Matching and ASM-based Choreographing of Semantic Web Services Using F-Logic and Flora-2

Shahin Mehdipour Ataee

Submitted to the
Institute of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

Eastern Mediterranean University
January 2018
Gazimağusa, North Cyprus

Approval of the Institute of Graduate Studies and Research

Assoc. Prof. Dr. Ali Hakan Ulusoy
Acting Director

I certify that this thesis satisfies the requirements as a thesis for the degree of Doctor of Philosophy in Computer Engineering.

Prof. Dr. Işık Aybay
Chair, Department of Computer Engineering

We certify that we have read this thesis and that in our opinion it is fully adequate in scope and quality as a thesis for the degree of Doctor of Philosophy in Computer Engineering.

Assoc. Prof. Dr. Zeki Bayram
Supervisor

Examining Committee

1. Prof. Dr. Ferda Nur Alpaslan
2. Prof. Dr. Can Özturan
3. Assoc. Prof. Dr. Zeki Bayram
4. Assoc. Prof. Dr. Alexander Chefranov
5. Asst. Prof. Dr. Ahmet Ünveren

ABSTRACT

In this dissertation, we mainly reincarnated the Semantic Web Service Choreography that had been abandoned for a decade in spite of its interesting utility, especially in today's dominant web service-based applications. We took the original Abstract State Machine-based (ASM) choreography execution algorithm of Web Service Modeling Ontology (WSMO), determined its weaknesses and improved it so that it can be used effectively in the context of semantic web services. We then implemented a completely new choreography engine based on our improved algorithm.

Our work has been done in two phases. First, we concentrated on the "capability" component of WSMO and used Frame Logic (F-Logic) to specify it. The new model allows very short but expressive descriptions of both goals and web service capabilities, which are then used by a matching engine to discover which web services can satisfy a given goal. The matching engine, using the meta-level F-logic inferencing capabilities of the underlying Flora-2 reasoner, is very efficient and has a very concise definition itself.

In the second phase, we again used F-Logic for specifying ASM-based modeling of interactions between a requester of service and provider of service, also called choreography, of semantic web services described in conformance to WSMO. Our choreography execution engine, implemented in Flora-2, remains loyal to the parallelism and branching paradigms of ASMs (unlike previous implementations), and is based on our improved choreography execution algorithm, which has the following novelties over the original algorithm: (i) it introduces the concept of initial state in the execution of ASM and links it to the precondition of the goal, (ii) it introduces the

concept of a final state in the execution of ASM and links it to the post-condition of the goal, (iii) it modifies the execution of ASM so that it stops when the final state conditions are satisfied by the current configuration of the machine, as opposed to stopping only when the machine has no more moves.

As part of our work, we also developed a visual tool for specifying web service choreographies in Flora-2, provided a mapping between JSON and Flora-2 Web Service specification, and lastly defined a formal mapping between traditional ASMs and ontological ASMs that are the basis of choreography specifications and proved their equivalence, which was missing in the literature before.

Keywords: Semantic web, web services, service matching, service choreography, abstract state machine, Flora-2, F-logic, web reasoning

ÖZ

Bu tez çalışmasında, özellikle bugünün baskın web hizmeti tabanlı uygulamalarında ilginç yararına rağmen, on yıl boyunca terk edilmiş Semantik Web Hizmeti Koreografisini yeniden canlandırdık. Web Service Modeling Ontology'nin (WSMO) orijinal Abstract State Machine tabanlı (ASM) koreografi yürütme algoritmasını aldık, zayıf yönlerini belirledik ve semantik web hizmetleri bağlamında etkin bir şekilde kullanılabilecek şekilde geliştirdik. Ardından, geliştirilmiş algoritmamızı temel alarak tamamen yeni bir koreografi motoru hazırladık.

Çalışmalarımız iki aşamalı olarak gerçekleştirildi. İlk olarak, WSMO'nun "yetenek" bileşenine yoğunlaştık ve onu belirtmek için Çerçeve Mantığı (F-Logic) kullandık. Yeni model, hem hedeflerin hem de web hizmet yeteneklerinin çok kısa ancak somut tanımlamalarına izin verir; bunlar da eşleşme motoru tarafından hangi web hizmetlerinin belirli hangi hedefle uyumlu olduğunu keşfetmek için kullanılır. Altta yatan Flora-2 mantığının meta düzeyinde F-Logic çıkarım yeteneklerini kullanan eşleşme motoru çok verimli ve çok özlü bir tanımlamaya sahiptir.

İkinci aşamada, yine F-Logic'i, WSMO'ya uygun olarak tanımlanan semantik web servislerinin, koreografi olarak da adlandırılan, hizmet talep eden ve hizmet sağlayıcısı arasındaki etkileşimlerin ASM tabanlı modellemesini belirtmek için tekrar kullandık. Flora-2'de uygulanan koreografi yürütme motorumuz ASM'lerin paralellik ve dallanma paradigmalarına (önceki uygulamalardan farklı olarak) sadık kalmaktadır ve orjinal algoritma üzerinde aşağıdaki yenilikleri içeren geliştirilmiş koreografi yürütme algoritmamızı temel almaktadır: (i) ASM'nin yürütülmesinde nihai bir durum kavramını getirir ve hedefin son durumuna bağlar; (ii) ASM'nin uygulanmasında

başlangıç hali kavramını getirir ve onu hedefin ön koşuluna bağlar; (iii) sadece makine daha fazla hareket olmadığında durdurulması yerine, makinenin geçerli konfigürasyonu ile nihai durum koşulları yerine getirildiğinde ASM'nin çalışmasını durdurur.

