

HTTP Hypertext Transfer Protocol.

ILL Intuitionistic Linear Logic.

JSON JavaScript Object Notation.

LHDL The Living Human Digital Library.

ReLL Resource Linking Language.

REST Representational State Transfer.

RMI	Remote Method Invocation.
RPC	Remote Procedure Call.
S3	Simple Storage Service.
SOAP	Simple Object Access Protocol.
UML	Unified Modelling Language.
URI	Uniform Resource Identifier.
URL	Uniform Resource Locator.
VTK	Visualisation Tool Kit.
W3C	World Wide Web Consortium.
WADL	Web Application Description Language.
WSDL	Web Service Description Language.
WWW	World Wide Web.
XML	eXtensible Markup Language.

Key Concepts

Atomic Service: a service is not decomposable, with whose business function does not break down into smaller components.

Composed Service: a service is created by the composition of a set of existing services that can be atomic or composed.

Coq: an interactive theorem prover which builds on the theory of the calculus of inductive constructions.

Linear Logic: a resource-sensitive logic which is a refinement of classical and intuitionistic logic.

π**-calculus**: a process calculus which allows processes to concurrently communicate through identified channels.

Program synthesis: a software engineering technique which automatically construct a program that provably satisfies a pre-defined high-level specification.

Proof-as-process: a paradigm for interpreting logical expressions into process calculus. The interpretation from Linear Logic to the π-calculus is used in this thesis.

RESTful Web Services: services are implemented with considerations of the REST architecture constraints: client-server, stateless, cache, layered system, code-on-demand and uniform interface.

Chapter 1

Introduction

This thesis discusses RESTful Web Service formalisation in the areas of modelling and composition. Modelling RESTful Web systems is demanding in order to guide the robust system implementation and maximise the benefits of using the REST architecture style. Composing RESTful Web Services is essential in order to save costs by efficiently re-using the large number of available services and exploiting services for complex business purposes.

The thesis investigates a logic-based approach to modelling and composing RESTful Web Services. It analyses the principles of the REST architecture style and the characteristics of RESTful Web Services and proposes resource-sensitive Linear Logic for the modelling task and the inference rule driven Linear Logic theorem proving for searching service composition results. The whole approach is performed semi-automatically in the Coq proof assistant to complete the composition search, obtain the composition results and verify Linear Logic theorem proving.

This introductory chapter explains the motivation that drives this research, the research aims and objectives, the main research questions and assumptions, the overall proposed solution, the key contributions and the organisation of this thesis.

1.1 Motivation

In the last few years, REST has gained increasing attention from the Web, the mobile and the service communities. REST, which stands for the Representation State Transfer, is an architecture style that was firstly introduced by Fielding [1] in his PhD thesis. The REST architecture aims to make the Web into a scalable yet reliable network-based hypermedia system. The World Wide Web (WWW) has evolved with the REST architecture and has become the largest REST system; this rapid and large-scale WWW development has shown the potentials of what REST can bring into a system, such as evolvability, simplicity and performance.

1.1.1 RESTful Web Service Modelling

The REST architecture style regulates distributed system implementations via a set of constraints, such as uniform interface and stateless client-server development. The applications and systems built in the REST style have several potential advantages, including being lightweight, being declarative and providing easy accessibility. Furthermore, the RESTful Web systems usually have better visibility and are easy to scale [1]. These benefits of REST create the demand for modelling techniques that can abstract the REST design from the implementation in order to better specify, analyse and implement large-scale RESTful Web systems.

Although there exist many systems and services that claim to be RESTful systems or RESTful Web Application Programming Interfaces (APIs), many of them do not conform the design constraints recommended by the definition of the REST architecture style in [1], and there are some misinterpretations of the REST constraints in the implementation.

For example, the development of Web Services is taking advantage of the benefits of the REST architecture style, and so-called RESTful Web Services [2]/RESTful Web APIs have become popular on the Web. When RESTful Web Services are being developed, the implementation of the REST architecture style typically uses the Hypertext Transfer Protocol (HTTP). This has led to some wrong assumptions in which services with eXtensible Markup Language (XML) over HTTP are treated as RESTful Web Services. REST has more constraints than HTTP; for instance, the hypermedia as the engine of application state (HATEOAS) constraint is not addressed in HTTP. As a result, this HATEOAS constraint is typically ignored by many RESTful Web Services/APIs. In addition, while HTTP is commonly used in Web Services/APIs, REST itself is not specifically dependent upon any particular communication protocol.

One vital reason for the misinterpretation is believed to be a lack of standard modelling languages with which to define RESTful Web Services [3]. As a consequence, it would be helpful for clear and sufficient models to be provided to assist Web engineers to interpret the RESTful architecture style properly and to design robust RESTful Web systems.

1.1.2 RESTful Web Service Composition

In addition to the general WWW use of REST, RESTful Web Services are important use cases of the REST architecture style. RESTful Web Services have gained more popularity only in recent years in contrast to over a decade of focus on developing and studying the traditional imperative style of Web Services (also known as Big Web Services, Remote Procedure Call (RPC) style Web Services or Simple Object Access Protocol (SOAP) Web Services) [4, 5]. The popularity of the development of RESTful Web Services is illustrated by the number of large service providers, such as Google [6], Amazon [7] and Yahoo [8], that offer most of their services in the RESTful style. However, debates on RESTful Web Services vs. the Big Web Services are still ongoing [9, 10, 11].

In contrast to the imperative style of Web Services, RESTful Web Services view business data and functionalities as identified resources, which brings the immediate advantage of being lightweight. The response of a service is the representation of the resource itself and does not involve extra encapsulation. RESTful Web Services typically use the direct HTTP protocol and have a uniform invocation interface as a result of using the same set of methods. Moreover, there is a standard set of HTTP status codes for understanding the response of the invocation, so RESTful Web Services are easy accessible by clients. Because RESTful Web Services focus on the data and resources themselves, they are claimed to be self-declarative. Service-oriented applications built using declarative, rather than imperative, approaches are more loosely coupled and offer better flexibility and scalability [12].

Service composition is important in the re-usage and exploitation of services for complex business purposes [13]. There are broad studies on composing the traditional imperative style of Web Services [14, 15, 16, 17, 18, 19]. However, being declarative makes RESTful Web Service composition different from the traditional form and introduces new challenges [20]. RESTful Web Services focus on resource exposure and representation, so their composition should integrate individual Web resources to create new resources or applications. In service-oriented research, the focus on composing services has mainly been devoted to conventional operation-oriented services, leaving the area of RESTful Web Services comparatively under-explored. Continued research in RESTful Web Service composition remains crucial to ensuring that the service community can understand and perform service composition.

1.2 Aims and Objectives

The aim of this thesis is to consider both RESTful Web Service modelling and composition from the logic-based point of view. It proposes to apply Linear Logic theorem proving to search and create composed services and to use the embedded π-calculus into Linear Logic inference rules for extracting the composed service in the process calculus.

This logic-based approach will guarantee the completeness and the correctness of the solution. The implementation of the π-calculus embedded Linear Logic in the theorem prover will ease the composition search and enable final composed results to be implementable at the executable level.

The following lists of objectives that have to be met in order to achieve the research aims:

- to analyse the features of RESTful Web services and the challenges in modelling and composing them;
- to investigate a formal method that can efficiently model RESTful Web Services and can automatically compose them according to user requirements; and
- to analyse the method and demonstrate its feasibility and versatility by applying it to different use case scenarios.

1.3 Research Questions and Key Assumptions

This thesis focuses on introducing a logic-based approach for modelling and composing RESTful Web Services. By conducting the research work, the thesis tries to answer the following questions.

- What are RESTful Web Services and why is formalising them necessary?
- What are the current methods for modelling and composing RESTful Web Services and what are their pros and cons?
- Is it feasible to model RESTful Web Services at the logic level and how can this be achieved?
- How can RESTful Web Services be composed by a Linear Logic based approach?
- How does this logical approach compare with other existing modelling and composition approaches?

Firstly, it is important to know what makes RESTful Web Services different from other types of Web Services, such as RPC-style Web Services. This involves analysing the REST architecture style, the features of overall Web

Services and the current Web development. The results of this analysis will allow a better understanding of why RESTful Web Services are gaining popularity in today's Web development and why feasible formalisations of RESTful Web Services are desired.

Secondly, it is essential to know what current formalisation methods have achieved and why other approaches are required. It will not only enable us to gain a clear picture of the research status in RESTul Web Service formalisation especially in the areas of modelling and composition but also allow us to outline future research work in this area.

Thirdly, logics have been used widely for modelling in different domains, so it is valuable to investigate whether a logic-based approach can bring benefits to the RESTful Web Service modelling and composition. The thesis tries to answer this key question with a set of sub-questions. For example, why is Linear Logic particularly selected for this purpose? How is the composition performed? How can service modelling and composing at the logic level be used in real cases?

Fourthly, in order to evaluate the proposed logical approach, it is important to compare it with other existing modelling and composition approaches. This comparison will not only help users to choose the appropriate modelling and composition method according to their specifications but also provide insight to future research in these areas.

In addition, a number of key assumptions have to be made in order to ensure that the thesis focuses on the research aim and research questions outlined.

1) Prior to the service composition, it is assumed that all available services are discovered and ready for access from a centralised repository. If no resources or services are matched during composition searching, the required services are assumed to be unavailable.

2) It is assumed that all specified business constraints/actions among the services are stored in a centralised repository which can be accessed during composition solution searching.

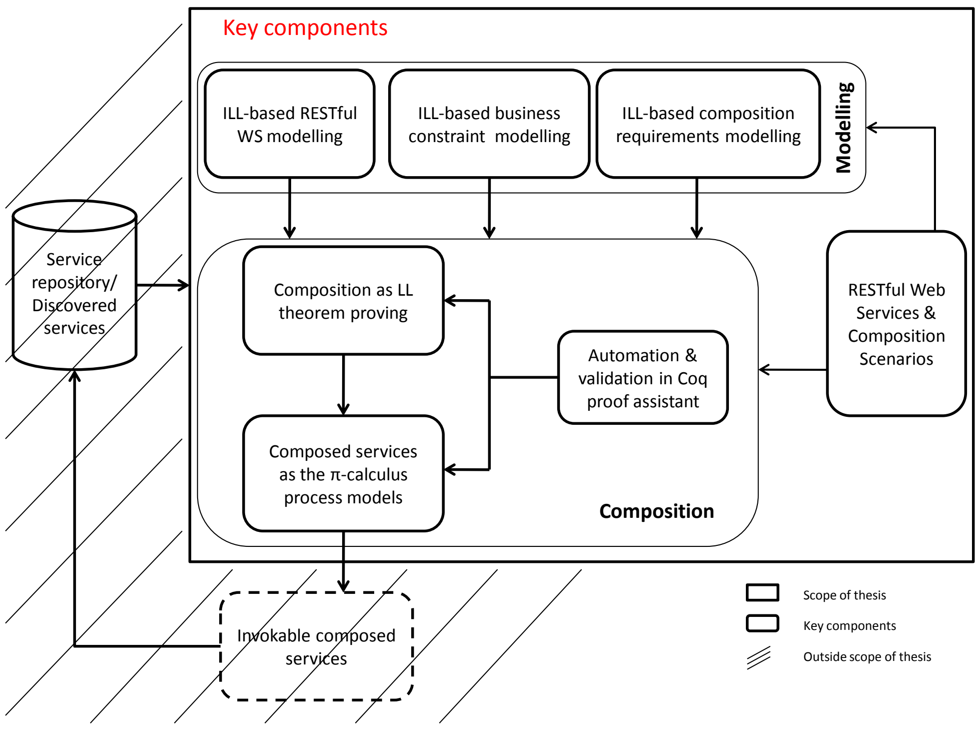
Key components
ILL-based RESTful WS modelling
ILL-based business constraint modelling
ILL-based composition requirements modelling
Modelling
Service repository/ Discovered services
Composition as LL theorem proving
Automation & validation in Coq proof assistant
RESTful Web Services & Composition Scenarios
Composed services as the π-calculus process models
Composition
Invokable composed services
Scope of thesis
Key components
Outside scope of thesis

Figure 1.1: Key research components in Linear Logic for RESTful Web Service Modelling and Composition.

3) It is assumed that the data resources and their categories have been defined by unified semantics according to the business models within the scenarios; there are no conflicting concepts among them once the business models are defined.

4) It is assumed that RESTful Web Services mentioned in this thesis are built on HTTP with four standard methods: GET, POST, PUT and DELETE. All these methods are used according to the original semantics, which was also pointed out as an important necessity when developing RESTful Web Services [21]. Therefore, GET is for retrieving the representation of a resource, POST is for creating a new resource, PUT is for updating a resource and DELETE is for removing a resource.

1.4 Proposed Solution

This thesis proposes Linear Logic as the base formalism by which to model RESTful Web Services and plan their composition. Figure 1.1 presents the key components in this research, in which LL stands for Linear Logic and ILL stands for Intuitionistic Linear Logic.

The proposed approach is achieved from the following considerations. The detailed descriptions of this approach are discussed in Chapter 3, 4 and 5.

- Because the representational state transfer feature of the REST architecture style, RESTful Web Service systems are viewed as state transition systems in which the resource representations are transferred to form the new application states.

- Linear Logic, which can explicitly model state transition systems, is chosen as a formal model for RESTful Web Services. The proposed service modelling techniques can be extended to the modelling of general RESTful systems.

- RESTful Web Service composition is performed via Linear Logic theorem proving and a two-stage composition method is proposed in order to improve the efficiency of proof searching.

- Based on the proof-as-process paradigm and the close relationship between Linear Logic and the π-calculus, the π-calculus is employed as the formalism for representing the composition result at the second composition stage.

- The modelling is encoded in a programming language and the theorem proving is completed semi-automatically in the Coq proof assistant.

The approach will be further demonstrated with real-world user scenarios and evaluated by analysing it scalability and comparing with other existing modelling and composition methods.

1.5 Contributions

The main contributions of this research are listed as follows. The detailed evaluation of these contributions is discussed in Chapter 7.

- Development of the first logic-based approach for modelling and composing RESTful Web Services (Chapter 3 and 4). As one type of formal methods, the logic-based approach can retain the simplicity of the models and representations. Furthermore, Linear Logic is particularly selected due to its explicit resource computation feature, its ability to represent state systems and its close relationship to the process calculus.

- Application of the combination of propositional Intuitionistic Linear Logic and the π-calculus for composing RESTful Web Services (Chapter 4). The key feature of this approach is combining two formalisms together that not only use propositional Linear Logic to guarantee the completeness and the correctness of the composed services but also use the π-calculus to create the process model for the composition result.

- An implementation of RESTful Web Service composition in Coq (Chapter 5). This allows the verification of the theorem proving and extraction of the process model. The use of Coq is motivated by the ease of mechanising logics as a one tier system. The type system supported by Coq also allows the fine specification of the π-calculus. And the graphic CoqIDE tool provides a user-friendly way to use the Coq proof assistant.

- Feasibility and versatility demonstration of the Linear Logic based approach for composing RESTful Web Services (Chapter 6). Two use scenarios in different domains (i.e. e-commerce and bioinformatics) are thoroughly studied and implemented in Coq. Feasibility is also demonstrated by scalability performance measurements of compositions with the large number of services and business specifications (Chapter 7).

In addition, earlier versions of several parts of this thesis have been published and presented in international conferences. These include discussions of the Linear Logic modelling at the IEEE ECOWS 2011 [22] and extraction of the process models at the IEEE NWeSP 2011 [23]. Some other publications, which are closely related to this research but not directly addressed in this thesis, can also be found in the publication list.

1.6 Thesis Outline

The thesis is organised as follows.

Chapter 2 provides the background of this research and an overview of the literature on RESTful Web Service modelling and composition, in which two surveys are conducted.

Chapter 3 discusses the reasons of choosing Linear Logic as the formalism for modelling and composing RESTful Web Services and how Linear Logic is used to model RESTful Web Services, in which RESTful Web systems are viewed as state transition systems.

Chapter 4 presents a two-stage method to compose RESTful Web Services and extract process models for the composition results. The π-calculus terms are attached to the Linear Logic inference rules, and the Linear Logic theorem proving is used for searching the composition results.

Chapter 5 implements the Linear Logic modelling and theorem proving in the Coq theorem prover, which greatly eases the theorem-proving process and double validate the proof.

Chapter 6 demonstrates the feasibility and versatility of this Linear Logic based approach to RESTful Web Service composition through four case studies in two user scenarios.

Chapter 7 evaluates the proposed Linear Logic based modelling and composition method and compares it to other approaches mentioned in the literature review.

Chapter 8 summarises the contribution of this thesis and proposes future research work.

Chapter 2

Background and Related Work

This chapter provides the background and the literature review for this thesis. The key scope of this thesis is RESTful Web Services which covers the concepts of the REST architecture style and Web Services.

The aim of this thesis is to formally model and compose RESTful Web Services. Thus, this chapter firstly demonstrates the core features of RESTful Web Services, and then discusses the importance of modelling and composing them. It then looks separately at existing methods in the areas of modelling and composing RESTful Web Services.

2.1 Representational State Transfer Architecture Style

REST was introduced as an architecture style for distributed hypermedia systems at the turn of the Century. It has been widely applied to the current World Wide Web and enables the Web to continue to expand and evolve for the future. The REST architecture style is regulated by six software engineering constraints [1]: client-server, stateless, cache, layered system, code-on-demand and uniform interface. With these constraints, the REST architecture style enables distributed hypermedia systems to be loosely-coupled, scalable, portable and reliable. Table 2.1 summarises the pros and cons of the above constraints

Table 2.1: Pros and cons of the REST constraints in the Web system development.

Constraints	Pros	Cons
Client-Sever	Improved portability of client interfaces and scalability of server components.	Increased network congestion.
Stateless	Increased visibility, reliability and scalability.	Decreased network performance; Reduction of the server's control on consistent application behavior.
Cache	Improved network efficiency	Decreased reliability
Layered System	Reduced complexity	Increased overhead and latency
Code-On-Demand	Improved extensibility	Reduced visibility
Uniform Interface	Improved simplicity and visibility	Decreased efficiency

with regard to the implementation criteria, such as portability, visibility, scalability, reliability and efficiency.

The *client-server* constraint separates the responsibilities of the client-side components and the server-side components; clients are not concerned with data storage, while servers are not concerned with user states. This separation maximizes the client-side portability and the server-side scalability. RESTful Web systems generally have two types of client-side agent: one is the human user agent and the other is the machine agent.

The *stateless* constraint requires that each request from the client to the server must contain all of the information necessary to fulfil it - the session state is kept only at the client side. This weakens the coupling between the client and the server.

The *cache* constraint allows the response to be defined as cacheable or non-cacheable, which reduces the number of network requests and improves the system performance.

The *layered system* constraint allows additional middle layers to be added between the client and the server. These layers separate the functions into hierarchies, thus increasing the scalability and flexibility of the system.

Code-on-demand is the only optional constraint and it allows the client to download and run the server-side code as scripts or applets, thus providing the possibility of enhanced functionality by consideration of context settings, such as firewall-prevented communications.

The *uniform interface* constraint is guided by multiple sub-constraints including identification of resources (e.g. Uniform Resource Identifier (URI)), manipulation of resources through representations, self-descriptive messages and Hypermedia as the Engine of Application State (HATEOAS). The uniform interface constraint simplifies the system architecture and improves the semantic understanding of the interactions. The HATEOAS constraint ensures that the systems interact entirely through hypermedia provided dynamically by the servers - the client needs to know only the entry address of a resource; it then follows the hyperlinks among the media to find the representations of other resources. It further decouples the client and the server, so both sides have the ability to evolve independently.

REST itself does not bind to any particular network protocols. However, because of the close matching to HTTP, REST together with HTTP has been widely used in today's WWW development.

The evolving RESTful systems can be viewed as state transition systems because of its representation state transfer feature, in which states are expressed as resource representations with links indicating transitions. Client agents manipulate resources at one state through their representations which contain links used by clients to reach the further desired application states.

For example, see Figure 2.1, the BBC website defines a media publication resource, and the client can access it through its Uniform Resource Locator (URL) (*http://bbc.co.uk*). The representation of the resource is returned to the client as a Web page in this case, and this representation places the client into a state *S1*. Links to further resources, such as News, Sport and Weather, are contained in the representation that allows the client to access them through their URLs. The client may obtain the representation of the *news* resource through the news link *http://www.bbc.co.uk/news*, which moves it into another application state, *S2*. Thus, the client application states change with each

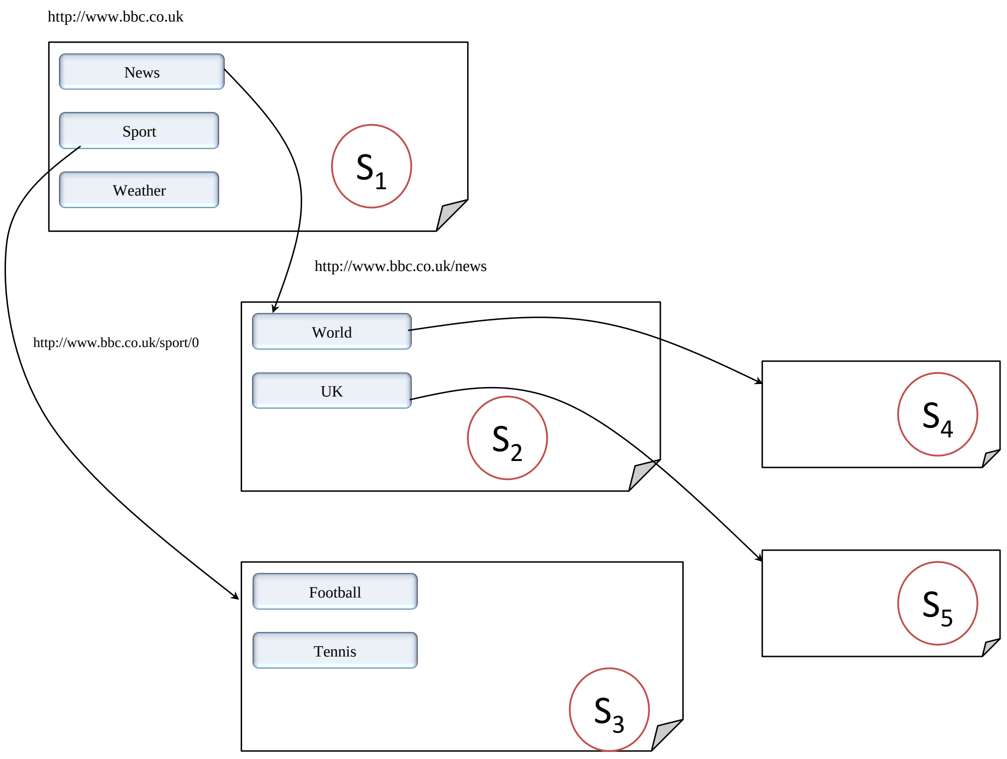

Figure 2.1: An example of RESTful Web system.

resource representation, and this state transfer can be expressed in the state transition system.

2.2 Web Services

The concept of Web Services first emerged around 2000 [24] based on the ideas of RPC, Common Object Request Broker Architecture (CORBA) and Remote Method Invocation (RMI) [25]. The World Wide Web Consortium (W3C) gave a definition of Web Services as [26]:

> "A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards."

Furthermore, W3C classified Web Services into two types [26]. One is REST-compliant Web Services in which Web resources are manipulated by XML representations through a set of uniform stateless methods. The other is arbitrary Web Services in which a variety of methods may be exposed.

Web Services that are built on the Web Service Description Language (WSDL) and SOAP technologies are nowadays referred to Big Web Services [2, 9]. This type of Web Services views the Web as the universal transport for communicating messages. The implementation of the services is operation-oriented and there are no regulations on the definitions of the operations. These services are typically seen as exposing internal functions through service endpoints (such as via WSDL) and transferring SOAP messages for service communications.

Although REST has been mentioned for Web Services, this original Web Service definition identifies the key supporting technologies for Web Services are WSDL, SOAP, XML and HTTP. The early implementations of Web Services have been largely demonstrated by enterprise and organisations exposing their mainframe functions into network-accessible services and publishing WSDL files as service access endpoints. The emergence of Amazon Web Services [7] in 2002 is one examples.

Web Services that are developed to adhere to the REST constraints are known as RESTful Web Services rather than REST-compliant services. Considering the W3C's Web Services definition and the characters of the REST architecture, the REST-compliant services do not exactly follow the REST architecture principle. This type of service is nowadays treated as hybrid [2], in which services use HTTP as the envelope format but do not use uniform access for the methods.

As a result of the development of Web 2.0, the implementation of real RESTful Web Services becomes popular. The shift from traditional operation-oriented Web Services to RESTful Web Services has been exemplified by the actions of Amazon introducing the Simple Storage Service (S3) [27] and Google deprecated the SOAP format search service in 2009 [28]. Although the debate on RESTful Web Services and Big Web Services still continues, RESTful Web Services are gaining increasing attentions due to their advantages of being lightweight, being declarative and providing easy accessibility. However, the majority of the REST movements is in industry. The research on RESTful Web Service formalisation is still under explored.

2.3 RESTful Web Services

RESTful Web Services, also known as RESTful Web APIs, refer to the type of RESTful Web Systems that particularly concentrates on systems with machine client agents. These Web Services are developed according to the REST architecture style and use the HTTP protocol. The key element of the service is a collection of resources, and each resource is identified by its URI. Resources are represented in different Internet media formats, such as JavaScript Object Notation (JSON), Atom and XML, and the change of the resource application state is managed through these representations. The representation of the resource contains links for driving the application states.

RESTful Web Services typically use standard HTTP operation methods (e.g. GET, POST, PUT and DELETE) with response status codes for uniform access. As mentioned in the previous chapter, this thesis assumes that all HTTP methods use the semantics expected, which was pointed out as an important necessity when developing RESTful Web Services [21]. GET is a safe method which can be performed repeatedly to retrieve the current representation of the target resource without any side effect. PUT is an idempotent method for updating the target resource; being idempotent means that it is replayable, so the effect of using PUT for N identical messages is the same as using PUT for one such message. DELETE is used for termination and is also an idempotent method. POST creates new resources and is thus not replayable.

To summarise, RESTful Web Services have the following characteristics.

- Addressable: Any piece of information that is related to a service should be defined as a resource, and this resource should be identified by a URI. Unlike SOAP-style services in which WSDL is used to specify the endpoints of the services, resources are directly accessible through the URI. This addressability feature enables RESTful Web Services to be extensible.

- Connectible: As required in the REST architecture style, resource representations should contain links to other resources, so resources in services are connected through this links. The links provide the next available resources after retrieval of the current resource. This connectivity feature enables RESTful Web Services to be compatible with the existing Web infrastructure and to be discovered by Web crawlers.

- Uniform Interface: Resources are typically defined as nouns, and all resources are manipulated through standard HTTP methods with uniform semantics. Thus, a representation obtained through the GET method is catchable at the client side.

- Stateless: As required in the REST architecture style, communication between service providers and service requesters should be kept stateless. This feature allows multiple requests to be handled simultaneously and facilitates the system scalability.

- Lightweight: Requests to RESTful Web Services are submitted directly through HTTP protocols without using any extra encapsulations for the messages, and responses are well represented in common Internet media formats without involving any extra encapsulations as well. Compared to SOAP-style services, which rely on SOAP envelopes to communicate messages, it is easier to consume RESTful Web Services, and the sizes of the messages communicated by RESTful Web Services are smaller.

- Declarative: RESTful Web Services view the services from the perspective of resources rather than the operating methods. Because they have a uniform interface, RESTful Web Services focus on describing the resources themselves, so the resources are more loosely-coupled and easier to be understood by the requesting clients.

2.4 Modelling RESTful Web Systems

Models provide a standardised format for analysing a problem and offer a systematic approach to problem solving. They can guide the development process and serve as consistent tools for system evaluation. RESTful Web Services are becoming popular in implementation and development. However, as mentioned in Chapter 1, the lack of formal modelling for RESTful Web Services has caused some misinterpretation during service implementation. It would be helpful for clear and sufficient models to be provided to assist Web engineers in interpreting the RESTful architecture style properly and designing robust RESTful Web systems.

Various modelling techniques may be used for different purposes, such as graphic/diagram models for requirement analysis in the visual form, symbolic mathematical models for reliability and performance evaluation, and semantic-based models for addressing system intelligence and interoperability. The remainder of this section will examine several approaches to modelling the complete RESTful Web systems with information on the techniques employed including Unified Modelling Language (UML), formal methods and semantic Web, and then compare them according to the REST constraints.

2.4.1 UML

UML [29, 30] is a typical diagram-based modelling language for software engineering. It uses class diagrams for modelling the static structures of a system, sequence diagrams for showing sequential interactions in a system, and state diagrams for representing abstract behaviours in a system.

To model resources and process interfaces, Porres et al. [31, 32] introduces conceptual UML and behavioural UML, respectively. The conceptual resource models are represented in UML class diagrams, in which each class represents a resource, the attributes of the class are data appearing in the resource representation, and the associations between classes are the links between resources; the starting point of the system is a class in the collection type. The process interfaces are modelled in UML state machines with guards, which aim to address states and transitions. A state is active if and only if the guard on the state is true. Transitions are said to be triggered only by HTTP POST, PUT and DELETE methods and are only enabled when the guard condition is true. This approach also extends the Web Application Description Language (WADL) with pre-conditions and post-conditions for publishing services and providing machine-readable service descriptions.

UML is also used in meta-modelling. Alarcon et al. [33] proposes the Resource Linking Language (ReLL) for describing RESTful services and detailed meta-models in UML class diagrams. The meta-model describes resources, representations, links and link types; as it is aimed at services, it exhibits server-side system characteristics, not those of the client side. ReLL enables machine clients to automatically retrieve Web resources, their domain semantics and the navigation mechanisms. However, this model is a purely static description of RESTful services and does not cover cases in which new resources or identification and access schemes are introduced.

2.4.2 Formal methods

Formal methods address system reliability and correctness by modelling and analyzing systems mathematically. The precision of mathematics helps in reduction of faults in systems and reveals inconsistencies, ambiguities and incompletion at the early stage of system specification. The following discusses four techniques that have been used in the recent literature.

A finite-state machine (FSM) is a mathematical model for designing computer programs. In an FSM, processes can have only a finite number of possible states and transitions. Computation begins at the start state and changes to a new state take place according to the transition function. FSMs are commonly used with model checking for the formal verification of a system.

Zuzak et al. [34] presents a view of RESTful systems as state transition systems. Non-deterministic finite-state machines (FSMs), in which both human and machine client-side agents are active, are used to model them. The research addresses the interactions between components and focuses on the functional properties of the system. In this approach, a RESTful system is modelled as a complete application in a single FSM. The transition functions are divided into client and server components. The model accepts the initial state of the system at startup, and input symbols are generated by the client component, which transforms input symbols into requests and integrates resource representations into the application state. The server component then processes the requests and gives responses.

Petri nets provide a promising graphical and mathematical modelling for distributed systems. A graphical Petri net is presented as a directed bipartite graph, in which nodes represent places and transitions, and tokens occupy places. A transition may fire when each of its input places has the required tokens. When a transition fires, all tokens from its input places are removed, after which, tokens are inserted into all of its output places.

When the original Petri nets are used, excessive proliferation of elements in the graph becomes a problem if the system being described is complex. To overcome this, high-level Petri nets were developed, which incorporate some high-level concepts, such as the use of complex data structures as tokens. Two existing approaches use high-level Petri nets to model RESTful Web systems.

Decker et al. [35] expresses RESTful process execution using the service net, a special class of Petri nets supporting value passing. The approach targets the composition of the RESTful processes and is akin to the traditional Business Process Execution Language (BPEL)/SOAP approach to process enactment. Tokens carry XML data that are consumed in, and produced by, communication transitions. Transitions are represented as URI passing, and URIs are considered to be of two types: static ports, which are independent of any particular process instance, and dynamic ports, which refer to exactly one activity process. The notation of dynamic ports, which take input tokens to generate new URIs, is important in realizing URI passing in RESTful systems.

Li and Chou [3] illustrates a REST chart model based on the Colored Petri Net topology, an extension to Petri nets that distinguishes tokens by colour. In the REST chart model, representations are divided into type representations and resource representations, with type representations being connected by transitions. A type representation is modelled as a Coloured Petri Net place that can have tokens denoting the resource representations of that type. The HATEOAS constraint is enforced by the original Petri Net transition firing rules. In RESTful systems, they are explained as: a state transition can be fired only when all of its input type representations have the correct resource representations, and after the firing, the input resource representations are consumed, and appropriate resource representations are created for the output type representations. The approach models the stateless constraint by introducing the idempotent transition, which is a transition performed by idempotent methods (i.e. GET, PUT, DELETE), and the stationary place, in which all hyperlinks in the representation are available.

The π-calculus, described as processes concurrently communicating through identified channels or ports, is seen as a powerful formalism to describe concurrency models and it is a foundation of business process management. It uses the concept of names to describe terms such as communication channels, links, and so on. An important characteristic is the mobility that allows processes to communicate with each other by exchanging messages through named channels that can also be sent over the names and be received by processes.

This π-calculus approach shows great promise for modelling many REST constraints including resources, representations, media types and HATEOAS. Indeed, the π-calculus has been mentioned by several groups of researchers [34, 35] as a good candidate for modelling RESTful systems.

In particular, Hernández and Garcíaone [36] models RESTful semantic Web Services using the π-calculus together with tuple space computing. The model is bound to the HTTP protocol. According to it, a semantic RESTful system is formalized as a set of processes with associated triple spaces that receive messages through URIs assigned. The approach models the exchange of Web resources containing resource identifiers between clients and servers as named channels being exchanged.

2.4.3 Semantic Web

The emergence of Semantic Web aims to transform the current Web, whose unstructured or semi-structured contents are understandable only by humans, into a machine-understandable Web with structured information. This would enable machines to perform more “human-like” tasks, such as discovering and combining information on the Web. Ontologies are commonly used in the Semantic Web context and play an important role in defining the concepts of the resources on the Web. Ontology-based annotations are often used to make machine-understandable service descriptions.

Zhao and Doshi [37] proposes an ontology-based approach to describing RESTful Web Services at the conceptual level. It classifies services into three types: Resource Set Service, Individual Resource Service and Transitional Service. Each type of service has a common definition, such as name, URI, descriptions, resource and HTTP methods. The Resource Set Service supports all four methods: GET, PUT, DELETE and POST, The Individual Resource Service does not support the POST method, and the Transitional Service supports only the POST method. This approach uses Transitional Services to enable the system state changes and proposes the use of situation calculus for automatically composing services based on these conceptual models.

hRESTs [38] provides machine-readable descriptions for RESTful Web Services and APIs by annotating them based on a simple service model. Although this research discusses the HATEOAS constraint, its model of RESTful Web Services still closely follows conventional thinking about the development of operational services (e.g. RPC-based services). Thus, services are still viewed as a set of operations rather than resources, and it addresses the input/output messages of the operations more than the representation and metadata of the resources. Consequently, it is not strictly compared to the other methods.

2.4.4 Evaluation of the Modelling Methods

The above modelling methods are summarised particularly with respect to the constraints in the REST architecture style. Table 2.2 shows results.

Client-server constraint

Two UML modelling approaches and the ontology-based approach discussed above focus mostly on the resources residing on the server, so they do not clearly show how requests are submitted from the client side. All other approaches address the client-server constraint in some way. Client agents are mostly seen as request providers and initialize the system transitions, while servers are request processors and deliver responses to clients.

Stateless constraint

The ReLL, service nets, the π-calculus and ontology approaches do not explicitly discuss the stateless constraint in their models. All other approaches including Porres's UML, FSM and REST Chart, which are based on the concept of state transition systems, address this constraint within their own models, but the understanding of the state is mixed with session state, representation state, application state and resource state.

Cache constraint

Cache is a non-functional constraint that is included in the REST architecture style in order to improve network efficiency. None of modelling techniques explicitly address this constraint, so it is omitted from Table 2.2.

Layered-system constraint

Layered system is the other non-functional constraint; it is included to make the REST architecture style more scalable. None of the approaches have addressed this constraint so far, so it is omitted from Table 2.2.

Table 2.2: Summary of RESTful Web system modelling approaches.

	Client-Server	Stateless	Code on Demand	Identification of resources	Resource manipulation via representations	Self-descriptive messages	HATEOAS
Porres et al. [31, 32]	Not explicit	State machine with guards	No	Classes in class diagram	Attributes in class diagram	No, extend WADL	Collection as entry point; Links as class diagram associations
ReLL [33]	Not explicit	Not explicit	No	Meta model as a class in the class diagram	Meta model as a class in the class diagram	Link, Link Type, Protocol, Representation in class diagram	Link and Link Type in class diagram
FSM [34]	Client: input symbol generator Server: request processor	Application state stored on client side	ε-transition	Resource identifier model	Representation model	Models of Links, LinkTypes, Representation, MediaTypeIdentifier	Transition function; Initial state defined for entry point
Service nets [35]	Send and receive transitions	Not explicit	Not explicit	Static and dynamic ports	XML tokens	Not explicit	URI passing with Petri net transition rules
REST chart [3]	Client: provide input tokens, fire transitions	Idemponent transition and stationary place	Not explicit	Hyperlinks in the type representation	Input and output places of a transition	Not explicit	Hypermedia dependent Petri net transition rules
π-calculus [36]	Request and response messages	Not explicit	Not explicit	URIs as name channels; HTTP resources as processes	triples in the triple space	Not explicit	Link passing
Ontology [37]	Not explicit	Not explicit	Not explicit	URI definition	Conceptual model in ontology	Type definition in ontology	Rule based transition

Code-on-demand constraint

FSM is the only approach that explicitly addresses the code-on-demand constraint by using the ε-transition in FSMs. However, this constraint is optional in the REST architecture style, and systems do not necessarily violate the RESTful design if they do not obey it.

Uniform interface constraint

All approaches represent the uniform interface constraint to some extent. The HATEOAS principle has been agreed by all approaches as the key for developing scalable RESTful systems. It has been modelled typically as link passing and state transitions in these existing approaches. The identification of resource and the representation principles are modelled more statically in the UML and ontology approaches. The π-calculus shows the dynamic creation of resources through the new resource induction.

It is still debatable whether RESTful systems should have description files, or not. According to the original principle of self-descriptive messages, RESTful systems should be capable of being understood by the message processors within their own system models. Porres's UML approach supports the provision of separate description files for the system by extending WADL. Other approaches - ReLL, FSM, and ontology - tend to detail models with representations, links, link types and media types in order to allow the client and server processors to understand the request and response messages. Petri nets and the π-calculus approaches do not explicitly address this principle.

It is also noted that, although the REST architecture style is not bound to any particular transport protocol, all the published descriptions of the various modelling approaches target the HTTP protocol. It is claimed that the FSM and ReLL approaches are not restricted to HTTP.

2.5 Composing RESTful Web Services

Service composition offers value-added dimensions to Web Services. The composition techniques allow service consumers to solve complex problems by reusing and combining existing services. The full cycle of service composition normally includes service discovery, service selection and service composition. This thesis assumes that all available services have been discovered in some way and focuses on the last stage of how services are combined to fulfil the business requirements.

With the increasing number of services available on the Web, feasible composition techniques are more demanding. RESTful Web Services focus on resource exposure and representation, so their composition should integrate individual Web resources to create new resources or applications. In service-oriented research, the focus in composing services has mainly been devoted to conventional operation-oriented services [14, 15, 16, 17, 18, 19], leaving the area of RESTful Web Services comparatively underexplored. Continued research in RESTful Web Service composition remains crucial to ensuring that the service community can understand and perform service composition. This section provides an overview of the recent notable work into three categories and compare them by the selected composition criteria.

2.5.1 Workflow-based Approaches

If the complex collaboration among services is viewed and implemented from a workflow perspective, Web Service composition is generally regarded as similar to workflow generation. The typical way to achieve service composition using a workflow is to program the executable workflow directly. Currently, BPEL [39] is the most common technique for specifying the interactions among services. However, BPEL was originally designed for process-oriented services, so workflow-based approaches to RESTful Web Service composition have to adapt or extend the current BPEL language to make it suitable for resource-oriented features.

Pautasso [40] identifies a set of requirements for RESTful Web Service composition and extends BPEL to accommodate the REST architecture, which aims to enable composition of both traditional Web Services and RESTful Web Services within the same process-oriented service composition language. Moreover, the work also allows publication of BPEL processes as RESTful Web Services. This work is summarised as a BPEL extension for REST that is, so far, the most mature approach for RESTful Web Service composition at the execution level.

Yu et al. [41] adopts the above BPEL extension for REST approach but focuses on the importance of roles in service description and composition. It argues that the emphasis on the impact of roles brings high usability, better security and improved flexibility. The resource meta-model is defined in order to build resources and their relationships to the roles.

Bite [42] is another workflow-based composition model for Web applications. It deals with both data interactions and control flows. The work uses a subset of the existing workflow models with the aim of providing simplicity and a short development cycle. As in the BPEL extension for REST, the composition workflow is published as a composed resource. However, Bite does not support the HTTP PUT method.

These workflow-based approaches provide good support for composing services at the execution level. However, a common issue is that additional correctness verification is required for the composition created by the business process. Furthermore, without a formal definition of RESTful Web Services, it is difficult to achieve automatic service composition.

2.5.2 Model-driven Approaches

In model-driven approaches of RESTful Web Service composition, models are used to describe user requirements, resources and composition processes. They see service composition from the high-level design-time point of view.

Rauf et al. [32] models composite RESTful Web Services with UML and is, so far, the only model-driven approach targeting RESTful Web Service composition. It introduces conceptual and behavioural models to model resources and composition processes, respectively, with conceptual resource models being represented in class diagrams. The RESTful process is a model in an activity diagram, and the composition process is detailed in a state machine diagram. The approach considers the composite service resource as the main resource in the model, and other partner services are navigated from it. The models are said to have a direct mapping to business flow languages such as the BPEL extension for REST; however, no detailed work is given.

2.5.3 AI Planning Approaches

In Artificial Intelligence (AI) planning approaches, service composition is regarded as a search problem using intelligent systems. The fundamental idea is to explore a large service space and produce a plan that can bridge the gap between the initial state (i.e. available services and composition requirements) and the final goal (i.e. composed services).

In Zhao and Doshi's [37] RESTful Web Service modelling approach mentioned in Section 2.4.3, it also proposes the use of situation calculus for automatically composing RESTful Web Services. However, it is arguable that the additional verb-like Transitional Service is not necessary in modelling RESTful Web Services if the URIs of services are meaningful and well defined. For example, the submit-payment service in their example can be achieved by using the POST method on the */payment* service.

Alarcon et al. [43] proposes a hypermedia-driven composition approach based on the previous work on ReLL (mentioned in 2.4.1) and Petri Nets. Because ReLL focuses on the hypermedia characteristics, it allows resources to be annotated explicitly with domain semantics. The Service Net approach introduced in [35] is adopted in this work for composing services, but with extra consideration being given to hypermedia constraints such as authentication and content negotiation.

Table 2.3: Summary of RESTful Web Service composition approaches.

	Automation	Scalability	Execution	Correctness
Pautasso [40]		Average	√	
Yu et al. [41]		Average	√	
Bite [42]		Average	√	
Rauf et al. [32]		Low		
Zhao and Doshi [37]	√	Good		√
Alarcon et al. [43]		Low		√

2.5.4 Comparison of Composition Methods

The thesis selects the following criteria [15, 44] to study the current composition methods; they will also be used later to evaluate the approach proposed. Table 2.3 summarise the existing composition approaches based on these criteria.

Automation

Automation is one of the ultimate goals of service composition. The automation criterion is used to measure the level of automation achieved by different composition approaches including the technology driven and the process of composition.

Workflow-based approaches do not have formal definitions of RESTful Web Services, which is an obstacle for automatic service composition. The model-driven approach based on UML focuses on modelling the composed services at a high level without particular targeting automation.AI-planning approaches greatly facilitate the automation of the composition process through formal techniques. Both the situation calculus and the service net approaches formally define the service composition process with the consideration of state transfer.

Scalability

The composition scalability criterion is used to indicate whether the composition approach is suitable for larger compositions. The complexity and the cost for composition increase at a higher level when the number of services and resources increases, then the composition scalability is low.

The approaches that rely on diagrams, such as UML and Petri-nets, tend to have low scalability as the increasing number of services involved increases, the size of the diagram increases and its legibility becomes lower. Approaches with formal mathematical or logical expressions, such as that based on the situation calculus, tend to have better scalability.

Execution

The execution criterion is used to ensure that a composition approach is not only sound at the design level but also feasible at the implementation level.

Workflow-based approaches put their main effort into defining business workflow languages, generating executable composition processes and publishing them as new services. They provide good support at the development level to allow services not only to be composed but also to be invokable.

In model-driven approaches, the models are typically independent and difficult to use at runtime. Although the resulting models can be transformed into executable composition specifications, no such work has yet been performed in the RESTful Web Service area.

In AI-planning approaches, the dedicated expressions can be transformed into BPEL-like executable languages, but again, no such work is available in the RESTful Web Service area.

This thesis argues that the primary reason for non-existing transformation from the design level to the implementation level is because the executable RESTful Web Service composition language is still under investigation.

Correctness

The correctness criterion is used to assess if the composition behaves as required in various circumstances. When the correctness of a composition approach is verified, the composed service should behave according to composition requirements.

Workflow-based approaches alone lack correctness verification for the composition created from the business processes. Further verification techniques have to be applied in order to ensure the composition correctness. However, none of the current approaches have addressed this aspect.

Model-driven approaches generally can better capture user requirements and present the software architecture. However, the correctness of the model and the dependency of the final implemented application are not assured [45, 46]. Further model checking techniques are required to ensure the models to be not only syntactically correct but also follow the correct semantics.

AI-planning techniques express the services, resources and related requirements in theoretical forms that provide a good foundation for correctness and verification.

2.6 Summary

This chapter has described the scope of this research on modelling and composing RESTful Web Services. Although RESTful Web Services are gaining more popularity in industry, the research work on both modelling and composition is still required.

The chapter has surveyed existing RESTful Web systems modelling methods and noteworthy RESTful Web Service composition approaches. Modelling methods were compared on whether they reflect to the constraints defined by the REST architecture style. Composition approaches were compared using a set of criteria including automation, scalability, execution and correctness.

These literature surveys have shown that despite the enthusiasm of the research community about formalising RESTful Web Services and their composition, research is still required to provide a uniform model that can help Web engineers to develop robust RESTful Web Services.

The following chapters will propose a logic-based approach to modelling and composing RESTful Web Services, which is not present in the current approaches.

Chapter 3

Modelling RESTful Web Services In Linear Logic

The previous chapter has studied the current approaches to modelling RESTful Web Services and highlighted that further research on modelling RESTful Web Services is still required. This chapter proposes, to our knowledge, the first logic-based approach to modelling RESTful Web Services, and Linear Logic is chosen for this purpose.

The chapter begins with a brief review of characteristics of Linear Logic and then discusses its expressiveness with particular focus on the reasons of choosing it for modelling RESTful Web Services and, later, for composing services. It then explains that how the key elements of RESTful Web Services are modelled in Linear Logic, how well this model works according to the six constraints defined for the REST architecture, together with the Amazon Simple Storage Service as an illustration example to model. The chapter finally summarises how the components discussed in this chapter contribute to the overall process described in Chapter 1.

3.1 Linear Logic

This thesis introduces a logical formalism based on Linear Logic [47, 48] to modelling RESTful Web Services. The clear advantage of this approach is that it provides uniform and explicit high-level specifications. Linear Logic was firstly introduced in [47] as a refinement of the classical logic, and later Intuitionistic Linear Logic (ILL) [49] was introduced.

Classical and intuitionistic logic focus only on the truth of a statement, in which there are no restrictions on the hypotheses. The hypotheses can be repeated or ignored. It is assumed that the hypotheses are still usable even though the conclusions have been obtained. These logics are sound in pure mathematics concepts. However, it is difficult to use them to explicitly model the real-world resource consumption, such as the memory consumption of the computer [48].

In Linear Logic, the weakness and the contraction rules supported in classical logic are removed by default. Conclusions have to be achieved by consuming the assumptions as resources. Each resource can be used only once, and two copies of the same resource are treated as distinct. Thus, Linear Logic is also known as the resource-sensitive logic.

Natural deduction in Linear Logic is commonly expressed in sequent calculus [50] form. A sequent is an expression of the form $\Gamma \vdash \Delta$, where Γ and Δ are sequences of formulae, and the sequent turnstile ($\vdash$) separates the assumption on the left from the conclusion on the right. In general, the deduction between the logic assumption and the logic conclusion is written as follows:

$$\texttt{assumption} \vdash \texttt{conclusion}$$

The thesis considers each composition requirement as a single goal, so it uses ILL for RESTful Web Service modelling and composition. ILL is typically written in a sequent form Γ, $\Delta \vdash$ G, where Γ is a set of formulae representing the intuitionistic context, Δ is a multiset of formulae representing the Linear Logic context, and G is a formula representing the goal. The sequent turnstile ($\vdash$) shows the transition between resource consumption and production. The sequent Γ, $\Delta \vdash$ G is described as: given a set of resources Γ, the goal G can be achieved by consuming resources Δ.

Table 3.1 summarises the linear connectives used in this thesis for modelling

Table 3.1: Linear Logic connectives used in the thesis.

Connective	Symbol	Example
Linear Implication	$\multimap$	A $\multimap$ B
Multiplicative Conjunction	$\otimes$	A $\otimes$ B
Additive Conjunction	&	A & B
Additive Disjunction	$\oplus$	A $\oplus$ B

RESTful Web Services. The detailed descriptions are as follows.

Linear Implication ($\multimap$) expresses the possibility of linear deduction. For example, $A \multimap B$ indicates that resource A is consumed, and resource B is produced as a result.