Çalışmamızın bir parçası olarak, ayni zamanda Flora-2'de web hizmeti Koreografilerini belirlemek için görsel bir araç geliştirdik, JSON ve Flora-2 Web Hizmeti spesifikasyonu arasında bir eşleme yaptık ve son olarak daha önce literatürde eksik olan geleneksel ASM'ler ve ontolojik ASM'ler arasındaki formel bir haritalama tanımladık ve eşdeğerliklerini ispatladık.

Anahtar Kelimeler: Semantik web, web servisler, servis eşleştirme, servis koreografi, Soyut durum makinesi, Flora-2, F-Logic, webde mantik yürütme

ACKNOWLEDGMENT

First of all, I would like to sincerely thank all the committee members of my PhD defense session for their time, presence and assessment of this work: Prof. Dr. Ferda Nur Alpaslan, Prof. Dr. Can Özturan, Assoc. Prof. Dr. Alexandre Chefranov, and Assist. Prof. Dr. Ahmet Ünveren.

I should thank my supervisor Assoc. Prof. Dr. Zeki Bayram for all the things I have learned from him during our research. I wish for him and his family forever peace and health.

I cannot explain my appreciation to Assoc. Prof. Dr. Muhammed Salamah (the ex-chairman of the department) as well as Prof. Dr. Işık Aybay (the present chairman of the department) for the opportunity they have given me to work beside them as an assistant and a faculty member.

During seven years of studying and working at the department of computer engineering, I have had a great time beside the other faculty members and the great people around and I want to thank every one of them:

Prof. Dr. Erden Başar, Prof. Dr. Doğu Arifler, Prof. Dr. Hasan Kömürcügil, Prof. Dr. Marifi Güler, Prof. Dr. Omar Ramadan, Assoc. Prof. Dr. Ekrem Varoğlu, Assoc. Prof. Dr. Gürcü Öz, Assoc. Prof. Dr. Mehmet Bodur, Assoc. Prof. Dr. Önsen Toygar, special thanks to Assist. Prof. Dr. Adnan Acan, Assist. Prof. Dr. Cem Ergün, Mrs. Sulan Yiğithan, Mr. Mehmet Topal, Mr. Erdal Altün, Ms. Semiha Sakalli, Ms. Zühre Derebey and others.

I would like to thank all my dear friends whose their presences make me a happier person in life specially, I thank Miss. Shadi Bouloukifar.

Words are not enough to describe how much I owe to my beloved Mom Dr. Parvin Khajeh Moloei, my beloved Dad Dr. Bahman Mehdipour, and my lovely brothers Iman and Yazdan. Nothing is enough to appreciate you for being in my life and I dedicate this dissertation to you.

Warmest regards–January 2018

TABLE OF CONTENTS

ABSTRACT iii

ÖZ v

ACKNOWLEDGMENT vii

LIST OF TABLES xii

LIST OF FIGURES xiii

LIST OF ABBREVIATIONS xiv

1 INTRODUCTION 1

2 PRELIMINARIES 8

2.1 WSMO and WSMO choreography 8

2.2 Flora-2 and F-Logic 9

2.3 Abstract State Machine (ASM) 10

2.4 Summary 11

3 RELATED WORK AND PROBLEM DEFINITION 12

3.1 Semantic service matching 12

3.2 Semantic Web Service choreography 13

3.3 Problem definition 17

3.4 Summary 18

4 F-LOGIC BASED SERVICE MATCHING OF SEMANTIC WEB SERVICES 19

4.1 Semantic Web Service specifications and the matching process 19

4.2 Use case: medical appointment finder 22

4.3 Summary 25

5 ASM-BASED CHOREOGRAPHY OF SWS WITH F-LOGIC 26

5.1 Service composition and semantic Web Service choreography 27

5.2 ASM-based choreography in WSMO 29

5.2.1 Specification of choreographies in WSMO....30
5.3 Choreography matching algorithm in WSMO....32
5.4 Problems with the WSMO choreography algorithm....33
5.5 Improved choreography execution algorithm....34
5.6 Implementing the improved choreography algorithm in Flora-2....36
5.6.1 Semantic specification of goal and Web Service in Flora-2....36
5.6.2 Proposed architecture....38
5.6.3 Major predicates of the choreography engine....40
5.6.4 Running (firing) the rules....42
5.6.5 Contradictions in applying the rules....43
5.6.6 Access control to objects of different types....44
5.7 Semantic choreography specification examples....46
5.7.1 Flight ticket reservation....46
5.7.2 Flight ticket reservation with data granularity mismatch....52
5.7.3 Choosing a branch by the goal or Web Service....55
5.8 Handling granularity mismatch problem....58
5.9 Relationship between Evolving Algebra and Evolving Ontologies....59
5.9.1 Simulation of choreography engine execution via ASM....61
5.9.2 Simulation of ASM execution via choreography engine....64
5.10 Comparison with IRS-III and WSMX....67
5.11 Summary....68
6 TIMING EVALUATION....69
6.1 Experimental environment....70
6.2 Response time w.r.t. the number of rules....72
6.3 Response time w.r.t. the number of rules (re-firing is prevented)....75
6.4 Response time w.r.t. complexity of LHSs and RHSs....78