Multiplicative Conjunction ($\otimes$) indicates that both resources coexist. When these resources appear as hypotheses, both resources are available and both of them have to be consumed in order to achieve the goal. In terms of service composition, it shows that two resources are combined for consumption. When these resources appear as the goal, both resources have to be produced in the goal. Thus, $A \otimes B \multimap C$ indicates that resources A and B are both consumed to produce resource C, and $C \multimap A \otimes B$ implies that both resource A and resource B are produced after consuming resource C.

Additive Conjunction (&) also indicates that both resources coexist; however, only one of them is used, and the user has the right to choose which it is. In computer systems, it can be understood as a "human-in-the-loop" choice. When these resources appear as hypotheses, both resources are available but users need to consume only one of them in order to obtain the goal. When these resources appear as the goal, it means that after the hypotheses are consumed, then the one or other goal is achieved, but not both. For example, $A\&B \multimap C$ indicates that one can choose to consume either resource A or resource B, but not both, to obtain resource C, while $C \multimap A\&B$ means that if resource C is consumed, then the user may choose to produce either resource A or B. In the service composition context, Additive Conjunction is able to express human-driven flows.

Additive Disjunction ($\oplus$) indicates that there are two possibilities, but only

$$\frac{}{A \vdash A}(id) \qquad \frac{\Gamma, A, B \vdash G}{\Gamma, B, A \vdash G}(Exchange) \qquad \frac{\Gamma_1 \vdash A \quad \Gamma_2, A \vdash G}{\Gamma_1, \Gamma_2 \vdash G}(Cut)$$

$$\frac{\Gamma_1 \vdash A \quad \Gamma_2, B \vdash G}{\Gamma_1, \Gamma_2, A \multimap B \vdash G}(\multimap L) \qquad \frac{\Gamma, A \vdash G}{\Gamma \vdash A \multimap G}(\multimap R) \qquad \frac{\Gamma \vdash A \multimap G}{\Gamma, A \vdash G}(Shift)$$

$$\frac{\Gamma, A, B \vdash G}{\Gamma, A \otimes B \vdash G}(\otimes L) \qquad \frac{\Gamma_1 \vdash A \quad \Gamma_2 \vdash B}{\Gamma_1, \Gamma_2 \vdash A \otimes B}(\otimes R)$$

$$\frac{\Gamma, A \vdash G}{\Gamma, A \& B \vdash G}(\& L_1) \qquad \frac{\Gamma, B \vdash G}{\Gamma, A \& B \vdash G}(\& L_2) \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \& B}(\& R)$$

$$\frac{\Gamma, A \vdash G \quad \Gamma, B \vdash G}{\Gamma, A \oplus B \vdash G}(\oplus L) \qquad \frac{\Gamma \vdash A}{\Gamma \vdash A \oplus B}(\oplus R_1) \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \oplus B}(\oplus R_2)$$

Figure 3.1: Inference rules of Intuitionistic Linear Logic.

one of them exists, so users do not have a choice. In terms of computer science, Additive Disjunction is mostly seen in the conclusion sequent to show either this or that result is produced. For example, $C \multimap A \oplus B$ indicates that after consuming resource C, either resource A or B is present, but which is present is out of the user's control. In the context of service invocation, it shows that service is invoked with successful or failed responses.

In order to preserve the logic strength, Linear Logic introduces exponentials to express unlimited number of hypotheses. For example, $!A$ means that A can be used as many times as possible. However, these exponentials are not used in this research for the following reasons.

- The provability of Linear Logic with exponentials is not decidable [51], so it will be difficult to show whether the composition theorem is provable or not.

- RESTful Web Services are stateless, although GET, PUT and DELETE methods are idempotent, so each invocation is treated as a new session.

Figure 3.1 lists the key inference rules of ILL, which will be used later in modelling RESTful Web Services and in Chapter 4 for composing services at the resource level.

id: presents the identity corresponding to a data resource in RESTful Web Services.

exchange: enables the arrangement of the data resource consumption.

cut: shows the resource composition flow through resource consumption and production.

$\multimap$***L***: introduces the link between two resources.

shift: shows the initial resource representations with links and changes this initial presentation into a sequent that can be used for future proofs.

$\multimap$***R***: reverses the change made by the *shift* rule.

$\otimes$***L***: generates a new data resource that is the composition of two separate resources.

$\otimes$***R***: generates a new data resource that consists of two resources in parallel.

&***L***: provides a redundant input resource that is not used by the process; however, it can be used in the cut elimination with an external choice.

&***R***: introduces external choices of the resources.

$\oplus$***L***: introduces internal choices of the resources.

$\oplus$***R***: indicates that either resource A or resource B is produced, which is used in the cut elimination with an internal choice.

3.2 Linear Logic for Modelling

Because of its resource-conscious and inference-rule-driven theorem characteristics, Linear Logic has been used to manage a number of problems with resources in different domains. This section begins with brief discussion of the expressive power of Linear Logic in current uses and then explains the reasons of selecting it as the fundamental formalism for modelling and composing RESTful Web Services.

Linear Logic has been used to model concurrent interactions, such as agent dialogues [52], interactions in multi-player games [53], where it helps to find and execute plans between different agents in a distributed network system. The state of an agent is typically modelled as a Linear Logic sequent; the agent finds a proof of the sequent which corresponds to a plan, and once a plan is found, it is executed until either a step fails or the plan is completed successfully. If it is successful, then all available resources are used up and all goals are achieved. When new resources are introduced or the available resources are not used up, the agent re-plans to try and find another solution given the available resources as assumptions.

Rao [54] applies ILL theorem proving to model semantic Web Service composition in a multi-agent environment. It attempts to exploit the expressiveness of Linear Logic to model both functional and non-functional properties of Web Services and focuses on the practical aspects of a multi-agent-based implementation. Rao's work motivates the ongoing research conducted by Papapanagiotou and Fleuriot [55, 56], in which the classical version of Linear Logic is used, and Web Service composition and validation are carried out fully within the Linear Logic theorem prover. Both publication consider the connection between Linear Logic and the π-calculus for extracting the process models of the composition results. However, both of them focus only on the traditional operation-oriented Web Services and do not take into account the resource-oriented RESTful Web Services.

Because of Linear Logic has the key resource-sensitive nature and the ability to present state transition systems explicitly, this thesis also selects it as the foundation for modelling and composing RESTful Web Services. The following further explains the reasons of this choice.

- Linear Logic is typically written in the style of sequent calculus, which provides clear indications of resources to be consumed and resources to be produced. This sequent calculus format enables us to present the REST client-server principle in the style of request and response, which means that once the request from the client component is consumed, the response from the server component is produced.

- The resource-sensitive characteristic of Linear Logic enables us to explicitly express the resource states and the usage of the resources. For example, $A \multimap B$ means that the conclusion resource B is achieved by consuming a single assumption resource A once. If a new conclusion has to be achieved, a new set of assumptions has to be used, again once only. This characteristic can clearly express the stateless feature of REST, in which each request is equipped with all the information required, and every request is treated as a new resource in Linear Logic.

- The Linear Implication ($\multimap$) shows the relationship between the assumption resources and the conclusion resources. It is capable of modelling the hyperlinks between different media used in REST systems. It can explicitly model the links that are produced by navigating from the existing resource representation.

- The Additive Conjunction (&) and the Additive Disjunction ($\oplus$) in Linear Logic can be used to distinguish between internal system choice and external user choice during system state transfer. These two types of choice are parts of common flows in service composition. For example, an internal system choice may be between a successful service invocation and an exception, and an external user choice may be selecting the most suitable service among several candidates. Thus, Linear Logic models RESTful Web systems as non-deterministic state transition systems.

- The Linear Logic inference rules show how representation states change when a given operation is performed. In particular, the dynamic elimination rule (i.e. Cut) and the induction rule (i.e. Linear Implication $\multimap$)

can be used to model the evolution of REST systems. The *cut* elimination rule shows the combination of resources and the *Linear Implication* induction rule shows the new resources being introduced from the existing resources.

- The Linear Logic theorem proving technique driven by its inference rules is widely used in modelling planning [52, 57] in the style of program synthesis whose aim is to automatically conduct a program that can provably satisfy a given high-level specification [58]. This approach not only ensures that the plans are discovered and valid, but also facilitates the automation process. Linear Logic theorem proving is used for generating valid RESTful Web Service composition plans.

- Linear Logic has close connections with process calculus (e.g.the π-calculus) which is the foundation of service composition [59, 60]. The translation between Linear Logic and the π-calculus has been studied by a number of researchers [61, 62, 63]. Moreover, as revealed in Chapter 2, the π-calculus is considered as a powerful model for REST systems. The combination of Linear Logic and the π-calculus would potentially contribute to both modelling and composition of RESTful Web Service. Chapter 4 will discuss this in further detail.

3.3 Modelling Key Elements of RESTful Web Services

This thesis models RESTful Web Services as evolving state transition systems using Intuitionistic Linear Logic (ILL). This section further analyses the key elements of RESTful Web Services as illustrated in Figure 3.2 and discusses how Linear Logic is used in modelling them. In general, RESTful Web Services are viewed in the client-server style, in which the client performs as a service requester and the server provides the actual service functions. The service requester submits requests to the service for processing and receives responses

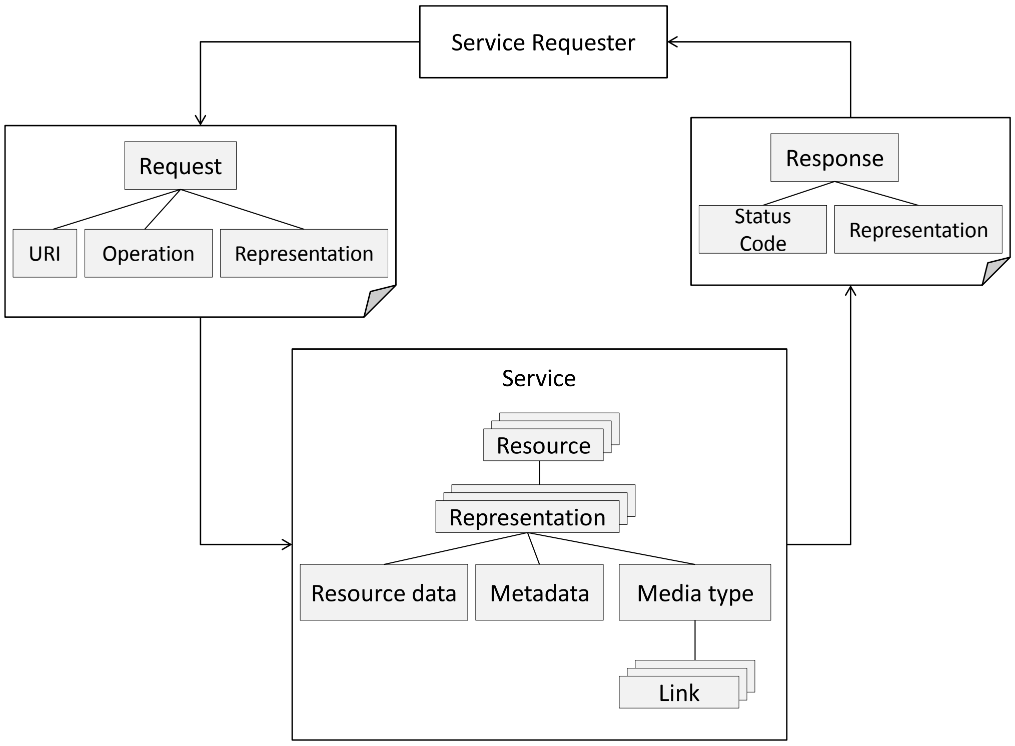

Figure 3.2: An overview of RESTful Web Services with key elements.

once they are ready.

The remainder of this section shows that Linear Logic is capable of modelling these key elements of the REST architecture style [1] in the fragment propositional Intuitionistic Linear Logic.

A **URI** is used as an identifier of a system resource. URIs are modelled as propositions in ILL. There are generally two types of URI: static and dynamic [35]. A static URI is independent of any particular resource instance, and POSTing to these URIs leads to the creation of activity resource instances; for example, *http://shop.example.com/order* is a static URI, which is modelled as an ILL proposition *uriordercollection*. A dynamic URI identifies exactly one activity resource instance and is normally written as a URI template; for example, *http://shop.example.com/order/*{*oid*} is a dynamic URI, which is modelled as an ILL proposition *uriorder*.

A **Resource** is a temporally varying mapping to a set of entities or values [1]. Resources are modelled as propositions in ILL. For example, a user resource is modelled as the ILL proposition *user*, and a book resource is modelled as the ILL proposition *book*.

In RESTful Web Services, actions on a resource are performed by using a representation to capture the current or intended state of that resource and transfer that representation for changing the state of the application. A resource may have a set of representations that may change over time.

A **Representation** is composed of the resource data, the metadata of the resource data and the media type, so it can be written as $Data \otimes Metadata \otimes MediaType$ in Linear Logic. **Metadata** defines the semantic structure of the data and is a key factor affecting the composition outcome. For example, the semantic structure of a User Service used in an online shopping scenario must have the properties for the payment method and the delivery address. Other User Services without this information are insufficient in an online shopping scenario because the lack of the essential payment and shipping functions for invocation. **MediaType** is a finite set of representation media types for RESTful Web Services, such as Atom and JSON.

A **Request** is a resource manipulation request for obtaining the representation and is composed of the URI, the Operation and the optional Representation, written as $Operation \otimes URI \otimes Representation$ in ILL. Here the **Operation** is a standard HTTP method, and this thesis covers the four main methods: GET, PUT, POST and DELETE. With PUT and POST methods, a representation is required to fulfil the request, while the representation is not needed for GET and DELETE methods. For example, a *place order* request in a shopping scenario requires the URI (http://shop.example.com/order), the HTTP POST operation and one representation of the order.

A **Response** is a resource manipulation response containing the response code and a representation, which is written as $Representation \otimes ResponseCode$ in ILL. The standard HTTP status codes are used as response codes in RESTful Web Services. For example, the response to the previous *place order* request is the representation of the order containing a new order (identified by *oid*) and HTTP status code 201, if successful; otherwise, the response contains HTTP status code 400 and an error representation.

Representations typically contain a series of **Links**. These links form the service communication workflow. Each link has a destination URI, link type and link relation, which is expressed as $URI \otimes LinkType \otimes LinkRelation$ in ILL. **LinkType** is defined by the MediaType definition, while **LinkRelation** shows the business-level semantics of a link, which will enable the link relation to be customised by the user. One important link relation is *next*, which indicates the next available service in the workflow. For example, the response representation of the *place order* request may contain two links as shown below: one is actual product item for this order, and the other is a possible next action after placing an order, which is the link to pay this order in this case.

```
<link type="application/atom+xml"
href="http://shop.example.com/order/abc/item"
rel=" http://item.example.com/rels/item"/>

<link type="application/atom+xml"
  href="http://shop.example.com/pay/order/abc"
  rel="next"/>
```

Linear Implication ($\multimap$) in ILL models links to the next available resources. It explicitly expresses the representation state transfer and allows new resources to be introduced dynamically. For example, $order \multimap orderpay$ indicates that the representation of the order resource contains a link to pay this order. Multiple links may be contained in one representation, for example, $order \multimap (orderpay \oplus ordercancel)$ indicates that after obtaining the representation of the order resource, two options (i.e. paying this order or cancelling this order) are available to choose for the next action.

The service invocation creates semantically equivalent representations through response to the client by consuming the service request, so it is expressed as a Linear Logic sequent in the form $Request \vdash Response$. For example, the *place order* invocation can be expressed as:

$$\texttt{URIOrder} \otimes \texttt{POST} \otimes \texttt{RepresentationOrderIn} \vdash$$
$$\texttt{RepresentationOrderOut} \otimes \texttt{ResponseCode}$$

where, URIOrder is the http://shop.example.com/order, POST is the HTTP POST method, *RepresentationOrderIn* is the representation at the request part (see Listing 3.1 for an example) and *RepresentationOrderOut* is the representation at the response part (see Listing 3.2 for an example).

Listing 3.1: An example input representation.

```
POST /order HTTP 1.1
HOST: shop.example.com
Content-Type: application/xml
<order xmlns=http://shop.example.com>
       <customer> ... </customer>
      <item> ... </item>
</order>
```

Listing 3.2: An example output representation.

```
201 Created
Location: http://shop.example.com/order/1234
Content-Type: application/atom
<feed>
<order xmlns=http://shop.example.com>
  <customer> ... </customer>
    <item>
      <cost>
      <link type="application/atom+xml"
          href="http://shop.example.com/order/1234/item"
          rel=" http://item.example.com/rels/item"/>
    </item>
  </order>
</feed>
```

The sequent turnstile ($\vdash$) in ILL is also used to model the representation state transition in RESTful Web Services. For example, the sequent $user, item \vdash order$ means that both user and item resources are consumed, and the order resource can be produced.

The links expressed in the Linear Impliction ($\multimap$) is also translated into state transition. For example, the sequent $\vdash (order \multimap orderpay)$ means that when assuming nothing, the representation of *order* has a next relation link *orderpay*. It is translated into $order \vdash orderpay$ according to the *Shift* rule defined in Figure 3.1, which means that if we assume that *order* is consumed, *orderpay* is produced.

In general, RESTful Web Services are modelled in the style of $Request \vdash Response$ as follows:

$$\texttt{Operation} \otimes \texttt{URI} \otimes \texttt{RepresentationIn} \\ \vdash \texttt{RepresentationOut} \oplus \texttt{ResponseCode}$$

where a request containing an operation, a URI, and an input resource representation is consumed , and a response is produced which contains an output resource representation and a response code. The output representation normally contains the links to the next available requests.

3.4 Modelling Corresponding to the REST Constraints

The proposed Linear Logic approach can address five of six REST constraints discussed in Chapter 2 in some degree except the layered-system constraint. The following presents the detail of how these constraints are modelled in Linear Logic.

Client-server constraint

Linear Logic models the separation of the client and the server in the two-side sequent calculus using turnstile ($\vdash$). The left side of the turnstile is the request from the client component, and the right side is the response from the server component. Thus, the client-server constraint is generally expressed in a pair of request and response as follows:

$$\texttt{request} \vdash \texttt{response}$$

where both request and response can be further detailed according to the models discussed earlier in Section 3.3.

Stateless constraint

As discussed earlier, the Intuitionistic Linear Logic applied in this thesis does not include exponentials, namely of-course (!) and why-not (?), and one main reason to do so is that all resources expressed in Linear Logic can be consumed only once. This naturally matches the stateless constraint in which any request has to be submitted freshly with all information required.

Cache constraint

This thesis will not discuss the mechanism of caching in detail, but explain the possibility of modelling some aspects of caching in Linear Logic. As resources expressed in Linear Logic are consumable, it is feasible to model consumable non-functional properties such as the duration of the cache or the size of the cached data. For example, if the duration of the cache is used to determine whether a new response or a cached response is returned, then the request and response interaction can be modelled as follows in Linear Logic:

$$\texttt{Operation} \otimes \texttt{URI} \otimes \texttt{RepresentationIn} \otimes \texttt{CacheDuration}^{1200} \\ \vdash \texttt{RepresentationOut} \oplus \texttt{Cache}$$

where the superscript (1200) on CacheDuration indicates that how long the cache is stored and Cache is the actual cached content. The detail of how browsers or applications handle the cache is beyond of this thesis, so they are omitted from this abstract expression. However, this model shows that if a request is cacheable, the cache duration should be considered when processing the response. Thus, if no updates are available, the cached content should be returned rather than a new output representation.

Code-on-demand constraint

This constraint is optional in the REST architecture and is generally used in REST Web applications rather than RESTful Web Services, so it will not be used later for service composition. Here, Linear Logic is used to model the code-on-demand constraint only in the context of Web applications. Recall the definition of the code-on-demand constraint in [1], it allows the client to download scripts from the server and then execute them on the client side. Thus, if an execution of a script (s) on the client side changes the application state from A to B, then this change can be modelled using the sequent turnstile ($\vdash$) in Linear Logic as follows:

$$\mathtt{A},\ \mathtt{s} \vdash \mathtt{B}$$

Taking the same example mentioned in the FSM modelling approach [34] for this constraint, if a script (s) changes the color of a hyperlink between red and blue, then it can be modelled as follows:

$$\mathtt{A}_{main\_link\_blue},\ \mathtt{s} \vdash \mathtt{A}_{main\_link\_red}$$
$$\mathtt{A}_{main\_link\_red},\ \mathtt{s} \vdash \mathtt{A}_{main\_link\_blue}$$

Uniform interface constraint

As mentioned in Chapter 2, the uniform interface constraint has four principles: identification of resources, manipulation of resources through representations, self-descriptive messages and HATEOAS.

Identification of resources is supported in the URI model presented earlier in Section 3.3, so it is shown as Linear Logic propositions. The resource identification is explicitly used in service requests and links. For example, the link of paying an order provided earlier has link type as “application/atom+xml”, has link URI as “http://shop.example.com/pay/order/abc” and has link relation as “next”. The link URI can be modelled by an URI as Linear Logic proposition.

Manipulation of resources through representations is supported by explicitly model RESTful resource representations as consumable Linear Logic resources, which are used in the service request and response and contain links for state transitions. For example, to place an order in an online shopping scenario, both input and output representations are essential in the service invocation. The request message performs resource requesting by providing resource URI (URIOrder), the request method (POST) and the request input representation. The response message replies to the request and provides access to the resource by the output representation (RepresentationOrderOut).

Self-descriptive messages is supported by the stateless interaction together with the limited consumable Linear Logic resources for resource representations, media types, links and link types, which are explicitly used in service request and response. For example, the above mentioned response representation of a *place order* request (see Listing 3.2) contains a complete set of information including the media type (application/atom), the representation of the order (within the *feed* tag), the links from the order (indicated by the *link* tag) as well as the type, URI and relation of each link.

HATEOAS is supported by modelling RESTful Web Services as state transition systems using sequent calculus and Linear Implication. The two-side sequent calculus for Linear Logic ($\Gamma \vdash \Delta$) explicitly shows the resource representation transition from one state (on the left of the sequent) to another (on the right of the sequent). The initial state can also be modelled as a sequent with empty left side, such as $\vdash \Delta$, which means that nothing is required in order to transfer to the next state. Linear Implication ($\multimap$) shows the potential links inside each resource representation, which may be invoked to transfer to the next state. For example, $order \multimap orderpay$ indicates that the representation of the order resource contains a link to pay this order.

Table 3.2: Amazon S3 Bucket RESTful Web Services.

URI	Method
BucketName.s3.amazonaws.com	GET, PUT, DELETE
/cors	GET, PUT, DELETE
/lifecycle	GET, PUT, DELETE
/policy	GET, PUT, DELETE
/tagging	GET, PUT, DELETE
/website	GET, PUT, DELETE
/acl	GET, PUT
/location	GET
/logging	GET, PUT
/notification	GET, PUT
/requestPayment	GET, PUT
/versioning	GET, PUT
/versions	GET

3.5 Linear Logic Model for an Example RESTful Web Service

This section demonstrates the generality of the proposed Linear Logic modelling methods by applying it to model the Amazon Simple Storage Service (Amazon S3) Bucket service [27]. More examples and further detail on case studies are discussed in Chapter 6.

Table 3.2 lists the key resources comprising this Web Service. The Amazon S3 Bucket service uses a base URI (*http://BucketName.amazonaws.com*), and other resources are identified by relative addressing.

The base URI (*http://BucketName.amazonaws.com*) of the Bucket service is expressed as an ILL proposition *uris3bucket*. Similarly, other resource URIs are modelled as ILL propositions in the style of *uris3bucketcors*, *uris3bucketlifecycle* and so on.

The abstract resources of these services are modelled as ILL propositions, such as *s3bucket*, *s3bucketcors*, *s3bucketlifecycle* and so on. The representations of these resources are written as *rs3bucket*, *rs3bucketcors*, *rs3bucketlifecycle*, etc. with each representation containing resource data, metadata and media type. The resource data are modelled in the same way as the abstract resources. The media type can be choose from *application/xml* or *application/text*, so it is modelled using an additive conjunction such as *xml* & *text* in ILL.

The request of the resource, for example *BucketName.s3.amazonaws.com*, is written as follows:

$\texttt{uris3bucket} \otimes \texttt{GET} \vdash (\texttt{200} \otimes \texttt{rbucketout}) \oplus \texttt{400}$

$\texttt{uris3bucket} \otimes \texttt{PUT} \otimes \texttt{rbucketin} \vdash (\texttt{200} \otimes \texttt{rbucketout}) \oplus \texttt{400}$

$\texttt{uris3bucket} \otimes \texttt{DELETE} \vdash \texttt{404}$

When the Bucket resource is requested by the GET method, no input representation is required and the response is either successful with output representation and HTTP success status code or failed with HTTP failure status code. When the Bucket resource is requested by the PUT method, an input resource representation is required and the response is in the style similar to that from the GET request. When the Bucket resource is requested by the DELETE method, the resource is removed and the HTTP status code shows no resource available.

3.6 Summary

Research on formalising RESTful Web Services is still under-explored. This chapter proposed the first logic approach to modelling RESTful Web Services in propositional Intuitionistic Linear Logic.

Reflecting the overall approach proposed, Figure 3.3 shows the components discussed in this chapter. Modelling RESTful Web Services in ILL has been particularly discussed, especially in modelling services themselves and the REST constraints. Modelling service composition requirements and business constraints will follow the similar approach, which will be discussed during service composition in the next chapter because their contexts are closely related to the composition process.

Linear Logic is chosen for the formal modelling purpose because of its potential expressiveness, its resource-sensitive characteristics, its capability of

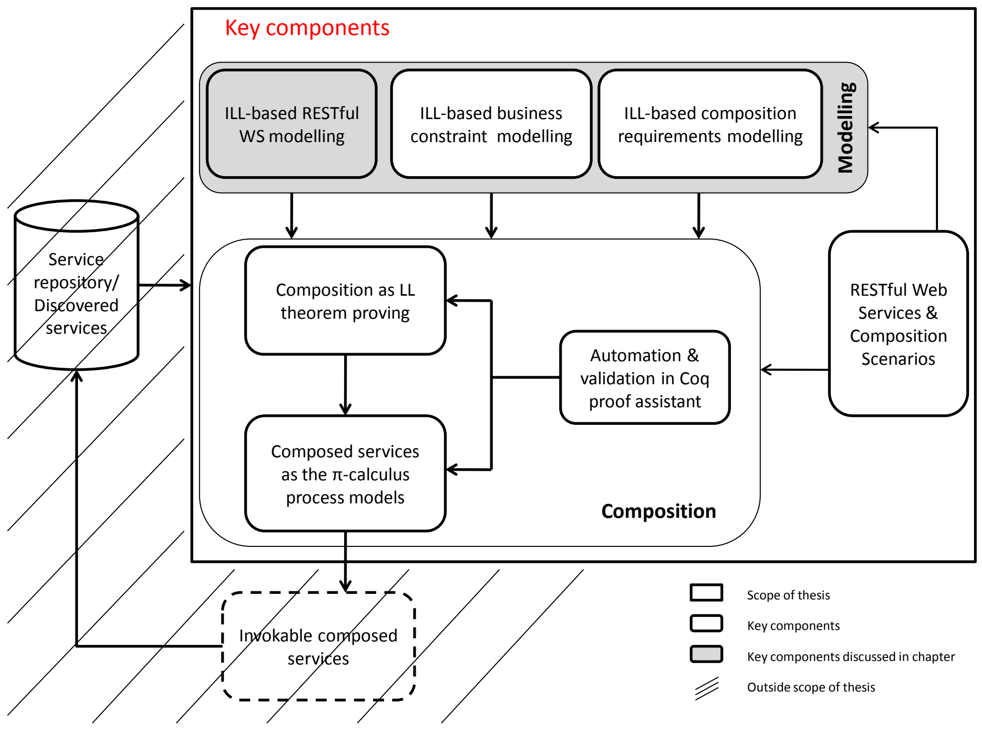
Key components
ILL-based RESTful WS modelling
ILL-based business constraint modelling
ILL-based composition requirements modelling
Modelling
Service repository/ Discovered services
Composition as LL theorem proving
Automation & validation in Coq proof assistant
RESTful Web Services & Composition Scenarios
Composed services as the π-calculus process models
Composition
Invokable composed services
Scope of thesis
Key components
Key components discussed in chapter
Outside scope of thesis

Figure 3.3: Key components discussed in Chapter 3.

explicitly express state transfer in state transition systems and its close relationship with process models. This chapter has shown that Linear Logic is capable of explicitly modelling the following most constraints of REST discussed in Chapter 2 including client-sever, stateless, cache, code-on-demand and uniform interface.

The next chapter will discuss in detail the relationship of Linear Logic and process models and will address the way in which they are used within RESTful Web Service composition.

Chapter 4

Composing RESTful Web Services using Linear Logic Theorem Proving

Although there are some emerging approaches to composing RESTful Web Services as studied in Chapter 2, the research in this area is still under-explored. The previous chapter has proposed and discussed Linear Logic as a suitable formalism for modelling RESTful Web Services. This chapter will continue the focus on the application of Linear Logic but will extend it to the composition of RESTful Web Services.

This chapter will first propose a two-stage RESTful Web Service composition method based on Linear Logic theorem proving and will then discuss each stage in detail. The first stage will concentrate on the abstract resource-level service composition, in which the theorem proving will be based on original ILL inference rules. The second stage will extend the ILL with the proof-as-process paradigm, which will perform at the operation level and allow the composed services to be extracted in process models. The π-calculus is chosen as the formalism for this process model due to its dynamic name-passing ability and its strong capability of modelling RESTful Web Services, as discussed in Chapter 2.

Furthermore, for theorem proving, a backward composition approach is proposed for both the resource and the operation level service composition. The chapter concludes by summarising its key contributions and reflecting on how the components discussed in this chapter contribute to the overall process described in Chapter 1.

4.1 Two-stage Composition based on Linear Logic Theorem Proving

Because of its resource-sensitive characteristic, i.e. assumptions can be consumed during inference, Linear Logic has been considered to provide a flexible approach to goal planning in evolving state transition systems [52, 57].

The proposed two-stage composition of RESTful Web Services specially addresses the way in which the hyperlinks modelled in Chapter 3 can drive the creation of the composition workflows; the composition systems are modelled as planning using Linear Logic, based on [52, 64, 65].

In this thesis, the search for a composed service is treated as a planning process using the propositional ILL theorem-proving. A proof in ILL is viewed as a composition plan for services. Thus, once a Linear Logic proof is found, the composed service is obtained.

Because of the completeness and soundness rules of propositional logic [66], if a solution exists, it will certainly be found, and the correctness of the composition services guarantees that user composition requirements are satisfied. Moreover, as mentioned in the previous chapter, without considering exponentials, such as !, Linear Logic has a firm control on normalization [47], which improves the efficiency of proof searching for composed services.

Linear Logic has the ability to provide natural encoding of notions such as resources, states and events, and it was used in modelling state updates and concurrent computations in complete logic settings [47]. There are extensive studies of the relationship between Linear Logic and process calculus [61, 67], which facilitates the translation of models from the logic framework into executable business process languages. This research embeds the π-calculus process model into the theorem proving process, which enables the result of the proof to be understood at the process level.

The composition is divided into two stages, as illustrated in Algorithm 1 and in Figures 4.1 and 4.2: firstly, a quick abstract resource-level planning is used to check whether the existing resources are sufficient to achieve the target abstract goal resource; secondly, an operation-level planning creates the goal resource with operations by composing existing services.

This two-stage approach improves the overall planning efficiency, especially when a large number of services/resources is available. The quick first-stage planning serves as a filter for checking and gathering the resources necessary for the composition. If the planning fails at this stage, there is no need to investigate the detail of the large number of resources, thus saving time and expense. However, when the number of resources is small, the first stage may be omitted.

Algorithm 1

```
Require: resources $R_1$, $R_2$, ..., $R_n$ as LL Axioms
Require: businessConstraints $BC_1$, $BC_2$, ..., $BC_n$ as LL Hypothesis
Require: compositionRequirement CR as LL theorem
Require: serviceMethod $SM_1$, $SM_2$, ..., $SM_n$ as $\pi$LL Axioms
Require: businessConstraints $BCM_1$, $BCM_2$, ..., $BCM_n$ as $\pi$LL Hypothesis
Require: compositionRequirement CRM as $\pi$LL Theorem
  Begin Stage 1
  prove CR using R and BC in theorem prover;
  if CR is proved then
    Begin Stage 2
    prove CRM using SM and BCM in theorem prover;
    if CRM is proved then
      extract composition process model (CPM) in $\pi$-calculus;
      return CPM in $\pi$-calculus;
    else
      return CRM is not achievable;
    end if
    End Stage 2
  else
    return CR is not achievable;
  end if
  End Stage 1
```

The proposed method for RESTful Web service composition has a number of advantageous characteristics.

- *It facilitates automated service composition.* The whole composition method based on Linear Logic theorem proving follows the program synthesis approach, which is a method used in software engineering to generate programs automatically [68]. The key ideas of correspondence between theorem with constructive proofs and specifications with programs are presented in [69]. In the proposed composition approach, the plan search for the composed service is performed with deductive program synthesis using Linear Logic theorems and proving techniques, which provides foundations for automatically composing services.

- *It ensures the correctness of the resulting composed service.* The service composition process finds a proof that satisfies the composition requirements by applying the business constraints and the available RESTful Web services, the composite service is, in fact, a proof of Linear Logic. Further verification of the composition is not necessary because the proof searching conducted during the propositional Linear Logic theorem proving guarantees that the resulting composed service will meet the defined business specifications. The first-stage proof ensures that the right types of resource exposed by RESTful Web services are available. Furthermore, the second-stage proof guarantees that resources with suitable metadata are available.

- *It reduces the gap between formal service modelling and executable implementation.* The second-stage theorem proving process adopts the proof-to-process paradigm, which bridges smoothly between Linear Logic and the π-calculus. The resulting composed service is ultimately expressed as a process model in the π-calculus. Furthermore, as the π-calculus has good connections with executable business process languages (e,g. BPEL) [70], it is possible to transform the outcome process model into an executable language to complete the composition from the logic level to the execution level.

- *It ensures the achievability of search for the composition.* The fundamental theorem proving process is performed within the propositional ILL. Because of the completeness and soundness rules of propositional Linear Logic, it is certain that all composition solutions that exist will be found.

The trade-off is that the proposed approach develops at the logic level, which may be difficult for Web engineers to understand. However, the transformation from the logic level to the process model level via the proof-to-process paradigm has brought this approach one step closer to the execution level. On one side, the theorem prover combined with the process model methods verifies the composition process and guarantees the correctness of the final composed service. On the other side, it complements the approaches that are closer to the execution level, such as workflow-based methods, which do not have an built-in verification mechanism.

4.2 Stage 1: Resource-level Composition Using Linear Logic Theorem Proving

This section elaborates the first-stage composition process that focuses on the resource level service composition. Figure 4.1 shows the general process of the composition in the style of program synthesis, in which high-level specifications, such as services, business constraints and composition requirements, are firstly expressed in a formal language (i.e. Linear Logic in this thesis), then it tries to prove if the specifications can be satisfied.

Here at the resource level, a RESTful Web Service is specified as a set R of a number of Web resources:

$$R = \{\vdash resource\_i : i = 1..n\}$$

where each resource is expressed as a Linear Logic proposition. For example,

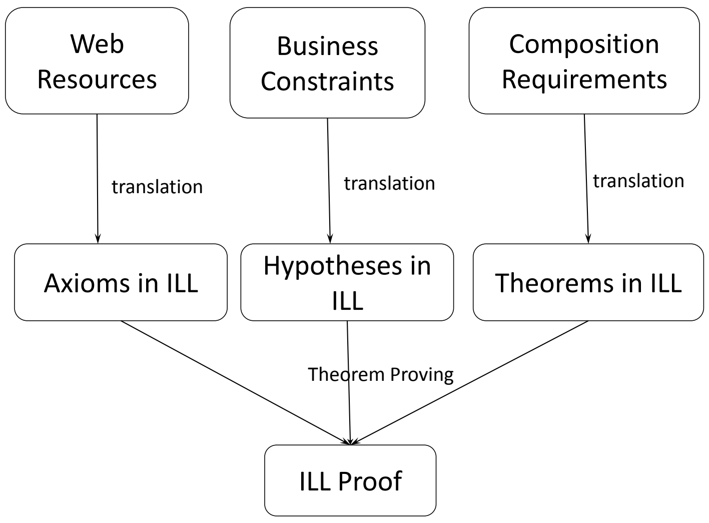

Figure 4.1: The first-stage resource level RESTful Web Service composition.

$\vdash$ *user* is defined as a Linear Logic proposition that indicates a user service resource.

Given the availability of the services, these resources are viewed as unlimited non-linear resources and each of them may be invoked as many times as required. Therefore, the of-course (!) modality may be used to model them to indicate they are unlimited. However, as mentioned in Chapter 3, Linear Logic with modalities such as of-course (!) is undecidable, so this thesis chooses not to use the of-course modality but rather to model all resources as limited resources. Another reason for this is that each service invocation is treated as a fresh one due to the stateless nature of the REST architecture style.

For the composition purpose, the business constraints among the service resources are also modelled in Linear Logic hypotheses in the following form.

$\texttt{constraint\_name: assumptions} \vdash \texttt{conclusion}$

where the left side of the turnstile shows the resources to be consumed and the right side shows the resources to be produced.

Take the e-shopping scenario (see Chapter 6 for the full detail) as an example, the *place order* constraints may be modelled as:

$\texttt{place\_order} : order\_empty, user, item \vdash order\_unpaid$

where the *order_unpaid* resource is produced by consuming resources *order_empty*, *user* and *item*.

The *place_order* Linear Logic model shows that only one user, one item and one empty order are occurred to be consumable and they must be consumed, and after a successful transaction only one unpaid order should be produced. If two items are required to be placed into orders, the quantity of the resource is able to be indicated in the Linear Logic model according to its resource-sensitive characteristic. For simplicity, if one item forms one order, the following model should be used to place two items into orders.

$$\begin{aligned}\texttt{place\_order\_two} : {} & order\_empty, order\_empty, user, item, item \\ & \vdash order\_unpaid, order\_unpaid\end{aligned}$$

Hence, the service composition requirement can be generally expressed as follows:

```
composition_name: existing_resources ⊢ composed_resources
```

For example, if in an e-shopping scenario, the service resources are available: ⊢order_empty, ⊢user, ⊢item, ⊢pay and the above *place_order* and *pay_order* business constraints are defined, the composition requirement in the following theorem is provable:

$$\texttt{comp\_theorem} : order\_empty \otimes user \otimes item \otimes payment \vdash order\_paid$$

As discussed earlier, the Linear Logic theorem proving is used for searching composition solutions. At this abstract resource level, the inference rules introduced in Figure 3.1 are used during theorem proving. Two inference rules (i.e. $\otimes L$ and *cut*) are particularly used for forming and planning composed resources. The $\otimes L$ rule enables the combination of existing resources shown as follows:

$$\frac{\text{resource}_1, \text{resource}_2 \vdash \text{resource}_3}{\text{resource}_1 \otimes \text{resource}_2 \vdash \text{resource}_3}\ (\otimes\text{L})$$

where resource$_1$ and resource$_2$ are composed via the $\otimes L$ rule to produce resource$_3$.

For example, the $\otimes L$ rule can be applied on the above *place_order* business constraint as follows:

$$\frac{\text{order\_empty, user, item} \vdash \text{order\_unpaid}}{\text{order\_empty} \otimes \text{user} \otimes \text{item} \vdash \text{order\_unpaid}} \ (\otimes\text{L})$$

where the $\otimes L$ rule ensures that all three resources (*order_empty*, *user*, *item*) have to be used together, which is a composition of these resources.

The *cut* elimination rule is important to drive the composition from one state to the other by applying the appropriate business constraints shown as follows:

$$\frac{\text{resourceA, resourceB} \vdash \text{resourceX} \qquad \text{resourceX, resourceC} \vdash \text{resourceY}}{\text{resourceA, resourceB, resourceC} \vdash \text{resourceY}} \ (\text{cut})$$

where $resourceY$ is obtained by cut $resourceX$ between $resourceA, resourceB \vdash resourceX$ and $resourceX, resourceC \vdash resourceY$.

Thus, for the e-shopping scenario, if the following business constraint is available:

$\texttt{pay\_order} : order\_unpaid, pay \vdash order\_paid$

then the *cut* rule can be applied as follows:

$$\frac{\text{order\_empty} \otimes \text{user} \otimes \text{item} \vdash \text{order\_unpaid} \qquad \vdash \text{order\_unpaid, pay} \multimap \text{order\_paid}}{\text{order\_empty} \otimes \text{user} \otimes \text{item, pay} \vdash \text{order\_paid}} \ (\text{cut})$$

where *order_unpaid* is eliminated through the *cut* rule.

Once the service composition goal is achievable at the resource level, the composition approach will continue to the operation level (see Section 4.3). On the other hand, if the composition theorem is not provable using the existing resources and business constraints, it means that the composition requirement is not achievable. Chapter 6 will illustrate this method using two complete use case scenarios.

4.3 Stage 2: Operation-level Composition Using the Proof-as-Process Paradigm

This section discusses in detail the second-stage RESTful Web Service composition, which focuses on the operation level. As shown in Figure 4.2, at this stage, RESTful Web Service composition is modelled as more concrete

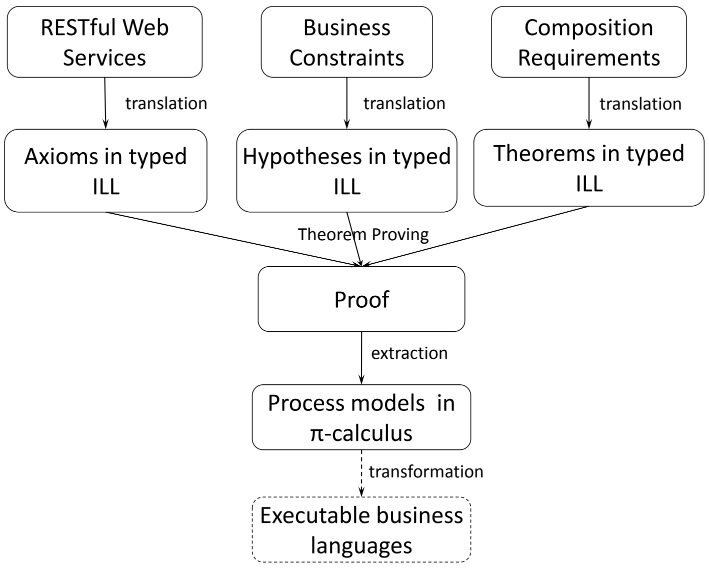

Figure 4.2: The second-stage operation level RESTful Web Service Composition.

operation-level information using Linear Logic planning techniques. The theorem proving at this stage demonstrates that the composition goal can be realised from the given services with given resources, metadata and operation methods by using the business actions in the main inference steps.

Furthermore, the second-stage adopts the proof-as-process paradigm by attaching the π-calculus to the original ILL inference rules (see Figure 3.1); by this, the proved goal can be mapped to the process model in the π-calculus which has a close relationship with executable business process languages [70].

This section begins with an introduction of the proof-as-process paradigm and the close relationship between Linear Logic and the π-calculus. It then discusses how the π-calculus embedded ILL is used in operation-level RESTful Web Service composition. The e-shopping scenario (see Chapter 6 for detail) is used in the explanations.

4.3.1 Linear Logic with the π-calculus

Process models are considered as important formation representations of the resulting composed services as well as the formalism of executable business composition language such as BPEL [59, 60].

Thus, in order to produce process models automatically from the logic proofs, this research adopts the proof-as-process paradigm as presented in [61] and the concurrent interpretation of ILL in [62], with the π-calculus used as the formalism for process models.

The π-calculus [71] is described as processes concurrently communicating through identified channels or ports and uses the concept of names to describe terms such as communication channels, links, and so on. One of its important characteristics is the mobility that allows processes to communicate with each other by exchanging messages through named channels that can also be sent over the names and be received by processes.

The π-calculus is seen as a powerful formalism to describe concurrency models and it is a foundation of business process management [70, 72]. Recently, it has also been considered as the formalism for the resource-oriented architecture [36, 73]. URI link passing in REST is described as mobility in the π-calculus. Messages, which are either request or response, can be sent through named channels (i.e. URIs in RESTful Web services), and processes can send or receive both types of messages. Additionally, (υx)P can be used to construct a new channel x for process P.

The synchronous π-calculus with guarding is used to represent sequential communications. In the grammar, sequencing is indicated by the "." symbol. The grammar of the synchronous π-calculus with guarding is defined as follows.

$$P ::= 0 \mid !P \mid (\upsilon x)P \mid x\langle y\rangle.P \mid x(y).P \mid P|Q \mid P + Q \mid P.Q$$

where lower case x, y are used for names ranging from variables to ports, and upper case P, Q are used for processes. 0 is an inactive action that does not perform anything. The restriction (υx)P defines a name x local to process P. The output prefix x⟨y⟩.P outputs message y at channel x, then behaves like process P. The input prefix x(y).P receives a message z from channel x then behaves like process P with the input message y replaced by the message z. The replication !x(y).P denotes an unlimited number of inputs. The composition P|Q indicates that the two processes P and Q execute in parallel, P.Q that they execute in sequence, and P + Q that either process P or process Q will execute.

The combination of the π-calculus and Linear Logic based on proof-as-process for composing RESTful Web Services makes two distinctive contributions to the whole composition process.

- It provides a process formalism for composed services. Process models have been widely used as a formalism for software/service composition [74, 75]. This formalism helps to verify the correctness of the composed services.

- It bridges the gap between the formal logic design and the real system implementation. The process models, especially the π-calculus, not only have a close relationship with logic formalisms (e.g. Linear Logic and the π-calculus through proof-as-process used in this thesis) but they have also been used to produce executable business languages, such as [76]. Therefore, it is feasible to use the π-calculus as the middle driver to enable RESTful Web Service composition to be achieved from the logical level to the execution level.

The π-calculus descriptions are directly attached to the Linear Logic inference rules in the style of type theory (see Figure 4.3), where actions are modelled as the π-calculus processes with attachments indicated by "::", and plan extraction notations are modelled as the π-calculus names with attachments indicated by ":".

Figure 4.3 lists the key inference rules of ILL with the π-calculus attachment, which are used for composing services at the operation level and extracting the process models for the composition.

id: shows the identity corresponding to a data resource in RESTful Web Services. It requires no action, so the corresponding process is empty as 0.

$$\frac{}{x : A \vdash (\upsilon x)0 :: x : A}(id) \qquad \frac{\Gamma, x : A, y : B \vdash P :: G}{\Gamma, y : B, x : A \vdash P :: G}(Exchange)$$

$$\frac{\Gamma_1 \vdash P :: x : A \quad \Gamma_2, x : A \vdash Q :: G}{\Gamma_1, \Gamma_2 \vdash (\upsilon x)(P|Q) :: G}(Cut)$$

$$\frac{\Gamma_1 \vdash P :: x : A \quad \Gamma_2, y : B \vdash Q :: G}{\Gamma_1, \Gamma_2, y : A \multimap B \vdash (\upsilon x)y\langle x\rangle.(P|Q) :: G}(\multimap L) \qquad \frac{\Gamma \vdash P :: y : A \multimap G}{\Gamma, x : A \vdash P :: y : G}(Shift)$$

$$\frac{\Gamma, x : A \vdash P :: y : B}{\Gamma \vdash y(x).P :: y : A \multimap B}(\multimap R) \qquad \frac{\Gamma, x : A, y : B \vdash P :: G}{\Gamma, z : A \otimes B \vdash y(x).P :: G}(\otimes L)$$

$$\frac{\Gamma_1 \vdash P :: x : A \quad \Gamma_2 \vdash Q :: y : B}{\Gamma_1, \Gamma_2 \vdash (\upsilon x)y\langle x\rangle.(P|Q) :: z : A \otimes B}(\otimes R)$$

$$\frac{\Gamma, x : A \vdash P :: G}{\Gamma, x : A\&B \vdash P :: G}(\&L_1) \qquad \frac{\Gamma, x : B \vdash P :: G}{\Gamma, x : A\&B \vdash P :: G}(\&L_2)$$

$$\frac{\Gamma \vdash P :: x : A \quad \Gamma \vdash Q :: x : B}{\Gamma \vdash x.(P + Q) :: x : A\&B}(\&R)$$

$$\frac{\Gamma, x : A \vdash P :: G \quad \Gamma, x : B \vdash Q :: G}{\Gamma, x : A \oplus B \vdash x.(P + Q) :: G}(\oplus L)$$

$$\frac{\Gamma \vdash P :: x : A}{\Gamma \vdash P :: x : A \oplus B}(\oplus R_1) \qquad \frac{\Gamma \vdash P :: x : B}{\Gamma \vdash P :: x : A \oplus B}(\oplus R_2)$$

Figure 4.3: Inference rules of Intuitionistic Linear Logic with the π-calculus attachments.

exchange: enables the consumption of the data resources to be arranged. The corresponding process remains unchanged.

cut: shows the composition of the resources through consumption and production. It corresponds to a fresh name passing in two parallel processes.

$\multimap$***L***: introduces the link between two resources. The corresponding processes run in parallel after sending out the first channel (x) as a fresh name through the second channel (y).

shift: shows the initial resource representations with links and changes this initial presentation into a sequent that can be used for the future proof. The initial channel has to be indicated and the corresponding process remains unchanged.

$\multimap$***R***: reverses the change made by the shift rule. The corresponding process remains unchanged after the first channel (x) is received through the second channel (y).

$\otimes$***L***: generates a new data resource, which is the composition of two separate resources. The corresponding process remains unchanged after receiving the first channel (x) through the second channel (y).

$\otimes$***R***: generates a new data resource that consists of two resources in parallel. The corresponding processes run in parallel on a composed channel (z) after sending out the first channel (x) as a fresh name through the second channel (y).