6.5 Summary 81

7 CONCLUSION AND FUTURE WORK 83

REFERENCES 84

APPENDICES 99

Appendix A: Visual editor for Flora-2 based SWS specifications (VSChor)......100

Appendix B: E-BNF grammar for Flora-2 goal and Web Service specifications106

Appendix C: Converting JSON to Flora-2...108

Appendix D: Choreography engine predicates list 112

Appendix E: Choreography engine source codes 115

Appendix F: More choreography specification examples...................................134

Appendix G: The source code of timing evaluation and the output raw data......153

LIST OF TABLES

Table 2.1: Summary of Flora-2 syntax used in the implementation 10

Table 5.1: The modes of concept in WSMO choreography 30

Table 5.2: Items added to WM at each choreography round 51

Table 5.3: Simulating a move of the choreography engine with an ASM 61

Table 5.4: Simulating a move of the ASM with a choreography engine 66

Table 5.5: Comparison of our approach with IRS-III and WSMX 67

Table 6.1: Changes in the choreography engine code 70

Table 6.2: Choreography engine response time 74

Table 6.3: Choreography engine response time 77

Table 6.4: Choreography engine response time 81

LIST OF FIGURES

Figure 5.1: Architectural view of the choreography engine 39

Figure 5.2: The UML sequence diagram for online ticket reservation 47

Figure 6.1: Trend of time w.r.t. the number of rules 75

Figure 6.2: Trend of time w.r.t. the number of rule firings 75

Figure 6.3: Trend of time w.r.t. the number of rules (re-firing is prevented) 78

Figure 6.4: Trend of time w.r.t. the number of terms in rule LHS and RHS 81

Figure 7.9: A snapshot of out.txt generated by the choreography engine 156

LIST OF ABBREVIATIONS

ΔWM	Delta-Working Memory
API	Application Programming Interface
ASM	Abstract State Machine
B2B	Business-to-Business
BPEL	Business Process Execution Language
BPEL4Chor	Business Process Execution Language for Choreography
BPEL^{gold}	Business Process Execution Language Global Definition
BPM	Business Process Modeling
CTR	Concurrent Transaction Logic
EBNF	Extended Backus-Naur Form
F-Logic	Frame Logic
Flora	F-Logic Translator
HiLog	Higher-order Logic
IRS	Internet Reasoning Service
JSON	Javascript Object Notation
MAP	Multi-Agent Protocl
OCML	Operational Conceptual Modeling Language
OWL-S	Web Ontology Language for Services
REST	Representational State Transfer
SOA	Service-Oriented Architecture
WM	Working Memory

WS-CDL	Web Service Choreography Description Language
WSCI	Web Service Choreography Interface
WSDL	Web Service Description Language
WSML	Web Service Modeling Language
WSMO	Web Service Modeling Ontology
WSMO4J	Web Service Modeling Ontology for Java
WSMX	Web Service Modeling eXecution Environment

Chapter 1

INTRODUCTION

In Service-Oriented Architecture (SOA), Web Services are defined, registered, invoked, and interconnected via some pre-agreed specifications [4]. As a result of emerging SOA applications in industry and B2B communication sectors, two major challenges have gained more attention. The first one is about how to find proper Web Services for a specific business; and the second one is about how to make the found Web Services cooperate together to make the specific business goal.

The first challenge is addressed by the concept known as *Service Discovery*; that is the process of finding one or more appropriate Web Service(s) among a possibly large pool of diverse Web Services. *Service Matching* is the main part of Service Discovery process. The non-intelligent way is to manually refer to Web Service repositories and use traditional text retrieval techniques to find some candidates for the specified application. On the other hand it can be done (semi) automatically by applying certain AI techniques such as logical inference. The latter approach requires the availability of rich semantic description of Web Service capabilities, user requirements and other related aspects of Web Services (such as non-functional properties), commonly in the form of logical statements in some appropriate form of logic, and in this case discovery amounts to proving certain logical inferences.

The second challenge is addressed by the concept known as *Service Composition*; that is the process of combining Web Services in a way that they use their outcomes in

a specific order which provides the specified goal. *Choreography* and *orchestration* are two complementary aspects of service composition. Choreography models the behavioral interface of a Web Service, and is used to realize (semi) automatic communication between the Web Service and its user. W3C defines choreography in this way: "Web Services Choreography concerns the interactions of services with their users. Any user of a Web service, automated or otherwise, is a client of that service. These users may, in turn, be other Web Services, applications or human beings. Transactions among Web Services and their clients must clearly be well defined at the time of their execution, and may consist of multiple separate interactions whose composition constitutes a complete transaction. This composition, its message protocols, interfaces, sequencing, and associated logic, is considered to be a choreography"[9]. Orchestration models the way of coordination among two or more Web Services in order to achieve a common goal [4][77]. W3C defines orchestration in this way: "Web service invokes other Web services in order to realize some useful function. I.e., an orchestration is the pattern of interactions that a Web service agent must follow in order to achieve its goal"[9].

An example scenario for describing choreography is the reservation process of an airline ticket. The actual scenario between a human and a web site providing flight reservation service can be as follows:

1. At the beginning, the user is usually able to set six items:
    - departure city/airport
    - arrival city/airport
    - whether the trip is roundtrip or one-way
    - departure date
    - return date

- the number of passengers

2. The website offers some candidate flight numbers and their details, including date, time, airport, and price.
3. The user has to choose one of the candidates.
4. The online ticket reservation site asks for passenger data, such as the full name, gender, and date of birth.
5. The system asks for credit card information, including the holder's name, credit-card number and its CVV code
6. The online ticket service queries the bank to validate the card.
7. Upon approval of the bank, the flight reservation service completes the transaction and issues a ticket to the user.

Similar to *Service Matching*, to automate Web Service choreography, the need for unambiguous, machine processable semantics is obvious.

A well-known semantic Web Service framework is the Web Service Modeling Ontology (WSMO) [26] – a meta-ontology for describing relevant aspects of Semantic Web Services that facilitates its logical description. In WSMO, Web Services and service requests are described using a rich semantic notation so that the meaning of either becomes understandable, both to man and machine. User requests are framed in the form of goals, common vocabulary is defined in the form of ontologies, and bridging the heterogeneities among Web Services and goals are resolved through mediation. Several languages of varying expressive power, such as WSML-Rule, WSML-Flight, and WSML-Full [10][55] have been proposed to specify Web Services according to WSMO, all based on Frame Logic (F-Logic) [60][58]. Both *Service Matching* and *Service Composition* are addressed by WSMO with its *Capability* and *Interface* com-

ponents respectively.

In spite of its comprehensive definitions and elegant service composition idea, WSMO has not been embraced by industry and no publicly available, fully working product has been made based on it. We found that the first barrier is the language used to describe WSMO-based Web Services, i.e. WSML. It is too verbose to be applied in practice. The second barrier is about its choreography component. Although WSMO choreography is built upon the well-founded theory of Abstract State Machines (ASM) [54][49][25], our investigations into the current, ASM-based algorithm advocated for being used in WSMO choreography [12][83][84] have revealed certain shortcomings of the algorithm, which make it unsuitable for matching the choreography requirements of the client and Web Service, or driving the interaction between the client and Web Service. Specifically, the current algorithm makes no connection between the pre and post conditions of the requester, and allows the execution of ASM to continue, even when the requester may be satisfied by the current state. Furthermore, the algorithm uses the paradigm of evolving ontologies where objects and relations on objects are evolving, whereas in the original ASM formalism, it was functions and relations that were evolving, and the mapping between the two formalisms has never been given. In this work, our goal has been to change this state of affairs by providing solutions to the mentioned problems.

Firstly, we use Flora-2[1] as a specification language for semantic description of Web Service components according to WSMO and implement a matching engine based on inference in F-logic in order to discover Web Services that can satisfy user requests specified in the form of goals. Our semantic specification is very concise since it makes

[1]F-logic Translator version 2 or Flora-2 by Michael Kifer et al. [56][20] is a powerful language for knowledge representation and reasoning. It is based on F-logic, HiLog [29][28], and Transactional logic [24].

use of the underlying Flora-2 syntax to the highest degree possible. The implementation of the matcher is also very compact since it makes effective use of the meta-level capabilities of the Flora-2 system.

Secondly, we use Flora-2 for adapting the definition and execution of abstract state machines to semantic Web Service choreography. We define a precise mapping from traditional ASMs to the version used in specifying choreographies. We modify the proposed ASM execution algorithm so that it takes into account the pre-post conditions of the requester, making it stop when the requester can be satisfied. We use a sub-language of F-logic as implemented in Flora-2 to semantically describe Web Service and goal (requester) choreographies, as well as functionalities, and implement a choreography engine in Flora-2 that realizes our algorithm. We thus not only propose a formalism for ASM-based choreography, but also demonstrate its viability though an actual implementation.

Outcomes of the above mentioned research have been published in the form of two articles. The service matching solution has been published in Lecture Notes in Electrical Engineering, by Springer [22] and the choreography solution has been published in the Scientific Programming journal, by Hindawi [71].