&***L***: provides a redundant input resource, which is not used by the process. However, it can be used in cut elimination with an external choice. The corresponding process remains unchanged.

&***R***: introduces external choices of the resources. The corresponding processes will be chosen accordingly and the name is passed in one of them.

$\oplus$***L***: introduces internal choices of the resources. The corresponding processes will be chosen accordingly and the name is passed in one of them.

$\oplus$***R***: indicates that either resource A or resource B is produced, which is used in cut elimination with an internal choice. The corresponding process remains unchanged but with name passing to only one of them.

4.3.2 Proof-as-process for RESTful Web Service Composition

Generally, RESTful Web Service composition at the operation level is achieved through theorem proving performed in successive steps in the form of:

```
Action: State ⊢ State'
```

where *Action* shows how a composition goal can be produced by using an action hypothesis or services resources in the existing non-linear context, and a *State* means the representation state of a data resource in the application system. When one representation state is available with a suitable action, a new representation state can be created.

In Linear Logic, RESTful Web Service composition is modelled particularly as probabilistic planning which enables us to express a set of possible state transitions following an action or a service request. Using Linear Logic disjunctions, probabilistic planning is expressed either as

$$\texttt{Action} : S \vdash S1 \oplus S2 \oplus ... \oplus Sn \ \texttt{or} \ \texttt{Action} : S \vdash S1\&S2\&...\&Sn$$

This means that only one state transition will be made after the action from state S. When this is Additive Disjunction ($\oplus$), the choice is decided by the server side. For example, after a single service invocation, two states may occur, one is success, and the other is an exception. When this is Additive Conjunction (&), the choice is decided by the client side. This is then used as user input from the next state transition. In this research, Additive Conjunction ($\oplus$) is specifically used to model service composition when there is more than one service available for inclusion in the work flow. For example, the order representation created in the place order request may contain a set of links for payment options, such as credit card or debit card payment. These links are modelled as the possible next state transitions.

Actions are further classified into two groups: OperationAction and BusinessAction. *OperationAction* refers to the internal operations (i.e. GET, POST, PUT, DELETE) modelled in Chapter 3, so the service invocation modelled is modified to:

```
Operation: URI, RepresentationIn
                                   ⊢ RepresentationOut ⊕ ResponseCode
```

In the perspective of the π-calculus, the action is viewed as the process that is conducted according to the specifications. In addition, each Linear Logic proposition is attached with a π-calculus name in the convention starting with letter "n" followed by the proposition in this thesis, so the above example is written in the π-calculus attachment format as follows:

```
nURI:URI, nRepresentationIn:RepresentationIn
        ⊢ Operation::nRepresentationOut:RepresentationOut
                ⊕ nResponseCode:ResponseCode
```

Thus, a typical RESTful Web Service with four possible operations - GET, PUT, POST, DELETE - may be modelled as follows:

```
nURI:URI ⊢ GET::nRepresentationOut:RepresentationOut
                ⊕ nResponseCode:ResponseCode


nURI:URI, nRepresentationIn:RepresentationIn
        ⊢ PUT::nRepresentationOut:RepresentationOut
                ⊕ nResponseCode:ResponseCode


nURI:URI, nRepresentationIn:RepresentationIn
        ⊢ POST::nRepresentationOut:RepresentationOut
                ⊕ nResponseCode:ResponseCode


nURI:URI ⊢ DELETE::0
                ⊕ nResponseCode:ResponseCode
```

BusinessAction refers to the external business constraints among the resources that drive the composition workflow to the ultimate goal. These actions are modelled as hyperlinks in the resource representations at the operation level.

For example, in the e-shopping scenario, the representation of the *create order* action may contain the link for paying this order. The client agent can choose different links to drive to different representation states.

```
CreateOrder: URIUser, URIProduct ⊢ URIOrder
LinkPayorder: ⊢ URIOrder ⊸ LinkPayorder
```

With the attachments of the π-calculus name, these business constraints are written as follows:

```
CreateOrder: nURIUser:URIUser, nURIProduct:URIProduct
        ⊢ CreateOrder::nURIOrder:URIOrder
LinkPayorder: ⊢ LinkPayorder::nLinkPayorder:
        (URIOrder ⊸ LinkPayorder)
```

For modelling composition requirements, the existential is introduced to show that there should exist such a process for the possible composed service, so the composition requirement is written as theorem in the following format:

```
Theorem composition_name: ∃ p, initial_services ⊢ composed_service
```

where the meta-variable $\exists p$ will become instantiated to the plan process in the π-calculus format as the proof proceeds. Backward reasoning is used during the theorem proving process, and the following section will explain this in detail.

For example, if a composition requirement, which obtains resource D through resource A and B, is defined as follows:

```
Theorem get_D_AB: ∃ P, a:A, b:B ⊢ P::d:D
```

The existing resources are A, B and C; and the following business constraints are available among them:

```
get_C_AB: a:A, b:B ⊢ GETC::c:C
get_D_C: c:C ⊢ GETD::d:D
```

The proof using the inference rules defined in Figure 4.3 is conducted as follows:

$$\dfrac{\dfrac{}{\text{a:A, b:B} \vdash \text{P}_0\text{::c:C}}\,(\text{get\_C\_AB}) \qquad \dfrac{}{\text{c:C} \vdash \text{Q}_0\text{::d:D}}\,(\text{get\_D\_C})}{\text{a:A, b:B} \vdash \text{P::d:D}}\,(\text{cut})$$

where the proof is performed backward as discussed in the next section, so process P is instantiated into two processes P_0 and Q_0 through the *cut* inference rule.

4.4 Backward Reasoning in Theorem Proving

For both Stages 1 and 2 described in Sections 4.2 and 4.3, the proposed composition method takes the backward reasoning approach during theorem proving. The advantage of doing so is to minimise the search space during proving. Reasoning in forward would cause heavy searching from the large pool of resources with less clues of organising them according to the business constraints. As Intuitionistic Linear Logic is used in the research, the desired composed service is expressed a single goal on the right side with a sequent calculus. It is easier to decompose this single goal into separated resources which can then be checked against existing resources. If all decomposed resources can be matched from the existing ones, it means that the proof is completed and the desired composed service can be achieved.

Depending on the syntax of the goal sequent, the following rules are applied during the decomposing process at both stages.

- If the goal sequent is of the form $A \multimap B$, the $\multimap$-introduction rule (i.e. $\multimap$R in Figure 3.1 or 4.3) is applied, which add A into the linear context and requires that only B is shown as result. No more decomposition is required after that. For example, a goal of form $\Gamma \vdash A \multimap B$ can be decomposed into $\Gamma, A \vdash B$.

- If the goal sequent is of the form $A \otimes B$, the $\otimes$-introduction rule (i.e. $\otimes$R in Figure 3.1 or 4.3) is applied. Meanwhile, the composition context is divided into two separate contexts corresponding to the two sub-goals and the decomposition process continues on both sub-goals. For example, a goal of form $\Gamma \vdash A \otimes B$ can be decomposed into two sub-goals: $\Gamma_1 \vdash A$ and $\Gamma_2 \vdash B$.

- If the goal sequent is of the form $A \oplus B$, both $\oplus$-introduction rules (i.e. $\oplus R_1$ and $\oplus R_2$ in Figure 3.1 or 4.3) can be applied. The decomposition process search for both possibilities. For example, a goal of form $\Gamma \vdash A \oplus B$ can be decomposed into two possibilities: $\Gamma \vdash A$ or $\Gamma \vdash B$, only one of which can be chosen.

- If the goal sequent is of the form $A\&B$, the &-introduction rule (i.e. &R) in Figure 3.1 or 4.3) is applied, which results in two sub-goals. Decomposition continues on both sub-goals. For example, a goal of form $\Gamma \vdash A\&B$ can be decomposed into two sub-goals: $\Gamma \vdash A$ and $\Gamma \vdash B$, only one of which can be chosen.

- If the goal sequent is not made up of the above forms ($\multimap, \otimes, \oplus, \&$), no further decomposition process is required. The backward reasoning method looks from the available services/resources to find one that realised it. If no services/resources in the context realises the given ones, the sub-goal is left to be solved later. If the sub-goal can not be realised at the end of theorem proving, the theorem proving finishes with non-completed proofs, which also means the composition requirements can not be achieved with the available services/resources.

The first stage composition can follow the above rules directly, in which the backward reasoning technique can be performed in the following two major steps: (i) decomposing the conclusion of the goal sequent; and (ii) applying inference rules in Figure 3.1.

At the second stage, the backward reasoning technique not only follows the above rules but also involves typed terms and actions. So a complete reasoning is done in the following three major steps: (i) decomposing the conclusion of the goal sequent; (ii) introducing meta-variables; and (iii) applying inference rules with actions.

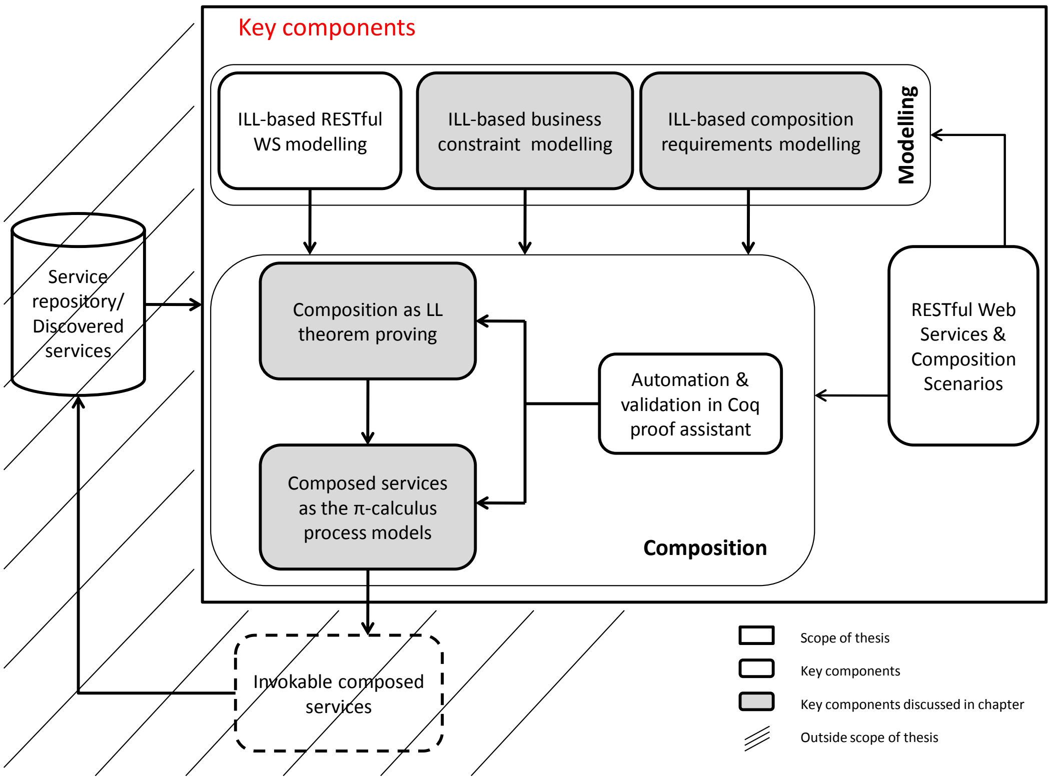
Key components
ILL-based RESTful WS modelling
ILL-based business constraint modelling
ILL-based composition requirements modelling
Modelling
Service repository/ Discovered services
Composition as LL theorem proving
Automation & validation in Coq proof assistant
RESTful Web Services & Composition Scenarios
Composed services as the π-calculus process models
Composition
Invokable composed services
Scope of thesis
Key components
Key components discussed in chapter
Outside scope of thesis

Figure 4.4: Key components discussed in Chapter 4.

When context splitting is required in the second stage as in the goal sequent is of the form $A \otimes B$, new meta-variables have to be introduced to represent terms of different types. For example, a goal of form $\Gamma \vdash z : (A \otimes B)$ can be decomposed into two sub-goals: $\Gamma_1 \vdash z_1 : A$ and $\Gamma_2 \vdash z_2 : B$. During theorem proving, the actions are extracted according to the predefined π-calculus embedded Linear Logic inference rules in Figure 4.3.

4.5 Summary

In response to the study of research in RESTful Web Service composition discussed in Chapter 2, this chapter proposed a two-stage Linear Logic theorem proving based method, which is the first logic-based approach to composing RESTful Web Services.

Reflecting to the overall proposed approach, Figure 4.4 shows the discussed components in this chapter. Following the modelling approach proposed in

Chapter 3, this chapter continues modelling the business constraints/actions and the composition requirements in ILL. The planning method based on the Linear Logic theorem proving is used for searching composed services, within which a backward reasoning technique is proposed to decompose the specification of composition requirements and to ensure that inference rules are properly applied.

One important contribution of the proposed Linear Logic based approach is the way of composing services as planning in the form of deductive program synthesis. It not only facilitates the automation of the service composition process, but also ensures the achievability of search for the composition and the correctness of the composed services produced.

The other key contribution of the proposed approach is the application of the proof-as-process paradigm in Linear Logic combined with the introduction of the π-calculus. This allows the resulting composed services to be automatically expressed in the form of process models, which brings achieving RESTful Web Service composition at the executable level one step closer.

This chapter has demonstrated the feasibility of composing RESTful Web Services based on Linear Logic theorem proving techniques. However, until now all theorem proving has been performed by applying the inference rules manually, which will be impractical when the number of services, resources and business constraints increases. The next chapter aims to automate theorem proving, which will use tool supported proving and validation by encoding the whole Linear Logic theorem proving process in the Coq proof assistant.

Chapter 5

Tool Supported Composition Validation

The previous chapters have described the proposed Linear Logic approach for modelling and planning RESTful Web Service composition. The manual approach to planning as theorem proving has been described in Chapter 4. This chapter encodes the proposed models and plans into the Coq theorem prover which not only facilitates automatic theorem proving but also validates the composition plan.

This chapter provides an overview of the possible theorem provers for Linear Logic and explains the reasons of choosing the Coq proof assistant. It then describes how the encoding of Linear Logic approach is implemented in Coq and concludes by summarising the findings.

5.1 Linear Logic Theorem Provers

Manual theorem proving is sufficient when analysing simple scenarios with only a few number of service resources. Whereas, when the number of resources involved in the proof increases, it becomes infeasible to complete the theorem proving manually. Tool-supported theorem proving is essential in the real-world scenario analysis. This section particularly discusses the available tools for possible Linear Logic theorem proving.

The Linear Logic theorem provers have been implemented in two main forms: one implements Linear Logic directly in logic programming languages such as Lolli [77], Forum [78] or Lygon [79], while the other mechanises Linear Logic in existing proof systems such as Coq [80] or Isabelle [81].

The implementations in logic programming languages aim to achieve high automation in the theorem proving and to improve time efficiency in the proof search. However, users have to code the proof deeply into the logic and understand the details of the theorem prover at the low implementation level.

Llprover [82] is another Linear Logic theorem prover which aims to achieve automated proving as well, but it is not efficient and lacks the capability of adding the π-calculus attachment to the existing implementation.

In contrast, mechanising Linear Logic in existing theorem proof systems allows users to build their theorem proofs quickly and control the proof process interactively in a user-friendly system. By utilising the built-in constructions or tactics in the theorem prover, the automation of the proof search can be achieved to a certain extent. Both Isabelle and Coq can be used to encode Linear Logic and the π-calculus.

Although the provision of tool-supported theorem proving is important in a logic-based approach for RESTful Web Service modelling and composition, this research does not delve into too much detail of implementing a full and powerful Linear Logic theorem prover. Instead, it adopts the approach to mechanising Linear Logic in the existing Coq proof assistant system. This implementation in Coq will perform well in searching and validating composition proofs.

Coq [80] is an interactive theorem prover which provides formal reasoning and extracts certified programs from the proof. Coq is written in OCaml [83] and is based on the Calculus of Inductive Constructions, which combines both higher-order logic and a functional programming language [84] which is rich in types.

Unlike Isabelle, which has a distinct object logic that differs from the system meta logic, Coq uses a single integrated system that allows built-in concepts to exist as an ordinary datatype within its system.

In addition to the command line operation, Coq provides a graphic user interface called CoqIde. The encoding and implementation done in this research are mainly completed in the CoqIde.

There have been previous formalisations of Linear Logic in Coq. Power and Webster [85] provides an encoding of ILL in the Coq proof assistant. However, it does not provide proof terms, so there is no explicit representation of the composition process model. Sadrzadeh [86] studies the feasibility of formalising classical modal Linear Logic with Coq. It provides an encoding that is particularly good in dealing with lists of formulae on both sides of the sequent relation, but no prove terms are involved, which again makes it unsuitable for the purpose of this thesis, as composition process models cannot be extracted. This thesis extends previous formalisations by encoding the π-calculus as representations of prove terms, which allows composition process models to be extracted. The following sections will discuss the detail of the encoding.

The main advantages of formalising Linear Logic theorem proving with the Coq proof assistant are summarised as follows.

- It provides soundness-preserving technology for exploring the automation of planning by developing proof tactics for the theorem prover. Coq has a number of built-in tactics, and users can also define customised tactics; all of these can be applied during the proving process.

- Linear Logic and the π-calculus rules can be integrated into Coq directly without changing Coq itself. Meanwhile, the data types already defined in Coq can be used when encoding Linear Logic and the π-calculus.

- The *cut* rule in Linear Logic can be defined inductively in Coq, which drives the automation of the whole synthesis process for obtaining the composed services.

- Encoding the Linear Logic theorem proving into the Coq proof assistant can further validate the correctness of the theorem proving.

5.2 Key Coq Syntax

This section provides a brief description of the Coq syntax that will be used later in the encoding. The complete Coq document is available in [80].

5.2.1 Inductive Definitions

An inductive definition is normally specified by giving the names and the type of the inductive sets followed by the constructor declarations. If parameters are required for the definition, they are added after the definition name.

The definition of a typical inductive type has the following form:

```
Inductive ident : sort :=
| ident1(param) : type1
... ...
| identn(param) : typen
.
```

where *ident* is the name of the inductively defined type; *sort* is the universe where it lives; *ident*$_1$ to *ident*$_n$ are the names of its constructors and *type*$_1$ to *type*$_n$ their respective types; *param* is the required parameter.

5.2.2 Assumptions

An assumption in Coq binds a definition to a type that can be default Coq types or inductive definitions. Three assumptions are used in this thesis: variables, axioms and hypotheses. Variables are used to introduce terms, axioms are used to introduce the existing services, and hypotheses are used to introduce business constraints.

Assumptions are normally written in the following forms:

```
Variable ident : type.
Axiom ident : type.
Hypothesis ident : type.
```

where *ident* is the name of the assumption and *type* is its type.

5.2.3 Assertions and Proofs

An assertion in Coq states a proposition of which the proof is interactively built using tactics. Two types of assertions are used in this thesis: theorems and lemmas. Theorems are used to represent service composition requirements, and lemmas are used to introduce assistant functions such as adding or removing an empty list. A proof starts by the keyword *Proof* completes by the keyword *Qed*. Thus, assertions and their proofs are generally written in the following forms:

```
Theorem ident : type.
Proof.
tactics
Qed.
```

5.2.4 Tactic-sytle Proving

In Coq, proofs are conducted by a series of tactics that are built-in or customised. Three built-in tactics are primarily used in the proofs in this thesis, which are *apply*, *econstructor* and *instantiate*.

The tactic *apply* tries to match the current goal against the conclusion of the type of term. If it succeeds, then the tactic returns as many subgoals as the number of non dependent premises of the type of term. For example, *apply TimesRight* will try to matched the goal against the result sequent $\Gamma_1, \Gamma_2 \vdash A \otimes B$ in the *TimesRight* inference rule as defined in the Linear Logic inference rules Figure 3.1.

The tactic *econstructor* used in this thesis introduces the existential (e.g. exists P) as a variable. In the Coq syntax, $\ll$ and [are used to introduce names and processes, respectively. The following provides an example of applying *econstructor*, in which the existential P is replaced with a variable *?173* defined by Coq automatically.

Before *econstructor*

```
exists P, ((A<<a)::nil ++ (B<<b)::nil) |- (D<<d[P)
```

After *econstructor*

```
(A << a) :: nil ++ (B << b) :: nil |- (D << d) [?173
```

The *instantiate* tactic allows one to refine an existential variable with a new term, which is normally used after *econstructor*. For example, applying *instantiate (1:= (nu c (par P Q)))* to the above example generates:

```
(A << a) :: nil ++ (B << b) :: nil |- (D << d) [(nu c (par P Q))
```

5.3 Encoding the Logic-based Approach in Coq

In this research, the composition planning is implemented in Coq which can use both built-in and customised tactics to assist theorem proving. The encoded implementation in Coq further validates the composition plans.

More specifically, the implementation is performed at two levels corresponding to the two-stage composition proposed in Chapter 4. The first and the second stage mainly use the rules defined in Figure 3.1 and 4.3, respectively. Thus, firstly, ILL and the inference rules defined in Figure 3.1 are encoded in Coq, and secondly, the π-calculus term attachments in ILL and the inference rules defined in Figure 4.3 are similarly encoded.

The ILL encoding follows the approach of [85, 86] but focuses on the connectives and inference rules used in this thesis. The code is implemented in the latest Coq version (v 8.3). The complete resource level encoding of Linear Logic in the Coq proof assistant is provided in Appendix A.1.

The encoding has been done in five key steps as follows. The next five sub-sections will present them in detail.

(i) Encoding ILL formulae;

(ii) Encoding the ILL sequent inference rules;

(iii) Encoding the π-calculus syntax;

(iv) Encoding the ILL with the π-calculus sequent inference rules;

(v) Encoding the business constraints and the composition requirements for theorem proving.

5.3.1 Encoding the ILL Formulas

The Intuitionistic Linear Logic propositions with connectives (e.g. Linear Implication $\multimap$, Multiplicative Conjuction $\otimes$, Additive Conjunction & and Additive Disjunction $\oplus$) provided in Table 3.1 are defined inductively as *ILinProp* in the *Set* type as follows:

```
Inductive ILinProp : Set :=
| Implies: ILinProp -> ILinProp -> ILinProp
| Times: ILinProp -> ILinProp -> ILinProp
| With: ILinProp -> ILinProp -> ILinProp
| Plus: ILinProp -> ILinProp -> ILinProp
.
```

where *Implies* ($\multimap$) takes two ILL propositions as input and its output is also an ILL proposition. Similar cases are for Times ($\otimes$), With (&) and Plus ($\oplus$).

Resources are then defined as propositions in the *ILinProp* variable type, for example, A and B are defined as linear propositions and Γ is defined as a list of linear propositions:

```
Variable A B : ILinProp.
Variable Γ : list ILinProp.
```

5.3.2 Encoding the ILL Sequent Inference Rules

The Intuitionistic Linear Logic inference rules listed in Figure 3.1 are defined inductively in Coq. The induction is made on the intuitionistic linear sequent relation $\Gamma \vdash G$. The sequent relation *LinCons* is represented as a 2-ary function that takes two arguments as input: the hypothesis Γ and the conclusion G. Γ is implemented as a list of formulas (*list ILinProp*), and G is a single goal formula implemented as the linear proposition (*ILinProp*). The output of the linear sequent relation *LinCons* is defined as a Coq proposition *Prop* that is either true or false. Thus the sequent relation *LinCons* definition is as follows:

```
Inductive LinCons : (list ILinProp) -> ILinProp -> Prop :=
... ...
```

However, formulas, which do not fit into the sequent relation *LinCons* style of *(list ILinProp) →ILinProp →Prop*, will cause errors in the proving process. For example, the sequents in the following format will not satisfy the encoding rules because A and B are not in the type of *list ILinProp* and the default list type Γ is missing in the sequent expression.

$$\frac{A, B \vdash G}{A \otimes B \vdash G}(\otimes L)$$

In order to solve this list problem and facilitate the proving process, two procedures have been introduced. Firstly, all single proposition defined on the left side of the sequent are modified as the list ILinProp type by attach ::nil at the end. For example, A is written as A::nil without the π-calculus attachment or (A≪x)::nil with the π-calculus attachment. Secondly, when the default list type, such as Γ, is missing during the proving processing, a Nil list is added to the front or end of the left side of the sequent. As a result, the above deduction is modified as the following:

$$\frac{Nil, A, B \vdash G}{Nil, A \otimes B \vdash G}(\otimes L)$$

The modifications are achieved through a set of lemmas: *AddNilFront*, *RemoveNilFront* and *AddNilEnd* as shown below.

Listing 5.1: Lemma: AddNilFront.

```
Lemma AddNilFront (A : ILinProp) (Γ : list ILinProp) :
(((nil ++ Γ) |- A) -> (Γ |- A)).
Proof.
intros.
apply H.
Qed.
```

Listing 5.2: Lemma: RemoveNilFront.

```
Lemma RemoveNilFront (A : ILinProp) (Γ : list ILinProp) :
((Γ |- A) -> ((nil ++ Γ) |- A)).
Proof.
intros.
apply H.
Qed.
```

Listing 5.3: Lemma: AddNilEnd.

```
Lemma AddNilEnd (A : ILinProp) (Γ : list ILinProp) :
(((Γ ++ nil) |- A) -> (Γ |- A)).
Proof.
intros.
replace Γ with (Γ ++ nil).
apply H.
elim Γ.
reflexivity.
simpl.
intros.
rewrite H0.
reflexivity.
Qed.
```

The complete encoding can be found in Appendix A.1. Here, the Cut and TimesLeft rules defined in Figure 3.1 are used as examples to explain this encoding in detail.

$$\frac{\Gamma_1 \vdash A \quad \Gamma_2, A \vdash G}{\Gamma_1, \Gamma_2 \vdash G}(Cut) \qquad \frac{\Gamma, A, B \vdash G}{\Gamma, A \otimes B \vdash G}(\otimes L)$$

The *cut* rule is encoded as:

```
Cut (A G : ILinProp)(Γ1 Γ2 : list ILinProp) :
((Γ1 |- A) -> ((Γ2 ++ (A::nil)) |- G) -> ((Γ1 ++ Γ2) |- G))
```

where A, G, $\Gamma\mathit{1}$ and $\Gamma\mathit{2}$ are parameters used in the definition; (Γ1 |- A) and ((Γ2 ++ (A::nil)) |- G) represent two existing sequents $(\Gamma_1 \vdash A)$ and $(\Gamma_2, A \vdash G)$,

Table 5.1: The π-calculus syntax encoded in Coq.

Syntax in π-calculus	Syntax in Coq	Description	Example
0	skip	null process	0
!	bang	replication	!P
υ	nu	restriction/new	υx
$\langle\rangle$	outp	output	x$\langle$y$\rangle$.P
()	inp	receive	x(y).P
|	par	parallel	P | Q
+	sum	choice	P + Q

respectively; ((Γ1 ++ Γ2) \|- G)) represents the concluded sequent $(\Gamma_1, \Gamma_2 \vdash G)$.

Similarly, the *TimesLeft* rule is encoded as follows:

```
TimesLeft (A B G : ILinProp) (Γ : list ILinProp) :
(Γ ++ ((A::nil)++ (B::nil)) |- G -> (Γ ++ ((A ⊗ B)::nil)) |- G)
```

where A, B, G and Γ are parameters used in the definition; (Γ ++ ((A::nil)++ (B::nil)) \|- G) and (Γ ++ ((A ⊗ B)::nil)) \|- G) represent the existing $(\Gamma, A, B \vdash G)$ and the concluded $(\Gamma, A \otimes B \vdash G)$ sequents, respectively.

5.3.3 Encoding the π-calculus Syntax

The key π-calculus syntax used in this thesis:

$$P ::= 0 \;|!P \mid (\upsilon x)P \mid x\langle y\rangle.P \mid x(y).P \mid P|Q \mid P + Q \mid P.Q$$

is encoded as in Listing 5.4, based on [87]. The notation and syntax are given in Table 5.1 for further reference. The full Coq code for Linear Logic with the π-calculus is given in Appendix A.2.

In Listing 5.4, *skip* is for the silent process (0), *bang* is for replication (!), *nu* is for introducing a new name (υ), *outp* is for output ($x\langle y\rangle.P$), *inp* is for receiving ($x(y).P$), *par* is for parallel processes (\|) and *sum* is for different options of process (+). Thus, in the next section inference rule encoding, the process for *nu* is written in the form of *nu name proc*; and similarly *outp name name proc* is for *outp*, *inp name name proc* is for *inp*.

Listing 5.4: Encoding the π-calculus in Coq.

```
Parameter name : Set.
Inductive proc : Set :=
| skip : proc
| bang : proc -> proc
| nu : name -> proc -> proc
| outp : name -> name -> proc -> proc
| inp : name -> name -> proc -> proc
| par : proc -> proc -> proc
| sum : proc -> proc -> proc
.
```

5.3.4 Encoding the ILL with the π-calculus Sequent Inference Rules

Together with the previous Linear Logic encoding, the name and process introduced by the π-calculus are defined as linear propositions AddName and AddProc, respectively, as shown in Listing 5.5. The remainder of the definition for *Implies*, *Times*, *With* and *Plus* is same as previous Linear Logic encoding.

Listing 5.5: Encoding Linear Logic with the π-calculus in Coq.

```
Inductive ILinProp : Set :=
| Implies: ILinProp -> ILinProp -> ILinProp
| Times: ILinProp -> ILinProp -> ILinProp
| With: ILinProp -> ILinProp -> ILinProp
| Plus: ILinProp -> ILinProp -> ILinProp
| AddName: ILinProp -> name -> ILinProp
| AddProc: ILinProp -> proc -> ILinProp
.
```

Table 5.2 summarises the notations and the corresponding syntax in the Coq implementation. The ILL inference rules with the π-calculus term attachments are formalised in Coq. Taking the Cut and TimesLeft rules as examples

Table 5.2: Syntax of Linear Logic with the π-calculus attachment encoded in Coq.

Syntax in LL with π-calculus	Syntax in Coq	Description	Example
$\multimap$	$\multimap$	Implies	A $\multimap$ B
$\otimes$	$\otimes$	Times	A $\otimes$ B
&	&	With	A & B
$\oplus$	$\oplus$	Plus	A $\oplus$ B
:	$\ll$	AddName	$\ll$x
::	[	AddProcess	[P

again, but with the definition containing the π-calculus in Figure 4.3:

$$\frac{\Gamma_1 \vdash P :: x : A \quad \Gamma_2, x : A \vdash Q :: G}{\Gamma_1, \Gamma_2 \vdash (\upsilon x)(P|Q) :: G}(Cut) \qquad \frac{\Gamma, x : A, y : B \vdash P :: G}{\Gamma, z : A \otimes B \vdash y(x).P :: G}(\otimes L)$$

The *cut* rule is encoded as follows:

```
Cut (A G : ILinProp)(Γ1 Γ2 : list ILinProp) (x : name) (P Q : proc):
  (exists P, (Γ1 |- (A<<x[P]))
  -> (exists Q, (Γ2 ++ ((A<<x)::nil)) |- (G[Q))
  -> (Γ1 ++ Γ2) |- (G[(nu x (par P Q)))
```

where A, G, $\Gamma 1$ and $\Gamma 2$ are parameters used in the definition; x is a π-calculus name; P and Q are π-calculus processes; (Γ1 |- (A≪x[P)) and ((Γ2 ++ ((A≪x)::nil)) |- G) represent two existing sequents ($\Gamma_1 \vdash P :: x : A$) and ($\Gamma_2, x : A \vdash$), respectively; ((Γ1 ++ Γ2) |- (G[(nu x (par P Q)))) represents the concluded sequent ($\Gamma_1, \Gamma_2 \vdash (\upsilon x)(P|Q) :: G$).

The *TimesLeft* ($\otimes$L) rule is encoded as follows:

```
TimesLeft (A B G : ILinProp)(Γ : list ILinProp)(x y z : name)
  (P : proc) :
  (exists P, (Γ ++ ((A<<x)::nil)++ ((B<<y)::nil)) |- (G[P))
  -> (Γ ++ (((A ⊗ B)<<z)::nil)) |- (G[(inp y x P))
```

where A, B, G and Γ are parameters used in the definition; x, y and z are π-calculus names; P is a π-calculus processes; (Γ ++ ((A≪x)::nil)++

((B≪y)::nil)) |- G) and (Γ ++ (((A ⊗ B)≪z)::nil)) |- G) represent the existing $(\Gamma, x : A, y : B \vdash P :: G)$ and the concluded $(\Gamma, z : A \otimes B \vdash y(x).P :: G)$ sequents, respectively.

5.3.5 Encoding the business constraints and the composition requirements

The business constraints are defined as hypotheses in Coq, for example, the *place order* constraint in an e-shopping scenario may be defined as follows:

```
Hypothesis place_order : ((item :: nil) ++ (user :: nil)
++ (order_empty :: nil)) |- order_unpaid.
```

The composition requirement is defined as theorem to be proven, for example, the *ship order* composition requirement in an e-shopping scenario may be defined as follows:

```
Theorem shipping_order : ((order_empty ⊗ user ⊗ item ⊗ payment ⊗
 shipment):: nil) |- order_shipped.
```

The tactic-style theorem proving is applied in Coq to conduct proofs. A successful proof shows that the composition requirement is achievable. The detailed example of theorem proving is explained in the next chapter, with complete code provided in Appendix B.

When encoding at the service method level, meta-variables (e.g. exists P) are introduced to represent the processes in the encoding of the inference rules as shown in the above and the composition requirement theorem at the below.

```
Theorem get_D_AB: exists P,
        ((A<<a)::nil ++ (B<<b)::nil) |- (D<<d[P).
```

The meta-variables introduced in the theorem will be gradually instantiated within the plan as the proof proceeds based on the inferences rules defined in Figure 4.3. *Theorem get_D_AB* listed above is used as an example to explain how the meta-variable instantiation is matched to the final process model in the π-calculus. It is assumed that the following is defined for this example:

```
Variable A B C D : ILinProp.
Variable P Q GETC GETD: proc.
Variable a b c d : name.
Hypothesis get_C_AB: ((A<<a)::nil ++ (B<<b)::nil) |- (C<<c[GETC).
Hypothesis get_D_C: (C<<c)::nil |- (D<<d[GETD).
```

The following tactics are applied to prove the theorem:

```
econstructor. (* (A<<a)::nil ++ (B<<b)::nil |- (D<<d[?173) *)
apply AddNilEnd.
(* (A<<a)::nil ++ (B<<b)::nil ++ nil |- (D<<d[?173) *)
instantiate (1:= (nu c (par P Q))).
(* (A<<a)::nil ++ (B<<b)::nil ++ nil |- (D<<d[nu c (par P Q)) *)
apply Cut with C.
(* exists P0 : proc, (A << a) :: nil ++ (B << b) :: nil
       |- ((C << c) [P0) *)
(* exists Q0 : proc, nil ++ (C << c) :: nil |- ((D << d) [Q0) *)
econstructor.
(* (A << a) :: nil ++ (B << b) :: nil |- ((C << c) [?179) *)
(* exists Q0 : proc, nil ++ (C << c) :: nil |- ((D << d) [Q0) *)
instantiate (1:= GETC).
(* (A << a) :: nil ++ (B << b) :: nil |- ((C << c) [GETC) *)
(* exists Q0 : proc, nil ++ (C << c) :: nil |- ((D << d) [Q0) *)
apply get_C_AB.
(* exists Q0 : proc, nil ++ (C << c) :: nil |- ((D << d) [Q0) *)
econstructor. (* nil ++ (C << c) :: nil |- ((D << d) [?182) *)
apply RemoveNilFront. (* (C << c) :: nil |- ((D << d) [?182) *)
instantiate (1:= GETD). (* (C << c) :: nil |- ((D << d) [GETD) *)
apply get_D_C. (* No more subgoals. *)
```

where contents expressed within (* *) after each tactic are results of applying that tactic.

The meta-variable P introduced in *Theorem get_D_AB* is firstly instantiated based on the *Cut* rule in the format of (nu y (Par P Q)), where any matched variables are also instantiated and any unmatched variables are in-

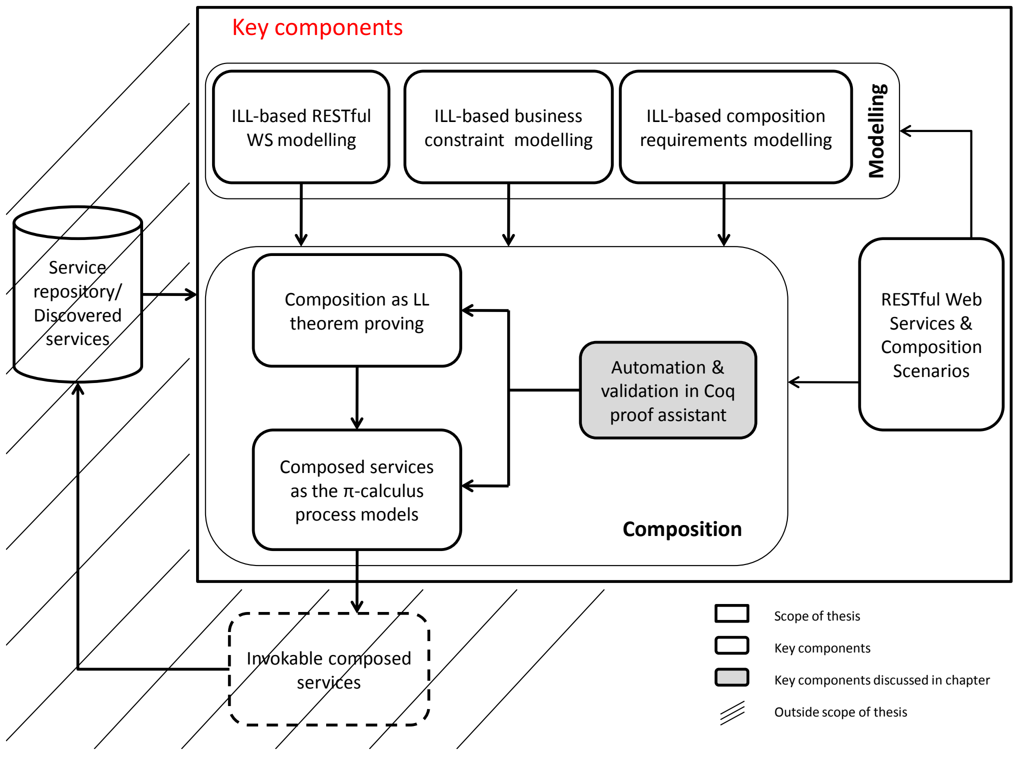
Key components
ILL-based RESTful WS modelling
ILL-based business constraint modelling
ILL-based composition requirements modelling
Modelling
Service repository/ Discovered services
Composition as LL theorem proving
Automation & validation in Coq proof assistant
Composed services as the π-calculus process models
Composition
RESTful Web Services & Composition Scenarios
Invokable composed services
Scope of thesis
Key components
Key components discussed in chapter
Outside scope of thesis

Figure 5.1: Key components discussed in Chapter 5.

troduced as new meta-variables. Thus P is written as (nu c (Par P Q)). Two new meta-variables P and Q are introduced, which will be instantiated in the next proving steps.

Following the proving process, meta-variable P is further instantiated into the process GETC, and meta-variable Q is instantiated into process GETD. Thus, the process corresponding to the composition requirement is eventually obtained at the proof progresses. The final process is written as (nu c (Par GETC GETD)) in the Coq encoding and $((\upsilon c)(\text{GETC} \mid \text{GETD}))$ in the π-calculus format defined in Chapter 4.

5.4 Summary

Reflecting the overall proposed approach, this chapter has discussed the implementation of the logic-based RESTful Web Service modelling and composition approach in the Coq proof assistant as shown in Figure 5.1.

This implementation in the tactic-style Coq theorem prover enables the efficient composition proof searching through well defined Linear Logic connectives and rules; therefore, it facilitates the automation of the composition. Although the completeness and soundness rules of the propositional Linear Logic ensure the correctness of the proof, the implementation and the theorem proving in Coq can further validate the whole proof process. The next chapter will use concrete examples to show how this implementation can be applied in the real use cases.

The Linear Logic encoding in Coq presented in this chapter can also potentially benefit researchers in other fields to perform Linear Logic theorem proving in the Coq proof assistant.

Chapter 6

Case Studies

This chapter demonstrates the feasibility and versatility of the logical approach proposed in the previous chapters by applying it to two real-world user scenarios.

One is a commonly-used e-shopping scenario which is retained in a simplified version for illustration, though it has the possibility to be considerably extended. It is detailed with three generic use cases, each of which is studied thoroughly at both the resource level and the service method level, as proposed in the two-stage service composition design. The other scenario lies in the field of biomedicine, in which a concrete service composition is discussed.

Because this thesis focuses on the composition aspect of the services, it is assumed that, in both scenarios, all existing services are discovered and available to be accessed in particular ways.

6.1 The E-shopping Scenario

E-shopping scenarios and similar holiday booking scenarios are commonly used in discussions of service composition research [20, 32, 40]. This research adopts an e-shopping scenario to thoroughly study how the proposed logical method can be applied in real cases.

This research divides composition into three detailed use cases by considering how services are involved in a composition process. Services are grouped by category according to their semantics and functions. Services in the same category have the same or very similar functionalities. For example, the British Airways flight booking service and the Air China flight booking service perform a similar function, so they are defined in the same category.

The first use case discusses a composition to be performed with only the minimal category of services/resources, and each category contains only one available service, so the availability of the services required is an important factor that determines whether the composition requirements can be achieved or not. The communication among these available services is mainly controlled by the constraints defined in the business models. For example, if both *Stock* and *Shipping* services are available, two service composition results may be created according to two different business model definitions. One composition may be a *product supplying* service that ships products to fill the stock, and the other may be a *product selling* service that ships products from the stock to a customer.

The second use case allows more than one service to be available in each category and keeps the number of categories low for simplicity. Services in the same category may be provided by different service providers at different levels of cost and quality. The availability of multiple services not only enables a particular service to be selected during composition but it also provides the possibility to replace one service by another in the same category if it is not available at a particular time. For example, in payment service, users may be given options of a debit-card or a credit-card payment service, so they can choose which one is used in the composition.

The third use case introduces more categories of service that may add extra value to the final composition. This will demonstrate that the proposed logic framework is capable of modelling and handling complex scenarios, in which the service composition specifications keep evolving with the user requirements and newly available services.

The remainder of this section provides an overview of these three use cases and the detail of modelling and composing services in each of them.

Table 6.1: Services and resources in core categories of service in the e-shopping scenario.

Service	URI	Method	Description
User Service	/example.user.com	POST	register as a new user
	/example.user.com/{uid}	GET	retrieve information of a user with uid
		PUT	update user information (e.g. address, payment method)
		DELETE	unregister user with uid
	/example.user.com/{uid}/payment	GET	retrieve payment information of a user with uid
Stock Service	/example.item.com	POST	add a new item into stock
		GET	request a list of items in stock
	/example.item.com/{iid}	GET	retrieve detail of an item with iid
		PUT	update an item with iid
Payment Service	/example.payment.com	POST	submit a payment
	/example.payment.com/{pid}	GET	retrieve detail of a payment with pid
		DELETE	cancel a payment with pid
Shipping Service	/example.shipment.com	POST	perform a shipment
	/example.shipment.com/{sid}	GET	retrieve detail of a shipment with sid

6.1.1 Case I: Core Categories of Service

Case I demonstrates a service composition scenario, in which only core categories of service are considered. In the case of e-shopping, realising a basic shopping scenario requires at least four core functional categories of atomic services as summarised in Table 6.1: a *User Service* for managing the data related to customers, a *Stock Service* for managing product information, a *Payment Service* for handling transactions between the customers and the shop, and a *Shipment Service* for delivering products from stock to the customers. This use case targets the minimal scenario, so it is assumed that only these four categories of services are available and that there is only one service in each category. The key resources represented by these services are *User*, *Item*, *Payment* and *Shipment*.

The flows among these services are defined in Figure 6.1, in which the e-shop acts as a composed service. Thus, the e-shop serves as the entry point to all other services. This entry URL is defined at *http://example.eshop.com* in this thesis. Users register themselves with information such as name and address via the e-shop service (*http://example.eshop.com/user*), which then initialises a POST method on the User Service defined at *http://example.user.com/user*. After that, a new user resource is created in the e-shop with user id, say, *http://example.eshop.com/user/1234* that links to the user in the User Service *http://example.user.com/user/1234*. Later, when users update their information, such as adding a payment method, the e-shop service performs a PUT method on *http://example.eshop.com/user/1234* that corresponds to the resource *http://example.user.com/user/1234* from the User Service.

When a user browses an item from a stock, the e-shop service performs a GET method on the Stock Service at *http://example.stock.com/item*. If a user wants to check the detail of a product, the e-shop service performs another GET method at *http://example.stock.com/item/{iid}*, where *iid* is replaced by the identification of the item. A user can perform a POST method at *http://example.eshop.com/order* to create a new order. The newly-added order resource in the e-shop service is viewed as a result of the composition

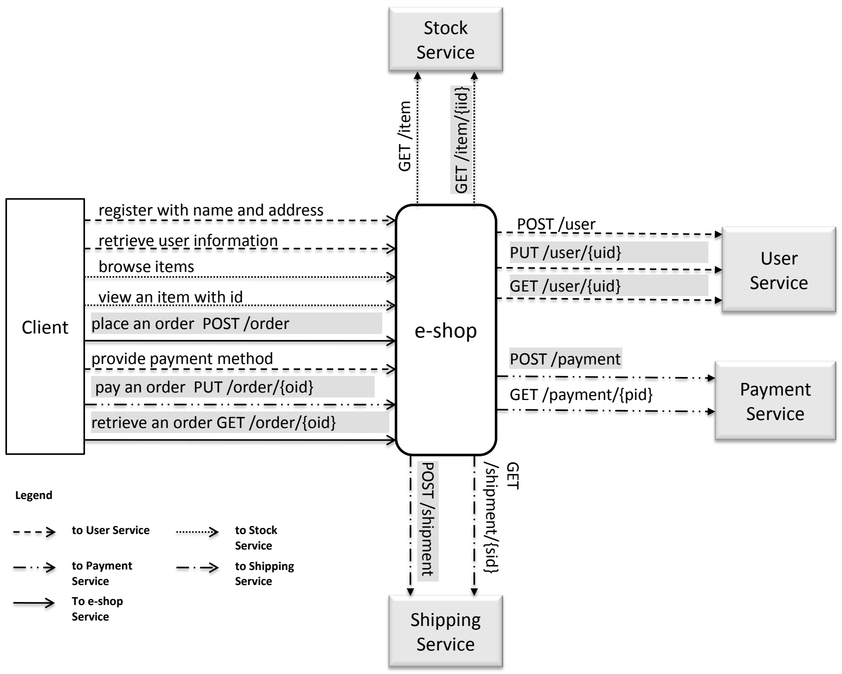

Figure 6.1: The flow of core categories of service in the e-shopping scenario.

of a User Service and a Stock Service, which allows a registered user to add an item as a purchasable order. When the payment information is submitted by the user, the e-shop service initialises a POST method to the Payment Service at *http://example.payment.com/payment*, which passes the payment detail from the user resource to the payment resource. Once the order is paid, the e-shop service initialises a POST method to the Shipping Service at *http://example.shipment.com/shipment* to ship the product to the address provided by the user. The user can retrieve the entire order information, including the product information, payment and shipment detail, using a GET method at *http://example.eshop.com/order/*{*oid*}, where *oid* is replaced by the identification of the order.

Composition at the First Stage

The application of the proposed composition method is first studied at the first-stage abstract resource level. The four categories of service defined earlier are represented by their data resources, which are then modelled as Linear Logic propositions. Thus, the minimal categories of existing RESTful Web

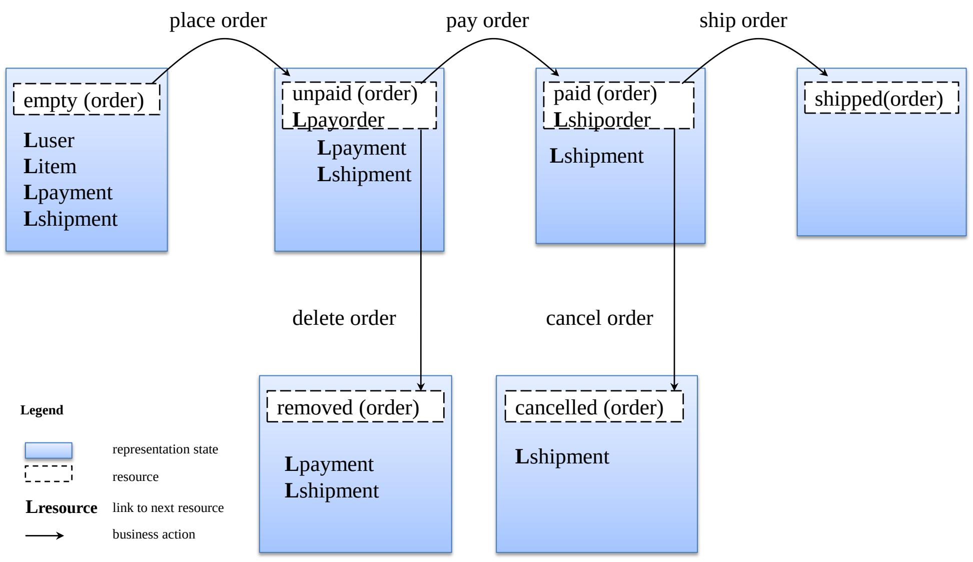

Figure 6.2: Case I flow at the abstract resource level.

services can be expressed in Linear Logic as follows:

$$\vdash user, \vdash item, \vdash payment, \vdash shipment$$

Linear Logic sequent calculus is used to model the constraints in a business model. According to the business constraints in the e-shopping scenario, the following Linear Logic hypotheses are defined to show possible relationships between the resources, in which Linear Implication ($\multimap$) is used to indicate the link to the next available resource.