The rest of this dissertation is structured as follows. In Chapter 2, we briefly explain the preliminaries, including short introductions to ASM theory, F-logic, Flora-2, and WSMO capability and choreography (as part of WSMO interface component) concepts, which are needed to understand the subsequent sections. Chapter 3 contains related work on choreography specification, matching and execution, with appropriate comparison with our approach along with the problem definition. In Chapter 4, we describe the semantic specification of goals and Web Services in WSMO, as well

as the logic we have used in the implementation of our service matching engine. We demonstrate the power and practicality of our service matching scheme through a simple but realistic appointment system scenario and show how our solution fulfills all the requirements needed to specify and discover Web Services/goals. Chapter 5 is dedicated to our choreography scheme and its implementation details. We give the existing WSMO choreography execution algorithm, point out its weaknesses, and present a rectified version. We prove the equivalence of ASMs (also known as *evolving algebras* [47]) and WSMO choreography specifications (commonly called *evolving ontologies* [44]) by providing appropriate mappings between them. Moreover, we see how our choice of Flora-2 as the specification language helps resolve the data granularity mismatch problem [13][70] that can occur between the goal and Web Service. The general form of goal and Web Service specifications in F-logic is given, as well as the implementation details of the improved algorithm, which works on the specifications. We provide three choreography scenario examples, demonstrating the capabilities of the proposed choreography specification and engine. We also prove the equivalence of ASM and choreography engine that was previously missing in the literature, and at the end of this chapter we highlight the major differences between our semantic choreography solution and the other available ones. In Chapter 6, we do benchmarking our choreography solution, and demonstrate that it is scalable; and finally in Chapter 7 we present the conclusion and future work in the area of semantic Web Service discovery and choreography based on our semantic Web Service specification approach.

In Appendix A, we illustrate the developed visual editor for semantic Web Service specification as a subset of Flora-2. In Appendix B, we give the grammar of the Web Service and goal choreography specifications in EBNF [1] notation. In Appendix

C we describe a scheme for converting JSON [15] content into its Flora-2 equivalent, a step that will be useful in grounding Flora-2 specified semantic Web Services into RESTful Web Services [79]. Appendix D contains a description of the predicates used in the implementation, which itself is available for download at [19]. In Appendix E the source codes of the choreography engine written in Flora-2 are given. More choreography specification examples are provided in Appendix F. Lastly, in Appendix G, the source codes of the choreography specification generator (written in C#), which is used in the benchmarking process, is given.

Chapter 2

PRELIMINARIES

In this chapter we briefly review the main concepts which are necessary to know to understand the consequent chapters.

2.1 WSMO and WSMO choreography

Web Service Modeling Ontology (WSMO) is a comprehensive framework that has the aim of enabling automatic service discovery, invocation, and composition [26]. It identifies four major concepts in semantic service oriented architecture: *ontology* (provides the common terminology between goals and web services), *web service* (models the functionality of the web service at a high, semantic level), *goal* (models the request of client at a high, semantic level), and *mediator* (resolves different types of possible incompatibilities, including process and terminological, between goals and web services).

The concept of Web Service in turn contains *nonfunctional properties*, *ontologies*, *mediators*, *capability*, and *interface* elements. The functionality of a WSMO Web Service is described by the *capability* element which contains *precondition*, *post-condition*, *assumption*, and *effect*. *Precondition* specifies what the web service requires from the goal before it can start execution. *Post-condition* specifies what the web service can provide to the client (i.e. goal) upon successful completion of its execution. Assumption is the state of the world which must hold true before the web service can be called. *Effect* is the change(s) caused to the state of the world through the execution

of the web service. A WSMO Web Service guarantees its post-condition and effect if its pre-condition and assumption are true. This feature of WSMO Web Services is used for automatic discovery purposes.

Choreography is part of Web Service *Interface* element which specifies the behavioral interaction of the web service with its client.

2.2 Flora-2 and F-Logic

Introduced by M. Kifer and G. Lausen, Frame Logic [60] or in short F-logic is a formalism that integrates first-order logic and the object-oriented paradigm. Equipped with predicate calculus [3], F-logic can easily model concepts, facts, rules, and specially ontologies in a very declarative fashion and is considered a rule-integrated ontological language [58]. In F-logic, classes and subclasses are modeled as concepts and sub-concepts respectively. Also, data members are represented by attributes and their assigned values. Detailed discussion about F-logic can be found in [60][61][76].

Flora-2 is a powerful, integrated system based on F-logic, HiLog [29] [29], and Transaction Logic [24]. It offers syntax similar to F-logic and by using the XSB inferencing engine [14], it can do reasoning on the facts and knowledge represented in F-logic or HiLog. The variety and multiplicity of logic and predicate operators makes Flora-2 a powerful reasoning system. Moreover, Flora-2 is being continuously extended and developed [56][17] and it can be integrated with Java through the provided APIs [18]. We use the latest version of Flora-2 [20] to implement a new Semantic Web Service choreography engine.

Flora-2 keeps knowledge in logical storage places named *modules*. By default, all the information and knowledge is stored in the *main* module. The user can create an arbitrary number of independent modules for organizing knowledge (referred to as

user-defined modules).

Flora-2 also has *built-in* modules which contain predefined predicates, such as the `\prolog` module where useful Prolog [75] utility predicates are kept.

In Table 2.1, some of the more prominent constructs of Flora-2 syntax, which we use in the choreography engine implementation, are given.