$\texttt{place\_order} : order\_empty, user, item \vdash order\_unpaid$

$\texttt{pay\_order} : Lpayorder, payment \vdash order\_paid$

$\texttt{ship\_order} : Lshiporder, shipment \vdash order\_shipped$

$\texttt{link\_payorder} : \vdash order\_unpaid \multimap Lpayorder$

$\texttt{link\_shiporder} : \vdash order\_paid \multimap Lshiporder$

Figure 6.2 shows the representation transfer flow among the resources. When a valid user selects a product, an order is created. The order becomes a paid order once it is paid, and the payment information is available for the order. The order becomes a shipped order when it is dispatched to the user, and the shipment information is available. The *unpaid order* representation contains a link for paying the order, which enables the client agent to choose to move to the next representation. The *paid order* representation contains a link for shipping the order. One possible composition requirement for obtaining the shipped order is defined as a Linear Logic theorem to be proved:

$$order\_empty \otimes user \otimes item \otimes payment \otimes shipment \vdash order\_shipped$$

where the state of the order transits from the empty state to the shipped state after consuming user, item, payment and shipment resources. The Linear Logic Multiple Conjunction (⊗) used here ensures that the required resources must be available and the composition is achieved by consuming all these resources.

Following the implementation discussed in Chapter 5, the RESTful resources, business constraints and the theorem proving process are implemented in the Coq proof assistant. The resources are defined as variables in the Linear Logic proposition type, the business constraints are defined as hypothesises, and the composition requirement is defined as a theorem. The following shows some example definitions.

```
Variable user item payment shipment order_empty order_unpaid
   order_paid order_shipped Lpayorder Lshiporder: ILinProp.
Hypothesis place_order : ((item :: nil) ++ (user :: nil)
                     ++ (order_empty :: nil)) ⊢ order_unpaid.
Hypothesis pay_order : ((Lpayorder :: nil) ++ (payment :: nil))
                   ⊢ order_paid.
Hypothesis ship_order : ((Lshiporder :: nil) ++ (shipment :: nil))
                   ⊢ order_shipped.
Hypothesis link_payorder: nil ⊢ order_unpaid ⊸ Lpayorder.
Hypothesis link_shiporder: nil ⊢ order_paid ⊸ Lshiporder.

Theorem 6.1.1.1 shipping_order : (order_empty ⊗ user ⊗ item ⊗
 payment ⊗ shipment) ⊢ order_shipped
Proof. see Appendix B.1.   □
```

$$\dfrac{\dfrac{\dfrac{\dfrac{(\text{O\_empty, U, I} \vdash \text{O\_unpaid})}{(\text{O\_empty} \otimes \text{U} \otimes \text{I} \vdash \text{O\_unpaid})}\,(\otimes\text{L}) \qquad \dfrac{(\vdash \text{O\_unpaid} \multimap L\text{PO})}{(\text{O\_unpaid} \vdash L\text{PO})}\,(\text{Shift})}{(\text{O\_empty} \otimes \text{U} \otimes \text{I} \vdash L\text{PO})}\,(\text{Cut}) \qquad (L\text{PO, P} \vdash \text{O\_paid})}{(\text{O\_empty} \otimes \text{U} \otimes \text{I, P} \vdash \text{O\_paid})}\,(\text{Cut})}{(\text{O\_empty} \otimes \text{U} \otimes \text{I} \otimes \text{P} \vdash \text{O\_paid})}\,(\otimes\text{L}) \qquad \textbf{(6.3.1)}$$

$$\dfrac{\dfrac{\dfrac{(\text{O\_empty} \otimes \text{U} \otimes \text{I} \otimes \text{P} \vdash \text{O\_paid}) \qquad \dfrac{(\vdash \text{O\_paid} \multimap L\text{SO})}{(\text{O\_paid} \vdash L\text{SO})}\,(\text{Shift})}{(\text{O\_empty} \otimes \text{U} \otimes \text{I} \otimes \text{P} \vdash L\text{SO})}\,(\text{Cut}) \qquad (L\text{SO, S} \vdash \text{O\_shipped})}{(\text{O\_empty} \otimes \text{U} \otimes \text{I} \otimes \text{P, S} \vdash \text{O\_shipped})}\,(\text{Cut})}{(\text{O\_empty} \otimes \text{U} \otimes \text{I} \otimes \text{P} \otimes \text{S} \vdash \text{O\_shipped})}\,(\otimes\text{L}) \qquad \textbf{(6.3.2)}$$

Figure 6.3: Linear Logic proof for composing e-shopping services case I at the resource level.

This composition requirement presented in the theorem is proved by applying the Intuitionistic Linear Logic inference rules defined in Figure 3.1 and implemented in Appendix A.1. The full detail of theorem proving in Coq are listed in Appendix B.1.

For legibility, the proof is summarised as proof trees and divided into two fragments (see Figure 6.3), in which the following abbreviations are used: U for user, I for item, P for payment, S for shipment, O for order, LPO for the link to pay the order, and LSO for the link to ship the order.

This proof tree shows two important characteristics that drive the whole composition planning process. One is the *link* business action offered by the nature of RESTful Web Services. The other is the cut inference rule presented by Linear Logic. For example, because there is a link to paying the order in the representation of order_unpaid ($\vdash$ O_unpaid $\multimap$ LPO) and a predefined business constraint (LPO, P $\vdash$ O_paid), the paid order (O_paid) is deducted by applying the *cut* rule.

Composition at the Second Stage

Once a proof is found at the abstract resource level at the first stage, the composition method continues to apply at the service method level using Linear Logic theorem proving. The proving task shows that the composition goal can be realised from the given services by using the business actions as the main inference steps. This step attaches the π-calculus to Linear Logic, so the process model in the π-calculus is directly extracted during the theorem proving. As discussed earlier, this process model can facilitate the translation of the logic model to executable business process languages.

The key methods in the existing four services are expressed as follows, with the π-calculus attachments. The operation actions are attached using ":: to represent the *processes* in the π-calculus. Variables are attached using ": to represent the *names* in the π-calculus.

User service:

$\texttt{nuriuser:uriuser, nusermsg:usermsg}$

$\quad\vdash \texttt{PostUser::nuriuid:((uriuser} \multimap \texttt{uriuid)} \oplus \texttt{err)}$

$\texttt{nuriuid:uriuid} \vdash \texttt{GetUid::nruid:(ruid} \oplus \texttt{err)}$

$\texttt{nuriuid:uriuid, nuidmsg:uidmsg} \vdash \texttt{PutUid::nruid:(ruid} \oplus \texttt{err)}$

$\texttt{nuriuid:uriuid} \vdash \texttt{DeleteUid::(0} \oplus \texttt{err)}$

$\texttt{nuriuidpay:uriuidpay} \vdash \texttt{GetUidPay::nruidpay:(ruipay} \oplus \texttt{err)}$

Stock service:

$\texttt{nuriitem:uriitem, nitemmsg:itemmsg}$

$\quad\vdash \texttt{PostItem::nuriiid:((uriitem} \multimap \texttt{uriiid)} \oplus \texttt{err)}$

$\texttt{nruiitem::uriitem} \vdash \texttt{GetItem::nritem:(ritem} \oplus \texttt{err)}$

$\texttt{nuriiid:uriiid} \vdash \texttt{GetIid::nriid:(riid} \oplus \texttt{err)}$

$\texttt{nuriiid:uriiid, niidmsg:iidmsg} \vdash \texttt{PutIid::nriid:(riid} \oplus \texttt{err)}$

Payment service:

$\texttt{nuripay:uripay, npaymsg:paymsg}$

$\quad\vdash \texttt{PostPay::nuripid:((uripay} \multimap \texttt{uripid)} \oplus \texttt{err)}$

$\texttt{nuripid:uripid} \vdash \texttt{GetPid::nrpid:(rpid} \oplus \texttt{err)}$

$\texttt{nuripid:uripid} \vdash \texttt{DeletePid::(0} \oplus \texttt{err)}$

Shipping service:

$\texttt{nuriship:uriship, nshipmsg:shipmsg}$

$\quad\vdash \texttt{PostShip::nurisid:((uriship} \multimap \texttt{urisid)} \oplus \texttt{err)}$

$\texttt{nurisid:urisid} \vdash \texttt{GetSid::nrsid:(rsid} \oplus \texttt{err)}$

It is noted that, to simplify the later proof process, the successful HTTP status codes are removed, and the error HTTP status codes are represented by *err* in the expressions. As an example, a complete expression for the PostUser method in the User service should be written as:

$\texttt{nuriuser:uriuser, nusermsg:usermsg} \vdash$

$\texttt{PostUser::nuriuid:(((uriuser} \multimap \texttt{uriuid)} \otimes \texttt{suc)} \oplus \texttt{err)}$

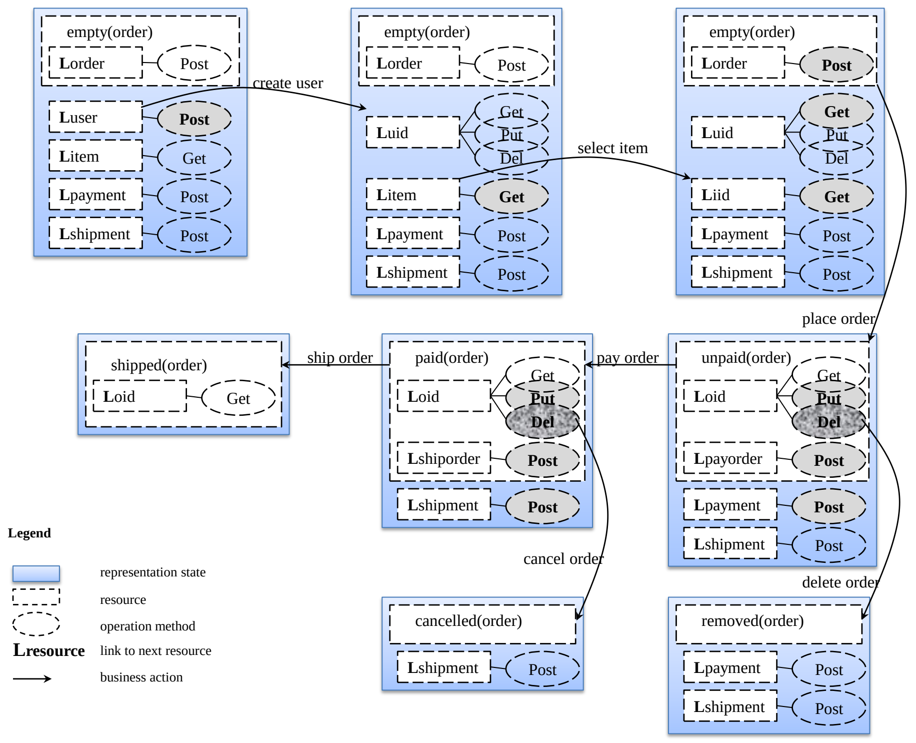

Figure 6.4: Case I flow at the service method level.

where *suc* stands for the successful HTTP status codes and *err* stands for the error HTTP status codes. The Additive Disjunction ($\oplus$) is used here to show the possibility of system exceptions, which is out of the control of the users.

The major representation transfer flows at the service method level are detailed in Figure 6.4. In order to start the e-shopping process, it is assumed that: there is a valid user identified by *http://user.example.com/1234* and an available item in the stock (*http://stock.example.com/001*); the payment service is available at the URL *http://payment.example.com*; the shipping service is available at the URL *http://shipping.example.com*; the order resource is identified by the URL *http://eshop.example.com/order*; and the initial state for an order is empty.

The key business actions are modelled in the following way.

$$\texttt{place\_order: nruid:(ruid} \oplus \texttt{err), nriid:(riid} \oplus \texttt{err),}$$
$$\texttt{nuriorder:uriorder} \vdash \texttt{PlaceOrder::nroid:(roid} \oplus \texttt{err)}$$
$$\texttt{pay\_order: n}L\texttt{payorder:(}L\texttt{payorder} \oplus \texttt{err), nruidpay:(ruidpay} \oplus \texttt{err),}$$
$$\texttt{nuripay:uripay} \vdash \texttt{PayOrder::nroidpaid:(roidpaid} \oplus \texttt{err)}$$

$\texttt{ship\_order: n}L\texttt{shiporder:(}L\texttt{shiporder} \oplus \texttt{err), nuriship:uriship}$

$\quad \vdash \texttt{ShipOrder::nroidshipped:(roidshipped} \oplus \texttt{err)}$

$\texttt{link\_payorder:} \vdash \texttt{LinkPayorder::n}L\texttt{payorder:((roid} \oplus \texttt{err)}$

$\quad \multimap \texttt{(}L\texttt{payorder} \oplus \texttt{err))}$

$\texttt{link\_shiporder:} \vdash \texttt{LinkShiporder::n}L\texttt{shiporder:((roidpaid} \oplus \texttt{err)}$

$\quad \multimap \texttt{(}L\texttt{shiporder} \oplus \texttt{err))}$

The *place_order* business action takes as input representation the user and item information, which are accessed through a GET method on the user URL and the item URL, respectively. It then performs a POST operation at the order resource URL (*http://eshop.example.com/order*) to create a new resource according to a dynamic URI template (a template of the dynamic order resource may be defined as *http://eshop.example.com/order/{oid}* for creating orders), so a newly-created order could, for example, be identified by the URL *http://eshop.example.com/order/abc*. This transfers the application state of an order from empty to unpaid. The output unpaid order representation contains a link to allow the next action to pay the order (e.g. *http://eshop.example.com/pay/order/abc*).

The *pay_order* business action uses the pay order link (Lpayorder) to POST to the payment service. After a successful payment, the order is updated from the unpaid state to the paid state. The output paid order representation contains a link (e.g. *http://eshop.example.com/ship/order/abc*) to allow the next action, which is to ship the order.

The *ship_order* business action uses the ship order link (Lshiporder) to POST to the shipment service and after a successful shipment, the order is updated from the paid state to the shipped state. It is noted that, in reality, the shipping service may periodically check the order state, rather than waiting for the POST action to start the shipping process.

The *delete_order* business action performs a DELETE operation on the order (*http://eshop.example.com/order/1234*), which transfers the order from the unpaid state to the removed state. There are no further links to the payment and the shipment services, so they remain uninvoked.

The *cancel_order* business action performs a DELETE operation on a paid-but-not-shipped order (*http://eshop.example.com/order/1234*), which transfers the order from the paid state to the cancelled state. The shipment service remains uninvoked. We note that the detailed refund process is ignored here to avoid overcomplicating the scenario.

One composition requirement for producing a shipped order is defined as a theorem to be proven (see Theorem 6.1.1.2). As at the resource level, the definitions and proofs are implemented in Coq, which is detailed in Appendix B.2. In the expression, the symbol $\ll$ and [are used to introduce names and processes, respectively.

```
Theorem 6.1.1.2   order_being_shipped: exists P,
        ((uriship ≪ nuriship), (uripay ≪ nuripay),
        (uriuidpay ≪ nuriuidpay), (uriorder ≪ nuriorder),
        (uripid ≪ nuripid), (uriuid ≪ nuriuid))
        ⊢ (roidshipped ≪ nroidshipped[P)
```

Proof. see Appendix $B.2$. □

This composition requirement is proved by applying the inference rules defined in Figure 4.3 and implemented in Appendix A.2. The full detail of theorem proving in Coq are listed in Appendix B.2. Appendix C.1 shows the proof trees for obtaining a shipped order.

During the implementation in Coq, the meta-variable (exists P) is used, which will be instantiated into the plan as the proof proceeds. Taking the proof process for Theorem 6.1.1.2 as an example, Listing 6.1 shows the first few lines of the proof (see Appendix B.2 for the whole proof).

The proof is performed in backward reasoning as proposed in Section 4.4. Thus, firstly, the *econstructor* tactic introduces the existential (exists P) as a variable. As underlined in Listing 6.2, the existential P is replaced by the variable *?2634* that is given randomly by Coq. This variable is then instantiated according to the inference rules defined in Figure 4.3. In this case, it is firstly instantiated (the 3^{rd} line at Listing 6.1) according to the *Cut* rule definition, whose π-calculus attachment is encoded in Coq as (nu y (par P Q)) (see Listing 6.3) before actually applying the *Cut* rule (the 4th line at Listing 6.1). After using the *Cut* rule, two fragments are created as shown in Listing 6.4 with two new existentials P0 and Q0.

Listing 6.1: An example Coq proof code.

```
Theorem order_being_shipped: exists P,
        (((uriship<<nuriship) :: nil)
        ++ ((uripay<<nuripay) :: nil)
        ++ ((uriuidpay<<nuriuidpay) :: nil)
        ++ ((uriorder<<nuriorder) :: nil)
        ++ ((uripid<<nuripid) :: nil)
        ++ ((uriuid<<nuriuid) :: nil))
        |- ((roidshipped ⊕ err) <<nroidshipped[P).


Proof.
1 econstructor.
2 apply AddNilRight.
3 instantiate (1:= (nu y (par P Q))).
4 apply Cut with (Times uriship (Plus lshiporder err)).
5 auto.
6 econstructor.
7 instantiate (1:= (nu nuriship (outp nlshiporder nuriship (par P Q)))).
8 apply TimesRight.
9 econstructor.
10 ... ...
```

Listing 6.2: After applying the econstructor tactic at line 1.

```
((uriship << nuriship) :: nil) ++ ((uripay << nuripay) :: nil)
++ ((uriuidpay << nuriuidpay) :: nil)
++ ((uriorder << nuriorder) :: nil)
++ ((uripid << nuripid) :: nil) ++ (uriuid << nuriuid) :: nil
|- (((roidshipped ⊕ err) << nroidshipped) [?2634)
```

Listing 6.3: After applying the instantiate tactic at line 3.

```
(((uriship << nuriship) :: nil) ++ ((uripay << nuripay) :: nil)
++ ((uriuidpay << nuriuidpay) :: nil)
++ ((uriorder << nuriorder) :: nil)
++ ((uripid << nuripid) :: nil)
++ (uriuid << nuriuid) :: nil) ++ nil
|- (((roidshipped ⊕ err) << nroidshipped) [nu y (par P Q))
```

Listing 6.4: After applying the Cut rule at line 4.

```
exists P0 : proc, ((uriship << nuriship) :: nil)
++ ((uripay << nuripay) :: nil)
++ ((uriuidpay << nuriuidpay) :: nil)
++ ((uriorder << nuriorder) :: nil)
++ ((uripid << nuripid) :: nil)
++ (uriuid << nuriuid) :: nil
|- (((uriship ⊗ (lshiporder ⊕ err)) << y) [P0)



exists Q0 : proc,
nil ++ ((uriship ⊗ (lshiporder ⊕ err)) << y) :: nil
|- (((roidshipped ⊕ err) << nroidshipped) [Q0)
```

The Coq theorem prover continues to apply tactics and inference rules until the proof is completed, the π-calculus process is instantiated during this proof. An example π-calculus process after the second instantiate at Line 7 is shown as follows.

```
nu y (par (nu nuriship (outp nlshiporder nuriship (par P Q))) Q0)
```

The whole π-calculus process for an order being shipped is shown in Listing 6.5. For legibility, the connectives are written in the style introduced in Chapter 4. The corresponding reference can be found in Table 5.1.

Listing 6.5: The π-calculus process model extracted for Case I of the e-shopping scenario.

```
1  (υy1)(((υnuriship)nuriship⟨nlshiporder⟩.(0
2     | ((υnroidpaid)(((υy2)(((υnuripay)nuripay⟨y3⟩.(0
3       | ((υnruidpay)nruidpay⟨nlpayorder⟩.(GETUIDPAY
4         | ((υnroid(((υy4)(((υnuriorder)nuriorder⟨y5⟩.(0
5             | ((υnriid)nriid⟨nruid⟩.(GETIID | GETUID))))
6           | (nuriorder(y6).nriid(nruid).PLACEORDER))))
7         | LINKPAYORDER)))))))
8       | (nuripay(y7).nruidpay(nlpayorder).PAYORDER))
9     | LINKSHIPORDER)))
10  | (nuriship(nlshiporder).SHIPORDER))
```

As the reasoning is performed backwards, the process extracted at the first outer layer (see the 1st and 10th lines in Listing 6.5) corresponds to the last business flow in the composition (i.e. shipping the order), and the last inner layer (see the 5th line in Listing 6.5) corresponds to the first business flow in the composition (i.e. obtaining user and item information through their ids: GETIID and GETUID). The unknown parameters, which assist the extraction but are not defined by the existing services, are represented by y.

Here, GETIID and GETUID can run in parallel. Once the user and the item information are retrieved, they are passed to the PLACEORDER process as shown in the 6th line; and the *place order* process is completed with the 4th line. After the PLACEORDER process, the LINKPAYORDER process (see the 7th line) is available and can run in parallel with the GETUIDPAY process (see the 3rd line) to make the user payment information ready in order to submit to the payment service. Once the PAYORDER process (see the 2nd and the 8th lines) is performed, the LINKSHIPORDER process is available; and finally the SHIPORDER process is performed at the 1st and 10th lines.

Table 6.2: Multiple payment services and resources in Case II of the e-shopping scenario.

Service	URI	Method	Description
Paypal Service	/example.payment.com	POST	submit a Paypal payment
	/example.paypal.com/{pid}	GET	retrieve detail of a payment with pid
		DELETE	cancel a payment with pid
CreditCard Service	/example.payment.com	POST	submit a creditcard payment
	/example.creditcard.com/{pid}	GET	retrieve detail of a payment with pid
		DELETE	cancel a payment with pid
DebitCard Service	/example.payment.com	POST	submit a debitcard payment
	/example.debitcard.com/{pid}	GET	retrieve detail of a payment with pid
		DELETE	cancel a payment with pid

6.1.2 Case II: Multiple Services in One Category

With the large number of services that can be found on the Web today, it is to be expected that one functional service category may have several services available. These services have the same, or very similar, functionalities to the end users, but they are offered by different providers with differing quality of service. For example, in the e-shopping scenario, there may be multiple stock providers for the same product, payment services may be offered by different providers, and shipment services may also have different providers. The e-shop service should be able to make all of the existing options available to users, allowing the user to decide upon which services to include in the composition.

Due to the provision of Additive Conjuction (&) in Linear Logic, the user choice among services can be explicitly modelled in the proposed logical approach. This second use case demonstrates this by analysing an example in which three different services are available in the Payment Service category. These services are summarised in Table 6.2: the *PayPal Service* for offering payment via the Paypal service, a *CreditCard Service* for allowing use of a credit card, and a *DebitCard Service* for allowing use of a debit card. Users themselves have the choice of which payment method to use through different payment services.

When the user reaches at the payment step from the e-shop service (i.e. via URL *http://example.eshop.com/payment*), links to the three payment options are available, which are defined by URLs: *http://example.paypal.com*, *http://example.creditcard.com* and *http://example.debitcard.com.* The POST payment method in the e-shop service corresponds to the selected payment services.

Composition at the First Stage

At the first-stage resource level, the Payment Service in Table 6.1 is extended into three services with detailed resources defined in Table 6.2. As in Case I, the resources are expressed as Linear Logic propositions as follows:

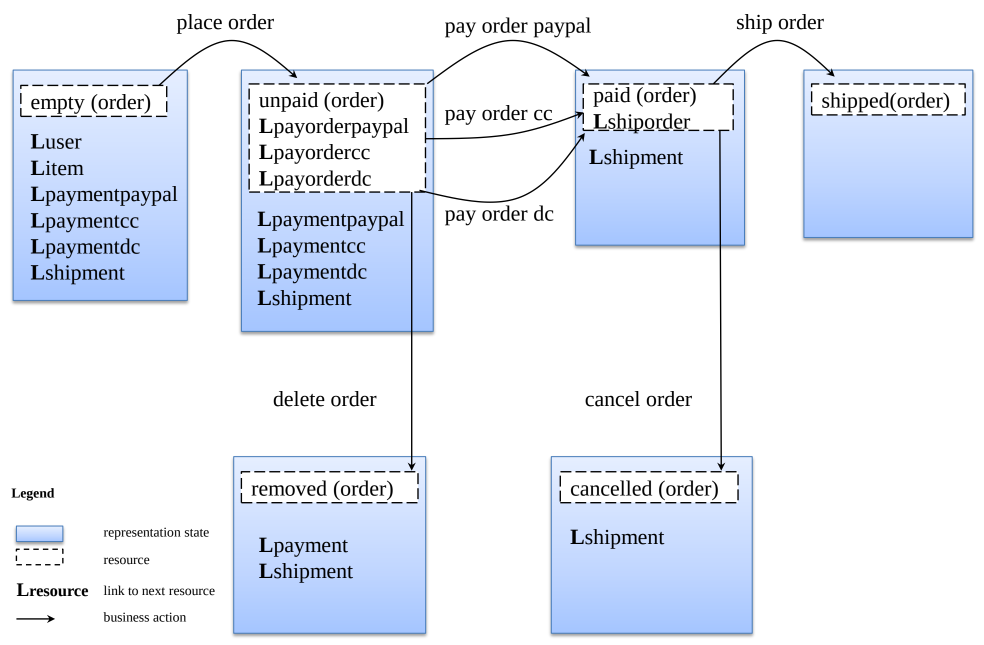

Figure 6.5: Case II flow at the abstract resource level.

$\vdash$`user`, $\vdash$`item`, $\vdash$`paymentpaypal`, $\vdash$`paymentcc`, $\vdash$`paymentdc`, $\vdash$`shipment`

The business constraints are modelled as Linear Logic sequent calculus as follows, where Additive Conjunction (&) is used to offer different payment choices to the users. The *pay_order* business action specified in Case I is now replaced by three sub business actions: *pay_order_paypal*, *pay_order_cc* and *pay_order_dc*.

```
place_order: order_empty, user, item ⊢ order_unpaid
pay_order_paypal: Lpayorderpaypal, paymentpaypal ⊢ order_paid
pay_order_cc: Lpayordercc, paymentcc ⊢ order_paid
pay_order_dc: Lpayorderdc, paymentdc ⊢ order_paid
ship_order: Lshiporder, shipment ⊢ order_shipped
link_payorder: ⊢ order_unpaid ⊸
       (Lpayorderpaypal & Lpayordercc & lpayorderdc)
link_shiporder: ⊢ order_paid ⊸ Lshiporder
```

Figure 6.5 shows the representation transfer flow among the resources for this use case. In addition to the flow of Case I illustrated in Figure 6.2, once an order is placed, three possible next payment transitions are available through links: *Lpayorderpaypal*, *Lpayordercc*, *Lpayorderdc*.

Once a payment link is selected, it will submit payment information to the corresponding payment service. For example, if *Lpayordercc* is selected, the credit card payment service (*paymentcc*) will be invoked. The use of Linear Logic Additive Conjunction (&) applies the restriction that only one payment method can be selected, so once *Lpayordercc* is selected, *Lpayorderpaypal* and *Lpayorderdc* will be ignored. After the payment is successfully performed, the representation state of the order changes from unpaid to paid.

Thus, the composition specification of producing a shipped order requires that three options of payment are available, as shown in Theorem 6.1.2.1. This composition requirement is proved by applying the Intuitionistic Linear Logic inference rules defined in Figure 3.1 and implemented in Appendix A.1. The full detail of theorem proving in Coq are listed in Appendix B.3. For legibility, the proof is summarised as proof trees as shown in Appendix C.2.

```
Theorem 6.1.2.1 ship_order_multipay :
(order_empty ⊗ user ⊗ item ⊗
(paymentpaypal & paymentcc & paymentdc) ⊗ shipment)
⊢ order_shipped

Proof. see Appendix B.3.  □
```

Composition at the Second Stage

At the second-stage service method level, the Payment Service defined in Case I is now replaced by three different services and the user provides the corresponding payment method in the User Service.

The following lists the full services available in this use case at the service method level, in which *Paypal Service*, *CreditCard Service* and *DebitCard Service* are newly-introduced for this use case. Again, the operation actions are attached using "":: to represent the *processes* in the π-calculus. Variables are attached using "": to represent the *names* in the π-calculus. As noted earlier in

Case I, the successful HTTP status codes are removed, and the error HTTP status codes are represented by *err* in the expressions.

User service:

$$\texttt{nuriuser:uriuser, nusermsg:usermsg}$$
$$\vdash \texttt{PostUser::nuriuid:((uriuser} \multimap \texttt{uriuid)} \oplus \texttt{err)}$$
$$\texttt{nuriuid:uriuid} \vdash \texttt{GetUid::nruid:(ruid} \oplus \texttt{err)}$$
$$\texttt{nuriuid:uriuid, nuidmsg:uidmsg} \vdash \texttt{PutUid::nruid:(ruid} \oplus \texttt{err)}$$
$$\texttt{nuriuid:uriuid} \vdash \texttt{DeleteUid::(0} \oplus \texttt{err)}$$
$$\texttt{nuriuidpay:uriuidpay} \vdash \texttt{GetUidPaypal::nruidpay:(ruipaypal} \oplus \texttt{err)}$$
$$\texttt{nuriuidpay:uriuidpay} \vdash \texttt{GetUidPaycc::nruidpay:(ruipaycc} \oplus \texttt{err)}$$
$$\texttt{nuriuidpay:uriuidpay} \vdash \texttt{GetUidPaydc::nruidpay:(ruipaydc} \oplus \texttt{err)}$$

Stock service:

$$\texttt{nuriitem:uriitem, nitemmsg:itemmsg}$$
$$\vdash \texttt{PostItem::nuriiid:((uriitem} \multimap \texttt{uriiid)} \oplus \texttt{err)}$$
$$\texttt{nruiitem::uriitem} \vdash \texttt{GetItem::nritem:(ritem} \oplus \texttt{err)}$$
$$\texttt{nuriiid:uriiid} \vdash \texttt{GetIid::nriid:(riid} \oplus \texttt{err)}$$
$$\texttt{nuriiid:uriiid, niidmsg:iidmsg} \vdash \texttt{PutIid::nriid:(riid} \oplus \texttt{err)}$$

Paypal service:

$$\texttt{nuripay:uripaypal, npaymsg:paymsg}$$
$$\vdash \texttt{PostPaypal::nuripid:((uripaypal} \multimap \texttt{uripid)} \oplus \texttt{err)}$$
$$\texttt{nuripid:uripid} \vdash \texttt{GetPid::nrpid:(rpid} \oplus \texttt{err)}$$
$$\texttt{nuripid:uripid} \vdash \texttt{DeletePid::(0} \oplus \texttt{err)}$$

CreditCard service:

$$\texttt{nuripay:uripaycc, npaymsg:paymsg}$$
$$\vdash \texttt{PostPaycc::nuripid:((uripaycc} \multimap \texttt{uripid)} \oplus \texttt{err)}$$
$$\texttt{nuripid:uripid} \vdash \texttt{GetPid::nrpid:(rpid} \oplus \texttt{err)}$$
$$\texttt{nuripid:uripid} \vdash \texttt{DeletePid::(0} \oplus \texttt{err)}$$

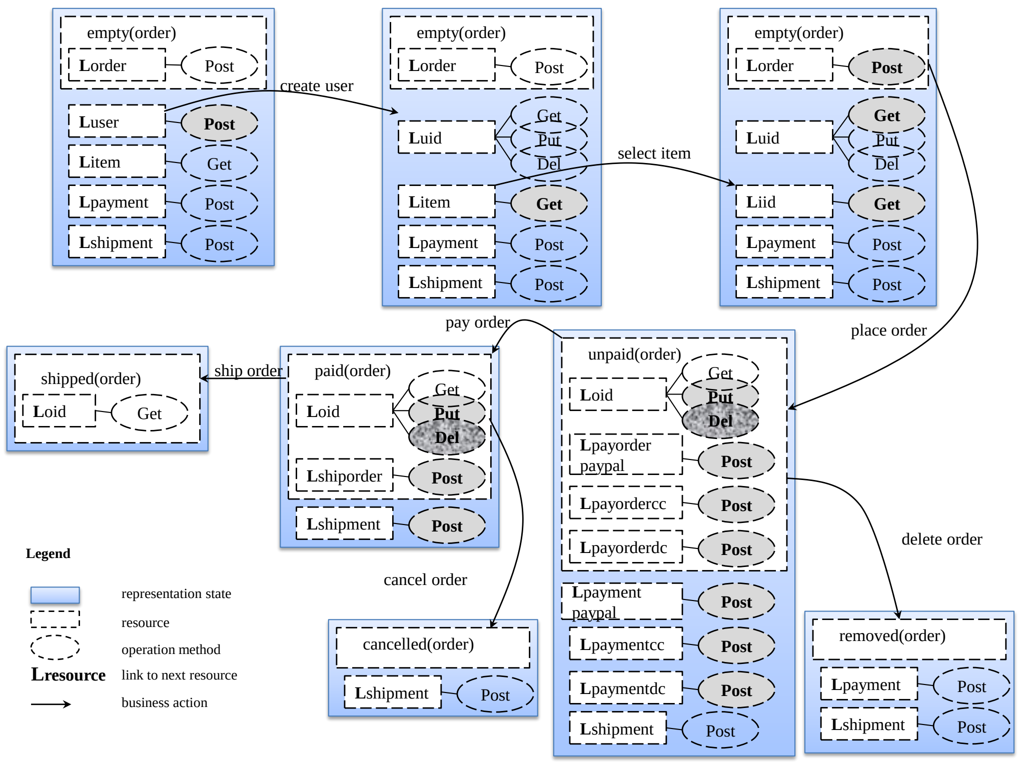

Figure 6.6: Case II flow at the service method level.

DebitCard service:

nuripay:uripaydc, npaymsg:paymsg

$\vdash$ PostPaydc::nuripid:((uripaydc $\multimap$ uripid) $\oplus$ err)

nuripid:uripid $\vdash$ GetPid::nrpid:(rpid $\oplus$ err)

nuripid:uripid $\vdash$ DeletePid::(0 $\oplus$ err)

Shipping service:

nuriship:uriship, nshipmsg:shipmsg

$\vdash$ PostShip::nurisid:((uriship $\multimap$ urisid) $\oplus$ err)

nurisid:urisid $\vdash$ GetSid::nrsid:(rsid $\oplus$ err)

The major representation transfer flows at this service method level are detailed in Figure 6.6. In contrast to that for Case I in Figure 6.4, the *unpaid order* resource is expanded with links to three available payment methods as well the actual payment services used for composition.

The business constraints are defined as follows. When the users reach the payment stage, they may choose their preferred payment method, which is modelled using the Linear Logic Additive Conjunction (&).

```
place_order: nruid:(ruid ⊕ err), nriid:(riid ⊕ err),
        nuriorder:uriorder ⊢ PlaceOrder::nroid:(roid ⊕ err)
pay_order_paypal: nLpayorderpaypal:(Lpayorderpaypal ⊕ err),
        nruidpaypal:(ruidpaypal ⊕ err), nuripaypal:uripaypal
        ⊢ PayOrderPaypal::nroidpaid:(roidpaid ⊕ err)
pay_order_cc: nLpayordercc:(Lpayordercc ⊕ err),
        nruidpaycc:(ruidpaycc ⊕ err), nuripaycc:uripaycc
        ⊢ PayOrderCC::nroidpaid:(roidpaid ⊕ err)
pay_order_dc: nLpayorderdc:(Lpayorderdc ⊕ err),
        nruidpaydc:(ruidpaydc ⊕ err), nuripaydc:uripaydc
        ⊢ PayOrderDC::nroidpaid:(roidpaid ⊕ err)
ship_order: nLshiporder:(Lshiporder ⊕ err), nuriship:uriship
        ⊢ ShipOrder::nroidshipped:(roidshipped ⊕ err)


link_payorder: ⊢ LinkPayorder::nLpayorder: ((roid ⊕ err)
        ⊸ ((Lpayorderpaypal ⊕ err) & (Lpayordercc ⊕ err)
                & (Lpayorderdc ⊕ err)))
link_shiporder: ⊢ LinkShiporder::nLshiporder:
        ((roidpaid ⊕ err) ⊸ (Lshiporder ⊕ err))
```

The three pay order business actions: *pay_order_paypal*, *pay_order_cc* and *pay_order_dc* use the corresponding pay order links: *Lpayorderpaypal*, *Lpayordercc* and *Lpayorderdc* to POST to the corresponding payment services, respectively. Other services remain the same as in Case I. After a successful payment, the order is updated from the unpaid state to the paid state. In a similar way to Case I, the output paid order representation contains a link (e.g. *http://eshop.example.com/ship/order/abc*) to allow the next action, which is to ship the order.

Theorem 6.1.2.2 below provides one composition requirement for obtaining a shipped order paid via the Paypal service. The proof for this composition will search whether the Paypal service is available and whether it can be applied in this composition. Other similar composition requirements, such as replacing the payment service with a CreditCard or DebitCard service, can also be performed by the same means.

```
Theorem 6.1.2.2   order_is_shipped_paypal: exists P,
((uriship ≪ nuriship), (uripaypal ≪ nuripay),
(uriuidpay ≪ nuriuidpay), (uriorder ≪ nuriorder),
(uripid ≪ nuripid), (uriuid ≪ nuriuid))
⊢ (roidshipped ≪ nroidshipped[P)
```

Proof. see Appendix $B.4$. □

This composition requirement is proved by applying the inference rules defined in Figure 4.3 and implemented in Appendix A.2. The full detail of theorem proving in Coq are listed in Appendix B.4. Appendix C.3 shows the proof trees constructed for obtaining a shipped order.

Listing 6.6: The π-calculus process model extracted for Case II of the e-shopping scenario.

```
(υy1)((υnuriship)(nuriship<nlshiporder>.(0
   | (υnroidpaid)((υy2)((υnuripay)nuripay<y3>.(0
     | (υnruidpay)nruidpay<nlpayorder>.(GETUIDPAYPAL
       | (υnroid( (υy4) ((υnuriorder)nuriorder<y5>.(0
           | (υnrpid)nrpid<nruid>.(GETPID | GETUID))
         | nuriorder(y6).nrpid(nruid).PLACEORDER)
       | (υnlpayorder)(LINKPAYORDER | 0))))
     | nuripay(y7).nruidpay(nlpayorder).PAYORDERPAYPAL)
   | LINKSHIPORDER)))
 | nuriship(nlshiporder).SHIPORDER)
```

The whole π-calculus process for an order being shipped is shown in Listing 6.6. For legibility, the connectives are written in the style introduced in Chapter 4. The corresponding reference can be found in Table 5.1. In contrast to the process model extracted for Case I listed in Listing 6.5, the payment process is changed. When obtaining the user payment method information, the GETUIDPAYPAL process (see the 3rd line in Listing 6.6) is instantiated for accessing the *Paypal* payment information; and the *pay order* process is instantiated by PAYORDERPAYPAL in the 8th line. Other extractions remain the same as in Case I.

6.1.3 Case III: Extended Categories of Service

This use case demonstrates the capability of the proposed Linear Logic based approach in terms of evolving the composition. In many composition scenarios, it is useful to enrich the basic business models with value-added services and to make the scenario evolve as required. Depending on the user's requirement specifications, such services may be added to achieve the ultimate result.

For example, a basic trip-planning scenario may include flight booking and hotel reservation services, but it may add hotel rating services in order to help users to select the most suitable hotels when booking the accommodation, and add currency converter services to allow users to calculate the cost immediately in a different currency.

Case III assumes that more than four categories of service are available in the e-shopping scenario. These services are summarised in Table 6.3. A user security checking service is added to authenticate users before allowing them to place an order. A rating service category is added to allow users to see the rating of the product before placing an order. An insurance service category is added to allow users to buy product protection while purchasing a product. Table 6.3 summaries these three services.

It is also assumed that there is at least one service available in each category. When users require such value-added functionalities, these related services are added into the composition specification.

In particular, this case study defines one service each in the security checking, product rating and product protection insurance service category, and they are identified by *http://example.securitychecker.com*, *http://example.rating.com* and *http://example.insurance.com*, respectively.

Table 6.3: Extended services and resources in Case III of the e-shopping scenario.

Service	URI	Method	Description
Security Check Service	/example.securitychecker.com	POST	submit a user to security check
Product Rating Service	/example.rating.com	POST	create a product rating
	/example.creditcard.com/{pid}	GET	retrieve detail of a product rating with rid
		DELETE	remove a product rating with rid
Travel Insurance Service	/example.insurance.com	POST	purchase a travel insurance
	/example.debitcard.com/{pid}	GET	retrieve detail of an insurance with insurid
		DELETE	cancel an insurance with insurid

Composition at the First Stage

At the first-stage resource level, three resources (i.e. product rating, user security check and product insurance) are provided on top of Case I, so the existing resources are expressed as Linear Logic propositions as follows:

$\vdash$`user`, $\vdash$`item`, $\vdash$`payment`, $\vdash$`shipment`, $\vdash$`rate`, $\vdash$`securitychecker`, $\vdash$`insurance`

The business constraints are modelled as follows:

```
validate_user: user, securityschecker ⊢ user_valid
rate_item: item, rate ⊢ item_rated
place_order: order_empty, user_valid, item_rated ⊢ order_unpaid
insure_unpaid_order: Linsureorder, insurance ⊢ order_unpaid_insured
insure_paid_order: Linsureorder, insurance ⊢ order_paid_insured
pay_order: Lpayorder, payment ⊢ order_paid
pay_insured_order: Lpayorder, payment ⊢ order_insured_paid
ship_order: Lshiporder, shipment ⊢ order_shipped
link_after_order: ⊢ order_unpaid ⊸ (Linsureorder & Lpayorder)
link_after_insured_unpaid_order: ⊢ order_unpaid_insured ⊸ Lpayorder
link_shiporder1: ⊢ order_paid_insured ⊸ Lshiporder
link_shiporder2: ⊢ order_insurerd_paid ⊸ Lshiporder
```

Figure 6.7 illustrates the overall representation transfer flow among the resources in this use case. A user becomes a valid user once they are authenticated by the Security Checking Service. For an item, its rating information can be checked via the Rating Service. A valid user may check the rating of an item and then place it as an order that remains unpaid. This *unpaid order* representation contains two links for the possible next state transition: pay this order or buy insurance for this order. If the *pay order* is chosen, the order becomes a paid but uninsured order, then users can choose between buy insurance for it or ship it for the next state. If the *insure order* is chosen for the unpaid order, the order becomes an insured order, and the link to pay order is available. If an order is paid and insured, the link for shipping it is available.

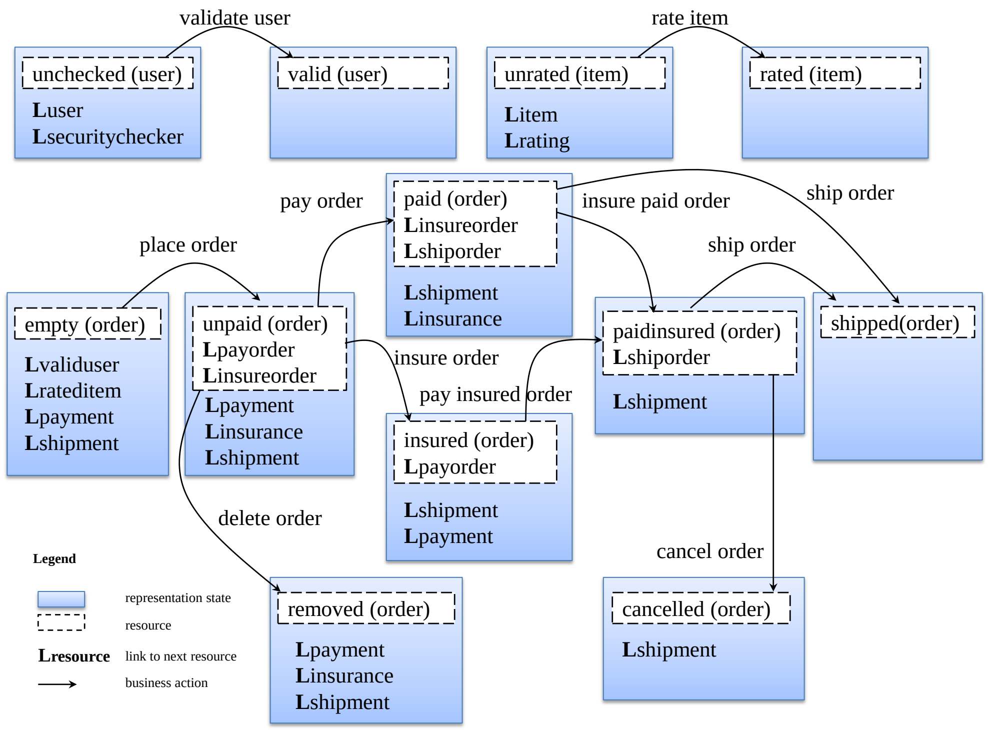

Figure 6.7: Case III flow at the abstract resource level.

In order to explicitly express these two different routes for the service composition in this use case (i.e. via pay unpaid order first and via insure unpaid order first), the business actions defined above deliberately split the paid and insured order into two resources: *order_paid_insured* and *order_insured_paid*. However, in real-world implementations, they can be represented by just a single resource.

Theorem 6.1.3.1 below shows an example service composition requirement that uses all seven existing services for obtaining a shipped order. As in previous cases, at the first-stage abstract level, this composition requirement is proved by applying the Intuitionistic Linear Logic inference rules defined in Figure 3.1 and implemented in Appendix A.1. The full detail of theorem proving in Coq are listed in Appendix B.5. The proof trees are shown in Appendix C.4.

```
Theorem 6.1.3.1 ship_order_extended :
(order_empty ⊗ (user ⊗ securitychecker) ⊗ (item ⊗ rate)
⊗ insurance ⊗ payment ⊗ shipment ) ⊢ order_shipped
```

`Proof. see Appendix` $B.3$. □

Composition at the Second Stage

At the second-stage service method level, the user security checking, product rating and insurance services are added, as shown in the following list. Similar to the expression in previous use cases, the operation actions are attached using ":: to represent the *processes* in the π-calculus. Variables are attached using ": to represent the *names* in the π-calculus. Again, as noted earlier in Case I, the successful HTTP status codes are removed, and the error HTTP status codes are represented by *err* in the expressions.

Security Checking Service

```
nurisecchec:urisecchec |- PostSecchec::nuriuid:(uriuid ⊕ err)
```

Product Rating Service

```
nurirate:urirate |- PostRate::nuririd:(uririd ⊕ err)
nuririd:uririd |- GetRid::nrrid:(rrid ⊕ err)
```

Product Insurance Service

```
nuriinsur:uriinsur |- PostInsure::nuriinid:(uriinid ⊕ err)
nuriinid:uriinid |- GetInid::nrinid:(rinid ⊕ err)
```

The overall representation transfer flows at this service method level are illustrated in Figure 6.8 and the business constraints are defined as follows. In order to show all seven services being used in this scenario, the *place_order* business action takes only valid users and rated items, and all orders have to be insured and paid via *insure_paid_order* or *pay_insured_order* before *ship_order*. For shipping the order, two links: *link_shiporder1* and *link_shiporder2*, are explicitly defined to show that the pre-shipped order may come from two different composition plans.

```
validate_user: nurisecchec:urisecchec, nruid:(ruid ⊕ err)
       |- VALIDUSER::nruidvalid:(ruidvalid ⊕ err)
```

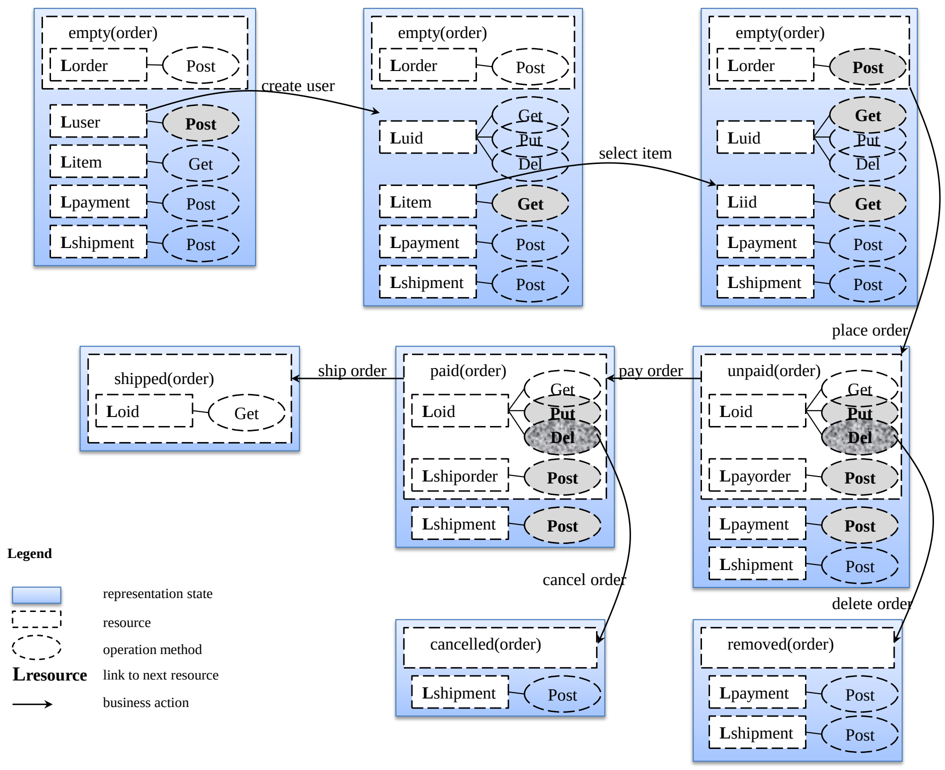

Figure 6.8: Case III flow at the service method level.

```
rate_item : nurirate:urirate, nriid:(riid ⊕ err)
       |- RATEITEM::nriidrated:(riidrated ⊕ err)
place_order: nuriorder:uriorder, nriidrated:(riidrated ⊕ err),
       nruidvalid:(ruidvalid ⊕ err)
       |- PLACEORDER::nroidunpaid(roidunpaid ⊕ err)
insure_unpaid_order : nuriinsur:uriinsur,
       nLinsureorder:(Linsureorder ⊕ err)
       |- INSUREUNPAIDORDER::nroidunpaidinsured
       :(roidunpaidinsured ⊕ err)
pay_order: nuripay:uripay, nruidpay:(ruidpay ⊕ err),
       nLpayorder:(Lpayorder ⊕ err)
       |- ORDERPAID::nroidpaid:(roidpaid ⊕ err)
insure_paid_order : nLinsureorder:(Linsureorder ⊕ err),
       nuriinsur:uriinsur
       |- INSUREUNPAIDORDER::nroidpaidinsured
       :(roidpaidinsured ⊕ err)
```

```
pay_insured_order: nuripay:uripay, nruidpay:(ruidpay ⊕ err),
       nLpayorder:(Lpayorder ⊕ err) |- ORDERINSUREDPAID
       ::nroidinsuredpaid:(roidinsuredpaid ⊕ err)
ship_order: nuriship:uriship, nLshiporder:(Lshiporder ⊕ err)
       |- ORDERSHIPPED::nroidshipped:(roidshipped ⊕ err)
link_order_insure: |- LINKORDERINSUR::nLorderinsure
       :((roidunpaid ⊕ err) ⊸ (Linsureorder ⊕ err))
link_order_pay: |- LINKORDERPAY::nLorderpay
       :((roidunpaid ⊕ err) ⊸ (Lpayorder ⊕ err))
link_after_insured_unpaid_order: |- LINKPAYORDER::nLpayorder
       :((roidunpaidinsured ⊕ err) ⊸ (Lpayorder ⊕ err))
link_after_order_pay: |- LINKPAIDORDERINSURE::nLpaidorderinsure
       :((roidpaid ⊕ err) ⊸ (Linsureorder ⊕ err))
link_shiporder1: |- LINKSHIPORDER1::nLshiporder:
       ((roidpaidinsured ⊕ err) ⊸ (Lshiporder ⊕ err))
link_shiporder2: |- LINKSHIPORDER2::nLshiporder:
       ((roidinsuredpaid ⊕ err) ⊸ (Lshiporder ⊕ err))
```

Theorem 6.1.3.2 provides the composition requirement that uses all of these services to obtain a shipped insured order. This composition is proved by applying the inference rules defined in Figure 4.3 and implemented in Appendix A.2. The full detail of theorem proving in Coq are listed in Appendix B.6. Appendix C.5 shows the proof trees for obtaining a shipped insured order.

```
Theorem 6.1.3.2 order_being_shipped_extended: exists P,
(((uriship<<nuriship) :: nil) ++ ((uripay<<nuripay) :: nil)
++ ((uriuidpay<<nuriuidpay) :: nil) ++ ((uriinsur<<nuriinsur) :: nil)
++ ((uriorder<<nuriorder) :: nil) ++ (((uriiid<<nuriiid) :: nil)
++ ((urirate<<nurirate) :: nil)) ++ (((uriuid<<nuriuid) :: nil))
++ ((urisecchec<<nurisecchec) :: nil))
|- ((roidshipped ⊕ err) <<nroidshipped[P).
```

Proof. see Appendix $B.6$. □

Listing 6.7: The π-calculus process model extracted for Case III of the e-shopping scenario.

```
1 (υy1)(((υnuriship)nuriship⟨nlshiporder⟩).(0
2     | ((υnroidinsuredpaid)(((υy2)(((υnuripay)nuripay⟨y3⟩.(0
3       | ((υnruidpay)nruidpay⟨nlpayorder⟩.(GETUIDPAY
4         | ((υnroidunpaidinsured)(((υnlinsureorder)
5           (((υnuriinsur)nuriinsur⟨nlorderinsure⟩.(0
6             | ((υnroidunpaid)
7               (((υy4)(((υnuriorder)nuriorder⟨y5⟩.(0
8                 | ((υnriidrated)nriidrated⟨nruidvalid⟩
9                   .(((υnriid)(GETIID | RATEITEM))
10                   | ((υnruid)(GETUID | VALIDUSER))))))
11                 | nuriorder(y6).nriidrated(nruidvalid).PLACEORDER)))
12               | LINKORDERINSURE)))))
13             | nuriinsur(nlinsureorder).INSUREUNPAIDORDER)))
14           | LINKPAYORDER))))))
15         | nuripay(y7).nruidpay(nlpayorder).PAYINSUREDORDER))
16       | LINKSHIPORDER2)))
17 | nuriship(nlshiporder).SHIPORDER)
```

The π-calculus process extracted from the above proof for an order being insured shipped is shown in Listing 6.7. Again, for legibility, the connectives are written in the style introduced in Chapter 4. The corresponding reference can be found in Table 5.1.

As shown in Listing 6.7, GETIID and GETUID processes can run in parallel as shown in the 9th and the 10th lines, in which GETIID runs with the RATEITEM process, and GETUID runs with the VALIDUSER process. Once the item and user information are retrieved, they are passed to the PLACEORDER process as shown in the 7th and the 11th lines. The LINKORDERINSURANCE process (see the 12th line) continues after the PLACEORDER process. After the insurance process at the 5th and the 13th lines, the LINKPAYORDER process is available together with user payment method information via the GETUIDPAY process, which invokes the PAYINSUREDORDER process at the 3^{rd} and the 15^{th} lines. Finally, the link to ship order process is available, the order is shipped as at the 1^{st} and the 17^{th} lines.

6.2 The Biomedical Service Composition Scenario

This section discusses how the proposed logical approach is used to compose services in real-world biomedical research. The scenario to be used here originates from one of the author's previous publications [37] which was based on a European Commission funded biomedical research project - The Living Human Digital Library (LHDL). In this project, the service composition has been remained mainly at the implementation level. This section re-visits the same scenario but focuses on the logical level modelling and composition.

In this scenario, a biomedical researcher would like to estimate the risk of bone fracture for a particular patient. The researcher has the 3D imaging data for the patient from Computed Tomography (CT) and a set of services for processing the image data. This thesis analyses a part of the flow that allows the researcher to build a 3D mesh from the CT data, as illustrated in Figure 6.9. After that, the researcher may continue to use motion-capture data to perform specific finite element simulations that require extensive computation, which is not discussed in this composition scenario.

The following lists the core services involved in this scenario in order to provide an overall picture of the scenario. However, it will not go into detail because this is beyond the scope of this thesis.

- *CT Service*: for managing the original 3D image data. It may have methods for obtaining and removing datasets.
- *Visualisation Tool Kit (VTK) Service*: for managing 3D image data in the VTK format. After the Importer Service, all 3D image data are represented in the VTK format to be used by other services. These VTK data are either used directly as 3D volumes or transferred to 3D

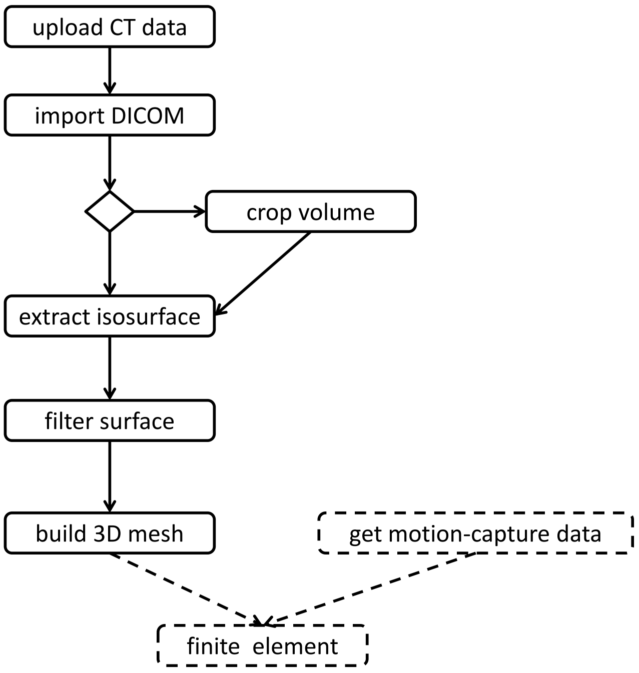

Figure 6.9: The overall flow of the services in the biomedical scenario modified from [37].

surfaces. Thus, in this scenario, VTK data may be referred as *vtkvolume* or *vtksurface*.

- *Importer Service*: for transforming the original image data into the VTK format. In this scenario, the Importer Service is mainly used to transform the original CT data in the Digital Imaging and Communications in Medicine (DICOM) format into the VTK format, and the results are VTK-compliant volume data.

- *Cropping Service*: for cropping image data to retain only the region of interest. The Cropping Service used in this scenario will mainly crop volume data according to the specific cropping parameters.

- *Isosurface Extractor Service*: for extracting isosurfaces from volume data, so the resulting data are surfaces.

- *Filter Service*: for filtering surfaces to create a new surface. Two types of filter are used in this scenario: decimation and smoothing. The surface decimation filter reduces the number of triangles in the surface but retains a good approximation to the original surface. Surface smoothing filter adjusts point coordinates using smoothing algorithms, such as Laplacian smoothing [88] to remove irregularities from the surface.

- *Motion-capture Data Service*: for managing data related to the movement of a patient, which is not used in the composition flow.

- *3D Mesh Service*: will produce the resulting composed service resource. It may have methods for obtaining and removing data.

The key resource in this scenario is the 3D dataset. The researcher initially has the CT data of the patient's femur as a DICOM file. An importer service converts the data from DICOM to a VTK volume dataset. If the data is large, it should be cropped to retain only the region of interest before processing; thus, a new VTK volume is created as output.

The researcher uses an *Isosurface Extractor* service, which accepts a vtk volume dataset as input and creates an isosurface as output.

The researcher builds a chain of services to filter the surface and improve its characteristics (smoothing, decimation). This processing chain may be saved by the researcher as a new service (which will then be available for use with other datasets on future occasions). The final surface dataset is created, which is used to build the 3D mesh.

To assess the risk of fracture, the researcher provides motion-capture data of stair climbing activities from a subject closely resembling his patient (same age, sex, physical characteristics) together with the pre-built 3D mesh to specific finite element simulations for further analysis.

Composition at the First Stage

At the first-stage abstract resource level, the available service resources are expressed as Linear Logic propositions as follows:

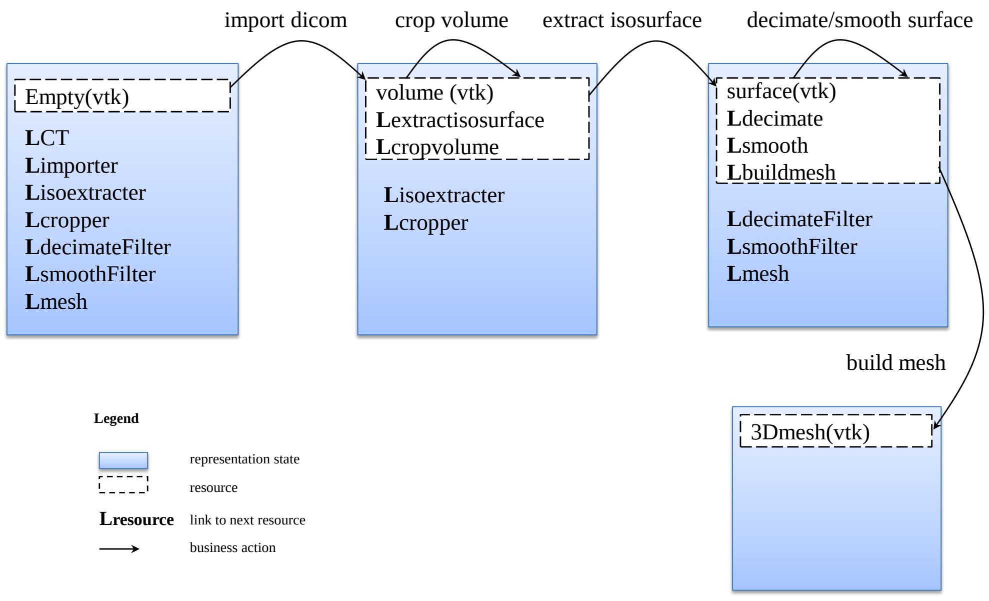

Figure 6.10: The 3D dataset resource flow in the biomedical scenario.

```
⊢CT, ⊢importer, ⊢isoextracter, ⊢smoothFilter, ⊢decimateFilter,
⊢mesh
```

The business constraints are modelled as Linear Logic sequent calculus as follows:

```
import_dicom: CT, importer ⊢ vtkvolume
extract_isosurface: lextractisosurface, isoextracter ⊢ vtksurface
crop_volume: vtkvolume, volumeCropper ⊢ vtkvolume
smooth_surface: lsmooth, smoothFilter ⊢ vtksurface
decimate_surface: ldecimate, decimateFilter ⊢ vtksurface
build_3Dmesh: lbuildMesh, mesh ⊢ 3DMesh
link_after_vloume: ⊢ vtkvolume
       ⊸ (lextractisosurface & lcropvolume)
link_after_surface: ⊢ vtksurface
       ⊸ (ldecimate & lsmooth & lbuildMesh)
```

Figure 6.10 shows the flow of the change to a 3D dataset during the execution in the service composition. When the original CT data (as in the

DICOM format) and the *importer* are available, the 3D dataset can be imported as *vtkvolume*. Two service links are possible for *vtkvolume* as defined in *link_after_volume*: *lextractisosurface* and *lcropvolume*. After the *extract_isosurface* action, a *vtksurface* is created that may have three links to the next service as defined in *link_after_surface*: *ldecimate*, *lsmooth* and *lbuildMesh*. Both *smooth_surface* and *decimate_surface* create new VTK surfaces. The *build_3Dmesh* creates a *3DMesh* from the available *vtksurface*.

Theorems 6.2.1 and 6.2.2 define two possible composition requirements: the first one produces a *3Dmesh* by importing the existing DICOM data and applying *isoextractor* and *decimateFilter* a single time, then using the motion capture data; the second one also produces a *3Dmesh* by importing the existing DICOM data, but as noticed in the theorem definition, the *decimateFilter* is applied twice before using the motion capture data. In the resource-sensitive Linear Logic, resources are defined as consumable, and the frequency of the resource usage, such as the *decimateFilter*, is explicitly expressed, which cannot be clearly emphasised in classical logic. Thus, if all *decimateFilter* used in these two theorems are given the same set of settings, the 3Dmesh results are different.

`Theorem 6.2.1: CT` $\otimes$ `importer` $\otimes$ `isoextracter` $\otimes$ `decimateFilter` $\otimes$ `mesh` $\vdash$ `3Dmesh`

`Proof. see Appendix` $B.7$. $\square$

`Theorem 6.2.2: CT` $\otimes$ `importer` $\otimes$ `isoextracter` $\otimes$ `decimateFilter` $\otimes$ `decimateFilter` $\otimes$ `mesh` $\vdash$ `3Dmesh`

`Proof. see Appendix` $B.7$. $\square$

These composition requirements are proved by applying the Intuitionistic Linear Logic inference rules defined in Figure 3.1 and implemented in Appendix A.1. The full detail of theorem proving in Coq are listed in Appendix B.7. For legibility, the proof is summarised as proof trees as shown in Appendix C.6 and C.7.

Composition at the Second Stage

At the second-stage service method level, the available services with their core methods are listed as follows. Again, the operation actions are attached using "":: to represent the *processes* in the π-calculus, and variables are attached using "": to represent the *names* in the π-calculus. As noted earlier in Case I of the e-shopping scenario, the successful HTTP status codes are removed, and the error HTTP status codes are represented by *err* in the expressions.

CT service

```
nurict:urict |- PostCt::nurictid:(urictid ⊕ err)
nurictid:urictid |- GetCtid::nrctid:(rctid ⊕ err)
```

Importer service

```
nuriimp:uriimp |- PostImporter::nrvtkvol:(rvtkvol ⊕ err)
```

Isoextractor service

```
nuriiso:uriiso |- PostIsoex::nrvtksur:(rvtksur ⊕ err)
```

Decimate filter

```
nuridec:uridec |- PostDecimate::nrvtksur:(rvtksur ⊕ err)
```

Smooth filter service

```
nurismo:urismo |- PostSmooth::nrvtksur:(rvtksur ⊕ err)
```

3D Mesh service

```
nurimesh:urimesh |- PostMesh::nrmesh:(rmesh ⊕ err)
```

The business constraints are defined as follows.

```
import_dicom: nuriimp:uriimp, nurictid:urictid
              |- ImportDicom::nrvtkvol:(rvtkvol ⊕ err)
extract_isosurface: nuriiso:uriiso, nlexiso:(lexiso ⊕ err)
              |- ExtractIso::nrvtksur:(rvtksur ⊕ err)
crop_volume: nuricrop:uricrop, nlcropvol:(lcropvol ⊕ err)
              |- CropVol::nrvtkvol:(rvtkvol ⊕ err)
```

```
decimate_surface: nuridec:uridec, nldec:(ldec ⊕ err)
              |- Deciamte::nrvtksur:(rvtksur ⊕ err)
smooth_surface: nurismo:urismo, nlsmo:(lsmo ⊕ err)
              |- Smooth::nrvtksur:(rvtksur ⊕ err)
build_mesh: nurimesh:urimesh, nlmesh:(lmesh ⊕ err)
              |- Mesh::nrmesh:(rmesh ⊕ err)
link_after_volume: |- (((rvtkvol ⊕ err) ⊸ LinfAfterVol::nlonvol
              :((lexiso ⊕ err) & (lcropvol ⊕ err)))
link_after_surface: |- (((rvtksur ⊕ err) ⊸ LinkAfterSur::nlonsur
              :((ldec ⊕ err) & (lsmo ⊕ err) & (lmesh ⊕ err)))
```

Theorem 6.2.3 below provides one composition requirement for obtaining a 3D mesh from the existing CT data and *extract isosurface* and *decimate surface* services. This composition is proved by applying the inference rules defined in Figure 4.3 and implemented in Appendix A.2. The full detail of theorem proving in Coq are listed in Appendix B.8. Appendix C.8 shows the proof trees constructed for obtaining a 3Dmesh.

```
Theorem 6.2.3 mesh_being_built: exists P,
       ((urimesh<<nurimesh) :: nil) ++ ((uridec<<nuridec) :: nil)
       ++ ((uriiso<<nuriiso) :: nil) ++ ((uriimp<<nuriimp) :: nil)
       ++ ((urictid<<nurictid) :: nil)
       |- ((rmesh ⊕ err) <<nrmesh[P).

Proof. see Appendix B.8.  □
```

The π-calculus process extracted from the above proof for a 3Dmesh is shown in Listing 6.8. Again, for legibility, the connectives are written in the style introduced in Chapter 4. The corresponding reference can be found in Table 5.1.

As shown in Listing 6.8, the LINKAFTERVOL process is available after IMPORTDICOM as shown in the 4^{th} line. The EXTRACTISO process is then invoked (see the 3^{rd} and the 5^{th} lines), and after that the LINKAFTER-SUR process is available (the 6^{th} line). Following the designated business constraints, the DECIMATE process is then performed, which makes the

LINKAFTERSUR process available. Finally, the surface information are passed to build the 3D mesh (see the 1st and the 8th lines).

Listing 6.8: The π-calculus process model extracted for the biomedical case.

```
1 (υnlonsur) ((υnurimesh)nurimesh⟨nlonsur⟩.(0
2    | ((υnlonsur)((υnuridec)nuridec⟨nlonsur⟩.(0
3      | ((υnlonvol)((υnuriiso)nuriiso⟨nlonvol⟩.(0
4        | ((υnrvtkvol) (IMPORTDICOM | LINKAFTERVOL)))
5      | ((υnrvtksur)(nuriiso(nlexiso).EXTRACTISO
6        | LINKAFTERSUR)))))
7      | ((υnrvtksur)(nuridec(nldec).DECIMATE | LINKAFTERSUR)))
8    | nurimesh(nlmesh).MESH)
```

In summary, the proposed Linear Logic theorem proving has been successfully applied to composing services in the context of a real biomedical scenario, which formed part of a recent successfully completed European project. In that project, the analysis was performed informally and involved the use of ad hoc procedures; if the approach advocated in this thesis had been available at the time, much less experimentation would have been required and a greater level of rigour could have been applied to the processes involved.

Taking advantage of recent developments in distributed systems and Web technologies, scientific and technological applications are increasingly using Web Services within a distributed computing environment, and as a result, increasing numbers of relevant Web Services are becoming available to support research in these fields. By providing a sound and rigorous method to support service composition, the proposed logical approach can potentially contribute not only to service composition relevant to the business community, where the issues originally arose, but also more generally to the broad range of activities in which web services are currently applied.

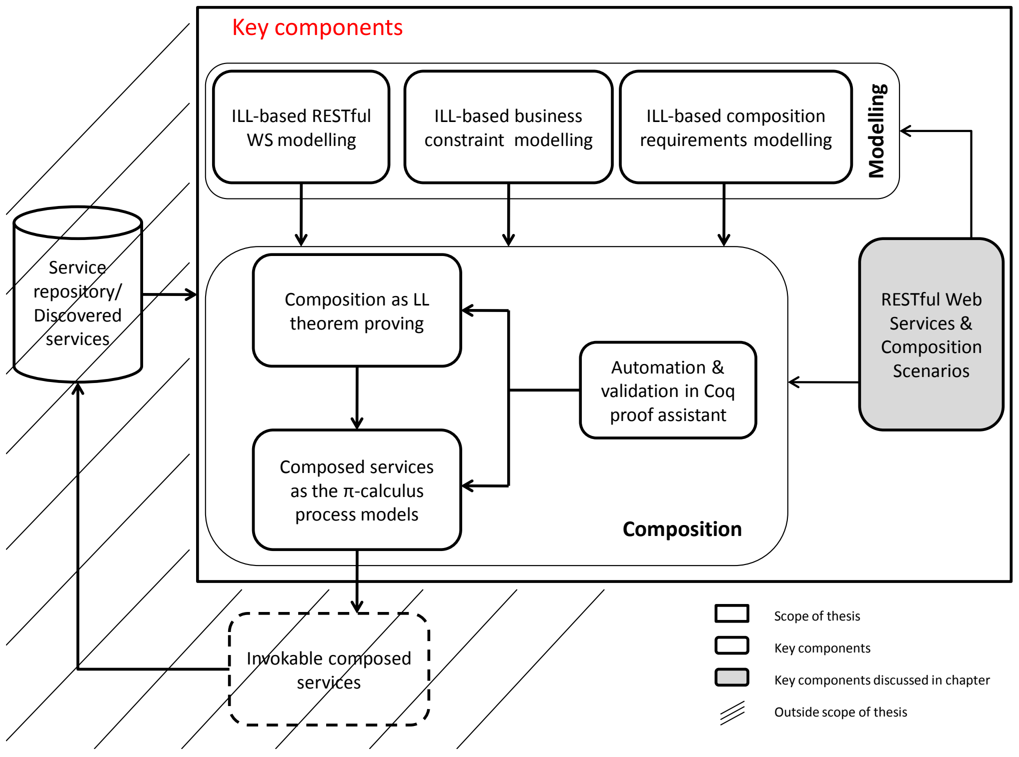
Key components
ILL-based RESTful WS modelling
ILL-based business constraint modelling
ILL-based composition requirements modelling
Modelling
Service repository/ Discovered services
Composition as LL theorem proving
Automation & validation in Coq proof assistant
Composed services as the π-calculus process models
Composition
RESTful Web Services & Composition Scenarios
Invokable composed services
Scope of thesis
Key components
Key components discussed in chapter
Outside scope of thesis

Figure 6.11: Key components discussed in Chapter 6.

6.3 Summary

This chapter has demonstrated the versatility of the proposed Linear Logic approach in modelling and composing RESTful Web Services through several case studies. Reflecting on the overall proposed approach, Figure 6.11 highlights the component discussed in this chapter.

Two use scenarios have been discussed in this chapter to illustrate that the logic approach and the tool-supported validation can be used to model and consolidate RESTful Web Service composition in real cases in different domains. Resources, service methods, business constraints and composition requirements are modelled in Linear Logic, and the encoding in the Coq theorem prover has efficiently assisted the theorem proving process for searching the proofs of the composed process and extracting the process models in the π-calculus. In particular, the detailed studies of the three use cases in the e-shopping scenario have provided an evaluation testbed in terms of the scalability of real number of services, service categories and composition scenarios.

The next chapter will fully evaluate the scalability and effectiveness of this logic approach taken in these case studies and will compare this approach with related methods mentioned in the literature review.

Chapter 7

Evaluation

This chapter provides an evaluation of the Linear Logic based RESTful Web Service modelling and composition approach proposed in this thesis. The evaluation will be performed from several aspects including evaluating the performance based on the use cases discussed in Chapter 6, comparing the proposed approach with the methods mentioned in Chapter 2 and evaluating the answers to the main research questions posed in Chapter 1. This chapter also points out the limitations of the proposed approach.

7.1 Performance Evaluation

Since this research investigates if the Linear Logic based approach is feasible to model and compose RESTful Web Services, it has focused on presentation RESTful Web Services using the existing Linear Logic fragments and the design of a generic method for composing RESTful Web Services based on Linear Logic theorem proving. The research has taken advantages of a well-developed existing theorem prover, namely the Coq proof assistant, at the implementation level, so, the performance of the composition task depends greatly on the performance of Coq and the predefined business scenarios.

Because Coq is a tactic-style semi-automatic theorem prover, the theorem proving time is affected mainly by the number of tactics used to complete the proof. The encoding in this thesis writes the application of one tactic as one

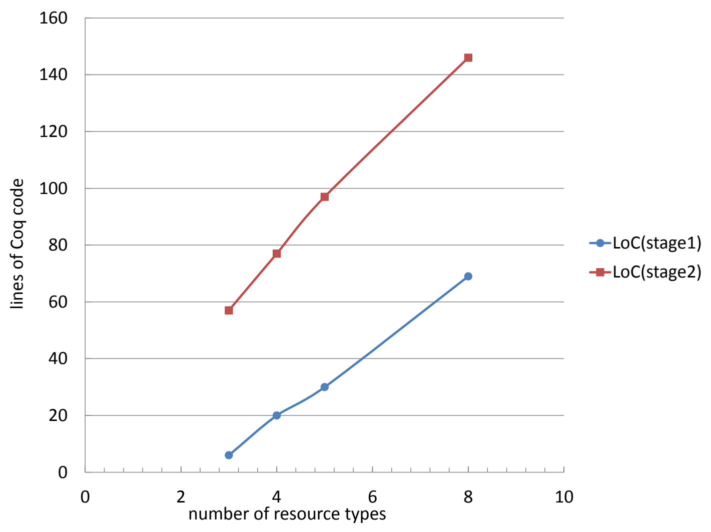

Figure 7.1: Number of service resource groups vs. lines of Coq code.

line of code, so the measurement will consider how the lines of Coq code (LoC) develop with the changes of service resources.

Experiments were built using the use cases discussed in Chapter 6 to test if the performance of the proposed method is sufficient for a real-time composition scenario. Although it is possible to increase the number of existing service resources to 100 or even 1000 and simulate them in theorem proving, the selection of the service resources in the actual theorem proving mainly depends on the complexity of the business scenarios (i.e. the business constraints among these service resources). Meanwhile, in real-world service composition scenarios, we rarely see one composition scenario that includes 100 or even 50 different types of existing service, so, it is believed that experiment analysis based on the use cases (with a maximum of 8 types of service resource and a maximum of 30 choices of resource in each type) discussed in this thesis provides a satisfactory indicator for the performance of the method.

Figure 7.1 shows the trends of the lines of Coq code associated with an increasing number of resource types for both stages. At each stage, the number of services has been chosen as 3, 4, 5, and 8 as presented in the e-shopping use cases. Stage 2 has a significant number of LoCs compared to Stage 1 because extra lines of code are required for extracting the π-calculus process model

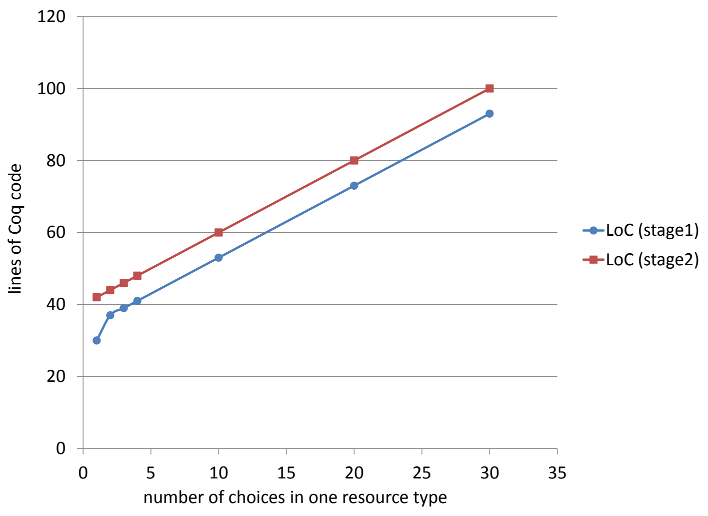

Figure 7.2: Number of resource choices in one resource groups vs. lines of Coq code.

during each step of logic inferencing. Overall, both stages show the linear trend of LoC with an increasing number of resource types, which indicates that RESTful Web Service composition based on Linear Logic theorem proving within the Coq proof assistant has reasonable scalability.

Figure 7.2 shows the trends of the lines of Coq code with regard to an increasing number of service resources in each resource type. In reality, it is common to have a number of services that provide similar functionalities, for example there are tens or maybe hundreds of online book selling services that allow one to buy the same titled book, so, in this analysis, the number of services in each resource type starts with 2, 3, 4 then increases to 10, 20 and 30. For both stages, when the number of choices in each resource type is more than 3, every new resource choice adds typically 2 extra lines of code for inferencing with the *With*(&) rule. The linear incrementation of LoC with the increasing number of choices in each resource type again indicates that RESTful Web Service composition based on Linear Logic theorem proving within the Coq proof assistant has reasonable scalability.

7.2 Comparison with Other Methods

Although RESTful Web Services have been used widely in implementation, associated research, especially in the areas of modelling and composition, is still under-developed. Compared to the other modelling and composition methods discussed in Chapter 2, the Linear Logic based approach proposed in this thesis has the following characteristics.

- It is the first logic-based approach to addressing both modelling and composition of RESTful Web Services.
- The resource-sensitive Linear Logic provides richer semantic connectives to model the key elements of RESTful Web Services and most constraints defined by the REST architecture style.
- The formalisation of state transition systems based on Linear Logic offers a promising way to explicitly model representation state transfer in RESTful Web Services.
- RESTful Web Service composition via Linear Logic theorem proving guarantees the completeness and correctness of the resulting composed services.
- The adoption of the proof-as-process paradigm with the π-calculus bridges the gap between formalisation and execution in service composition.
- The implementation of theorem proving in the semi-automated Coq proof assistant takes the overall composition approach one step closer to the ultimate goal of full automation.

The remainder of this section provides the comparison with exisiting modelling and composition methods.

7.2.1 Comparison with Other Modelling Methods

This section compares the proposed Linear Logic based modelling approach to others discussed in Chapter 2 with respect to the six REST constraints.

Table 7.1: Summary of the Linear Logic approach for RESTful Web system modelling.

Client-Server	Modelling service request and response in sequent calculus: Request $\vdash$ Response
Stateless	Not using Linear Logic exponentials and modelling key service elements as consumable Linear Logic resources
Cache	Modelling the non-functional properties, such as cache size, cache duration, as consumable Linear Logic resources in the service request
Code on Demand	Modelling client scripts as linear resources to be consumed in order to transfer from one state to another
Identification of resources	Modelling resources by URI as Linear Logic propositions
Resource manipulation via representations	Representations are modelled in both service request and response
Self-descriptive messages	Not using Linear Logic exponentials for modelling resource representations, media types, links, link types and link relations
HATEOAS	Modelling initial state with one-side sequent ($\vdash \Delta$), modelling state transition with two-side sequent ($\Gamma \vdash \Delta$) and modelling links with Linear Implication ($\multimap$)

The key characteristics that make Linear Logic a good candidate for modelling RESTful Web Services are its resource-sensitive nature and its ability to model state transition systems explicitly. Table 7.1 summarises the Linear Logic approach, which can be used together with Table 2.2 for the comparison.

Firstly, in the Linear Logic approach, there is no need to introduce extra elements in order to model the client and server interaction and the state transition. The sequent turnstile ($\vdash$) can clearly express the request and response interaction between client and server, and the Linear Implication ($\multimap$) can clearly indicate the next state for transition. In the existing approaches discussed in Chapter 2, the π-calculus approach [36] naturally represents the client and server interaction through sending request messages and receiving response messages. The FSM approach [34] has to introduce ε-transition to represent the interaction between client and server. In approaches based on

Petri-nets, such as service nets [35] and REST chart [3], transitions have to be defined explicitly. Other approaches do not explicitly address the client-server constraint.

Secondly, the proposed Linear Logic approach adapted in this thesis addresses the statelessness constraint in a straightforward way by not using the exponentials, such as of-course (!) and why-not(?), so resources represented are consumed once only. In this way, all invocations between client and server are naturally treated separately in the new sessions. Only two existing approaches, which also consider RESTful systems as state transition systems, address this statelessness constraint, but both of them have to introduce a particular state mechanism to achieve that. The FSM approach [34] stores the current state of the system, and the REST chart approach [3] introduces a stationary place for storing states.

Thirdly, Linear Logic is able to model services' non-functional properties as abstract consumable resources, such as cache size and duration although more technical detail have to be considered in the real Web applications. None of existing modelling approaches have addressed any non-functional constraints of REST.

Fourthly, the Linear Logic approach is capable of modelling the optional code-on-demand constraint for RESTful Web applications with the sequent turnstile ($\vdash$). Only the FSM approach [34] explicitly addresses this constraint with the use of ε-transition.

Fifthly, the chosen propositional Intuitionistic Linear Logic in the sequent calculus form can explicitly support the uniform interface constraints in all of its four principles. Linear Logic models all key service elements as propositions, including resource representation, resource identifier, media types, links, link types, link relations, which provides an abstract view of the service resource. The application state transfer is naturally modelled as Linear Logic in two-side sequent calculus and the potential links within the resource representation are explicitly modelled by the Linear Implication connective. Three existing approaches, namely ReLL [33], FSM [34] and ontology [37], model these four

Table 7.2: Comparison of the proposed logical method with other RESTful Web Service composition approaches.

	Automation	Scalability	Execution	Correctness
Pautasso [40]		Average	$\surd$	
Yu et al. [41]		Average	$\surd$	
Bite [42]		Average	$\surd$	
Rauf et al. [32]		Low		
Zhao and Doshi [37]	$\surd$	Good		$\surd$
Alarcon et al. [43]		Low		$\surd$
Logical method in this thesis	Semi-automatic	Good	Towards execution with extracted π-calculus process	$\surd$

principles to a certain degree, another three, namely REST Chart [3], Service nets [35] and the π-calculus [36], do not explicitly model the self-descriptive message principle, and one approach (i.e. the UML approach [31, 32]) introduced extended WADL to describe RESTful services which clearly violate the self-descriptive concept.

7.2.2 Comparison with Other Composition Methods

This section compares the proposed Linear Logic theorem proving approach to other composition methods discussed in Chapter 2 with respect to four composition criteria: automation, scalability, execution and correctness (see Table 7.2).

Firstly, the proposed theorem proving approach has considered automation as an ultimate goal. The method implemented in this thesis has achieved semi-automated composition by applying Linear Logic theorem proving in the style of program synthesis and encoding the complete theorem proving process in the Coq proof assistant. Although the program synthesis approach has the potential to achieve full automation, for a practical implementation, there are limited tools for achieving automation in all stages of the synthesis including automated requirement analysis and modelling, fully automated theorem proving and automated transformation to executable languages. Only AI-planning approaches, such as situation calculus [37] and the service net [43], have automation as a goal, but neither of them provide a complete automated composition environment.

Both situation calculus and the service net consider RESTful Web Services as state transition systems. Unlike the situation calculus approach using fluents, the resource-sensitive characteristic and the sequent calculus expression allow the proposed Linear Logic approach to model state transition systems more naturally without using extra elements to represent the transition for state change. Compared with the other non-logic formalisations, Linear Logic is more abstract and is able to model many other formal languages, such as Petri-nets. The abstract logic-level approach with proofs ensures the completeness of the composition outcome.

Secondly, as shown in Section 7.1, the proposed Linear Logic theorem proving approach and the implementation in Coq of the extraction to the π-calculus being implemented provide good level of scalability as the number of resource types and resources increases. Formal mathematics and logic based approaches, such as situation calculus, tend to provide better scalability compared with those relying on diagrams, such as approaches based on UML and Petri-nets, comprehensive omission from the existing research work surveyed in Chapter 2 is a discussion on the scalability of their performance.

Thirdly, the proposed research considers service execution as an important factor in the composition process. Although the implementation in this thesis has not produced a final composed result in a fully executable language, the automatic extraction of the π-calculus process models from Linear Logic theorem proving has enabled the proposed approach to move one step closer to the executable level, with the guarantee of composition completeness and correctness. Among the existing research, only the workflow-based approaches directly focus on service composition at the executable level. Other approaches, such as model-driven and AI-planning, consider the execution requirements, they leave the detailed implementation to future work.

Fourthly, the proposed composition approach, which uses theorem proving and proof-as-process based on propositional Linear Logic and the π-calculus, ensures two important elements of service composition: completeness and correctness. The completeness and the soundness rules from propositional logic guarantee that a composed service will be found if it exists, and once a proof is obtained through Linear Logic theorem proving, it is guaranteed that the corresponding composition is achieved correctly. Moreover, with the process model extraction in the formal π-calculus, the composition result is further verified, so the correctness of the composition outcome is doubly certain. In comparison, none of existing methods discussed in Chapter 2 guarantees the completeness of the composition, though the AI-planning approach can ensure the correctness of the composed service through verification by the formalism itself. Workflow-based and model-driven approaches do not provide mechanisms to ensure correctness by themselves, and extra work, such as model checking or formal verification, have to be performed to verify the correctness of the outcome.

7.3 Answers to Research Questions

The following summarises the answers for the underlying research questions posed in Chapter 1.

Research question 1: What are RESTful Web Services and why is formalising them necessary?

- Due to the current over-use of the term "REST", it is necessary to define what should exactly be referred to as RESTful Web Services. Chapter 2 has discussed RESTful Web Services as services/APIs that follow the principles of the REST architecture style. These services should be defined in a declarative resource-oriented way with at least the characteristics of addressability, connectivity, statelessness and uniform interfaces. This thesis has distinguished RESTful Web Services from those Web Services over HTTP that do not follow the REST principles. Thus in this research RESTful Web Services are considered to be those that follow the REST principles.

- Chapters 1 and 2 have pointed out that some so-called RESTful Web Services are merely those implemented over HTTP without considering the REST principles proposed initially in [1], which has caused variation in the implementations of RESTful Web Services. It is important to have techniques that can guide the implementation of RESTful Web Services within one concept. Studies on current modelling approaches in Chapter 2 suggests that formal modelling is a technique that can achieve this purpose, because formal models can not only intuitively express fundamental principles but also keep knowledge focused by omitting unnecessary information during the formalisation.

Research question 2: What are the current methods for modelling and composing RESTful Web Services and what are their pros and cons?

- Chapter 2 has provided a survey on the existing modelling approaches and has evaluated them based on the key principles of the REST architecture style. The survey shows that although there are different approaches to modelling RESTful Web systems, none of them can fully express the principles of the REST, so investigating other modelling approaches is still necessary. All existing approaches ignore the non-functional type of principles such as cache and layered-system. Apart from the FSM approach presented in [34], all other existing approaches have not modelled the services in clear correspondence with the REST principles. Approaches such as the UML modelling presented by [31] still favour the introduction of service description files (e.g. WADL) without treating RESTful Web Services in the declarative resource-oriented style. Table 2.2 shows that the formal method approaches including FSM, service nets, REST chart and the π-calculus have a better capability for expressing the overall principles. The results of this survey motivated this research to continue investigating a formal method to model RESTful Web Services, thus a Linear Logic approach has been proposed, to our knowledge, as the first logic-based approach to address the modelling of RESTful Web Services.

- Chapter 2 also provided a survey on existing approaches to composing RESTful Web Services, as well as a comparison of these methods based on a number of service composition criteria: automation, scalability, execution and correctness. This study showed that research on composing RESTful Web Services is still under-explored, and most of the current approaches are still at their initial stages. There has been no detailed evaluation in such work, and the study performed in this research is the first one to summarise and compare them. Furthermore, this survey found that current approaches are either working at the executable level without a correctness guarantee or focusing on correctness and automation without connecting to the executable level. This lack of connection between formal methods and executable languages has motivated this research to investigate a method that will behave as follows: 1) it should be able to perform at the level of the formal method to verify the correctness of the resulting composed service; 2) it should be able to drive the composition towards automation; and 3) the formalism should be capable of being transformed into an executable language for implementation. In response to these, a program synthesis approach based on Linear Logic and the proof-as-process paradigm with the π-calculus was investigated in this research.

Research question 3: Is it feasible to model RESTful Web Services in a Linear Logic framework and how can this be achieved?

- On one hand, this research has viewed RESTful Web Services from the perspective of being parts of a system, in which services are presented by resources that are manipulated through representation state transfer. RESTful Web Services, together with the communications among them, can be summarised as state transition systems as discussed in Chapter 2. On the other hand, the resource-sensitive Linear Logic is well suited to explicitly expressing state transition systems, as pointed out in Chapter 3. Hence, Linear Logic can be a good candidate for modelling RESTful Web Services and representing the composition communications.

- RESTful Web Services themselves have been modelled in Chapter 3 with regard to the key elements discussed in the REST architecture style in Chapter 2. This research has focused on the use of the propositional Linear Logic, and Linear Logic written in the sequent style has been used throughout the thesis. The sequent turnstile ($\vdash$) indicates the resource representations transitioning from one state to another, and the Linear Implication ($\multimap$) connective has been used to explicitly express the hyperlinks within the resource representations. When working on the composition aspect, existing RESTful Web Services are modelled as Linear Logic axioms. Chapter 4 has also shown ways to model business constraints/actions as Linear Logic hypotheses and composition requirements as Linear Logic theorems.

Research question 4: How can RESTful Web Services be composed by a Linear Logic based approach?

- At the logic level, the existing service resources are expressed as Linear Logic axioms, the business constraints among the services are described as Linear Logic hypotheses, and the composition requirements are described as Linear Logic theorems. Chapters 3 and 4 have provided guidelines for the translation from RESTful Web Services to Linear Logic expressions.

- The foundation of the proposed RESTful Web Service composition approach is deductive program synthesis via Linear Logic theorem proving. Deductive program synthesis observes proofs as equivalent to programs because each step of a proof can be interpreted as a step of a computation, which transforms the problems of software composition or program synthesis into a theorem proving task. This research has translated the characteristics of RESTful Web Services and the hyperlinks among the service resources into Linear Logic expressions, then used Linear Logic theorem proving for searching and forming services that satisfy the composition requirements specified.

- The research performed transforms logic models into the process models via the proof-as-process paradigm. The process models are generated in the π-calculus. Chapter 4 pointed out that the proofs produced by Linear Logic can guarantee that the outcome of the composed service is correct, which is the reason for not modelling RESTful Web Services and their compositions directly from the π-calculus. During theorem proving, the process information is attached to the logical formulae as proof terms. The original inference rules presented in Figure 3.1 have been studied from the point of view of giving each inference rule a concrete computational interpretation in the context of RESTful Web Service composition. Thus, a set of inference rules with proof terms attached were presented in Figure 4.3. They are used during Linear Logic theorem proving to construct the π-calculus process models from the steps of the proof.

- The research adopted the Coq proof assistant to implement the entire theorem proving and to facilitate the automation of the composition. In Chapter 5, Linear Logic, the π-calculus and the inference rules were encoded in the Coq proof assistant. Coq ensures that the Linear Logic theorem proving will perform with the behaviour expected. Although Coq is not a fully-automated theorem prover, the encoding has shown that its tactic style definition enables the theorem to be proved semi-automatically, which also facilitates moving the composition process towards automation.

Research question 5: How does this logical approach compare with other existing modelling and composition approaches?

- Section 7.2 of this chapter has conducted two sets of comparison for the proposed Linear Logic approach against existing modelling and composition approaches surveyed in Chapter 2. In terms of RESTful Web Service modelling corresponding to the REST architecture constraints, nevertheless, none approach discussed in this thesis can address all six constraints, the proposed Linear Logic approach has shown the capability of modelling five of them: client-server, statelessness, cache, code on demand and uniform interface.

- In terms of RESTful Web Service composition, the comparison is performed with respect to four important composition criteria: automation, scalability, execution and correctness. As discussed in Section 7.2, existing approaches studied in Chapter 2 address some of the criteria but none of them can cover all; and more research work is still required in this area. Although further work is still required for the proposed logical and proof-as-process composition approach to achieve the ultimate level of all these four criteria, the work performed in this thesis provides a feasible approach that can run in semi-automated tool supported environment, has good level of scalability, has correctness guarantee, and can produce process models for possible transformation to executable level languages.

7.4 Limitations

This research concentrated purely on how a Linear Logic based approach would benefit the modelling and composition of RESTful Web Services, so it did not address issues such as user authorisation/authentication, or the detail of how services are invoked or discovered.

While this research provides a possible approach to creating executable programs from the abstract logic level via the connection to the π-calculus, it does not include detailed examples for transforming the resulting π-calculus process model to any particular executable language. The main reason for this is that existing executable languages for RESTful Web Services are not sufficiently mature indeed, many of them are still under development. One business execution language that may be considered is the BPEL extension for REST mentioned in literature, but a comprehensive study is needed to decide on its suitability.

The research in this thesis has remained at the formal method level including the implementation as formal in the semi-automated Coq proof assistant, which provides good support for high-level service analysis and correct service composition. However, it requires users to have a reasonable knowledge of the underlying logic used, as well as the theorem prover. This may represent an obstacle for Web engineers when considering this approach in practice. Further research is required in order to make the whole approach more user friendly, such as providing an inter-layer to hide the detail of theorem proving at the back end but still to offer users opportunities to specify services and constraints in a user friendly environment.

7.5 Summary

This chapter has evaluated the Linear Logic based RESTful Web Service modelling and composition method proposed in this thesis. The evaluation was conducted by summarising how the research questions introduced in Chapter 1 are answered by the thesis, by examining the scalability performance of the proposed method, and by comparing the proposed approach to other existing modelling and composition methods.

This chapter also highlighted the limitations of the proposed logic-based approach. Some issues, such as translating the resulting π-calculus model to a specific executable language and providing user friendly access will be investigated in future work.

Chapter 8

Conclusion and Future Work

This thesis has proposed a formal approach, based on Linear Logic, to modelling and composing RESTful Web Services. This approach uses the set of semantic connectives provided by Linear Logic to model most of the architecture constraints defined by REST and uses the inference rule driven Linear Logic theorem proving to compose RESTful Web Services. The proposed approach was conducted semi-automatically in the Coq proof assistant and its versatility was demonstrated by being applied to a number of real-world use cases. The evaluation showed that this composition method scales well as the number of services and resources grows.

This chapter concludes this thesis by summarising the research performed, highlighting the key contributions and listing future research in the related area.

8.1 Thesis Summary and Contributions

This research concentrated on formalising RESTful Web Services rather than the traditional RPC-style Big Web Services because the popularity of RESTful Web Services is growing for implementation, despite a lack of formal research on its models and compositions, and this lack of formalism has been seen as a serious obstacle to Web engineers implementing proper and robust RESTful Web Services.

From the literature review conducted in Chapter 2, it can be seen that formal methods are important for abstracting service resources during modelling and ensuring correct outcomes during service composition. This thesis took advantage of the resource-sensitive nature of Linear Logic and its close relationship to the π-calculus process model to present the first logic-based method to address the modelling and composition issues of RESTful Web Services.

Being aware of the trade-offs that are necessary between the expressiveness and efficiency and the completeness of the usage of logic, this research selected the propositional fragment of Linear Logic to ensure the completeness of the resulting composed services and chose the semi-automatic Coq proof assistant to allow the whole method to be rigorously expressed in a theorem prover. This provides efficiency and completeness for the proof searching needed in the composition process while reducing "human-in-the-loop" activities, thus moving the whole process closer to full automation.

Modelling RESTful Web Services using propositional Intuitionistic Linear Logic was presented in Chapter 3. The modelling method specially referred to the 6 constraints (5 compulsory and 1 optional) defined by the REST architecture style discussed in Chapter 2. The proposed Linear Logic based method explicitly modelled four compulsory constraints: client-server, statelessness, cache and uniform interface. Because of the Linear Implication ($\multimap$) connective and the general expression of Linear Logic in sequent calculus, this method is particularly good at modelling hyperlinks and state transitions, known as a sub constraint - HATEOAS within the uniform interface constraint.

The composition method was presented in Chapter 4 as follows: 1) two-stage Intuitionistic Linear Logic theorem proving was proposed; 2) a backward reasoning method was introduced to decompose the desired composed service during proofing; 3) the π-calculus was attached as type terms in each ILL inference rule. The major advantage of using two stages is to increase the proof search efficiency especially when the composition requirement is complicated and the number of services is high. The first stage, at the abstract resources level, would determine if the existing types of resource are sufficient to accomplish a given composition requirement. If no complete proof is found, it saves the effort of performing detailed theorem proving at the concrete service level. Thus, the second stage determines if the existing resources can be planned to achieve the composition requirements. The advantage of using backward reasoning is to minimise the resource search effort during proving by decomposing the single composition goal into resources and matching them to the existing resources. The advantage of adding the π-calculus into Linear Logic inference rules is that process models can be directly extracted during the second stage theorem proving, and because of the close relationship between the π-calculus and business process executable languages, the gap between the logic level and the executable level will be largely reduced.