Table 2.1: Summary of Flora-2 syntax used in the implementation

<table>
<tr><th>Flora-2 Syntax</th><th>Meaning</th></tr>
<tr><td>concept[|attribute => type|].</td><td>Defining a concept, its attributes, and their types</td></tr>
<tr><td>object[attribute -> value].</td><td>Specifying an instance object (frame) attribute value</td></tr>
<tr><td>subconcept::concept.</td><td>Defining inheritance between two concepts</td></tr>
<tr><td>object:concept.</td><td>Instance declaration</td></tr>
<tr><td>${. . . }</td><td>Reifying any kind if object on Flora-2</td></tr>
<tr><td>~</td><td>Meta-unification operator</td></tr>
<tr><td>\object</td><td>Base-type in Flora-2</td></tr>
<tr><td>\if. . . \then. . . \else</td><td>If-then-else formula</td></tr>
<tr><td>Predicate(parameters)</td><td>Defining predicate in Flora-2</td></tr>
<tr><td>%Predicate(parameters)</td><td>Forces Flora-2 to apply non-tabled predicate.</td></tr>
<tr><td>?variable</td><td>Unifiable variable</td></tr>
<tr><td>?_</td><td>Don’t care unifiable variable</td></tr>
<tr><td>?_name</td><td>Don’t care identifiable unifiable variable</td></tr>
<tr><td>,(\and) ;(\or) \+</td><td>Logical AND, Logical OR, Logical NOT</td></tr>
<tr><td>L :- R</td><td>Logical implication operator ($R \rightarrow L$)</td></tr>
<tr><td>!</td><td>Prolog cut operator</td></tr>
<tr><td>@\Prolog</td><td>Denotes using Prolog predefined functions.</td></tr>
<tr><td>@\btp</td><td>Denotes using embedded base-types or predicates.</td></tr>
<tr><td>setof{?X | any formula containing variable ?X}</td><td>Generating list of all X’s where the formula is verifiable by them.</td></tr>
<tr><td>//comment /*comment*/</td><td>Commenting</td></tr>
</table>

2.3 Abstract State Machine (ASM)

ASMs or Evolving Algebras were first introduced by Y. Gurevich [47][48]. ASM theory says that every algorithm can be modeled as a step-by-step evolving system containing two main components: a state signature which represents the current status of the system and a finite set of transition rules which determine the next state of the

system based on the current one. Finite State Machines (FSMs) [53] can be seen as a specific instance of ASMs. An important point about ASMs is that transition rules are applied in parallel at each evolution step and they are categorized into three types: *if-(else)-rule*, *forall-rule*, and *choose-rule* [25].

ASMs are generally categorized into Basic-ASMs and Multi-agent ASMs. The discussion here is based on Basic-ASMs, similarly to WSMO which configured and extended Basic-ASMs to model choreographies. In WSMO, evolving ontologies (ontologized ASM [93]) are used to represent the state of the choreography, instead of the evolving algebra of ASMs. It turns out that evolving ontologies are equivalent to evolving algebras, and we prove this in Section 5.9. For further reading about ASMs the reader is referred to [54][49][25]. More detailed and formal explanation about ASM is also given in Section 5.9.

2.4 Summary

In this chapter we briefly explained the fundamental concepts which are used in the subsequent sections. These concepts include ASM theory, F-logic, Flora-2, and WSMO capability and choreography.

Chapter 3

RELATED WORK AND PROBLEM DEFINITION

In this chapter, we review the related work for semantic service matching and semantic Web Service choreography. In general, we found that in service matching purposes, F-logic and Flora-2 are more adequate in compare with the previously proposed solutions. For semantic Web Service choreography, we found none of the existent frameworks adheres to important ASM features including parallel state changing, choose branching, and concept modes.

3.1 Semantic service matching

In [73], the authors propose a framework for Semantic Web Service discovery using FIPA multi-agents. They have a broker architecture and deal with OWL-S [68] rather than WSMO, as we do. In [86] the authors use WSML to specify goals and Web Services, which is very verbose. Furthermore, they do not state the proof commitments that are needed for a successful match in a logical way.

In [59], the authors use Flora-2 to present a logical framework for automated Web Service discovery. Moreover, they use WSMO specification as the conceptual description of Web Services as we do. However, their specification of Web Services and goals are very involved, and they resort to Transaction logic for proof commitments. We will see that our specifications, as apparent in the given realistic example in Section 4.2, are quite intuitive and simple. Furthermore, we make use of only Frame-logic for stating our proof commitments, which itself is equivalent to first-order predicate logic

[62], and is much more accessible to the reader. The simplicity of our approach can be an important facilitator in its adaptation by industry (after necessary enhancements to deal with the web environment).

There are several surveys and reviews about semantical as well as non-semantical Web Service discovery proposals, which give a general overview of this field of study [74][64][67]. Many of the surveyed proposals are based on OWL-S. As M. Kifer et al. stated in [60], such approaches rely on subsumption reasoning [86] and due to the lack of rules in OWL, they are not able to exactly guarantee goal post-conditions.

3.2 Semantic Web Service choreography

Semantic Web Service frameworks such WSMO and OWL-S [68] use rich semantic reasoning systems to realize semantic Web Service choreography. They model the interaction between the client and the service as a bi-directional conversation, with implementations such as WSMX [51] [94] [80], WSMO Studio (a visual editor for WSML) [35][36], WSMO4J API [34], IRS-III [38] [27], and OWL-S tools [85].

OWL-S is not well aligned with the WSMO framework. It "does not provide an explicit definition of choreography, but instead focuses on a process based description of how complex Web Services invoke atomic Web Services" [39]. In [66], WSMO and OWL-S are compared in detail and the author concludes that "WSMO presents some important advantages when compared to OWL-S". Here, we point out some general issues about OWL-S:

- OWL-S does not properly decouple the viewpoint of service requester and service provider.
- OWL-S service profile mixes the information of WSMO goal, WSMO capability, and non-functional properties.

- In OWL-S, the requester has to formulate its request based on the descriptions of profiles.
- OWL-S does not clearly define how logical expressions are used to describe conditions and results.
- In spite of its incompleteness, WSMO choreography provides ASM as its formal model, whereas a formal semantic OWL-S process model is still missing.

WSMX is known as the reference prototype implementation of WSMO [52][13][16]. WSMX offers a flexible architecture that can accept different components as its plug-ins. The project has been implemented in Java, can handle service and goal specifications that are written in WSML [10][31], and uses WSML2Reasoner [63][65], which converts WSML into the internal representation of external reasoning engines in order to do the reasoning tasks. KAON2 [72] is the external reasoner used to deal with choreography reasoning tasks [93].

We have thoroughly investigated WSMX using publicly available documents, including published papers and source code [52][13]. We have found that:

- the implementation of choreography in WSMX was started but not completed.
- the implementation does not support parallelism and consequently inconsistency checking is not even an issue.
- the implementation does not support intentional non-deterministic behavior necessitated by the *Choose* rule type.
- in the case of *if-then* rules, if more than one left-hand side (antecedent) are satisfied by the current ontology state, right-hand sides of all matching rules are executed sequentially, without any consistency check of the actions performed, resulting in behavior that depends on the order of the rules.

Furthermore, in the last version of WSML2Reasoner, which is used by WSMX to translate WSML logical expressions to the native language of the used reasoner, there is no translation of the *Forall* or *Choose* rule types, confirming our findings. It is clear that several of the most fundamental features of ASMs remain unimplemented in WSMX.

IRS-III (The Internet Reasoning Service: 3rd version) [38] [27] [37] provides an infrastructure that utilizes the WSMO framework. The IRS system is composed of three major components: *server*, *client*, and *publisher*. Choreography between a client and a Web Service is not done directly, but through the IRS choreography engine, which acts as a broker between the available Web Services and user requests. IRS takes the responsibility of service discovery, mediation, communication and invocation of the Web Services and provides the result for the goal; however, clients should formulate their needs to IRS in the specific representation language of IRS [27]. IRS uses the OCML ontology representation language and its server has been implemented in Lisp [69][88].

IRS does not adhere to either original ASM, or WSMO choreography, because:

- transition rules of IRS are not run in parallel. In the case that more than one transition rule applies to the current state of the choreography, only one is selected using an internal function for which no further details are available.
- actions in the rules are tightly coupled with the actual messages sent to the Web Service, which makes the choreography specification inflexible; the actual call sequence of operations are pre-determined for different kinds of requests.
- goals are modeled by pre and post conditions only and do not contain a choreography component at all. The interaction is between the IRS, acting on behalf

of the goal, and the Web Service, using solely the choreography specification of the Web Service.

- the concept of modes is completely absent; flexible interaction between the requester and service provider that is made possible by having modes is replaced by a rigid communication model where the *actor* which has the *initiative* can update data.

In comparison with our approach, IRS does not support parallel firing of transition rules and does not check for consistency of the updates. Consequently, the next state of the ontology is not unique, and depends on the choice of the rule to be fired, leading to nondeterministic behavior. Whereas we make full use of modes and enforce their compliance, as already mentioned, IRS completely ignores them. Most importantly, it ignores the obvious connection between the initial choreography execution state and the precondition of the goal, as well as the final state of the choreography execution and the goal post-condition, relying instead on the built-in predicate *init-choreography* to start the chain of rule firing, and the action *end-choreography* to terminate the choreography run. If the choreography is not designed carefully, the situation where the choreography run terminates without the goal post-condition being satisfied could arise.

There are other notable works on the analysis, formalization, and modeling of choreographies. D. Roman et. al. in [82] argue that choreographies specified in the original ASM model become quite involved when they contain contracting and enactment (additional policies and constraints imposed by Web Service and goal). In [82], they extend the current model of WSMO with Concurrent Transaction Logic (CTR) [81] [57] to simplify the representation; however, the CTR implementation is still in

its prototype stage [2]. M. Stollberg in [89] discusses the main reasoning and mediation activities required for choreography and orchestration, both in general and in the context of WSMO. SWORD authors use rule-based expert systems to "determine whether a desired composite service can be realized using existing services" [78]. This approach is similar to ours in that it uses forward-chaining reasoning to develop knowledge in a stepwise manner, but for the purpose of Web Service composition, and not choreography.