Chapter 5 encoded ILL connectives, its inference rules and its attachments with the π-calculus and performed theorem proving in a semi-automatic theorem prover - the Coq proof assistant. Considering the trade-offs discussed earlier, the main advantages of choosing Coq over other theorem provers are that both ILL and the π-calculus can be suitably encoded on top of the Coq system while using the theorem prover facilities to ensure that the proof is performed correctly, and the tactic-style proving provided by Coq provides a certain level of automation, though the user still has some control over the proving.

The thesis provided a feasibility study based on four use cases in two real-world scenarios in Chapter 6 and presented a scalability evaluation in Chapter 7. The proposed logical composition approach is not only capable of addressing typical service composition scenarios, such as the commercial e-shopping scenario, but also feasible for tackling non-trivial scientific examples, such as a real-world scenario within an European Commission funded biomedical project. The results showed that the proposed logic-based approach can successfully represent possible resource relationships during composition, such as the sequence of resource introduction (by sequent turnstile $\vdash$ or Linear Implication $\multimap$), resource combination (by Multiplicative Conjunction $\otimes$), choice of resources by users (by Additive Conjunction &), and service exceptions (by

Additive Disjunction $\oplus$). The composition method also scales well when the number of services and service types grows.

The contributions of this thesis are summarised as follows. Firstly, a novel logic-based approach was developed, the first of its kind, for the purposes of modelling and composing RESTful Web Services. Secondly, the proof-as-process paradigm using Linear Logic and the π-calculus was used to perform service composition, which not only ensures the completeness and the correctness of the resulting composed services but also produces their process models naturally, providing the possibility to translate them into executable business/programming languages. Thirdly, the proposed composition method was successfully implemented in the Coq proof assistant, which allows both Linear Logic theorem proving and the π-calculus extraction to be conducted semi-automatically. Fourthly, scenario-based feasibility studies were performed, and the method showed good scalability when the number of services and resources grows.

8.2 Future Work

Although the proposed method demonstrated that Linear Logic can be a good approach to the modelling and composition of RESTful Web Services, further research is required for improvement. The following provides a list of possible future research directions.

- **An executable composition engine for RESTful Web Services.** The implementation described in Chapter 5 has kept at the logic level. The full implementation of an executable engine which can work efficiently for Web engineers is far from complete. Two main steps have to be completed in order to achieve that.

  (i) Defining methods that enable the resulting π-calculus process to be translated into a form of executable language. Previous work on translating the π-calculus to BPEL [76] exists, but BPEL was introduced for process-oriented RPC-style Web Services, so it is not reasonable for RESTful Web Services to be translated into BPEL. Although the BPEL extension for REST method discussed in Chapter 2 provides an approach that allows REST to be embedded into BPEL, further justification is required to decide to translate the π-calculus to the BPEL extension for REST or to investigate new executable languages.

(ii) Developing an application, ideally a Web application, that wraps the Coq proving process to the back end and presents the executable result directly to users. In this way, Web engineers can easily define service resources and business constraints as well as obtain the resulting proofs without deep knowledge of the underlying logic and theorem provers.

- **Semantic models for service resource.** The composition approach proposed in this thesis has considered using the types of service resource for the first stage composition search and the concrete service resources for the second stage. Because the focus of the research is on the feasibility of Linear Logic, it has not provided detail of how the types of resource and resources themselves are specified and discovered during the composition search. In real-world applications, it is important to specifically know the semantics of the resources and their types in order to choose the correct ones during the composition search. It would be valuable to investigate Semantic Web techniques and embed them into the current approach. In this way, the resources could be better identified by the use of semantics and more accurately chosen during theorem proving for composition.

- **Exploration of the use of first order Linear Logic.** The propositional Linear Logic used in this research is suitable for modelling the type of resource as well as the resources at a more abstract level, with a completeness guarantee, but it has less expressiveness regarding the detail of the resources. Whereas, high-order logics such as first-order

Linear Logic will provide more expressive power when modelling service resources, they do not guarantee completeness and they may be less efficient for service composition. The research performed in this thesis has demonstrated the feasibility of applying Linear Logic, so from the logical perspective, it would be worthwhile investigating different fragments of Linear Logic in order to obtain the best results for both RESTful Web Service modelling and composition.

Appendix A

Encodings in Coq

A.1 Encoding Intuitionistic Linear Logic in Coq

```
Require Import Utf8_core.

Require Import List.

(* Encoding Linear Logic connectives *)

Inductive ILinProp : Set :=

| Implies: (ILinProp) → (ILinProp) → ILinProp

| One: ILinProp

| Plus: (ILinProp) → (ILinProp) → ILinProp

| Times: (ILinProp) → (ILinProp) → ILinProp

| Top: ILinProp

| With: (ILinProp) → (ILinProp) → ILinProp

| Zero: ILinProp

.

Reserved Notation "x ⊢ y" (at level 85, no associativity).

Infix " ⊗" := Times (at level 80).

Infix "&" := With (at level 80).

Infix "⊕" := Plus (at level 80).

Infix "⊸" := Implies (at level 80).
```

(Encoding Intuitionistic Linear Logic Inference rules *)*

`Inductive` *LinCons* : (*list ILinProp*) $\rightarrow$ *ILinProp* $\rightarrow$ `Prop` :=

| *Identity* (A : *ILinProp*) : $((A{::}nil) \vdash A)$

| *Exchange* ($A\ B\ G$: *ILinProp*) (Γ : *list ILinProp*) : $((\Gamma +\!\!+ (A{::}nil) +\!\!+ (B{::}nil)) \vdash G) \rightarrow ((\Gamma +\!\!+ (B{::}nil) +\!\!+ (A{::}nil)) \vdash G)$

| *Cut* ($A\ G$: *ILinProp*)($\Gamma 1\ \Gamma 2$: *list ILinProp*) : $((\Gamma 1 \vdash A) \rightarrow ((\Gamma 2 +\!\!+ (A{::}nil)) \vdash G) \rightarrow ((\Gamma 1 +\!\!+ \Gamma 2) \vdash G))$

| *ImpliesLeft* ($A\ B\ G$: *ILinProp*) ($\Gamma 1\ \Gamma 2$: *list ILinProp*) : $((\Gamma 1 \vdash A) \rightarrow ((\Gamma 2 +\!\!+ (B{::}nil)) \vdash G) \rightarrow (((\Gamma 1 +\!\!+ \Gamma 2) +\!\!+ ((A \multimap B){::}nil)) \vdash G))$

| *ImpliesRight* ($A\ G$: *ILinProp*) (Γ : *list ILinProp*) : $((\Gamma +\!\!+ (A{::}nil) \vdash G) \rightarrow (\Gamma \vdash (A \multimap G)))$

| *TimesLeft* ($A\ B\ G$: *ILinProp*) (Γ : *list ILinProp*) : $(\Gamma +\!\!+ ((A{::}nil) +\!\!+ (B{::}nil)) \vdash G \rightarrow (\Gamma +\!\!+ ((A \otimes B){::}nil)) \vdash G)$

| *TimesRight* ($A\ B$: *ILinProp*) ($\Gamma 1\ \Gamma 2$: *list ILinProp*) : $((\Gamma 1 \vdash A) \rightarrow (\Gamma 2 \vdash B) \rightarrow ((\Gamma 1 +\!\!+ \Gamma 2) \vdash (A \otimes B)))$

| *WithLeft1* ($A\ B\ G$: *ILinProp*) (Γ : *list ILinProp*) : $(((\Gamma +\!\!+ (A{::}nil)) \vdash G) \rightarrow (((\Gamma +\!\!+ (A \mathbin{\&} B) :: nil)) \vdash G))$

| *WithLeft2* ($A\ B\ G$: *ILinProp*) (Γ : *list ILinProp*) : $(((\Gamma +\!\!+ (B{::}nil)) \vdash G) \rightarrow (((\Gamma +\!\!+ (A \mathbin{\&} B) :: nil)) \vdash G))$

| *WithRight* ($A\ B$: *ILinProp*) (Γ : *list ILinProp*): $((\Gamma \vdash A) \rightarrow (\Gamma \vdash B) \rightarrow (\Gamma \vdash (A \mathbin{\&} B)))$

| *PlusLeft* $(A\ B\ G : \mathit{ILinProp})\ (\Gamma : \mathit{list\ ILinProp}) : ((\Gamma \mathbin{++} (A{::}\mathit{nil})) \vdash G) \to ((\Gamma \mathbin{++} (B{::}\mathit{nil})) \vdash G) \to ((\Gamma \mathbin{++} ((A \oplus B){::}\mathit{nil})) \vdash G)$

| *PlusRight1* $(A\ B : \mathit{ILinProp})\ (\Gamma : \mathit{list\ ILinProp}) : (\Gamma \vdash A) \to (\Gamma \vdash (A \oplus B))$

| *PlusRight2* $(A\ B : \mathit{ILinProp})\ (\Gamma : \mathit{list\ ILinProp}) : (\Gamma \vdash B) \to (\Gamma \vdash (A \oplus B))$

| *AssociateLeft* $(A\ B\ G : \mathit{ILinProp})\ (\Gamma : \mathit{list\ ILinProp}) : (((\Gamma \mathbin{++} (A :: \mathit{nil})) \mathbin{++} (B :: \mathit{nil})) \vdash G) \to ((\Gamma \mathbin{++} (A :: \mathit{nil}) \mathbin{++} (B :: \mathit{nil})) \vdash G)$
where "x ⊢ y" := $(\mathit{LinCons}\ x\ y)$
.

`Lemma` *AddNilLeft* $(A : \mathit{ILinProp})\ (\Gamma : \mathit{list\ ILinProp})$:
 $(((\mathit{nil} \mathbin{++} \Gamma) \vdash A) \to (\Gamma \vdash A))$.

```
Proof.
   intros.
   apply H.
Qed.
```

`Lemma` *RemoveNilLeft* $(A : \mathit{ILinProp})\ (\Gamma : \mathit{list\ ILinProp})$:
 $((\Gamma \vdash A) \to ((\mathit{nil} \mathbin{++} \Gamma) \vdash A))$.

```
Proof.
   intros.
   apply H.
Qed.
```

`Lemma` *AddNilRight* $(A : \mathit{ILinProp})\ (\Gamma : \mathit{list\ ILinProp})$:
 $((\Gamma \mathbin{++} \mathit{nil}) \vdash A) \to (\Gamma \vdash A))$.

```
Proof.
   intros.
```

```
    replace Γ with (Γ ++ nil).
    apply H.
    elim Γ.
    reflexivity.
    simpl.
    intros.
    rewrite H0.
    reflexivity.
Qed.
```

A.2 Encoding Intuitionistic Linear Logic and the π-caluclus in Coq

```
Require Import List.
Require Import Setoid.
Parameter name : Set.
(* Encoding the π-calculus syntax *)
    Inductive proc : Set :=
| skip : proc
| nu : name → proc → proc
| tau_pref : proc → proc
| par : proc → proc → proc
| sum : proc → proc → proc
| inp : name → name → proc → proc
| outp : name → name → proc → proc
.
(* Encoding the π-calculus attachments to Linear Logic *)
    Inductive ILinProp : Set :=
| Implies: (ILinProp) → (ILinProp) → (ILinProp)
| Plus: (ILinProp) → (ILinProp) → ILinProp
| Times: (ILinProp) → (ILinProp) → ILinProp
| With: (ILinProp) → (ILinProp) → ILinProp
| AddName: ILinProp → name → ILinProp
| AddProc: ILinProp → proc → ILinProp
.

Infix "⊗" := Times (at level 80).
Infix "&" := With (at level 80).
Infix "⊕" := Plus (at level 80).
Infix "⊸" := Implies (at level 80).
Infix "≪" := AddName (at level 75).
Infix "[" := AddProc (at level 75).

Reserved Notation "x ⊢ y" (at level 85, no associativity).

(* Encoding the Intuitionistic Linear Logic inference rules with the π-calculus
attachments *)
Inductive LinCons : (list ILinProp) → ILinProp → Prop :=
| Identity (A : ILinProp) (x : name): (((A≪x)::nil) ⊢ (A≪x[skip))

| Exchange (A B G : ILinProp) (Γ : list ILinProp) (x y z : name) (P Q :
proc):
    ((Γ ++ ((A≪x)::nil) ++ ((B≪y)::nil))
    ⊢ (G≪z[P)) → ((Γ ++ ((B≪y)::nil) ++ ((A≪x)::nil)) ⊢ (G≪z[P))

| Cut (A G : ILinProp)(Γ1 Γ2 : list ILinProp) (x y z : name) (P Q : proc):
    (∃ P, (Γ1 ⊢ (A≪x[P))) → (∃ Q, (Γ2 ++ ((A≪x)::nil)) ⊢ (G≪z[Q)) →
    (Γ1 ++ Γ2) ⊢ (G≪z[(nu x (par P Q)))

| TimesLeft (A B G : ILinProp)(Γ : list ILinProp)(x y z z1 : name)(P :
proc) :
    (∃ P, (Γ ++ ((A≪x)::nil)++ ((B≪y)::nil)) ⊢ (G≪z[P)) →
    (Γ ++ (((A ⊗ B)≪z1)::nil)) ⊢ (G≪z[(inp y x P))
```

| *TimesRight* (*A B* : *ILinProp*) (*Γ1 Γ2* : *list ILinProp*)(*x y z* : *name*) (*P Q*
:
proc) : (∃ *P*, (*Γ1* ⊢ (*A*≪*x*[*P*]))) → (∃ *Q*, (*Γ2* ⊢ (*B*≪*y*[*Q*]))) → (*Γ1* ++
Γ2) ⊢
((*A* ⊗ *B*)≪*z*[(*nu x* (*outp y x* (*par P Q*))))

| *ImpliesLeft* (*A B G* : *ILinProp*)(*Γ1 Γ2* : *list ILinProp*)(*x y*: *name*) (*P Q* :
proc):
(∃ *P*, (*Γ1* ⊢ (*A*≪*x*[*P*]))) → (∃ *Q*, (*Γ2* ++ ((*B*≪*y*)::*nil*)) ⊢ (*G*[*Q*]) →
(*Γ1* ++ *Γ2* ++ ((*A*⊸*B*)≪*y*)::*nil*) ⊢ (*G*[(*nu x* (*outp y x* (*par P Q*))))

| *ImpliesRight* (*A B* : *ILinProp*)(*Γ* : *list ILinProp*)(*x y*: *name*) (*P* : *proc*):
(∃ *P*, (*Γ* ++ (*A*≪*x*)::*nil*) ⊢ (*B*≪*y*[*P*]) → (*Γ* ⊢ ((*A*⊸*B*)≪*y*[(*inp y x*
P)))

| *Shift* (*A G* : *ILinProp*)(*Γ* : *list ILinProp*)(*x y*: *name*) (*P* : *proc*):
(∃ *P*, (*Γ* ⊢ ((*A*⊸*G*)≪*y*[*P*]))) → ((*Γ* ++ (*A*≪*x*)::*nil*) ⊢ (*G*≪*y*[*P*])

| *WithLeft1* (*A B G* : *ILinProp*) (: *list ILinProp*)(*x*: *name*)(*P* : *proc*):
(∃ *P*, ((++ ((*A*≪*x*)::*nil*)) ⊢ (*G*[*P*])) → ((++ (((*A* &&& *B*)≪*x*) ::
nil)) ⊢ (*G*[*P*])

| *WithLeft2* (*A B G* : *ILinProp*) (: *list ILinProp*)(*x*: *name*)(*P* : *proc*):
(∃ *P*, ((++ ((*B*≪*x*)::*nil*)) ⊢ (*G*[*P*])) → ((++ (((*A* &&& *B*)≪*x*) ::
nil)) ⊢ (*G*[*P*])

| *WithRight* (*A B* : *ILinProp*) (: *list ILinProp*)(*x*: *name*)(*P Q*: *proc*):
(∃ *P*, (⊢ (*A*≪*x*[*P*]))) → (∃ *Q*, (⊢ (*B*≪*x*[*Q*]))) → (⊢ ((*A* &&&
B)≪*x*[(*sum P Q*)))

| *PlusLeft* $(A\ B\ G : ILinProp)\ (\ : list\ ILinProp)(x{:}\ name)(P\ Q{:}\ proc)$:

$(\exists\ P, ((\ {+}{+}\ ((A{\ll}x){::}nil)) \vdash (G[P])) \to (\exists\ Q, ((\ {+}{+}\ ((B{\ll}x){::}nil)) \vdash (G[Q])) \to ((\ {+}{+}\ (((A \oplus B){\ll}x){::}nil)) \vdash (G{\ll}x[(sum\ P\ Q)))$

| *PlusRight1* $(A\ B : ILinProp)\ (\ : list\ ILinProp)(x{:}\ name)(P : proc)$:

$(\exists\ P, (\ \vdash (A{\ll}x[P]))) \to (\ \vdash ((A \oplus B){\ll}x[P])$

| *PlusRight2* $(A\ B : ILinProp)\ (\ : list\ ILinProp)(x{:}\ name)(P : proc)$:

$(\exists\ P, (\ \vdash (B{\ll}x[P]))) \to (\ \vdash ((A \oplus B){\ll}x[P])$

where ”x ⊢ y” := $(LinCons\ x\ y)$

.

`Lemma` *AddNilLeft* $(A : ILinProp)\ (\Gamma : list\ ILinProp)\ (x : name)\ (P : proc)$:

$(((nil\ {+}{+}\ \Gamma) \vdash (A{\ll}x[P])) \to (\Gamma \vdash (A{\ll}x[P])))$.

```
Proof.
   intros.
   apply H.
Qed.
```

`Lemma` *RemoveNilLeft* $(A : ILinProp)\ (\Gamma : list\ ILinProp)\ (x : name)\ (P : proc)$:

$((\Gamma \vdash (A{\ll}x[P])) \to ((nil\ {+}{+}\ \Gamma) \vdash (A{\ll}x[P])))$.

```
Proof.
   intros.
   apply H.
Qed.
```

`Lemma` *AddNilRight* $(A : ILinProp)\ (\Gamma : list\ ILinProp)\ (x : name)\ (P : proc)$:

$(((\Gamma\ {+}{+}\ nil) \vdash (A{\ll}x[P])) \to (\Gamma \vdash (A{\ll}x[P])))$.

```
Proof.
```

```
  intros.
  replace Γ with (Γ ++ nil).
  apply H.
  elim Γ.
  reflexivity.
  simpl.
  intros.
  rewrite H0.
  reflexivity.
Qed.
```

Appendix B

Use case implementations in Coq

B.1 E-shopping Scenario - Case I Resource Level Implementation in Coq

`Variable` *user item payment shipment order_empty order_unpaid order_paid order_shipped Lpayorder Lshiporder*: *ILinProp.*

`Hypothesis` *place_order* : $((order\_empty :: nil) ++ (user :: nil) ++ (item :: nil)) \vdash order\_unpaid.$

`Hypothesis` *pay_order* : $((Lpayorder :: nil) ++ (payment :: nil)) \vdash order\_paid.$

`Hypothesis` *ship_order* : $((Lshiporder :: nil) ++ (shipment :: nil)) \vdash order\_shipped.$

`Hypothesis` *link_payorder*: $nil \vdash order\_unpaid \multimap Lpayorder.$

`Hypothesis` *link_shiporder*: $nil \vdash order\_paid \multimap Lshiporder.$

`Theorem` *shipping_order* : $((order\_empty \otimes user \otimes item \otimes payment \otimes shipment) :: nil) \vdash order\_shipped.$

`Proof.`

`apply` *AddNilLeft.*

`apply` *TimesLeft.*

```
  apply AssociateLeft.
  apply Cut with Lshiporder.
  apply TimesLeft.
  apply AssociateLeft.
  apply Cut with order_unpaid.
  apply TimesLeft.
  apply Exchange.
  apply TimesLeft.
  apply Exchange.
  apply place_order.
  apply AddNilLeft.
  apply Exchange.
  apply RemoveNilLeft.
  apply Cut with Lpayorder.
  apply AddNilLeft.
  apply Shift.
  apply link_payorder.
  apply AddNilRight.
  apply Cut with order_paid.
  apply AddNilLeft.
  apply Exchange.
  apply pay_order.
  apply Shift.
  apply link_shiporder.
  apply AddNilLeft.
  apply Exchange.
  apply RemoveNilLeft.
  apply ship_order.
Qed.
```

B.2 E-shopping Scenario - Case I Service Method Level Implementation in Coq

Variable *uriuser uriuid ruid uidmsg uriuidpay ruidpay* : *ILinProp.*

Variable *uriitem uriiid ritem riid iidmsg* : *ILinProp.*

Variable *uripay uripayid rpayid* : *ILinProp.*

Variable *uriship urisid rsid* : *ILinProp.*

Variable *nuriuser nuriuid nruid nuidmsg nuriuidpay nruidpay* : *name.*

Variable *nuriitem nuriiid nritem nriid niidmsg* : *name.*

Variable *nuripay nuripayid nrpayid* : *name.*

Variable *nuriship nurisid nrsid* : *name.*

Variable *GETUID GETIID CREATEORDERURI PLACEORDER GETUIDPAY PAYSHIPORDER POSTORDER POSTOIDPAY POSTOIDSHIP POSTPAY GETPAYID POSTSHIP GETSID GETOID PUTOIDPAY PUTOIDSHIP PAYORDER SHIPORDER*: *proc.*

Variable *uriorder urioid uriorderpay roid roidpaid roidshipped urioidpay urioidship*: *ILinProp.*

Variable *nuriorder nurioid nuriorderpay nroid nroidpaid nroidshipped nurioidpay nurioidship moidpaymsg moidshipmsg*: *name.*

Variable *lpayorder lshiporder*: *ILinProp.*

Variable *nlpayorder nlshiporder*: *name.*

Variable *LINKPAYORDER LINKSHIPORDER*: *proc.*

Variable *err* : *ILinProp.*

Variable $P\ Q$: *proc.*

Variable $x\ y$: *name.*

Axiom *post_user* : $(uriuser :: nil) \vdash uriuid.$

Axiom *get_uid* : $(uriuid \ll nuriuid) :: nil \vdash ((ruid \oplus err) \ll nruid[GETUID).$

Axiom *put_uid* : $((uriuid :: nil) ++ (uidmsg :: nil)) \vdash ruid.$

Axiom *delete_uid* : $(uriuid :: nil) \vdash Zero.$

Axiom get_uid_pay : $((uriuidpay \ll nuriuidpay) :: nil)$

$\vdash ((ruidpay \oplus err) \ll nruidpay[GETUIDPAY).$

Axiom $post\_item$: $(uriitem :: nil) \vdash uriiid.$

Axiom get_item : $(uriitem :: nil) \vdash ritem.$

Axiom get_iid : $(uriiid \ll nuriiid) :: nil \vdash ((riid \oplus err) \ll nriid[GETIID).$

Axiom put_iid : $((uriiid :: nil) +\!+ (iidmsg :: nil)) \vdash riid.$

Axiom $post\_pay$: $((uripay \ll nuripay) :: nil)$

$\vdash ((uripayid \oplus err) \ll nuripayid[POSTPAY).$

Axiom get_payid : $((uripayid \ll nuripayid) :: nil)$

$\vdash ((rpayid \oplus err) \ll nrpayid[GETPAYID).$

Axiom $delete\_payid$: $(uripayid :: nil) \vdash Zero.$

Axiom $post\_ship$: $((uriship \ll nuriship) :: nil)$

$\vdash ((urisid \oplus err) \ll nurisid[POSTSHIP).$

Axiom get_sid : $((urisid \ll nurisid) :: nil) \vdash ((rsid \oplus err) \ll nrsid[GETSID).$

Axiom $post\_order$: $((uriorder \ll nuriorder) :: nil)$

$\vdash ((urioid \oplus err) \ll nurioid[POSTORDER).$

Axiom get_oid : $((urioid \ll nurioid) :: nil)$

$\vdash (((roid \multimap urioidpay) \oplus err) \ll nroid[GETOID).$

Axiom put_oid_pay : $((urioid \ll nurioid) :: nil +\!+ (rpayid \ll nrpayid) :: nil)$

$\vdash ((roidpaid \oplus err) \ll nroidpaid[PUTOIDPAY).$

Axiom put_oid_ship : $((urioid \ll nurioid) :: nil +\!+ (rsid \ll nrsid) :: nil)$

$\vdash ((roidshipped \oplus err) \ll nroidshipped[PUTOIDSHIP).$

Axiom $post\_oid\_ship$: $((urioidship \ll nurioidship) :: nil)$

$\vdash ((roidshipped \oplus err) \ll nroidshipped[POSTOIDSHIP).$

Hypothesis $place\_order$: $(((uriorder \ll nuriorder) :: nil)$

$+\!+ (((riid \oplus err) \ll nriid) :: nil)$

$+\!+ (((ruid \oplus err) \ll nruid) :: nil))$

$\vdash ((roid \oplus err) \ll nroid[PLACEORDER).$

Hypothesis pay_order: $(((uripay \ll nuripay) :: nil)$

$+\!+ (((ruidpay \oplus err) \ll nruidpay) :: nil)$

$++\ (((lpayorder \oplus err) \ll nlpayorder) :: nil))$

$\vdash ((roidpaid \oplus err) \ll\ nroidpaid[PAYORDER).$

`Hypothesis` *ship_order*: $(((uriship \ll nuriship) :: nil)$

$++\ (((lshiporder \oplus err) \ll nlshiporder) :: nil))$

$\vdash ((roidshipped \oplus err) \ll\ nroidshipped[SHIPORDER).$

`Hypothesis` *link_payorder*: *nil*

$\vdash (((roid \oplus err) \multimap (lpayorder \oplus err)) \ll nlpayorder[LINKPAYORDER).$

`Hypothesis` *link_shiporder*: *nil*

$\vdash (((roidpaid \oplus err) \multimap (lshiporder \oplus err)) \ll nlshiporder[LINKSHIPORDER).$

`Theorem` *order_being_shipped*: $\exists\ P,\ (((uriship \ll nuriship) :: nil)$

$++\ ((uripay \ll nuripay) :: nil)\ ++\ ((uriuidpay \ll nuriuidpay) :: nil)$

$++\ ((uriorder \ll nuriorder) :: nil)\ ++\ ((uriiid \ll nuriiid) :: nil)$

$++\ ((uriuid \ll nuriuid) :: nil)) \vdash ((roidshipped \oplus err) \ll nroidshipped[P).$

```
Proof.
  econstructor.
  apply AddNilRight.
  instantiate (1:= (nu y (par P Q))).
  apply Cut with (Times uriship (Plus lshiporder err)).
  auto.
  econstructor.
  instantiate (1:= (nu nuriship (outp nlshiporder nuriship (par P Q)))).
  apply TimesRight.
  econstructor.
  instantiate (1:= skip).
  apply Identity.
  econstructor.
  apply AddNilRight.
  instantiate (1:= (nu nroidpaid (par P Q))).
  apply Cut with (Plus roidpaid err).
  auto.
  econstructor.
```

```
apply AddNilRight.
instantiate (1:= (nu y (par P Q))).
apply Cut with (Times uripay (Times (Plus ruidpay err) (Plus lpayorder
err))).
auto.
econstructor.
instantiate (1:= (nu nuripay (outp y nuripay (par P Q)))).
apply TimesRight.
econstructor.
instantiate (1:= skip).
apply Identity.
econstructor.
instantiate (1:= (nu nruidpay (outp nlpayorder nruidpay (par P Q)))).
apply TimesRight.
econstructor.
instantiate (1:= GETUIDPAY).
apply get_uid_pay.
econstructor.
apply AddNilRight.
instantiate (1:= (nu nroid (par P Q))).
apply Cut with (Plus roid err).
auto.
econstructor.
apply AddNilRight.
instantiate (1:= (nu y (par P Q))).
apply Cut with (Times uriorder (Times (Plus riid err) (Plus ruid err))).
auto.
econstructor.
instantiate (1:= (nu nuriorder (outp y nuriorder (par P Q)))).
apply TimesRight.
econstructor.
```

```
instantiate (1:= skip).
apply Identity.
econstructor.
instantiate (1:= (nu nriid (outp nruid nriid (par P Q)))).
apply TimesRight.
econstructor.
instantiate (1:= GETIID).
apply get_iid.
econstructor.
instantiate (1:= GETUID).
apply get_uid.
econstructor.
instantiate (1:= (inp y nuriorder P)).
apply TimesLeft.
econstructor.
apply RemoveNilLeft.
instantiate (1:= (inp nruid nriid P)).
apply TimesLeft.
econstructor.
instantiate (1:= PLACEORDER).
apply place_order_.
econstructor.
instantiate (1:= P).
apply Shift.
econstructor.
instantiate (1:= LINKPAYORDER).
apply link_payorder.
econstructor.
instantiate (1:= (inp y nuripay P)).
apply TimesLeft.
econstructor.
```

```
   apply RemoveNilLeft.
   instantiate (1:= (inp nlpayorder nruidpay P)).
   apply TimesLeft.
   econstructor.
   instantiate (1:= PAYORDER).
   apply pay_order.
   econstructor.
   instantiate (1:= P).
   apply Shift.
   econstructor.
   instantiate (1:= LINKSHIPORDER).
   apply link_shiporder.
   econstructor.
   instantiate (1:= (inp nlshiporder nuriship P)).
   apply TimesLeft.
   econstructor.
   apply RemoveNilLeft.
   instantiate (1:= SHIPORDER).
   apply ship_order.
Qed.
```

B.3 E-shopping Scenario - Case II Resource Level Implementation in Coq

```
Variable user item paymentpaypal paymentcc paymentdc shipment order_empty
         order_unpaid order_paid order_shipped Lpayorderpaypal Lpayordercc
         Lpayorderdc Lshiporder: ILinProp.

Hypothesis place_order : ((item :: nil) ++ (user :: nil)
         ++ (order_empty :: nil)) ⊢ order_unpaid.
Hypothesis pay_order_paypal : ((Lpayorderpaypal :: nil)
```

$++ \ (paymentpaypal :: nil)) \vdash order\_paid.$

`Hypothesis` $pay\_order\_cc : ((Lpayordercc :: nil)$

$++ \ (paymentcc :: nil)) \vdash order\_paid.$

`Hypothesis` $pay\_order\_dc : ((Lpayorderdc :: nil)$

$++ \ (paymentdc :: nil)) \vdash order\_paid.$

`Hypothesis` $ship\_order : ((Lshiporder :: nil) ++ (shipment :: nil))$

$\vdash order\_shipped.$

`Hypothesis` $link\_payorder$: $nil \vdash order\_unpaid$

$\multimap (Lpayorderpaypal \ \& \ Lpayordercc \ \& \ Lpayorderdc).$

`Hypothesis` $link\_shiporder$: $nil \vdash order\_paid \multimap Lshiporder.$

`Theorem` $ship\_order\_paypal : nil \vdash ((((order\_empty \otimes user \otimes item \otimes$

$(paymentpaypal \ \& \ paymentcc \ \& \ paymentdc) \otimes shipment)))$

$\multimap order\_shipped).$

`Proof.`

`apply` *ImpliesRight.*

`apply` *TimesLeft.*

`apply` *AssociateLeft.*

`apply` *Cut* `with` *order_paid.*

`apply` *TimesLeft.*

`apply` *RemoveNilLeft.*

`apply` *Cut* `with` *order_unpaid.*

`apply` *AddNilLeft.*

`apply` *TimesLeft.*

`apply` *Exchange.*

`apply` *RemoveNilLeft.*

`apply` *TimesLeft.*

`apply` *Exchange.*

`apply` *place_order.*

`apply` *AddNilLeft.*

`apply` *Exchange.*

`apply` *RemoveNilLeft.*

apply *WithLeft1*.

apply *WithLeft1*.

apply *Cut* with $(Lpayorderpaypal \,\&\, Lpayordercc \,\&\, Lpayorderdc)$.

apply *AddNilLeft*.

apply *Shift*.

apply *link_payorder*.

apply *WithLeft1*.

apply *WithLeft1*.

apply *AddNilLeft*.

apply *Exchange*.

apply *RemoveNilLeft*.

apply *pay_order_paypal*.

apply *AddNilLeft*.

apply *Exchange*.

apply *RemoveNilLeft*.

apply *Cut* with *Lshiporder*.

apply *AddNilLeft*.

apply *Shift*.

apply *link_shiporder*.

apply *AddNilLeft*.

apply *Exchange*.

apply *ship_order*.

Qed.

Theorem *ship_order_creditcard* : $nil \vdash ((((order\_empty \otimes user \otimes item \otimes (paymentpaypal \,\&\, paymentcc \,\&\, paymentdc) \otimes shipment))) \multimap order\_shipped)$.

Proof.

apply *ImpliesRight*.

apply *TimesLeft*.

apply *AssociateLeft*.

apply *Cut* with *order_paid*.

```
apply TimesLeft.
apply RemoveNilLeft.
apply Cut with order_unpaid.
apply AddNilLeft.
apply TimesLeft.
apply Exchange.
apply RemoveNilLeft.
apply TimesLeft.
apply Exchange.
apply place_order.
apply AddNilLeft.
apply Exchange.
apply RemoveNilLeft.
apply WithLeft1.
apply WithLeft2.
apply Cut with (Lpayorderpaypal & Lpayordercc & Lpayorderdc).
apply AddNilLeft.
apply Shift.
apply link_payorder.
apply WithLeft1.
apply WithLeft2.
apply AddNilLeft.
apply Exchange.
apply RemoveNilLeft.
apply pay_order_cc.
apply AddNilLeft.
apply Exchange.
apply RemoveNilLeft.
apply Cut with Lshiporder.
apply AddNilLeft.
apply Shift.
```

apply *link_shiporder*.

apply *AddNilLeft*.

apply *Exchange*.

apply *ship_order*.

Qed.

Theorem *ship_order_debitcard* : $nil \vdash ((((order\_empty \otimes user \otimes item \otimes (paymentpaypal \,\&\, paymentcc \,\&\, paymentdc) \otimes shipment))) \multimap order\_shipped)$.

Proof.

apply *ImpliesRight*.

apply *TimesLeft*.

apply *AssociateLeft*.

apply *Cut* with *order_paid*.

apply *TimesLeft*.

apply *RemoveNilLeft*.

apply *Cut* with *order_unpaid*.

apply *AddNilLeft*.

apply *TimesLeft*.

apply *Exchange*.

apply *RemoveNilLeft*.

apply *TimesLeft*.

apply *Exchange*.

apply *place_order*.

apply *AddNilLeft*.

apply *Exchange*.

apply *RemoveNilLeft*.

apply *WithLeft2*.

apply *Cut* with $(Lpayorderpaypal \,\&\, Lpayordercc \,\&\, Lpayorderdc)$.

apply *AddNilLeft*.

apply *Shift*.

apply *link_payorder*.

```
    apply WithLeft2.
    apply AddNilLeft.
    apply Exchange.
    apply RemoveNilLeft.
    apply pay_order_dc.
    apply AddNilLeft.
    apply Exchange.
    apply RemoveNilLeft.
    apply Cut with Lshiporder.
    apply AddNilLeft.
    apply Shift.
    apply link_shiporder.
    apply AddNilLeft.
    apply Exchange.
    apply ship_order.
Qed.
```

B.4 E-shopping Scenario - Case II Service Method Level Implementation in Coq

```
Variable uriuser uriuid ruid uidmsg uriuidpay ruidpay : ILinProp.
Variable uriprod uriiid rprod riid iidmsg : ILinProp.
Variable uripay uripayid rpayid : ILinProp.
Variable uriship urisid rsid : ILinProp.

Variable nuriuser nuriuid nruid nuidmsg nuriuidpay nruidpay : name.
Variable nuriprod nuriiid nrprod nriid niidmsg : name.
Variable nuripay nuripayid nrpayid : name.
Variable nuriship nurisid nrsid : name.

Variable GETUID GETIID CREATEORDERURI PLACEORDER
         GETUIDPAYPAL GETUIDPAYCC GETUIDPAYDC PAYORDER
```

$SHIPORDER\ PAYSHIPORDER\ POSTORDER\ POSTOIDPAY$
$POSTOIDSHIP\ POSTPAY\ GETPAYID\ POSTSHIP\ GETSID$
$GETOID\ PUTOIDPAY\ PUTOIDSHIP$: $proc.$

`Variable` $uriorder\ urioid\ uriorderpay\ roid\ roidpaid\ roidshipped\ urioidpay$
$urioidship$: $ILinProp.$

`Variable` $nuriorder\ nurioid\ nuriorderpay\ nroid\ nroidpaid\ nroidshipped$
$nurioidpay\ nurioidship\ moidpaymsg\ moidshipmsg$: $name.$

`Variable` err : $ILinProp.$

`Variable` $uripaypal\ uripaycc\ uripaydc\ ruidpaypal\ ruidpaycc\ ruidpaydc$
$lpayorderpaypal\ lpayordercc\ lpayorderdc$: $ILinProp.$

`Variable` $nlpayorderpaypal\ nlpayordercc\ nlpayorderdc$: $name.$

`Axiom` $post\_user$: $(uriuser :: nil) \vdash uriuid.$

`Axiom` get_uid : $(uriuid \ll nuriuid) :: nil \vdash ((ruid \oplus err) \ll nruid[GETUID).$

`Axiom` put_uid : $((uriuid :: nil) ++ (uidmsg :: nil)) \vdash ruid.$

`Axiom` $delete\_uid$: $(uriuid :: nil) \vdash Zero.$

`Axiom` get_uid_paypal : $((uriuidpay \ll nuriuidpay) :: nil)$
$\vdash ((ruidpaypal \oplus err) \ll nruidpay[GETUIDPAYPAL).$

`Axiom` get_uid_paycc : $((uriuidpay \ll nuriuidpay) :: nil)$
$\vdash ((ruidpaycc \oplus err) \ll nruidpay[GETUIDPAYCC).$

`Axiom` get_uid_paydc : $((uriuidpay \ll nuriuidpay) :: nil)$
$\vdash ((ruidpaydc \oplus err) \ll nruidpay[GETUIDPAYDC).$

`Axiom` $post\_prod$: $(uriprod :: nil) \vdash uriiid.$

`Axiom` get_prod : $(uriprod :: nil) \vdash rprod.$

`Axiom` get_iid : $(uriiid \ll nuriiid) :: nil \vdash ((riid \oplus err) \ll nriid[GETIID).$

`Axiom` put_iid : $((uriiid :: nil) ++ (iidmsg :: nil)) \vdash riid.$

`Axiom` $post\_pay$: $((uripay \ll nuripay) :: nil)$
$\vdash ((uripayid \oplus err) \ll nuripayid[POSTPAY).$

`Axiom` get_payid : $((uripayid \ll nuripayid) :: nil)$
$\vdash ((rpayid \oplus err) \ll nrpayid[GETPAYID).$

`Axiom` $delete\_payid$: $(uripayid :: nil) \vdash Zero.$

Axiom $post\_ship$: $((uriship \ll nuriship) :: nil)$

$\vdash ((urisid \oplus err) \ll nurisid[POSTSHIP).$

Axiom get_sid : $((urisid \ll nurisid) :: nil) \vdash ((rsid \oplus err) \ll nrsid[GETSID).$

Axiom $post\_order$: $((uriorder \ll nuriorder) :: nil)$

$\vdash ((urioid \oplus err) \ll nurioid[POSTORDER).$

Axiom get_oid : $((urioid \ll nurioid) :: nil)$

$\vdash (((roid \multimap urioidpay) \oplus err) \ll nroid[GETOID).$

Axiom put_oid_pay : $((urioid \ll nurioid) :: nil ++ (rpayid \ll nrpayid) :: nil)$

$\vdash ((roidpaid \oplus err) \ll nroidpaid[PUTOIDPAY).$

Axiom put_oid_ship : $((urioid \ll nurioid) :: nil ++ (rsid \ll nrsid) :: nil)$

$\vdash ((roidshipped \oplus err) \ll nroidshipped[PUTOIDSHIP).$

Axiom $post\_oid\_ship$: $((urioidship \ll nurioidship) :: nil)$

$\vdash ((roidshipped \oplus err) \ll nroidshipped[POSTOIDSHIP).$

Variable $lpayorder\ lshiporder$: $ILinProp.$

Variable $nlpayorder\ nlshiporder$: $name.$

Variable $LINKPAYORDER\ LINKSHIPORDER\ PAYORDERPAYPAL$
$PAYORDERCC\ PAYORDERDC$: $proc.$

Variable $P\ Q$: $proc.$

Variable $x\ y$: $name.$

Hypothesis $place\_order\_$: $(((uriorder \ll nuriorder) :: nil) ++$

$(((riid \oplus err) \ll nriid) :: nil) ++ (((ruid \oplus err) \ll nruid) :: nil))$

$\vdash ((roid \oplus err) \ll nroid[PLACEORDER).$

Hypothesis pay_order: $(((uripay \ll nuripay) :: nil) ++$

$(((ruidpay \oplus err) \ll nruidpay) :: nil) ++$

$(((lpayorder \oplus err) \ll nlpayorder) :: nil))$

$\vdash ((roidpaid \oplus err) \ll nroidpaid[ORDERPAID).$

Hypothesis pay_order_paypal: $(((uripaypal \ll nuripay) :: nil) ++$

$(((ruidpaypal \oplus err) \ll nruidpay) :: nil) ++$

$(((lpayorderpaypal \oplus err) \ll nlpayorder) :: nil))$

$\vdash ((roidpaid \oplus err) \ll nroidpaid[PAYORDERPAYPAL).$

`Hypothesis` *pay_order_cc*: $(((uripaycc \ll nuripay) :: nil) ++$
$(((ruidpaycc \oplus err) \ll nruidpay) :: nil) ++$
$(((lpayordercc \oplus err) \ll nlpayorder) :: nil))$
$\vdash ((roidpaid \oplus err) \ll nroidpaid[PAYORDERCC).$

`Hypothesis` *pay_order_dc*: $(((uripaydc \ll nuripay) :: nil) ++$
$(((ruidpaydc \oplus err) \ll nruidpay) :: nil) ++$
$(((lpayorderdc \oplus err) \ll nlpayorder) :: nil))$
$\vdash ((roidpaid \oplus err) \ll nroidpaid[PAYORDERDC).$

`Hypothesis` *ship_order*: $(((uriship \ll nuriship) :: nil) ++$
$(((lshiporder \oplus err) \ll nlshiporder) :: nil))$
$\vdash ((roidshipped \oplus err) \ll nroidshipped[ORDERSHIPPED).$

`Hypothesis` *link_payorder*: $nil \vdash (((roid \oplus err) \multimap ((lpayorderpaypal \oplus err) \&$
$(lpayordercc \oplus err) \& (lpayorderdc \oplus err)))$
$\ll nlpayorder[LINKPAYORDER).$

`Hypothesis` *link_shiporder*: $nil \vdash (((roidpaid \oplus err) \multimap$
$(lshiporder \oplus err)) \ll nlshiporder[LINKSHIPORDER).$

`Theorem` *order_being_shipped_paypal*: $\exists\ P, (((uriship \ll nuriship) :: nil) ++$
$(((uripaypal \ \& \ uripaycc \ \& \ uripaydc) \ll nuripay) :: nil) ++$
$((uriuidpay \ll nuriuidpay) :: nil) ++ ((uriorder \ll nuriorder) :: nil) ++$
$((uriiid \ll nuriiid) :: nil) ++ ((uriuid \ll nuriuid) :: nil))$
$\vdash ((roidshipped \oplus err) \ll nroidshipped[P).$

`Proof`.

`econstructor`.

`apply` *AddNilRight*.

`instantiate` *(1:= (nu y (par P Q)))*.

`apply` *Cut* `with` *(Times uriship (Plus lshiporder err))*.

`auto`.

`econstructor`.

`instantiate` *(1:= (nu nuriship (outp nlshiporder nuriship (par P Q))))*.

`apply` *TimesRight*.

`econstructor`.

`instantiate` *(1:= skip).*

`apply` *Identity.*

`econstructor`.

`apply` *AddNilRight.*

`instantiate` *(1:= (nu nroidpaid (par P Q))).*

`apply` *Cut* `with` *(Plus roidpaid err).*

`auto`.

`econstructor`.

`apply` *AddNilRight.*

`instantiate` *(1:= (nu y (par P Q))).*

`apply` *Cut* `with` *(Times uripaypal (Times (Plus ruidpaypal err) (Plus lpay-orderpaypal err))).*

`auto`.

`econstructor`.

`instantiate` *(1:= (nu nuripay (outp y nuripay (par P Q)))).*

`apply` *TimesRight.*

`econstructor`.

`instantiate` *(1:= P).*

`apply` *AddNilLeft.*

`apply` *WithLeft1.*

`econstructor`.

`instantiate` *(1:= P).*

`apply` *WithLeft1.*

`econstructor`.

`instantiate` *(1:= skip).*

`apply` *Identity.*

`econstructor`.

`instantiate` *(1:= (nu nruidpay (outp nlpayorder nruidpay (par P Q)))).*

`apply` *TimesRight.*

`econstructor`.

`instantiate` *(1:= GETUIDPAYPAL).*

```
apply get_uid_paypal.
econstructor.
apply AddNilRight.
instantiate (1:= (nu nroid (par P Q))).
apply Cut with (Plus roid err).
auto.
econstructor.
apply AddNilRight.
instantiate (1:= (nu y (par P Q))).
apply Cut with (Times uriorder (Times (Plus riid err) (Plus ruid err))).
auto.
econstructor.
instantiate (1:= (nu nuriorder (outp y nuriorder (par P Q)))).
apply TimesRight.
econstructor.
instantiate (1:= skip).
apply Identity.
econstructor.
instantiate (1:= (nu nriid (outp nruid nriid (par P Q)))).
apply TimesRight.
econstructor.
instantiate (1:= GETPID).
apply get_iid.
econstructor.
instantiate (1:= GETUID).
apply get_uid.
econstructor.
instantiate (1:= (inp y nuriorder P)).
apply TimesLeft.
econstructor.
apply RemoveNilLeft.
```

`instantiate` *(1:= (inp nruid nriid P))*.

`apply` *TimesLeft*.

`econstructor`.

`instantiate` *(1:= PLACEORDER)*.

`apply` *place_order_*.

`econstructor`.

`apply` *RemoveNilLeft*.

`apply` *AddNilRight*.

`instantiate` *(1:= (nu nlpayorder (par P Q)))*.

`apply` *Cut* `with` *((lpayorderpaypal ⊕ err) & (lpayordercc ⊕ err) & (lpay-orderdc ⊕ err))*.

`auto`.

`econstructor`.

`apply` *AddNilLeft*.

`instantiate` *(1:= P)*.

`apply` *Shift*.

`econstructor`.

`instantiate` *(1:= LINKPAYORDER)*.

`apply` *link_payorder*.

`econstructor`.

`instantiate` *(1:= P)*.

`apply` *WithLeft1*.

`econstructor`.

`instantiate` *(1:= P)*.

`apply` *WithLeft1*.

`econstructor`.

`instantiate` *(1:= skip)*.

`apply` *Identity*.

`econstructor`.

`instantiate` *(1:= (inp y nuripay P))*.

`apply` *TimesLeft*.

econstructor.

apply *RemoveNilLeft.*

instantiate *(1:= (inp nlpayorder nruidpay P)).*

apply *TimesLeft.*

econstructor.

instantiate *(1:= PAYORDERPAYPAL).*

apply *pay_order_paypal.*

econstructor.

instantiate *(1:= P).*

apply *Shift.*

econstructor.

instantiate *(1:= LINKSHIPORDER).*

apply *link_shiporder.*

econstructor.

instantiate *(1:= (inp nlshiporder nuriship P)).*

apply *TimesLeft.*

econstructor.

apply *RemoveNilLeft.*

instantiate *(1:= ORDERSHIPPED).*

apply *ship_order.*

Qed.

Theorem *order_being_shipped_cc*: $\exists$ P, $(((uriship \ll nuriship) :: nil) +\!+$
$(((uripaypal \ \& \ uripaycc \ \& \ uripaydc) \ll nuripay) :: nil) +\!+$
$((uriuidpay \ll nuriuidpay) :: nil) +\!+ ((uriorder \ll nuriorder) :: nil) +\!+$
$((uriiid \ll nuriiid) :: nil) +\!+ ((uriuid \ll nuriuid) :: nil))$
$\vdash ((roidshipped \oplus err) \ll nroidshipped[P).$

Proof.

econstructor.

apply *AddNilRight.*

instantiate *(1:= (nu y (par P Q))).*

`apply` *Cut* `with` *(Times uriship (Plus lshiporder err)).*

`auto.`

`econstructor.`

`instantiate` *(1:= (nu nuriship (outp nlshiporder nuriship (par P Q)))).*

`apply` *TimesRight.*

`econstructor.`

`instantiate` *(1:= skip).*

`apply` *Identity.*

`econstructor.`

`apply` *AddNilRight.*

`instantiate` *(1:= (nu nroidpaid (par P Q))).*

`apply` *Cut* `with` *(Plus roidpaid err).*

`auto.`

`econstructor.`

`apply` *AddNilRight.*

`instantiate` *(1:= (nu y (par P Q))).*

`apply` *Cut* `with` *(Times uripaycc (Times (Plus ruidpaycc err) (Plus lpay-ordercc err))).*

`auto.`

`econstructor.`

`instantiate` *(1:= (nu nuripay (outp y nuripay (par P Q)))).*

`apply` *TimesRight.*

`econstructor.`

`instantiate` *(1:= P).*

`apply` *AddNilLeft.*

`apply` *WithLeft1.*

`econstructor.`

`instantiate` *(1:= P).*

`apply` *WithLeft2.*

`econstructor.`

`instantiate` *(1:= skip).*

```
apply Identity.
econstructor.
instantiate (1:= (nu nruidpay (outp nlpayorder nruidpay (par P Q)))).
apply TimesRight.
econstructor.
instantiate (1:= GETUIDPAYCC).
apply get_uid_paycc.
econstructor.
apply AddNilRight.
instantiate (1:= (nu nroid (par P Q))).
apply Cut with (Plus roid err).
auto.
econstructor.
apply AddNilRight.
instantiate (1:= (nu y (par P Q))).
apply Cut with (Times uriorder (Times (Plus riid err) (Plus ruid err))).
auto.
econstructor.
instantiate (1:= (nu nuriorder (outp y nuriorder (par P Q)))).
apply TimesRight.
econstructor.
instantiate (1:= skip).
apply Identity.
econstructor.
instantiate (1:= (nu nriid (outp nruid nriid (par P Q)))).
apply TimesRight.
econstructor.
instantiate (1:= GETPID).
apply get_iid.
econstructor.
instantiate (1:= GETUID).
```

```
apply get_uid.
econstructor.
instantiate (1:= (inp y nuriorder P)).
apply TimesLeft.
econstructor.
apply RemoveNilLeft.
instantiate (1:= (inp nruid nriid P)).
apply TimesLeft.
econstructor.
instantiate (1:= PLACEORDER).
apply place_order_.
econstructor.
apply RemoveNilLeft.
apply AddNilRight.
instantiate (1:= (nu nlpayorder (par P Q))).
apply Cut with ((lpayorderpaypal ⊕ err) & (lpayordercc ⊕ err) & (lpayorderdc ⊕ err)).
auto.
econstructor.
apply AddNilLeft.
instantiate (1:= P).
apply Shift.
econstructor.
instantiate (1:= LINKPAYORDER).
apply link_payorder.
econstructor.
instantiate (1:= P).
apply WithLeft1.
econstructor.
instantiate (1:= P).
apply WithLeft2.
```

```
    econstructor.
    instantiate (1:= skip).
    apply Identity.
    econstructor.
    instantiate (1:= (inp y nuripay P)).
    apply TimesLeft.
    econstructor.
    apply RemoveNilLeft.
    instantiate (1:= (inp nlpayorder nruidpay P)).
    apply TimesLeft.
    econstructor.
    instantiate (1:= PAYORDERCC).
    apply pay_order_cc.
    econstructor.
    instantiate (1:= P).
    apply Shift.
    econstructor.
    instantiate (1:= LINKSHIPORDER).
    apply link_shiporder.
    econstructor.
    instantiate (1:= (inp nlshiporder nuriship P)).
    apply TimesLeft.
    econstructor.
    apply RemoveNilLeft.
    instantiate (1:= ORDERSHIPPED).
    apply ship_order.
Qed.

Theorem order_being_shipped_dc: ∃ P, (((uriship≪nuriship) :: nil) ++
    (((uripaypal & uripaycc & uripaydc)≪nuripay):: nil) ++
    ((uriuidpay≪nuriuidpay) :: nil) ++ ((uriorder≪nuriorder) :: nil) ++
    ((uriiid≪nuriiid) :: nil) ++ ((uriuid≪nuriuid) :: nil))
```

$\vdash ((roidshipped \oplus err) \ll nroidshipped[P).$

`Proof.`

`econstructor.`

`apply` *AddNilRight.*

`instantiate` *(1:= (nu y (par P Q))).*

`apply` *Cut* `with` *(Times uriship (Plus lshiporder err)).*

`auto.`

`econstructor.`

`instantiate` *(1:= (nu nuriship (outp nlshiporder nuriship (par P Q)))).*

`apply` *TimesRight.*

`econstructor.`

`instantiate` *(1:= skip).*

`apply` *Identity.*

`econstructor.`

`apply` *AddNilRight.*

`instantiate` *(1:= (nu nroidpaid (par P Q))).*

`apply` *Cut* `with` *(Plus roidpaid err).*

`auto.`

`econstructor.`

`apply` *AddNilRight.*

`instantiate` *(1:= (nu y (par P Q))).*

`apply` *Cut* `with` *(Times uripaydc (Times (Plus ruidpaydc err) (Plus lpay-orderdc err))).*

`auto.`

`econstructor.`

`instantiate` *(1:= (nu nuripay (outp y nuripay (par P Q)))).*

`apply` *TimesRight.*

`econstructor.`

`instantiate` *(1:= P).*

`apply` *AddNilLeft.*

`apply` *WithLeft2.*

```
econstructor.
instantiate (1:= skip).
apply Identity.
econstructor.
instantiate (1:= (nu nruidpay (outp nlpayorder nruidpay (par P Q)))).
apply TimesRight.
econstructor.
instantiate (1:= GETUIDPAYDC).
apply get_uid_paydc.
econstructor.
apply AddNilRight.
instantiate (1:= (nu nroid (par P Q))).
apply Cut with (Plus roid err).
auto.
econstructor.
apply AddNilRight.
instantiate (1:= (nu y (par P Q))).
apply Cut with (Times uriorder (Times (Plus riid err) (Plus ruid err))).
auto.
econstructor.
instantiate (1:= (nu nuriorder (outp y nuriorder (par P Q)))).
apply TimesRight.
econstructor.
instantiate (1:= skip).
apply Identity.
econstructor.
instantiate (1:= (nu nriid (outp nruid nriid (par P Q)))).
apply TimesRight.
econstructor.
instantiate (1:= GETPID).
apply get_iid.
```

```
econstructor.
instantiate (1:= GETUID).
apply get_uid.
econstructor.
instantiate (1:= (inp y nuriorder P)).
apply TimesLeft.
econstructor.
apply RemoveNilLeft.
instantiate (1:= (inp nruid nriid P)).
apply TimesLeft.
econstructor.
instantiate (1:= PLACEORDER).
apply place_order_.
econstructor.
apply RemoveNilLeft.
apply AddNilRight.
instantiate (1:= (nu nlpayorder (par P Q))).
apply Cut with ((lpayorderpaypal ⊕ err) & (lpayordercc ⊕ err) & (lpayorderdc ⊕ err)).
auto.
econstructor.
apply AddNilLeft.
instantiate (1:= P).
apply Shift.
econstructor.
instantiate (1:= LINKPAYORDER).
apply link_payorder.
econstructor.
instantiate (1:= P).
apply WithLeft2.
econstructor.
```

```
    instantiate (1:= skip).
    apply Identity.
    econstructor.
    instantiate (1:= (inp y nuripay P)).
    apply TimesLeft.
    econstructor.
    apply RemoveNilLeft.
    instantiate (1:= (inp nlpayorder nruidpay P)).
    apply TimesLeft.
    econstructor.
    instantiate (1:= PAYORDERDC).
    apply pay_order_dc.
    econstructor.
    instantiate (1:= P).
    apply Shift.
    econstructor.
    instantiate (1:= LINKSHIPORDER).
    apply link_shiporder.
    econstructor.
    instantiate (1:= (inp nlshiporder nuriship P)).
    apply TimesLeft.
    econstructor.
    apply RemoveNilLeft.
    instantiate (1:= ORDERSHIPPED).
    apply ship_order.
Qed.
```

B.5 E-shopping Scenario - Case III Resource Level Implementation in Coq

`Variable` *user user_valid item rate item_rated payment securitychecker shipment insurance order_empty order_unpaid order_unpaid_insured order_paid order_paid_insured order_shipped order_insured_paid Linsureorder Lpayorder Lshiporder*: *ILinProp*.

`Hypothesis` *validate_user* : $((user :: nil) ++ (securitychecker :: nil)) \vdash user\_valid.$

`Hypothesis` *rate_item* : $((item :: nil) ++ (rate :: nil)) \vdash item\_rated.$

`Hypothesis` *place_order* : $((item\_rated :: nil) ++ (user\_valid :: nil) ++ (order\_empty :: nil)) \vdash order\_unpaid.$

`Hypothesis` *insure_unpaid_order* : $((Linsureorder :: nil) ++ (insurance :: nil)) \vdash order\_unpaid\_insured.$

`Hypothesis` *insure_paid_order* : $((Linsureorder :: nil) ++ (insurance :: nil)) \vdash order\_paid\_insured.$

`Hypothesis` *pay_order* : $((Lpayorder :: nil) ++ (payment :: nil)) \vdash order\_paid.$

`Hypothesis` *pay_insured_order* : $((Lpayorder :: nil) ++ (payment :: nil)) \vdash order\_insured\_paid.$

`Hypothesis` *link_after_order* : $nil \vdash order\_unpaid \multimap (Linsureorder \mathbin{\&} Lpayorder).$

`Hypothesis` *link_after_insured_unpaid_order* : $nil \vdash order\_unpaid\_insured \multimap (Lpayorder).$

`Hypothesis` *link_shiporder1* : $nil \vdash order\_paid\_insured \multimap Lshiporder.$

`Hypothesis` *link_shiporder2* : $nil \vdash order\_insured\_paid \multimap Lshiporder.$

`Hypothesis` *ship_order* : $((Lshiporder :: nil) ++ (shipment :: nil)) \vdash order\_shipped.$

`Theorem` *ship_order_extendedservices* : $((order\_empty \otimes (user \otimes securitychecker) \otimes (item \otimes rate) \otimes insurance \otimes payment \otimes$

```
        shipment) :: nil) ⊢ order_shipped.

Proof.
  apply AddNilLeft.
  apply TimesLeft.
  apply AssociateLeft.
  apply Cut with order_insured_paid.
  apply TimesLeft.
  apply RemoveNilLeft.
  apply Cut with order_unpaid_insured.
  apply AddNilLeft.
  apply TimesLeft.
  apply RemoveNilLeft.
  apply Cut with order_unpaid.
  apply AddNilLeft.
  apply TimesLeft.
  apply RemoveNilLeft.
  apply Cut with (order_empty ⊗ user_valid).
  apply AddNilLeft.
  apply TimesLeft.
  apply RemoveNilLeft.
  apply TimesRight.
  apply Identity.
  apply AddNilLeft.
  apply TimesLeft.
  apply RemoveNilLeft.
  apply validate_user.
  apply Cut with item_rated.
  apply AddNilLeft.
  apply TimesLeft.
  apply RemoveNilLeft.
  apply rate_item.
apply AddNilLeft.
apply Exchange.
apply RemoveNilLeft.
apply TimesLeft.
apply Exchange.
apply place_order.
apply AddNilLeft.
apply Exchange.
apply RemoveNilLeft.
apply Cut with (Linsureorder & Lpayorder).
apply AddNilLeft.
apply Shift.
apply link_after_order.
apply WithLeft1.
apply AddNilLeft.
apply Exchange.
apply RemoveNilLeft.
apply insure_unpaid_order.
apply AddNilLeft.
apply Exchange.
apply RemoveNilLeft.
apply Cut with Lpayorder.
apply AddNilLeft.
apply Shift.
apply link_after_insured_unpaid_order.
apply AddNilLeft.
apply Exchange.
apply RemoveNilLeft.
apply pay_insured_order.
apply AddNilLeft.
apply Exchange.
```

```
    apply RemoveNilLeft.
    apply Cut with Lshiporder.
    apply AddNilLeft.
    apply Shift.
    apply link_shiporder2.
    apply AddNilLeft.
    apply Exchange.
    apply RemoveNilLeft.
    apply ship_order.
Qed.
```

B.6 E-shopping Scenario - Case III Service Method Level Implementation in Coq

```
Variable uriuser uriuid ruid uidmsg uriuidpay ruidpay ruidvalid : ILinProp.
Variable uriitem uriiid ritem riid iidmsg riidrated : ILinProp.
Variable uripay uripayid rpayid : ILinProp.
Variable uriship urisid rsid : ILinProp.
Variable urirate uririd rrid : ILinProp.
Variable urisecchec : ILinProp.
Variable uriinsur uriinid rinid: ILinProp.
Variable uriorder urioid uriorderpay roid roidpaid roidshipped urioidpay
         urioidship : ILinProp.
Variable roidunpaid roidunpaidinsured roidpaidinsured
         roidinsuredpaid : ILinProp.
Variable nuriuser nuriuid nruid nuidmsg nuriuidpay nruidpay : name.
Variable nuriitem nuriiid nritem nriid niidmsg nriidrated : name.
Variable nuripay nuripayid nrpayid : name.
Variable nuriship nurisid nrsid : name.
Variable nurirate nuririd nrrid : name.
Variable nuriinsur nuriinid nrinid : name.
Variable nurisecchec : name.
Variable nuriorder nurioid nuriorderpay nroid nroidpaid nroidshipped
         nurioidpay : name.
Variable nurioidship moidpaymsg moidshipmsg nroidunpaid
         nroidunpaidinsured :name.
Variable nroidpaidinsured nroidinsuredpaid: name.
Variable nruidvalid : name.

Variable lpayorder lshiporder linsureorder : ILinProp.
Variable nlpayorder nlshiporder nlinsureorder nlorderinsure
         nlorderpay nlpaidorderinsure : name.

Variable GETUID GETUIDPAY : proc.
Variable GETIID : proc.
Variable POSTPAY GETPAYID : proc.
Variable POSTSHIP GETSID : proc.

Variable CREATEORDERURI PLACEORDER POSTORDER
         POSTOIDPAY POSTOIDSHIP : proc.
Variable GETOID PUTOIDPAY PUTOIDSHIP PAYORDER
         SHIPORDER : proc.
Variable PAYSHIPORDER PAYINSUREDORDER : proc.

Variable RATEITEM POSTRATE GETRID : proc.
Variable POSTSECCHEC : proc.
Variable POSTINSUR GETINID : proc.
Variable VALIDUSER INSUREUNPAIDORDER LINKORDERINSURE
         LINKORDERPAY : proc.
Variable LINKPAYORDER LINKSHIPORDER1 LINKSHIPORDER2
         LINKPAIDORDERINSURE : proc.

Variable err : ILinProp.

Variable P Q : proc.
Variable x y : name.
```

Axiom $post\_user : (uriuser :: nil) \vdash uriuid.$

Axiom $get\_uid : (uriuid \ll nuriuid) :: nil \vdash ((ruid \oplus err) \ll nruid[GETUID).$

Axiom $put\_uid : ((uriuid :: nil) ++ (uidmsg :: nil)) \vdash ruid.$

Axiom $delete\_uid : (uriuid :: nil) \vdash Zero.$

Axiom $get\_uid\_pay : ((uriuidpay \ll nuriuidpay) :: nil)$

$\vdash ((ruidpay \oplus err) \ll nruidpay[GETUIDPAY).$

Axiom $post\_item : (uriitem :: nil) \vdash uriiid.$

Axiom $get\_item : (uriitem :: nil) \vdash ritem.$

Axiom $get\_iid : (uriiid \ll nuriiid) :: nil \vdash ((riid \oplus err) \ll nriid[GETIID).$

Axiom $put\_iid : ((uriiid :: nil) ++ (iidmsg :: nil)) \vdash riid.$

Axiom $post\_pay : ((uripay \ll nuripay) :: nil)$

$\vdash ((uripayid \oplus err) \ll nuripayid[POSTPAY).$

Axiom $get\_payid : ((uripayid \ll nuripayid) :: nil)$

$\vdash ((rpayid \oplus err) \ll nrpayid[GETPAYID).$

Axiom $delete\_payid : (uripayid :: nil) \vdash Zero.$

Axiom $post\_ship : ((uriship \ll nuriship) :: nil)$

$\vdash ((urisid \oplus err) \ll nurisid[POSTSHIP).$

Axiom $get\_sid : ((urisid \ll nurisid) :: nil) \vdash ((rsid \oplus err) \ll nrsid[GETSID).$

Axiom $post\_rate : ((urirate \ll nurirate) :: nil)$

$\vdash ((uririd \oplus err) \ll nuririd[POSTRATE).$

Axiom $get\_rid : ((uririd \ll nuririd) :: nil) \vdash ((rrid \oplus err) \ll nrrid[GETRID).$

Axiom $post\_secchec : ((urisecchec \ll nurisecchec) :: nil)$

$\vdash ((uriuid \oplus err) \ll nuriuid[POSTSECCHEC).$

Axiom $post\_insur : ((uriinsur \ll nuriinsur) :: nil)$

$\vdash ((uriinid \oplus err) \ll nuriinid[POSTINSUR).$

Axiom $get\_inid : ((uriinid \ll nuriinid) :: nil) \vdash ((rinid \oplus err) \ll nrinid[GETINID).$

Axiom $post\_order : ((uriorder \ll nuriorder) :: nil)$

$\vdash ((urioid \oplus err) \ll nurioid[POSTORDER).$

Axiom $get\_oid : ((urioid \ll nurioid) :: nil)$

$\vdash (((roid \multimap urioidpay) \oplus err) \ll nroid[GETOID).$

Axiom put_oid_pay : $((urioid \ll nurioid) :: nil ++ (rpayid \ll nrpayid) :: nil)$

$\vdash ((roidpaid \oplus err) \ll nroidpaid[PUTOIDPAY)$.

Axiom put_oid_ship : $((urioid \ll nurioid) :: nil ++ (rsid \ll nrsid) :: nil)$

$\vdash ((roidshipped \oplus err) \ll nroidshipped[PUTOIDSHIP)$.

Axiom $post\_oid\_ship$: $((urioidship \ll nurioidship) :: nil)$

$\vdash ((roidshipped \oplus err) \ll nroidshipped[POSTOIDSHIP)$.

Hypothesis $validate\_user$: $((urisecchec \ll nurisecchec) :: nil$

$++ ((ruid \oplus err) \ll nruid) :: nil) \vdash ((ruidvalid \oplus err)$

$\ll nruidvalid[VALIDUSER)$.

Hypothesis $rate\_item$: $((urirate \ll nurirate) :: nil$

$++ ((riid \oplus err) \ll nriid) :: nil) \vdash ((riidrated \oplus err)$

$\ll nriidrated[RATEITEM)$.

Hypothesis $place\_order$: $(((uriorder \ll nuriorder) :: nil)$

$++ (((riidrated \oplus err) \ll nriidrated) :: nil)$

$++ (((ruidvalid \oplus err) \ll nruidvalid) :: nil))$

$\vdash ((roidunpaid \oplus err) \ll nroidunpaid[PLACEORDER)$.

Hypothesis $insure\_unpaid\_order$: $(((uriinsur \ll nuriinsur) :: nil)$

$++ (((linsureorder \oplus err) \ll nlinsureorder) :: nil))$

$\vdash ((roidunpaidinsured \oplus err) \ll nroidunpaidinsured$

$[INSUREUNPAIDORDER)$.

Hypothesis pay_order: $(((uripay \ll nuripay) :: nil)$

$++ (((ruidpay \oplus err) \ll nruidpay) :: nil)$

$++ (((lpayorder \oplus err) \ll nlpayorder) :: nil))$

$\vdash ((roidpaid \oplus err) \ll nroidpaid[PAYORDER)$.

Hypothesis $insure\_paid\_order$: $((((linsureorder \oplus err) \ll nlinsureorder) ::$

$nil) ++ ((uriinsur \ll nuriinsur) :: nil))$

$\vdash ((roidpaidinsured \oplus err) \ll nroidpaidinsured[INSUREUNPAIDORDER)$.

Hypothesis $pay\_insured\_order$: $(((uripay \ll nuripay) :: nil)$

$++ (((ruidpay \oplus err) \ll nruidpay) :: nil)$

$++ (((lpayorder \oplus err) \ll nlpayorder) :: nil))$

$\vdash ((roidinsuredpaid \oplus err) \ll nroidinsuredpaid[PAYINSUREDORDER)$.

`Hypothesis` *ship_order*: $(((uriship \ll nuriship) :: nil)$

$++ (((lshiporder \oplus err) \ll nlshiporder) :: nil))$

$\vdash ((roidshipped \oplus err) \ll nroidshipped[SHIPORDER).$

`Hypothesis` *link_order_insure*: $nil \vdash (((roidunpaid \oplus err)$

$\multimap (linsureorder \oplus err)) \ll nlorderinsure[LINKORDERINSURE).$

`Hypothesis` *link_order_pay*: $nil \vdash (((roidunpaid \oplus err)$

$\multimap (lpayorder \oplus err)) \ll nlorderpay[LINKORDERPAY).$

`Hypothesis` *link_after_insured_unpaid_order*: $nil \vdash (((roidunpaidinsured$

$\oplus err) \multimap (lpayorder \oplus err)) \ll nlpayorder[LINKPAYORDER).$

`Hypothesis` *link_after_order_pay*: $nil \vdash (((roidpaid \oplus err)$

$\multimap (linsureorder \oplus err)) \ll nlpaidorderinsure$

$[LINKPAIDORDERINSURE).$

`Hypothesis` *link_shiporder1*: $nil \vdash (((roidpaidinsured \oplus err)$

$\multimap (lshiporder \oplus err)) \ll nlshiporder[LINKSHIPORDER1).$

`Hypothesis` *link_shiporder2*: $nil \vdash (((roidinsuredpaid \oplus err)$

$\multimap (lshiporder \oplus err)) \ll nlshiporder[LINKSHIPORDER2).$

`Theorem` *order_being_shipped_extended*: $\exists\ P, (((uriship \ll nuriship) :: nil)$

$++ ((uripay \ll nuripay) :: nil) ++ ((uriuidpay \ll nuriuidpay) :: nil)$

$++ ((uriinsur \ll nuriinsur) :: nil)$

$++ ((uriorder \ll nuriorder) :: nil) ++ (((uriiid \ll nuriiid) :: nil)$

$++ ((urirate \ll nurirate) :: nil))$

$++ (((uriuid \ll nuriuid) :: nil)) ++ ((urisecchec \ll nurisecchec)$

$:: nil)) \vdash ((roidshipped \oplus err) \ll nroidshipped[P).$

`Proof`.

`econstructor`.

`apply` *AddNilRight*.

`instantiate` *(1:= (nu y (par P Q)))*.

`apply` *Cut* `with` *(Times uriship (Plus lshiporder err))*.

`auto`.

`econstructor`.

```
instantiate (1:= (nu nuriship (outp nlshiporder nuriship (par P Q)))).
apply TimesRight.
econstructor.
instantiate (1:= skip).
apply Identity.
econstructor.
apply AddNilRight.
instantiate (1:= (nu nroidinsuredpaid (par P Q))).
apply Cut with (Plus roidinsuredpaid err).
auto.
econstructor.
apply AddNilRight.
instantiate (1:= (nu y (par P Q))).
apply Cut with (Times uripay (Times (Plus ruidpay err) (Plus lpayorder
err))).
auto.
econstructor.
instantiate (1:= (nu nuripay (outp y nuripay (par P Q)))).
apply TimesRight.
econstructor.
instantiate (1:= skip).
apply Identity.
econstructor.
instantiate (1:= (nu nruidpay (outp nlpayorder nruidpay (par P Q)))).
apply TimesRight.
econstructor.
instantiate (1:= GETUIDPAY).
apply get_uid_pay.
econstructor.
apply AddNilRight.
instantiate (1:= (nu nroidunpaidinsured (par P Q))).
```

apply Cut with *(Plus roidunpaidinsured err).*

econstructor.

apply *AddNilRight.*

instantiate *(1:= (nu nlinsureorder (par P Q))).*

apply *Cut* with *(Times uriinsur (Plus linsureorder err)).*

econstructor.

instantiate *(1:= (nu nuriinsur (outp nlorderinsure nuriinsur (par P Q)))).*

apply *TimesRight.*

econstructor.

instantiate *(1:= skip).*

apply *Identity.*

econstructor.

apply *AddNilRight.*

instantiate *(1:= (nu nroidunpaid (par P Q))).*

apply *Cut* with *(Plus roidunpaid err).*

auto.

econstructor.

apply *AddNilRight.*

instantiate *(1:= (nu y (par P Q))).*

apply *Cut* with *(Times uriorder (Times (Plus riidrated err) (Plus ruid-valid err))).*

auto.

econstructor.

instantiate *(1:= (nu nuriorder (outp y nuriorder (par P Q)))).*

apply *TimesRight.*

econstructor.

instantiate *(1:= skip).*

apply *Identity.*

econstructor.

instantiate *(1:= (nu nriidrated (outp nruidvalid nriidrated (par P Q)))).*

apply *TimesRight.*

```
econstructor.
instantiate (1:= (nu nriid (par P Q))).
apply Cut with (Plus riid err).
econstructor.
instantiate (1:= GETIID).
apply get_iid.
econstructor.
instantiate (1:= RATEITEM).
apply rate_item.
econstructor.
instantiate (1:= (nu nruid (par P Q))).
apply Cut with (Plus ruid err).
econstructor.
instantiate (1:= GETUID).
apply get_uid.
econstructor.
instantiate (1:= VALIDUSER).
apply validate_user.
econstructor.
instantiate (1:= (inp y nuriorder P)).
apply TimesLeft.
econstructor.
apply RemoveNilLeft.
instantiate (1:= (inp nruidvalid nriidrated P)).
apply TimesLeft.
econstructor.
instantiate (1:= PLACEORDER).
apply place_order.
econstructor.
instantiate (1:= P).
apply Shift.
```

```
econstructor.
instantiate (1:= LINKORDERINSURE).
apply link_order_insure.
econstructor.
instantiate (1:= (inp nlinsureorder nuriinsur P)).
apply TimesLeft.
econstructor.
apply RemoveNilLeft.
instantiate (1:= INSUREUNPAIDORDER).
apply insure_unpaid_order.
econstructor.
instantiate (1:= P).
apply Shift.
econstructor.
instantiate (1:= LINKPAYORDER).
apply link_after_insured_unpaid_order.
econstructor.
instantiate (1:= (inp y nuripay P)).
apply TimesLeft.
econstructor.
apply RemoveNilLeft.
instantiate (1:= (inp nlpayorder nruidpay P)).
apply TimesLeft.
econstructor.
instantiate (1:= PAYINSUREDORDER).
apply pay_insured_order.
econstructor.
instantiate (1:= P).
apply Shift.
econstructor.
instantiate (1:= LINKSHIPORDER2).
```

```
    apply link_shiporder2.
    econstructor.
    instantiate (1:= (inp nlshiporder nuriship P)).
    apply TimesLeft.
    econstructor.
    apply RemoveNilLeft.
    instantiate (1:= SHIPORDER).
    apply ship_order.
Qed.
```

B.7 Biomedical Scenario - Resource Level Implementation in Coq

```
Variable ct importer isoextracter cropper smoothfilter decimatefilter
         mesh : ILinProp.
Variable vtkvolume vtksurface mesh3d : ILinProp.
Variable Lextractisosurface Lcropvolume Ldecimate Lsmooth
         Lbuildmesh : ILinProp.
Hypothesis importdicom : ((ct::nil) ++ (importer::nil)) ⊢ vtkvolume.
Hypothesis extractisosurface : ((Lextractisosurface::nil)
         ++ (isoextracter::nil)) ⊢ vtksurface.
Hypothesis cropvolume : ((Lcropvolume::nil)
         ++ (cropper::nil)) ⊢ vtkvolume.
Hypothesis decimatesurface : ((Ldecimate::nil)
         ++ (decimatefilter::nil)) ⊢ vtksurface.
Hypothesis smoothsurface : ((Lsmooth::nil)
         ++ (smoothfilter::nil)) ⊢ vtksurface.
Hypothesis buildmesh : ((Lbuildmesh::nil)
         ++ (mesh::nil)) ⊢ mesh3d.
```

Hypothesis $link\_after\_volume$: $nil \vdash (vtkvolume$
$\multimap (Lextractisosurface \,\&\, Lcropvolume))$.

Hypothesis $link\_after\_surface$: $nil \vdash (vtksurface$
$\multimap (Ldecimate \,\&\, Lsmooth \,\&\, Lbuildmesh))$.

Theorem $buildmesh\_decimate\_1$: $((ct \otimes importer \otimes isoextracter$
$\otimes\, decimatefilter \otimes mesh)::nil) \vdash mesh3d$.

Proof.

apply *AddNilLeft.*

apply *TimesLeft.*

apply *AssociateLeft.*

apply *Cut* with *(Ldecimate & Lsmooth & Lbuildmesh).*

apply *RemoveNilLeft.*

apply *AddNilRight.*

apply *Cut* with *vtksurface.*

apply *AddNilLeft.*

apply *TimesLeft.*

apply *AssociateLeft.*

apply *Cut* with *(Ldecimate & Lsmooth & Lbuildmesh).*

apply *RemoveNilLeft.*

apply *AddNilRight.*

apply *Cut* with *vtksurface.*

apply *AddNilLeft.*

apply *TimesLeft.*

apply *AssociateLeft.*

apply *Cut* with *(Lextractisosurface & Lcropvolume).*

apply *RemoveNilLeft.*

apply *AddNilRight.*

apply *Cut* with *vtkvolume.*

apply *AddNilLeft.*

apply *TimesLeft.*

apply *RemoveNilLeft.*

`apply` *importdicom.*

`apply` *Shift.*

`apply` *link_after_volume.*

`apply` *WithLeft1.*

`apply` *AddNilLeft.*

`apply` *Exchange.*

`apply` *RemoveNilLeft.*

`apply` *extractisosurface.*

`apply` *Shift.*

`apply` *link_after_surface.*

`apply` *WithLeft1.*

`apply` *WithLeft1.*

`apply` *AddNilLeft.*

`apply` *Exchange.*

`apply` *RemoveNilLeft.*

`apply` *decimatesurface.*

`apply` *Shift.*

`apply` *link_after_surface.*

`apply` *WithLeft2.*

`apply` *AddNilLeft.*

`apply` *Exchange.*

`apply` *RemoveNilLeft.*

`apply` *buildmesh.*

Qed.

`Theorem` *buildmesh_decimate_2* : $((ct \otimes importer \otimes isoextracter$
$\otimes decimatefilter \otimes decimatefilter \otimes mesh)::nil) \vdash mesh3d.$

Proof.

`apply` *AddNilLeft.*

`apply` *TimesLeft.*

`apply` *AssociateLeft.*

```
apply Cut with (Ldecimate & Lsmooth & Lbuildmesh).
apply RemoveNilLeft.
apply AddNilRight.
apply Cut with vtksurface.
apply AddNilLeft.
apply TimesLeft.
apply AssociateLeft.
apply Cut with (Ldecimate & Lsmooth & Lbuildmesh).
apply RemoveNilLeft.
apply AddNilRight.
apply Cut with vtksurface.
apply AddNilLeft.
apply TimesLeft.
apply AssociateLeft.
apply Cut with (Ldecimate & Lsmooth & Lbuildmesh).
apply RemoveNilLeft.
apply AddNilRight.
apply Cut with vtksurface.
apply AddNilLeft.
apply TimesLeft.
apply AssociateLeft.
apply Cut with (Lextractisosurface & Lcropvolume).
apply RemoveNilLeft.
apply AddNilRight.
apply Cut with vtkvolume.
apply AddNilLeft.
apply TimesLeft.
apply RemoveNilLeft.
apply importdicom.
apply Shift.
apply link_after_volume.
```

```
    apply WithLeft1.
    apply AddNilLeft.
    apply Exchange.
    apply RemoveNilLeft.
    apply extractisosurface.
    apply Shift.
    apply link_after_surface.
    apply WithLeft1.
    apply WithLeft1.
    apply AddNilLeft.
    apply Exchange.
    apply RemoveNilLeft.
    apply decimatesurface.
    apply Shift.
    apply link_after_surface.
    apply WithLeft1.
    apply WithLeft1.
    apply AddNilLeft.
    apply Exchange.
    apply RemoveNilLeft.
    apply decimatesurface.
    apply Shift.
    apply link_after_surface.
    apply WithLeft2.
    apply AddNilLeft.
    apply Exchange.
    apply RemoveNilLeft.
    apply buildmesh.
Qed.
```

B.8 Biomedical Scenario - Service Method Level Implementation in Coq

`Variable` $urict\ urictid\ rct\ rctid$: $ILinProp.$

`Variable` $uriimp$: $ILinProp.$

`Variable` $uriiso$: $ILinProp.$

`Variable` $uridec$: $ILinProp.$

`Variable` $uricrop$: $ILinProp.$

`Variable` $urimesh\ rmesh\ rvtkvol\ rvtksur$: $ILinProp.$

`Variable` $lexiso\ ldec\ lsmo\ lmesh\ lcropvol$: $ILinProp.$

`Variable` $nurict\ nurictid\ nrctid$: $name.$

`Variable` $nuriimp$: $name.$

`Variable` $nuriiso$: $name.$

`Variable` $nuridec$: $name.$

`Variable` $nurismo$: $name.$

`Variable` $nuricrop$: $name.$

`Variable` $nurimesh\ nrmesh\ nrvtkvol\ nrvtksur\ nlonvol\ nlonsur$: $name.$

`Variable` $nlexiso\ nldec\ nlsmo\ nlmesh\ nlcropvol$: $name.$

`Variable` $POSTCT\ GETCTID\ POSTIMPORTER\ POSTISOEX$
$POSTDECIMATE\ POSTSMOOTH\ POSTMOT$
$POSTMESH$: $proc.$

`Variable` $IMPORTDICOM\ EXTRACTISO\ CROPVOL\ DECIMATE$
$SMOOTH\ MESH\ LINKAFTERVOL\ LINKAFTERSUR$: $proc.$

`Variable` err : $ILinProp.$

`Variable` $P\ Q$: $proc.$

`Variable` $x\ y$: $name.$

`Axiom` $post\_ct : (urict \ll nurict) :: nil \vdash ((urictid \oplus err) \ll nurictid[POSTCT).$

`Axiom` $get\_ctid : (urictid \ll nurictid) :: nil \vdash ((rctid \oplus err) \ll nrctid[GETCTID).$

`Axiom` $post\_importer : (uriimp \ll nuriimp) :: nil$
$\vdash ((rvtkvol \oplus err) \ll nrvtkvol[POSTIMPORTER).$

`Axiom` $post\_isoex$: $(uriiso \ll nuriiso) :: nil$

$\vdash ((rvtksur \oplus err) \ll nrvtksur[POSTISOEX).$

`Axiom` $post\_decimate$: $(uridec \ll nuridec) :: nil$

$\vdash ((rvtksur \oplus err) \ll nrvtksur[POSTDECIMATE).$

`Axiom` $post\_smooth$: $(urismo \ll nurismo) :: nil$

$\vdash ((rvtksur \oplus err) \ll nrvtksur[POSTSMOOTH).$

`Axiom` $post\_mesh$: $(urimesh \ll nurimesh) :: nil$

$\vdash ((rmesh \oplus err) \ll nrmesh[POSTMESH).$

`Hypothesis` $import\_dicom$: $(uriimp \ll nuriimp)::nil ++ (urictid \ll nurictid)::nil$

$\vdash ((rvtkvol \oplus err) \ll nrvtkvol[IMPORTDICOM).$

`Hypothesis` $extract\_isosurface$: $(uriiso \ll nuriiso)::nil$

$++ ((lexiso \oplus err) \ll nlexiso)::nil \vdash ((rvtksur \oplus err)$

$\ll nrvtksur[EXTRACTISO).$

`Hypothesis` $crop\_volume$: $(uricrop \ll nuricrop)::nil$

$++ ((lcropvol \oplus err) \ll nlcropvol)::nil \vdash ((rvtkvol \oplus err)$

$\ll nrvtkvol[CROPVOL).$

`Hypothesis` $decimate\_surface$: $(uridec \ll nuridec)::nil$

$++ ((ldec \oplus err) \ll nldec)::nil \vdash ((rvtksur \oplus err)$

$\ll nrvtksur[DECIMATE).$

`Hypothesis` $smooth\_surface$: $(urismo \ll nurismo)::nil$

$++ ((lsmo \oplus err) \ll nlsmo)::nil \vdash ((rvtksur \oplus err) \ll nrvtksur[SMOOTH).$

`Hypothesis` $build\_mesh$: $(urimesh \ll nurimesh)::nil$

$++ ((lmesh \oplus err) \ll nlmesh)::nil \vdash ((rmesh \oplus err) \ll nrmesh[MESH).$

`Hypothesis` $link\_after\_volume$: $nil \vdash (((rvtkvol \oplus err)$

$\multimap ((lexiso \oplus err) \,\&\, (lcropvol \oplus err))) \ll nlonvol[LINKAFTERVOL).$

`Hypothesis` $link\_after\_surface$: $nil \vdash (((rvtksur \oplus err)$

$\multimap ((ldec \oplus err) \,\&\, (lsmo \oplus err) \,\&\, (lmesh \oplus err))) \ll nlonsur[LINKAFTERSUR).$

`Theorem` $mech\_being\_built$: $\exists\ P, ((urimot \ll nurimot) :: nil)$

$++ ((uridec \ll nuridec) :: nil) ++ ((uriiso \ll nuriiso) :: nil)$

$++ ((uriimp \ll nuriimp) :: nil) ++ ((urictid \ll nurictid) :: nil)$

$\vdash ((rmesh \oplus err) \ll nrmesh[P)$.

`Proof.`

`econstructor.`

`apply` *AddNilRight.*

`instantiate` *(1:= (nu nlonsur (par P Q))).*

`apply` *Cut* `with` *(Times urimesh (With (With (Plus ldec err) (Plus lsmo err)) (Plus lmesh err))).*

`econstructor.`

`instantiate` *(1:= (nu nurimot (outp nlonsur nurimot (par P Q)))).*

`apply` *TimesRight.*

`econstructor.`

`instantiate` *(1:= skip).*

`apply` *Identity.*

`econstructor.`

`apply` *AddNilRight.*

`instantiate` *(1:= (nu nlonsur (par P Q))).*

`apply` *Cut* `with` *(Times uridec (With (With (Plus ldec err) (Plus lsmo err)) (Plus lmesh err))).*

`econstructor.`

`instantiate` *(1:= (nu nuridec (outp nlonsur nuridec (par P Q)))).*

`apply` *TimesRight.*

`econstructor.`

`instantiate` *(1:= skip).*

`apply` *Identity.*

`econstructor.`

`apply` *AddNilRight.*

`instantiate` *(1:= (nu nlonvol (par P Q))).*

`apply` *Cut* `with` *(Times uriiso (With (Plus lexiso err) (Plus lcropvol err))).*

`econstructor.`

`instantiate` *(1:= (nu nuriiso (outp nlonvol nuriiso (par P Q)))).*

`apply` *TimesRight.*

```
econstructor.
instantiate (1:= skip).
apply Identity.
econstructor.
apply AddNilRight.
instantiate (1:= (nu nrvtkvol (par P Q))).
apply Cut with (Plus rvtkvol err).
econstructor.
instantiate (1:= IMPORTDICOM).
apply import_dicom.
econstructor.
instantiate (1:= P).
apply Shift.
econstructor.
instantiate (1:= LINKAFTERVOL).
apply link_after_volume.
econstructor.
apply RemoveNilLeft.
apply AddNilRight.
instantiate (1:= (nu nrvtksur (par P Q))).
apply Cut with (Plus rvtksur err).
econstructor.
apply AddNilLeft.
instantiate (1:= (inp nlexiso nuriiso P)).
apply TimesLeft.
econstructor.
instantiate (1:= P).
apply WithLeft1.
econstructor.
instantiate (1:= EXTRACTISO).
apply extract_isosurface.
```

```
econstructor.
instantiate (1:= P).
apply Shift.
econstructor.
instantiate (1:= LINKAFTERSUR).
apply link_after_surface.
econstructor.
apply AddNilRight.
instantiate (1:= (nu nrvtksur (par P Q))).
apply Cut with (Plus rvtksur err).
econstructor.
instantiate (1:= (inp nldec nuridec P)).
apply TimesLeft.
econstructor.
instantiate (1:= P).
apply WithLeft1.
econstructor.
instantiate (1:= P).
apply WithLeft1.
econstructor.
instantiate (1:= DECIMATE).
apply decimate_surface.
econstructor.
instantiate (1:= P).
apply Shift.
econstructor.
instantiate (1:= LINKAFTERSUR).
apply link_after_surface.
econstructor.
instantiate (1:= (inp nlmesh nurimesh P)).
apply TimesLeft.
```

```
econstructor.
apply RemoveNilLeft.
instantiate (1:= P).
apply WithLeft2.
econstructor.
instantiate (1:= MESH).
apply build_mesh.
Qed.
```

Appendix C

Proof Trees

C.1 E-shopping Scenario Case I at the Second Stage

C.1.1

$$\dfrac{\dfrac{(\text{uriuid} \vdash (\text{ruid} \oplus \text{err})) \qquad (\text{uriid} \vdash (\text{riid} \oplus \text{err}))}{(\text{uriuid, uriid} \vdash (\text{ruid} \oplus \text{err}) \otimes (\text{riid} \oplus \text{err}))}\,(\otimes\ \text{R}) \qquad (\text{uriorder} \vdash \text{uriorder})}{(\text{uriuid, uriid, uriorder} \vdash (\text{ruid} \oplus \text{err}) \otimes (\text{riid} \oplus \text{err}) \otimes \text{uriorder})}\,(\otimes\ \text{R})$$

C.1.2

$$\dfrac{\dfrac{\text{C.1.1} \qquad \dfrac{\dfrac{((\mathrm{ruid} \oplus \mathrm{err}), (\mathrm{riid} \oplus \mathrm{err}), \mathrm{uriorder} \vdash \mathrm{roid})}{((\mathrm{ruid} \oplus \mathrm{err}) \otimes (\mathrm{riid} \oplus \mathrm{err}), \mathrm{uriorder} \vdash (\mathrm{roid} \oplus \mathrm{err}))}\,(\otimes\ \mathrm{L})}{((\mathrm{ruid} \oplus \mathrm{err}) \otimes (\mathrm{riid} \oplus \mathrm{err}) \otimes \mathrm{uriorder} \vdash (\mathrm{roid} \oplus \mathrm{err}))}\,(\otimes\ \mathrm{L})}{(\mathrm{uriuid}, \mathrm{uriid}, \mathrm{uriorder} \vdash (\mathrm{roid} \oplus \mathrm{err}))}\,(\mathrm{Cut}) \qquad \dfrac{(\vdash (\mathrm{roid} \oplus \mathrm{err}) \multimap (L\mathrm{payorder} \oplus \mathrm{err}))}{((\mathrm{roid} \oplus \mathrm{err}) \vdash (L\mathrm{payorder} \oplus \mathrm{err}))}\,(\mathrm{Shift})}{(\mathrm{uriuid}, \mathrm{uriid}, \mathrm{uriorder} \vdash (L\mathrm{payorder} \oplus \mathrm{err}))}\,(\mathrm{Cut})$$

C.1.3

$$\dfrac{\dfrac{\text{C.1.2} \qquad (\mathrm{uriuidpay} \vdash (\mathrm{ruidpay} \oplus \mathrm{err}))}{(\mathrm{uriuid}, \mathrm{uriid}, \mathrm{uriorder}, \mathrm{uriuidpay} \vdash (L\mathrm{payorder} \oplus \mathrm{err}) \otimes (\mathrm{ruidpay} \oplus \mathrm{err}))}\,(\otimes\ \mathrm{R}) \qquad (\mathrm{uripay} \vdash \mathrm{uripay})}{(\mathrm{uriuid}, \mathrm{uriid}, \mathrm{uriorder}, \mathrm{uriuidpay}, \mathrm{uripay} \vdash (L\mathrm{payorder} \oplus \mathrm{err}) \otimes (\mathrm{ruidpay} \oplus \mathrm{err}) \otimes \mathrm{uripay})}\,(\otimes\ \mathrm{R})$$

C.1.4

$$\dfrac{\dfrac{\text{C.1.3} \qquad \dfrac{\dfrac{((L\mathrm{payorder} \oplus \mathrm{err}), (\mathrm{ruidpay} \oplus \mathrm{err}), \mathrm{uripay} \vdash (\mathrm{roidpaid} \oplus \mathrm{err}))}{((L\mathrm{payorder} \oplus \mathrm{err}) \otimes (\mathrm{ruidpay} \oplus \mathrm{err}), \mathrm{uripay} \vdash (\mathrm{roidpaid} \oplus \mathrm{err}))}\,\otimes\ \mathrm{L}}{((L\mathrm{payorder} \oplus \mathrm{err}) \otimes (\mathrm{ruidpay} \oplus \mathrm{err}) \otimes \mathrm{uripay} \vdash (\mathrm{roidpaid} \oplus \mathrm{err}))}\,\otimes\ \mathrm{L}}{(\mathrm{uriuid}, \mathrm{uriid}, \mathrm{uriorder}, \mathrm{uriuidpay}, \mathrm{uripay} \vdash (\mathrm{roidpaid} \oplus \mathrm{err}))}\,(\mathrm{Cut}) \qquad \dfrac{(\vdash (\mathrm{roidpaid} \oplus \mathrm{err}) \multimap (L\mathrm{shiporder} \oplus \mathrm{err}))}{((\mathrm{roidpaid} \oplus \mathrm{err}) \vdash (L\mathrm{shiporder} \oplus \mathrm{err}))}\,(\mathrm{Shift})}{(\mathrm{uriuid}, \mathrm{uriid}, \mathrm{uriorder}, \mathrm{uriuidpay}, \mathrm{uripay} \vdash (L\mathrm{shiporder} \oplus \mathrm{err}))}\,(\mathrm{Cut})$$

C.1.5

$$\dfrac{((L\mathrm{shiporder} \oplus \mathrm{err}), \mathrm{uriship} \vdash (\mathrm{roidshipped} \oplus \mathrm{err}))}{((L\mathrm{shiporder} \oplus \mathrm{err}) \otimes \mathrm{uriship} \vdash (\mathrm{roidshipped} \oplus \mathrm{err}))}\,(\otimes\ \mathrm{L})$$

C.1.6

$$\dfrac{\dfrac{\text{C.1.4} \qquad (\text{uriship} \vdash \text{uriship})}{(\text{uriuid, uriid, uriorder, uriuidpay, uripay, uriship} \vdash (L\text{shiporder} \oplus \text{err}) \otimes \text{uriship})}\,(\otimes\ \text{R}) \qquad \text{C.1.5}}{(\text{uriuid, uriid, uriorder, uriuidpay, uripay, uriship} \vdash (\text{roidshipped} \oplus \text{err}))}\,(\text{Cut})$$

C.2 E-shopping Scenario Case II at the First Stage

C.2.1

$$\dfrac{\dfrac{\dfrac{\dfrac{(\text{O\_empty, U, I} \vdash \text{O\_unpaid}) \qquad \dfrac{(\vdash \text{O\_unpaid} \multimap L\text{PO})}{(\text{O\_unpaid} \vdash L\text{PO})}\,(\text{Shift})}{(\text{O\_empty, U, I} \vdash L\text{PO})}\,(\text{Cut}) \qquad \dfrac{(L\text{PO}, \text{P}_1 \vdash \text{O\_paid})}{(L\text{PO}, \text{P}_1 \mathbin{\&} \text{P}_2 \vdash \text{O\_paid})}\,(\&\ \text{L}_1)}{(\text{O\_empty, U, I}, \text{P}_1 \mathbin{\&} \text{P}_2 \vdash \text{O\_paid})}\,(\text{Cut}) \qquad \dfrac{(\vdash \text{O\_paid} \multimap L\text{SO})}{(\text{O\_paid} \vdash L\text{SO})}\,(\text{Shift})}{(\text{O\_empty, U, I}, \text{P}_1 \mathbin{\&} \text{P}_2 \vdash L\text{SO})}\,(\text{Cut}) \qquad (L\text{SO}, \text{S} \vdash \text{O\_shipped})}{(\text{O\_empty, U, I}, \text{P}_1 \mathbin{\&} \text{P}_2, \text{S} \vdash \text{O\_shipped})}\,(\text{Cut})$$

C.3 E-shopping Scenario Case II at the Second Stage

C.3.1

$$\dfrac{\dfrac{(\text{uriid} \vdash (\text{riid} \oplus \text{err})) \qquad (\text{uriuid} \vdash (\text{ruid} \oplus \text{err}))}{(\text{uriid, uriuid} \vdash ((\text{riid} \oplus \text{err} \otimes (\text{ruid} \oplus \text{err})))}\,(\otimes\ \text{R}) \qquad (\text{uriorder} \vdash \text{uriorder})}{(\text{uriorder, uriid, uriuid} \vdash (\text{uriorder} \otimes (\text{riid} \oplus \text{err} \otimes (\text{ruid} \oplus \text{err})))}\,(\otimes\ \text{R})$$

C.3.2

$$\dfrac{(\vdash (\text{roid} \oplus \text{err}) \multimap (L\text{payorderpaypal} \oplus \text{err}) \mathbin{\&} (L\text{payordercc} \oplus \text{err}) \mathbin{\&} (L\text{payorder} \oplus \text{err}))}{((\text{roid} \oplus \text{err}) \vdash (L\text{payorderpaypal} \oplus \text{err}) \mathbin{\&} (L\text{payordercc} \oplus \text{err}) \mathbin{\&} (L\text{payorderdc} \oplus \text{err}))}\ (\text{Shift})$$

C.3.3

$$\dfrac{\text{C.3.2} \qquad ((L\text{payorderpaypal} \oplus \text{err}) \mathbin{\&} (L\text{payordercc} \oplus \text{err}) \mathbin{\&} (L\text{payorderdc} \oplus \text{err}) \vdash (\text{lpayorderpaypal} \oplus \text{err}))}{((\text{roid} \oplus \text{err}) \vdash (\text{lpayorderpaypal} \oplus \text{err}))}\ (\text{Cut})$$

C.3.4

$$\dfrac{\dfrac{\dfrac{\text{C.3.1} \qquad \dfrac{\dfrac{((\text{uriorder}, \text{riid} \oplus \text{err}, \text{ruid} \oplus \text{err}) \vdash (\text{roid} \oplus \text{err}))}{((\text{uriorder}, (\text{riid} \oplus \text{err}) \otimes (\text{ruid} \oplus \text{err})) \vdash (\text{roid} \oplus \text{err}))}\ (\otimes\ \text{L})}{((\text{uriorder} \otimes (\text{riid} \oplus \text{err}) \otimes (\text{ruid} \oplus \text{err})) \vdash (\text{roid} \oplus \text{err}))}\ (\otimes\ \text{L})}{(\text{uriorder}, \text{uriid}, \text{uriuid} \vdash (\text{roid} \oplus \text{err}))}\ (\text{Cut}) \qquad \text{C.3.3}}{(\text{uriorder}, \text{uriid}, \text{uriuid} \vdash (\text{lpayorderpaypal} \oplus \text{err}))}\ (\text{Cut}) \qquad (\text{uriuidpay} \vdash (\text{ruidpaypal} \oplus \text{err}))}{(\text{uriuidpay}, \text{uriorder}, \text{uriid}, \text{uriuid} \vdash (((\text{ruidpaypal} \oplus \text{err} \otimes (\text{lpayorderpaypal} \oplus \text{err})))}\ (\otimes\ \text{R})$$

C.3.5

$$\dfrac{\text{C.3.4} \qquad (\text{uripaypal} \vdash \text{uripaypal})}{(\text{uripaypal}, \text{uriuidpay}, \text{uriorder}, \text{uriid}, \text{uriuid} \vdash ((\text{uripaypal} \otimes (\text{ruidpaypal} \oplus \text{err} \otimes (\text{lpayorderpaypal} \oplus \text{err})))}\ (\otimes\ \text{R})$$

C.3.6

$$\dfrac{\dfrac{\text{C.3.5} \qquad (((\text{uripaypal} \otimes (\text{ruidpaypal} \oplus \text{err} \otimes (\text{lpayorderpaypal} \oplus \text{err}))\vdash (\text{roidpaid} \oplus \text{err}))}{(\text{uripaypal}, \text{uriuidpay}, \text{uriorder}, \text{uriid}, \text{uriuid} \vdash (\text{roidpaid} \oplus \text{err}))}\ (\text{Cut}) \qquad ((\text{roidpaid} \oplus \text{err}) \vdash (\text{lshiporder} \oplus \text{err}))}{(\text{uripaypal}, \text{uriuidpay}, \text{uriorder}, \text{uriid}, \text{uriuid} \vdash (\text{lshiporder} \oplus \text{err}))}\ (\text{Cut})$$

C.3.7

$$\dfrac{\dfrac{\text{C.3.6} \qquad (\text{uriship} \vdash \text{uriship})}{(\text{uriship, uripaypal, uriuidpay, uriorder, uriid, uriuid} \vdash (\text{uriship} \otimes (\text{lshiporder} \oplus \text{err})))}\,(\otimes\,\text{R}) \qquad ((\text{uriship} \otimes (\text{lshiporder} \oplus \text{err})) \vdash (\text{roidshipped} \oplus \text{err}))}{(\text{uriship, uripaypal, uriuidpay, uriorder, uriid, uriuid} \vdash (\text{roidshipped} \oplus \text{err}))}\,(\text{Shift})$$

C.4 E-shopping Scenario Case III at the First Stage

C.4.1

$$\dfrac{\dfrac{(\text{item, rate} \vdash \text{item\_rated})}{(\text{item} \otimes \text{rate} \vdash \text{item\_rated})}\,(\otimes\,\text{L}) \qquad \dfrac{(\text{order\_empty, user\_valid, item\_rated} \vdash \text{order\_unpaid})}{(\text{order\_empty} \otimes \text{user\_valid, item\_rated} \vdash \text{order\_unpaid})}\,(\otimes\,\text{L})}{((\text{item} \otimes \text{rate}), (\text{order\_empty} \otimes \text{user\_valid}) \vdash \text{order\_unpaid})}\,(\text{cut})$$

C.4.2

$$\dfrac{\dfrac{\dfrac{\dfrac{(\text{order\_empty} \vdash \text{order\_empty}) \qquad \dfrac{(\text{user, securitychecker} \vdash \text{user\_valid})}{((\text{user} \otimes \text{securitychecker}) \vdash \text{user\_valid})}\,(\otimes\,\text{L})}{(\text{order\_empty}, (\text{user} \otimes \text{securitychecker}) \vdash (\text{order\_empty} \otimes \text{user\_valid})}\,(\otimes\,\text{R})}{(\text{order\_empty} \otimes (\text{user} \otimes \text{securitychecker}) \vdash (\text{order\_empty} \otimes \text{user\_valid}))}\,(\otimes\,\text{L}) \qquad \text{C.4.1}}{(\text{order\_empty} \otimes (\text{user} \otimes \text{securitychecker}), (\text{item} \otimes \text{rate}) \vdash \text{order\_unpaid})}\,(\text{cut})}{(\text{order\_empty} \otimes (\text{user} \otimes \text{securitychecker}) \otimes (\text{item} \otimes \text{rate}) \vdash \text{order\_unpaid})}\,(\otimes\,\text{L})$$

C.4.3

$$\dfrac{\dfrac{\text{C.4.2} \qquad \dfrac{\dfrac{(\vdash \text{order\_unpaid} \multimap \text{Linsureorder} \,\&\, \text{Lpayorder})}{(\text{order\_unpaid} \vdash \text{Linsureorder} \,\&\, \text{Lpayorder})}\,(\text{shift}) \qquad \dfrac{(\text{insurance, Linsureorder} \vdash \text{order\_unpaid\_insured})}{(\text{insurance, Linsureorder} \,\&\, \text{Lpayorder} \vdash \text{order\_unpaid\_insured})}\,(\&\ \text{L\_1})}{(\text{insurance, order\_unpaid} \vdash \text{order\_unpaid\_insured})}\,(\text{cut})}{(\text{order\_empty} \otimes (\text{user} \otimes \text{securitychecker}) \otimes (\text{item} \otimes \text{rate}), \text{insurance} \vdash \text{order\_unpaid\_insured})}\,(\text{cut})}{(\text{order\_empty} \otimes (\text{user} \otimes \text{securitychecker}) \otimes (\text{item} \otimes \text{rate}) \otimes \text{insurance} \vdash \text{order\_unpaid\_insured})}\,(\otimes\ \text{L})$$

C.4.4

$$\dfrac{\dfrac{\text{C.4.3} \qquad \dfrac{\dfrac{(\vdash \text{order\_unpaid\_insured} \multimap \text{Lpayorder})}{(\text{order\_unpaid\_insured} \vdash \text{Lpayorder})}\,(\text{shift}) \qquad (\text{payment, Lpayorder} \vdash \text{order\_insured\_paid})}{(\text{payment, order\_unpaid\_insured} \vdash \text{order\_paid\_insured})}\,(\text{cut})}{(\text{order\_empty} \otimes (\text{user} \otimes \text{securitychecker}) \otimes (\text{item} \otimes \text{rate}) \otimes \text{insurance, payment} \vdash \text{order\_insured\_paid})}\,(\text{cut})}{(\text{order\_empty} \otimes (\text{user} \otimes \text{securitychecker}) \otimes (\text{item} \otimes \text{rate}) \otimes \text{insurance} \otimes \text{payment} \vdash \text{order\_insured\_paid})}\,(\otimes\ \text{L})$$

C.4.5

$$\dfrac{\dfrac{\text{C.4.4} \qquad \dfrac{\dfrac{(\vdash \text{order\_insured\_paid} \multimap \text{Lshiporder})}{(\text{order\_insured\_paid} \vdash \text{Lshiporder})}\,(\text{shift}) \qquad (\text{shipment, Lshiporder} \vdash \text{order\_shipped})}{(\text{shipment, order\_insured\_paid} \vdash \text{order\_shipped})}\,(\text{cut})}{(\text{order\_empty} \otimes (\text{user} \otimes \text{securitychecker}) \otimes (\text{item} \otimes \text{rate}) \otimes \text{insurance} \otimes \text{payment, shipment} \vdash \text{order\_shipped})}\,(\text{cut})}{(\text{order\_empty} \otimes (\text{user} \otimes \text{securitychecker}) \otimes (\text{item} \otimes \text{rate}) \otimes \text{insurance} \otimes \text{payment} \otimes \text{shipment} \vdash \text{order\_shipped})}\,(\otimes\ \text{L})$$

C.5 E-shopping Scenario Case III at the Second Stage

C.5.1

$$\dfrac{\dfrac{}{\text{uriuid}, \vdash \text{ruid} \oplus \text{err}}\,(\text{get\_uid}) \qquad \dfrac{}{\text{urisechec, ruid} \oplus \text{err} \vdash \text{ruidvalid} \oplus \text{err}}\,(\text{validate\_user})}{\text{uriuid, urisechec} \vdash \text{ruidvalid} \oplus \text{err}}\,(\text{Cut})$$

C.5.2

$$\dfrac{\dfrac{}{\text{uriorder} \vdash \text{uriorder}}\,(\text{Id}) \qquad \dfrac{\dfrac{\dfrac{}{\text{uriiid} \vdash \text{riid} \oplus \text{err}}\,(\text{get\_iid}) \qquad \dfrac{}{\text{urirate}, \text{riid} \oplus \text{err} \vdash \text{riidrated} \oplus \text{err}}\,(\text{rate\_item})}{\text{uriiid}, \text{urirate} \vdash \text{riidrated} \oplus \text{err}}\,(\text{Cut}) \qquad \text{C.5.1}}{\text{uriiid}, \text{urirate}, \text{uriuid}, \text{urisechec} \vdash (\text{riidrated} \oplus \text{err}) \otimes (\text{ruidvalid} \oplus \text{err})}\,(\otimes\text{R})}{\text{uriorder}, \text{uriiid}, \text{urirate}, \text{uriuid}, \text{urisechec} \vdash \text{uriorder} \otimes (\text{riidrated} \oplus \text{err}) \otimes (\text{ruidvalid} \oplus \text{err})}\,(\otimes\text{R})$$

C.5.3

$$\dfrac{\text{C.5.2} \qquad \dfrac{\dfrac{\dfrac{}{\text{uriorder}, (\text{riidrated} \oplus \text{err}), (\text{ruidvalid} \oplus \text{err}) \vdash \text{roidunpaid} \oplus \text{err}}\,(\text{place\_order})}{\text{uriorder}, (\text{riidrated} \oplus \text{err}) \otimes (\text{ruidvalid} \oplus \text{err}) \vdash \text{roidunpaid} \oplus \text{err}}\,(\otimes\text{L})}{\text{uriorder} \otimes (\text{riidrated} \oplus \text{err}) \otimes (\text{ruidvalid} \oplus \text{err}) \vdash \text{roidunpaid} \oplus \text{err}}\,(\otimes\text{L})}{\text{uriorder}, \text{uriiid}, \text{urirate}, \text{uriuid}, \text{urisechec} \vdash \text{roidunpaid} \oplus \text{err}}\,(\text{Cut})$$

C.5.4

$$\dfrac{\text{C.5.3} \qquad \dfrac{\dfrac{}{\vdash (\text{roidunpaid} \oplus \text{err}) \multimap (\text{linsureorder} \oplus \text{err})}\,(\text{link\_order\_insurance})}{\text{roidunpaid} \oplus \text{err} \vdash \text{linsureorder} \oplus \text{err}}\,(\text{Shift})}{\text{uriorder}, \text{uriiid}, \text{urirate}, \text{uriuid}, \text{urisechec} \vdash \text{linsureorder} \oplus \text{err}}\,(\text{Cut})$$

C.5.5

$$\dfrac{\dfrac{\dfrac{}{\text{uriinsur} \vdash \text{uriinsur}}\,(\text{Id}) \qquad \text{C.5.4}}{\text{uriinsur}, \text{uriorder}, \text{uriiid}, \text{urirate}, \text{uriuid}, \text{urisechec} \vdash \text{uriinsur} \otimes (\text{linsureorder} \oplus \text{err})}\,(\otimes\text{R}) \qquad \dfrac{\dfrac{}{\text{uriinsur}, (\text{linsureorder} \oplus \text{err}) \vdash \text{roidunpaidinsured} \oplus \text{err}}\,(\text{insure\_unpaid\_order})}{\text{uriinsur} \otimes (\text{linsureorder} \oplus \text{err}) \vdash \text{roidunpaidinsured} \oplus \text{err}}\,(\otimes\text{L})}{\text{uriinsur}, \text{uriorder}, \text{uriiid}, \text{urirate}, \text{uriuid}, \text{urisechec} \vdash \text{roidunpaidinsured} \oplus \text{err}}\,(\text{Cut})$$

C.5.6

$$\dfrac{\text{C.5.5} \qquad \dfrac{\dfrac{}{\vdash \text{roidunpaidinsured} \oplus \text{err} \multimap \text{lpayorder} \oplus \text{err}}\;\text{(link\_after\_insured\_unpaid\_order)}}{\text{roidunpaidinsured} \oplus \text{err} \vdash \text{lpayorder} \oplus \text{err}}\;\text{(Shift)}}{\text{uriinsur, uriorder, uriiid, urirate, uriuid, urisechec} \vdash \text{lpayorder} \oplus \text{err}}\;\text{(Cut)}$$

C.5.7

$$\dfrac{\dfrac{}{\text{uripay} \vdash \text{uripay}}\;\text{(Id)} \qquad \dfrac{\dfrac{}{\text{uriuidpay} \vdash \text{ruidpay} \oplus \text{err}}\;\text{(get\_uid\_pay)} \qquad \text{C.5.6}}{\text{uriuidpay, uriinsur, uriorder, uriiid, urirate, uriuid, urisechec} \vdash (\text{ruidpay} \oplus \text{err}) \otimes (\text{lpayorder} \oplus \text{err})}\;(\otimes\text{R})}{\text{uripay, uriuidpay, uriinsur, uriorder, uriiid, urirate, uriuid, urisechec} \vdash \text{uripay} \otimes (\text{ruidpay} \oplus \text{err}) \otimes (\text{lpayorder} \oplus \text{err})}\;(\otimes\text{R})$$

C.5.8

$$\dfrac{\text{C.5.7} \qquad \dfrac{\dfrac{\dfrac{}{(\text{uripay}, ((\text{ruidpay} \oplus \text{err}), (\text{lpayorder} \oplus \text{err})) \vdash (\text{roidinsuredpaid} \oplus \text{err})}\;\text{(pay\_insured\_order)}}{(\text{uripay}, ((\text{ruidpay} \oplus \text{err}) \otimes (\text{lpayorder} \oplus \text{err})) \vdash (\text{roidinsuredpaid} \oplus \text{err})}\;(\otimes\text{L})}{(\text{uripay} \otimes ((\text{ruidpay} \oplus \text{err}) \otimes (\text{lpayorder} \oplus \text{err})) \vdash (\text{roidinsuredpaid} \oplus \text{err})}\;(\otimes\text{L})}{\text{uripay, uriuidpay, uriinsur, uriorder, uriiid, urirate, uriuid, urisechec} \vdash (\text{roidinsuredpaid} \oplus \text{err})}\;\text{(Cut)}$$

C.5.9

$$\dfrac{\dfrac{}{\text{uriship} \vdash \text{uriship}}\;\text{(Id)} \qquad \dfrac{\text{C.5.8} \qquad \dfrac{\dfrac{}{\vdash (\text{roidinsuredpaid} \oplus \text{err}) \multimap (\text{lshiporder} \oplus \text{err})}\;\text{(link\_shiporder2)}}{(\text{roidinsuredpaid} \oplus \text{err}) \vdash (\text{lshiporder} \oplus \text{err})}\;\text{(Shift)}}{\text{uripay, uriuidpay, uriinsur, uriorder, uriiid, urirate, uriuid, urisechec} \vdash (\text{lshiporder} \oplus \text{err})}\;\text{(Cut)}}{\text{uriship, uripay, uriuidpay, uriinsur, uriorder, uriiid, urirate, uriuid, urisechec} \vdash \text{uriship} \otimes (\text{lshiporder} \oplus \text{err})}\;(\otimes\text{R})$$

C.5.10

$$\dfrac{\text{C.5.9} \qquad \dfrac{\dfrac{}{\text{uriship}, (\text{lshiporder} \oplus \text{err}) \vdash (\text{roidshipped} \oplus \text{err})}\;\text{(ship\_order)}}{\text{uriship} \otimes (\text{lshiporder} \oplus \text{err}) \vdash (\text{roidshipped} \oplus \text{err})}\;(\otimes\text{L})}{\text{uriship, uripay, uriuidpay, uriinsur, uriorder, uriiid, urirate, uriuid, urisechec} \vdash (\text{roidshipped} \oplus \text{err})}\;\text{(Cut)}$$

C.6 Biomedical Scenario at the First Stage - Surface Decimation Applied Once

C.6.1

$$\dfrac{\dfrac{\dfrac{}{\text{Lextractisosurface, isoextracter} \vdash \text{vtksurface}}\ (\text{extractisosurface})}{\text{isoextracter, Lextractisosurface} \vdash \text{vtksurface}}\ (\text{Exchange})}{\text{isoextracter, Lextractisosurface \& Lcropvolume} \vdash \text{vtksurface}}\ (\&\ L_1)$$

C.6.2

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\text{ct, importer} \vdash \text{vtkvolume}}\ (\text{importdicom})}{\text{ct} \otimes \text{importer} \vdash \text{vtkvolume}}\ (\otimes\text{L}) \qquad \dfrac{\dfrac{}{\vdash \text{vtkvolume} \multimap \text{Lextractisosurface \& Lcropvolume}}\ (\text{link\_after\_volume})}{\text{vtkvolume} \vdash \text{Lextractisosurface \& Lcropvolume}}\ (\text{Shift})}{\text{ct} \otimes \text{importer} \vdash \text{Lextractisosurface \& Lcropvolume}}\ (\text{Cut}) \qquad \text{C.6.1}}{\text{ct} \otimes \text{importer, isoextracter} \vdash \text{vtksurface}}\ (\text{Cut})$$

C.6.3

$$\dfrac{\dfrac{\text{C.6.2}}{\text{ct} \otimes \text{importer} \otimes \text{isoextracter} \vdash \text{vtksurface}}\ (\otimes\text{L}) \qquad \dfrac{\dfrac{}{\vdash \text{vtksurface} \multimap \text{(Ldecimate \& Lsmooth) \& Lbuildmesh}}\ (\text{link\_after\_surface})}{\text{vtksurface} \vdash \text{(Ldecimate \& Lsmooth) \& Lbuildmesh}}\ (\text{Shift})}{\text{ct} \otimes \text{importer} \otimes \text{isoextracter} \vdash \text{(Ldecimate \& Lsmooth) \& Lbuildmesh}}\ (\text{Cut})$$

C.6.4

$$\dfrac{\text{C.6.3} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\text{Ldecimate, decimatefilter} \vdash \text{vtksurface}}\ (\text{decimatesurface})}{\text{decimatefilter, Ldecimate} \vdash \text{vtksurface}}\ (\text{Exchange})}{\text{decimatefilter, (Ldecimate \& Lsmooth)} \vdash \text{vtksurface}}\ (\&\ L_1)}{\text{decimatefilter, (Ldecimate \& Lsmooth) \& Lbuildmesh} \vdash \text{vtksurface}}\ (\&\ L_1)}{\text{ct} \otimes \text{importer} \otimes \text{isoextracter, decimatefilter} \vdash \text{vtksurface}}\ (\text{Cut})$$

C.6.5

$$\dfrac{\dfrac{\text{C.6.4}}{\text{ct} \otimes \text{importer} \otimes \text{isoextracter} \otimes \text{decimatefilter} \vdash \text{vtksurface}}\,(\otimes\text{L}) \qquad \dfrac{\dfrac{}{\vdash \text{vtksurface} \multimap (\text{Ldecimate} \mathbin{\&} \text{Lsmooth}) \mathbin{\&} \text{Lbuildmesh}}\,(\text{link\_after\_surface})}{\text{vtksurface} \vdash (\text{Ldecimate} \mathbin{\&} \text{Lsmooth}) \mathbin{\&} \text{Lbuildmesh}}\,(\text{Shift})}{\text{ct} \otimes \text{importer} \otimes \text{isoextracter} \otimes \text{decimatefilter} \vdash (\text{Ldecimate} \mathbin{\&} \text{Lsmooth}) \mathbin{\&} \text{Lbuildmesh}}\,(\text{Cut})$$

C.6.6

$$\dfrac{\text{C.6.5} \qquad \dfrac{\dfrac{\dfrac{}{\text{Lbuildmesh, mesh} \vdash \text{mesh3d}}\,(\text{buildmesh})}{\text{mesh, Lbuildmesh} \vdash \text{mesh3d}}\,(\text{Exchange})}{\text{mesh, } (\text{Ldecimate} \mathbin{\&} \text{Lsmooth}) \mathbin{\&} \text{Lbuildmesh} \vdash \text{mesh3d}}\,(\mathbin{\&} L_2)}{\text{ct} \otimes \text{importer} \otimes \text{isoextracter} \otimes \text{decimatefilter} \otimes \text{mesh} \vdash \text{mesh3d}}\,(\text{Cut})$$

C.7 Biomedical Scenario at the First Stage - Surface Decimation Applied Twice

C.7.1

$$\dfrac{\dfrac{\dfrac{}{\text{Lextractisosurface, isoextracter} \vdash \text{vtksurface}}\,(\text{extractisosurface})}{\text{isoextracter, Lextractisosurface} \vdash \text{vtksurface}}\,(\text{Exchange})}{\text{isoextracter, Lextractisosurface} \mathbin{\&} \text{Lcropvolume} \vdash \text{vtksurface}}\,(\mathbin{\&} L_1)$$

C.7.2

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\text{ct, importer} \vdash \text{vtkvolume}}\,(\text{importdicom})}{\text{ct} \otimes \text{importer} \vdash \text{vtkvolume}}\,(\otimes\text{L}) \qquad \dfrac{\dfrac{}{\vdash \text{vtkvolume} \multimap \text{Lextractisosurface} \mathbin{\&} \text{Lcropvolume}}\,(\text{link\_after\_volume})}{\text{vtkvolume} \vdash \text{Lextractisosurface} \mathbin{\&} \text{Lcropvolume}}\,(\text{Shift})}{\text{ct} \otimes \text{importer} \vdash \text{Lextractisosurface} \mathbin{\&} \text{Lcropvolume}}\,(\text{Cut}) \qquad \text{C.7.1}}{\text{ct} \otimes \text{importer, isoextracter} \vdash \text{vtksurface}}\,(\text{Cut})$$

C.7.3

$$\dfrac{\dfrac{\text{C.7.2}}{\text{ct} \otimes \text{importer} \otimes \text{isoextracter} \otimes \text{decimatefilter} \vdash \text{vtksurface}}\,(\otimes\text{L}) \qquad \dfrac{\dfrac{}{\vdash \text{vtksurface} \multimap (\text{Ldecimate} \,\&\, \text{Lsmooth}) \,\&\, \text{Lbuildmesh}}\,(\text{link\_after\_surface})}{\text{vtksurface} \vdash (\text{Ldecimate} \,\&\, \text{Lsmooth}) \,\&\, \text{Lbuildmesh}}\,(\text{Shift})}{\text{ct} \otimes \text{importer} \otimes \text{isoextracter} \vdash (\text{Ldecimate} \,\&\, \text{Lsmooth}) \,\&\, \text{Lbuildmesh}}\,(\text{Cut})$$

C.7.4

$$\dfrac{\text{C.7.3} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\text{Ldecimate}, \text{decimatefilter} \vdash \text{vtksurface}}\,(\text{decimatesurface})}{\text{decimatefilter}, \text{Ldecimate} \vdash \text{vtksurface}}\,(\text{Exchange})}{\text{decimatefilter}, (\text{Ldecimate} \,\&\, \text{Lsmooth}) \vdash \text{vtksurface}}\,(\&\, L_1)}{\text{decimatefilter}, (\text{Ldecimate} \,\&\, \text{Lsmooth}) \,\&\, \text{Lbuildmesh} \vdash \text{vtksurface}}\,(\&\, L_1)}{\text{ct} \otimes \text{importer} \otimes \text{isoextracter}, \text{decimatefilter} \vdash \text{vtksurface}}\,(\text{Cut})$$

C.7.5

$$\dfrac{\dfrac{\text{C.7.4}}{\text{ct} \otimes \text{importer} \otimes \text{isoextracter} \otimes \text{decimatefilter} \vdash \text{vtksurface}}\,(\otimes\text{L}) \qquad \dfrac{\dfrac{}{\vdash \text{vtksurface} \multimap (\text{Ldecimate} \,\&\, \text{Lsmooth}) \,\&\, \text{Lbuildmesh}}\,(\text{link\_after\_surface})}{\text{vtksurface} \vdash (\text{Ldecimate} \,\&\, \text{Lsmooth}) \,\&\, \text{Lbuildmesh}}\,(\text{Shift})}{\text{ct} \otimes \text{importer} \otimes \text{isoextracter} \otimes \text{decimatefilter} \vdash (\text{Ldecimate} \,\&\, \text{Lsmooth}) \,\&\, \text{Lbuildmesh}}\,(\text{Cut})$$

C.7.6

$$\dfrac{\text{C.7.5} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\text{Ldecimate}, \text{decimatefilter} \vdash \text{vtksurface}}\,(\text{decimatesurface})}{\text{decimatefilter}, \text{Ldecimate} \vdash \text{vtksurface}}\,(\text{Exchange})}{\text{decimatefilter}, (\text{Ldecimate} \,\&\, \text{Lsmooth}) \vdash \text{vtksurface}}\,(\&\, L_1)}{\text{decimatefilter}, (\text{Ldecimate} \,\&\, \text{Lsmooth}) \,\&\, \text{Lbuildmesh} \vdash \text{vtksurface}}\,(\&\, L_1)}{\text{ct} \otimes \text{importer} \otimes \text{isoextracter} \otimes \text{decimatefilter}, \text{decimatefilter} \vdash \text{vtksurface}}\,(\text{Cut})$$

C.7.7

$$\dfrac{\dfrac{\text{C.7.6}}{\text{ct} \otimes \text{importer} \otimes \text{isoextracter} \otimes \text{decimatefilter} \otimes \text{decimatefilter} \vdash \text{vtksurface}}\,(\otimes\text{L}) \qquad \dfrac{\dfrac{}{\vdash \text{vtksurface} \multimap (\text{Ldecimate} \,\&\, \text{Lsmooth}) \,\&\, \text{Lbuildmesh}}\,(\text{link\_after\_surface})}{\text{vtksurface} \vdash (\text{Ldecimate} \,\&\, \text{Lsmooth}) \,\&\, \text{Lbuildmesh}}\,(\text{Shift})}{\text{ct} \otimes \text{importer} \otimes \text{isoextracter} \otimes \text{decimatefilter} \otimes \text{decimatefilter} \vdash (\text{Ldecimate} \,\&\, \text{Lsmooth}) \,\&\, \text{Lbuildmesh}}\,(\text{Cut})$$

C.7.8

$$\dfrac{\text{C.7.7} \qquad \dfrac{\dfrac{\dfrac{}{\text{Lbuildmesh}, \text{mesh} \vdash \text{mesh3d}}\,(\text{buildmesh})}{\text{mesh}, \text{Lbuildmesh} \vdash \text{mesh3d}}\,(\text{Exchange})}{\text{mesh}, (\text{Ldecimate} \,\&\, \text{Lsmooth}) \,\&\, \text{Lbuildmesh} \vdash \text{mesh3d}}\,(\&\, L_2)}{\text{ct} \otimes \text{importer} \otimes \text{isoextracter} \otimes \text{decimatefilter} \otimes \text{decimatefilter} \otimes \text{mesh} \vdash \text{mesh3d}}\,(\text{Cut})$$

C.8 Biomedical Scenario at the Second Stage

C.8.1

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\text{uriiso}, (\text{lexiso} \oplus \text{err}) \vdash \text{rvtksur} \oplus \text{err}}\,(\text{extract\_isosurface})}{\text{uriiso}, ((\text{lexiso} \oplus \text{err}) \,\&\, (\text{lcropvol} \oplus \text{err})) \vdash \text{rvtksur} \oplus \text{err}}\,(\&\text{L}_1)}{\text{uriiso} \otimes ((\text{lexiso} \oplus \text{err}) \,\&\, (\text{lcropvol} \oplus \text{err})) \vdash \text{rvtksur} \oplus \text{err}}\,(\otimes\text{L}) \qquad \dfrac{\dfrac{}{\vdash (\text{rvtksur} \oplus \text{err}) \multimap (\text{ldec} \oplus \text{err}) \,\&\, (\text{lsmo} \oplus \text{err})) \,\&\, (\text{lmesh} \oplus \text{err})}\,(\text{link\_after\_surface})}{\text{rvtksur} \oplus \text{err} \vdash (\text{ldec} \oplus \text{err}) \,\&\, (\text{lsmo} \oplus \text{err})) \,\&\, (\text{lmesh} \oplus \text{err})}\,(\text{Shift})}{\text{uriiso} \otimes ((\text{lexiso} \oplus \text{err}) \,\&\, (\text{lcropvol} \oplus \text{err})) \vdash (\text{ldec} \oplus \text{err}) \,\&\, (\text{lsmo} \oplus \text{err})) \,\&\, (\text{lmesh} \oplus \text{err})}\,(\text{Cut})$$

C.8.2

$$\dfrac{\dfrac{\dfrac{}{\text{uriiso} \vdash \text{uriiso}}\,(\text{Id}) \quad \dfrac{\dfrac{}{\text{uriimp, urictid} \vdash \text{rvtkvol} \oplus \text{err}}\,(\text{import\_dicom}) \quad \dfrac{\dfrac{}{\vdash (\text{rvtkvol} \oplus \text{err}) \multimap ((\text{lexiso} \oplus \text{err}) \mathbin{\&} (\text{lcropvol} \oplus \text{err}))}\,(\text{import\_after\_volume})}{\text{rvtkvol} \oplus \text{err} \vdash ((\text{lexiso} \oplus \text{err}) \mathbin{\&} (\text{lcropvol} \oplus \text{err}))}\,(\text{Shift})}{\text{uriimp, urictid} \vdash ((\text{lexiso} \oplus \text{err}) \mathbin{\&} (\text{lcropvol} \oplus \text{err}))}\,(\text{Cut})}{\text{uriiso, uriimp, urictid} \vdash \text{uriiso} \otimes ((\text{lexiso} \oplus \text{err}) \mathbin{\&} (\text{lcropvol} \oplus \text{err}))}\,(\otimes\text{R}) \quad \text{C.8.1}}{\text{uriiso, uriimp, urictid} \vdash (\text{ldec} \oplus \text{err}) \mathbin{\&} (\text{lsmo} \oplus \text{err})) \mathbin{\&} (\text{lmesh} \oplus \text{err})}\,(\text{Cut})$$

C.8.3

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\text{uridec}, (\text{ldec} \oplus \text{err}) \vdash \text{rvtksur} \oplus \text{err}}\,(\text{decimate\_surface})}{\text{uridec}, ((\text{ldec} \oplus \text{err}) \mathbin{\&} (\text{lsmo} \oplus \text{err})) \vdash \text{rvtksur} \oplus \text{err}}\,(\&\text{L}_1)}{\text{uridec}, ((\text{ldec} \oplus \text{err}) \mathbin{\&} (\text{lsmo} \oplus \text{err})) \mathbin{\&} (\text{lmesh} \oplus \text{err}) \vdash \text{rvtksur} \oplus \text{err}}\,(\&\text{L}_1)}{\text{uridec} \otimes ((\text{ldec} \oplus \text{err}) \mathbin{\&} (\text{lsmo} \oplus \text{err})) \mathbin{\&} (\text{lmesh} \oplus \text{err}) \vdash \text{rvtksur} \oplus \text{err}}\,(\otimes\text{L}) \quad \dfrac{\dfrac{}{\vdash (\text{rvtksur} \oplus \text{err}) \multimap (\text{ldec} \oplus \text{err}) \mathbin{\&} (\text{lsmo} \oplus \text{err}) \mathbin{\&} (\text{lmesh} \oplus \text{err})}\,(\text{link\_after\_surface})}{\text{rvtksur} \oplus \text{err} \vdash (\text{ldec} \oplus \text{err}) \mathbin{\&} (\text{lsmo} \oplus \text{err}) \mathbin{\&} (\text{lmesh} \oplus \text{err})}\,(\text{Shift})}{\text{uridec} \otimes ((\text{ldec} \oplus \text{err}) \mathbin{\&} (\text{lsmo} \oplus \text{err}) \mathbin{\&} (\text{lmesh} \oplus \text{err})) \vdash (\text{ldec} \oplus \text{err}) \mathbin{\&} (\text{lsmo} \oplus \text{err})) \mathbin{\&} (\text{lmesh} \oplus \text{err})}\,(\text{Cut})$$

C.8.4

$$\dfrac{\dfrac{\dfrac{}{\text{uridec} \vdash \text{uridec}}\,(\text{Id}) \quad \text{C.8.2}}{\text{uridec, uriiso, uriimp, urictid} \vdash \text{uridec} \otimes ((\text{ldec} \oplus \text{err}) \mathbin{\&} (\text{lsmo} \oplus \text{err})) \mathbin{\&} (\text{lmesh} \oplus \text{err}))}\,(\otimes\text{R}) \quad \text{C.8.3}}{\text{uridec, uriiso, uriimp, urictid} \vdash (\text{ldec} \oplus \text{err}) \mathbin{\&} (\text{lsmo} \oplus \text{err})) \mathbin{\&} (\text{lmesh} \oplus \text{err})}\,(\text{Cut})$$

C.8.5

$$\dfrac{\dfrac{\dfrac{}{\text{urimesh}, (\text{lmesh} \oplus \text{err}) \vdash \text{rmesh} \oplus \text{err}}\,(\text{build\_mesh})}{\text{urimesh}, ((\text{ldec} \oplus \text{err}) \mathbin{\&} (\text{lsmo} \oplus \text{err})) \mathbin{\&} (\text{lmesh} \oplus \text{err})) \vdash \text{rmesh} \oplus \text{err}}\,(\&\text{L}_2)}{\text{urimesh} \otimes ((\text{ldec} \oplus \text{err}) \mathbin{\&} (\text{lsmo} \oplus \text{err})) \mathbin{\&} (\text{lmesh} \oplus \text{err})) \vdash \text{rmesh} \oplus \text{err}}\,(\otimes\text{L})$$

C.8.6

$$\dfrac{\dfrac{\dfrac{}{\text{urimesh} \vdash \text{urimesh}}\,(\text{Id}) \qquad \text{C.8.4}}{\text{urimesh, uridec, uriiso, uriimp, urictid} \vdash \text{urimesh} \otimes ((\text{ldec} \oplus \text{err}) \mathbin{\&} (\text{lsmo} \oplus \text{err})) \mathbin{\&} (\text{lmesh} \oplus \text{err}))}\,(\otimes\text{R}) \qquad \text{C.8.5}}{\text{urimesh, uridec, uriiso, uriimp, urictid} \vdash \text{rmesh} \oplus \text{err}}\,(\text{Cut})$$

References

[1] R. T. Fielding, *Architectural styles and the design of network-based software architectures.* PhD thesis, University of California, Irvine, 2000.

[2] L. Richardson and S. Ruby, *RESTful Web Services.* O'Reilly Media, 2007.

[3] L. Li and W. Chou, "Design and describe REST API without violating REST: A Petri net based approach," in *Proc of 2011 IEEE International Conference on Web Services (ICWS)*, pp. 508–515, IEEE, 2011.

[4] E. Cerami and S. St Laurent, *Web Services essentials.* O'Reilly & Associates, Inc., 2002.

[5] F. Coyle, *XML, Web Services, and the data revolution.* Addison-Wesley Longman Publishing Co., Inc., 2002.

[6] Google Developers, "Google custom search." Retrieved July 2012 from `https://developers.google.com/custom-search/`.

[7] Amazon, "Amazon Web Services." Retrieved July 2012 from `http://aws.amazon.com/`.

[8] Yahoo! Developer Network, "Yahoo! Web Services and APIs." Retrieved July 2012 from `http://developer.yahoo.com/about/`.

[9] C. Pautasso, O. Zimmermann, and F. Leymann, "RESTful Web Services vs. Big' Web Services: making the right architectural decision," in *Proc of the 17th international conference on World Wide Web*, pp. 805–814, ACM, 2008.

[10] T. Singh, "REST vs. SOAP: The right Web Service." Retrieved July 2012 from `http://geeknizer.com/rest-vs-soap-using-http-choosing-the-right-webservice-protocol/`.

[11] M. Rozlog, "REST and SOAP: When should I use each (or both)." Retrieved July 2012 from `http://www.infoq.com/articles/rest-soap-when-to-use-each/`.

[12] E. Wilde, "Declarative Web 2.0," in *Proc of IEEE International Conference on Information Reuse and Integration (IRI 2007)*, pp. 612–617, IEEE, 2007.

[13] M. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-oriented computing: State of the art and research challenges," *Computer*, vol. 40, no. 11, pp. 38–45, 2007.

[14] B. Srivastava and J. Koehler, "Web service composition-current solutions and open problems," in *Proc of the 2003 Workshop on Planning for Web Services (ICAPS)*, vol. 35, 2003.

[15] N. Milanovic and M. Malek, "Current solutions for web service composition," *IEEE Internet Computing*, vol. 8, no. 6, pp. 51–59, 2004.

[16] J. Rao and X. Su, "A survey of automated web service composition methods," *Semantic Web Services and Web Process Composition*, pp. 43–54, 2005.

[17] A. Bucchiarone and S. Gnesi, "A survey on services composition languages and models," in *Proc of the International Workshop on Web Services–Modeling and Testing (WS-MaTe 2006)*, p. 51, 2006.

[18] M. ter Beek, A. Bucchiarone, and S. Gnesi, "Web service composition approaches: From industrial standards to formal methods," in *Proc of the 2nd International Conference on Internet and Web Applications and Services (ICIW)*, pp. 15–15, IEEE, 2007.

[19] Z. Li, L. O'Brien, J. Keung, and X. Xu, "Effort-oriented classification matrix of web service composition," in *Proc of the 5th International Conference on Internet and Web Applications and Services (ICIW)*, pp. 357–362, IEEE, 2010.

[20] H. Zhao and P. Doshi, "Towards automated RESTful Web Service composition," in *Proc of IEEE International Conference on Web Services (ICWS 2009)*, pp. 189–196, IEEE, 2009.

[21] S. Vinoski, "RESTful Web Services development checklist," *Internet Computing, IEEE*, vol. 12, no. 6, pp. 96–95, 2008.

[22] X. Zhao, E. Liu, and G. J. Clapworthy, "A two-stage RESTful Web Service composition method based on Linear Logic," in *Proc of the 9th IEEE European Conference on Web Services (ECOWS 2011)*, pp. 39–46, IEEE, 2011.

[23] X. Zhao, E. Liu, G. J. Clapworthy, N. Ye, and Y. Lu, "RESTful Web Service composition: Extracting a process model from Linear Logic theorem proving," in *Proc of the 7th International Conference on Next Generation Web Services Practices (NWeSP 2011)*, pp. 398–403, IEEE, 2011.

[24] J. Levitt, "From EDI to XML and UDDI: A brief history of Web Services," *Information Week*, 2001.

[25] G. Alonso, *Web Services: concepts, architectures and applications.* Springer Verlag, 2004.

[26] W3C Working Group and others, "Web Services architecture," *W3C Note*, 2004.

[27] Amazon, "Amazon Simple Storage Services." Retrieved July 2012 from `http://aws.amazon.com/s3/`.

[28] E. Tholomé, "A well earned retirement for the SOAP search API." Retrieved October 2012 from `http://googlecode.blogspot.co.uk/2009/08/well-earned-retirement-for-soap-search.html`.

[29] P. Stevens, R. J. Pooley, and R. Pooley, *Using UML: software engineering with objects and components.* Addison-Wesley Longman, 2006.

[30] H. E. Eriksson and M. Penker, *Business modeling with UML.* John Wiley & Sons, 2000.

[31] I. Porres and I. Rauf, "Modeling behavioral RESTful Web Service interfaces in UML," in *Proc of the 2011 ACM Symposium on Applied Computing*, pp. 1598–1605, ACM, 2011.

[32] I. Rauf, A. Ruokonen, T. Systa, and I. Porres, "Modeling a composite RESTful Web Service with UML," in *Proc of the 4th European Conference on Software Architecture: Companion Volume*, pp. 253–260, ACM, 2010.

[33] R. Alarcón and E. Wilde, "Linking data from RESTful services," in *Proc of the 3rd International Workshop on Linked Data on the Web*, 2010.

[34] I. Zuzak, I. Budiselic, and G. Delac, "A finite-state machine approach for modeling and analyzing RESTful systems," *Journal of Web Engineering*, vol. 10, no. 4, pp. 353–390, 2011.

[35] G. Decker, A. Lüders, H. Overdick, K. Schlichting, and M. Weske, "RESTful Petri net execution," *Web Services and Formal Methods*, pp. 73–87, 2009.

[36] A. G. Hernández and M. N. M. García, "A formal definition of RESTful Semantic Web Services," in *Proc of the 1st International Workshop on RESTful Design (WS-REST '10)*, pp. 39–45, ACM, 2010.

[37] X. Zhao, E. Liu, G. J. Clapworthy, M. Viceconti, and D. Testi, "SOA-based digital library services and composition in biomedical applications," *Computer Methods and Programs in Biomedicine*, vol. 106, no. 3, pp. 219 – 233, 2012.

[38] J. Kopecky, K. Gomadam, and T. Vitvar, "hRESTs: An HTML microformat for describing RESTful Web Services," in *Proc of the*

IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT'08), vol. 1, pp. 619–625, IEEE, 2008.

[39] J. Pasley, "How BPEL and SOA are changing web services development," *Internet Computing, IEEE*, vol. 9, no. 3, pp. 60–67, 2005.

[40] C. Pautasso, "RESTful Web Service composition with BPEL for REST," *Data & Knowledge Engineering*, vol. 68, no. 9, pp. 851–866, 2009.

[41] H. Yu, C. Zhu, H. Cai, and B. Xu, "Role-centric RESTful services description and composition for e-business applications," in *Proc of IEEE International Conference on e-Business Engineering (ICEBE'09)*, pp. 103–110, IEEE, 2009.

[42] F. Rosenberg, F. Curbera, M. J. Duftler, and R. Khalaf, "Composing RESTful services and collaborative workflows: A lightweight approach," *Internet Computing, IEEE*, vol. 12, no. 5, pp. 24–31, 2008.

[43] R. Alarcón, E. Wilde, and J. Bellido, "Hypermedia-driven RESTful service composition," *Service-Oriented Computing*, pp. 111–120, 2011.

[44] H. Q. Yu and S. Reiff-Marganiec, "Semantic web services composition via planning as model checking," *Department of Computer Science, University of Leicester*, 2006.

[45] A. Derezinska and R. Pilitowski, "Correctness issues of UML class and state machine models in the c$\sharp$ code generation and execution framework," in *Proc of International Multiconference on Computer Science and Information Technology*, pp. 517 –524, oct. 2008.

[46] G. Csertán, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varró, "VIATRA - visual automated transformations for formal verification and validation of UML models," in *Proc of the 17th IEEE International Conference on Automated Software Engineering*, pp. 267 – 270, 2002.

[47] J. Y. Girard, "Linear Logic," *Theoretical Computer Science*, vol. 50, no. 1, pp. 1–101, 1987.

[48] J.-Y. Girard, “Linear logic: its syntax and semantics,” in *Proc of the workshop on Advances in Linear Logic*, (New York), pp. 1–42, Cambridge University Press, 1995.

[49] G. M. Bierman, *On Intuitionistic Linear Logic.* PhD thesis, University of Cambridge, 1994.

[50] G. Gentzen, “Collected works. edited by m. e. szabo,” 1969.

[51] P. Lincoln, “Deciding provability of Linear Logic formulas,” *London Mathematical Society Lecture Note Series*, pp. 109–122, 1995.

[52] L. Dixon, A. Smaill, and T. Tsang, “Plans, actions and dialogues using Linear Logic,” *Journal of Logic, Language and Information*, vol. 18, no. 2, pp. 251–289, 2009.

[53] F. Collé, R. Champagnat, and A. Prigent, “Scenario analysis based on Linear Logic,” in *Proc of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology*, ACE ’05, (New York, NY, USA), ACM, 2005.

[54] J. Rao, *Semantic web service composition via logic-based program synthesis.* PhD thesis, Norwegian University of Science and Technology, 2004.

[55] P. Papapanagiotou and J. Fleuriot, “Formal verification of web services composition using linear logic and the π-calculus,” in *Proc of 9th IEEE European Conference on Web Services (ECOWS)*, pp. 31–38, IEEE, 2011.

[56] P. Papapanagiotou and J. D. Fleuriot, “A theorem proving framework for the formal verification of web services composition,” in *Proc of WWV’11*, pp. 1–16, 2011.

[57] L. Dixon, A. Smaill, and A. Bundy, “Planning as deductive synthesis in intuitionistic Linear Logic,” tech. rep., Technical Report EDI-INF-RR-0786, School of Informatics, University of Edinburgh, 2006.

[58] D. Basin, Y. Deville, P. Flener, A. Hamfelt, and J. Fischer Nilsson, "Synthesis of programs in computational logic," *Program Development in Computational Logic*, pp. 103–111, 2004.

[59] R. Lucchi and M. Mazzara, "A pi-calculus based semantics for WS-BPEL," *Journal of Logic and Algebraic Programming*, vol. 70, no. 1, pp. 96–118, 2007.

[60] W. M. P. van der Aalst, "Pi calculus versus Petri nets: Let us eat humble pie rather than further inflate the Pi hype," *BPTrends*, vol. 3, no. 5, pp. 1–11, 2005.

[61] G. Bellin and P. J. Scott, "On the π-calculus and Linear Logic," *Theoretical Computer Science*, vol. 135, no. 1, pp. 11–65, 1994.

[62] L. Caires and F. Pfenning, "A concurrent interpretation of intuitionistic Linear Logic." 2009.

[63] D. Miller, "The pi-calculus as a theory in Linear Logic: Preliminary results," in *Proc of the 3rd International Workshop on Extensions of Logic Programming*, ELP '92, (London, UK, UK), pp. 242–264, Springer-Verlag, 1993.

[64] M. Masseron, C. Tollu, and J. Vauzeilles, "Generating plans in Linear Logic: I. Actions as proofs," *Theoretical Computer Science*, vol. 113, no. 2, pp. 349 – 370, 1993.

[65] M. Masseron, "Generating plans in Linear Logic: II. A geometry of conjunctive actions," *Theoretical Computer Science*, vol. 113, no. 2, pp. 371 – 375, 1993.

[66] S. BuvaE and I. A. Mason, "Propositional logic of context," in *Proc of the 11th national conference on artificial intelligence*, 1993.

[67] S. Abramsky, "Proofs as processes," *Theoretical Computer Science*, vol. 135, pp. 5–9, April 1992.

[68] Z. Manna and R. J. Waldinger, "Toward automatic program synthesis," *Communications of the ACM*, vol. 14, no. 3, pp. 151–165, 1971.

[69] Z. Manna and R. Waldinger, "Fundamentals of deductive program synthesis," *IEEE Transactions on Software Engineering*, vol. 18, no. 8, pp. 674–704, 1992.

[70] H. Smith and P. Fingar, "Workflow is just a Pi process," *BPTrends, November*, pp. 1–36, 2003.

[71] R. Milner, *Communicating and mobile systems: the π-calculus.* Cambridge University Press, 1999.

[72] F. Puhlmann, "Why do we actually need the Pi-calculus for business process management," in *Proc of the 9th International Conference on Business Information Systems (BIS 2006)*, vol. 85, pp. 77–89, 2006.

[73] H. Overdick, "The resource-oriented architecture," in *Proc of 2007 IEEE Congress on Services*, pp. 340–347, IEEE, 2007.

[74] M. Lumpe, *A Pi-Calculus Based Approach to Software Composition.* PhD thesis, University of Bern, Institute of Computer Science and Applied Mathematics, 1999.

[75] J. Rao, P. Kungas, and M. Matskin, "Logic-based web services composition: from service description to process model," in *Proc of IEEE International Conference on Web Services*, pp. 446–453, IEEE, 2004.

[76] F. Abouzaid, "A mapping from Pi-calculus into BPEL," in *Proc of the 2006 conference on Leading the Web in Concurrent Engineering: Next Generation Concurrent Engineering*, (Amsterdam), pp. 235–242, IOS Press, 2006.

[77] J. S. Hodas, "Lolli: An extension of prolog with linear logic context management," *Proc of the workshop on the λProlog Programming Language*, 1992.

[78] D. Miller, "Forum: A multiple-conclusion specification logic," *Theoretical Computer Science*, vol. 165, no. 1, pp. 201–232, 1996.

[79] J. Harland, D. Pym, and M. Winikoff, "Programming in lygon: An overview," *Algebraic Methodology and Software Technology*, pp. 391–405, 1996.

[80] The Coq Development Team, "The Coq proof assistant reference manual v8.3," *INRIA*, July 20, 2011.

[81] L. C. Paulson, *Isabelle: A generic theorem prover*, vol. 828. Springer, 1994.

[82] N. Tamura, "User's guide of a linear logic theorem prover (llprover)," tech. rep., Technical report, Kobe University, Japan, 1998.

[83] D. Rémy, "Using, understanding, and unraveling the OCaml language from practice to theory and vice versa," *Applied Semantics*, pp. 115–137, 2002.

[84] T. Coquand and G. Huet *Information and Computation*, vol. 76, no. 2, pp. 95–120, 1988.

[85] J. Power and C. Webster, "Working with Linear Logic in Coq," in *Proc of the 12th International Conference on Theorem Proving in Higher Order Logics*, 1999.

[86] M. Sadrzadeh, "Modal Linear Logic in higher order logic, an experiment in Coq," in *Proc of Theorem Proving in Higher Order Logics*, no. 187, pp. 75–93, Aracne, 2003.

[87] D. Hirschkoff, "A full formalisation of π-calculus theory in the calculus of constructions," *Theorem Proving in Higher Order Logics*, pp. 153–169, 1997.

[88] G. A. Hansen, R. W. Douglass, and A. Zardecki, *Mesh enhancement: selected elliptic methods, foundations and applications*. Imperial College Press, 2005.