3.3 Problem definition

In spite of its comprehensive definitions and elegant service composition idea, WSMO has not been embraced by industry and no publicly available, fully working product has been made based on it. Although WSMO choreography is built upon the well-founded theory of Abstract State Machines (ASM) [54][49][25], our investigations into the current, ASM-based algorithm advocated for being used in WSMO choreography [12][83][84] have revealed certain shortcomings of the algorithm, which make it unsuitable for matching the choreography requirements of the client and Web Service, or driving the interaction between the client and Web Service. Specifically, the current algorithm makes no connection between the pre and post conditions of the requester, and allows the execution of ASM to continue, even when the requester may be satisfied by the current state. Furthermore, the algorithm uses the paradigm of evolving ontologies where objects and relations on objects are evolving, whereas in the original ASM formalism, it was functions and relations that were evolving, and the mapping between the two formalisms has never been given. In this work, our goal has been to change this state of affairs by providing solutions to the mentioned problems: we rectify the original WSMO choreography algorithm. We introduce a new language

for writing choreography specification adhering to WSMO choreography model. We implement a choreography engine which supports all the types of transition rules defined by ASM. By variety of examples, we show that the implemented choreography engine can run the choreography specifications (written in the new language) correctly and completely. Moreover, in Chapter 6, we experimentally show that the time needed for the choreography engine to complete choreography scenarios is linearly changed with regard to the size of the choreography specifications.

3.4 Summary

In this chapter, we briefly reviewed the existing semantic Web Service choreography solutions. We explained in short why none of them adhered to the available theoretical basis provided by WSMO and in general semantic Web Service choreography definition.

Chapter 4

F-LOGIC BASED SERVICE MATCHING OF SEMANTIC WEB SERVICES

In this chapter, we use Flora-2 as a specification language for semantic description of Web Service components according to WSMO and implement a matching engine based on inference in F-logic in order to discover Web Services that can satisfy user requests specified in the form of goals. Our semantic specification is very concise since it makes use of the underlying Flora-2 syntax to the highest degree possible. The implementation of the matcher is also very compact since it makes effective use of the meta-level capabilities of the Flora-2 system.

The rest of this chapter is structured as follows. In Section 4.1, we describe the semantic specification of goals and Web Services in WSMO, as well as the logic we have used in the implementation of our service matching engine. In Section 4.2, we demonstrate the power and practicality of our scheme through a simple but realistic appointment system scenario and show how our solution fulfills all the requirements needed to specify and discover Web Services/goals.

4.1 Semantic Web Service specifications and the matching process

The functionality of a WSMO Web Service is defined under the *capability* tag (element) which contains four axioms: *precondition*, *assumption*, *post-condition* and *effect* [12]. Pre and post conditions represent the internal state of the Web Service, whereas assumption and effect represent the state of the outside world (environment).

A WSMO Web Service guarantees its post-condition and effect if its pre-condition and assumption are true. This feature of WSMO Web Services is used for discovery and selection purposes. For the sake of simplicity, we only consider Web Service pre-conditions (shown by *web.pre*) and Web Service post-conditions (shown by *web.post*), since assumptions and effects can be handled in a similar way.

Logically, the functionality of a Web Service can be shown by the following formula (the arrow represents the *implication* operator).

$$\forall x : web.pre(x) \Rightarrow web.post(x) \tag{4.1}$$

This formula means that for all instantiations of the free variables in the formula (represented by x), if the pre-condition is true, the Web Service guaranties that the post-condition will also be true after the Web Service has finished its execution.

The definition of a logical match between a goal and a Web Service can be described precisely with the formula below:

$$\begin{aligned} &\forall x_\mathrm{i} \forall y_\mathrm{i} : (goal.pre(x_\mathrm{i}) \Rightarrow web.pre(y_\mathrm{i})) \\ &\quad \wedge \\ &(goal.pre(x_\mathrm{i}) \wedge (web.pre(y_\mathrm{i}) \Rightarrow web.post(y_\mathrm{i})) \Rightarrow goal.post(x_\mathrm{i})) \end{aligned} \tag{4.2}$$

This formula should be shown to be a valid statement in F-logic before we can say that the Web Service completely satisfies the functional requirements of the goal. In this formula, x_i represents the free variables in the goal and y_j represents the free variables in the Web Service.

Informally, the formula above checks that the goal pre-condition logically implies the pre-condition of the Web Service (hence guarantying that the Web Service has all it needs before it gets executed) and that the goal pre-condition, together with the implicit statement of the Web Service functionality (that the Web Service pre-condition implies the Web Service post-condition) logically implies the goal post-condition, thus

guarantying that the goal will get the desired result with the execution of the Web Service.

In our implementation of Web Service specifications and the matcher, we diverge slightly from the logical definition given above in order to take advantage of the meta-logical capabilities of Flora-2 i.e. insertion of new facts which is a meta-logic operation. Specifically, the post-condition of Web Services can contain the *insert* predicate of Flora-2, so that the post-condition, instead of just being stated as being true, is made to be true by insertion of facts into the knowledge base. Then the post-condition of the goal can be tested against the new knowledge base.

Listing 4.1 depicts how our matching logic is implemented in Flora-2. Predicate `%match` in line 1 takes two variables, `?goal` representing a goal object and `?WS` representing a Web Service object as its parameters. In line 2, a new variable, `?module` is defined and assigned to the Web Service tag. In line 3, a new Flora-2 module with the same name as the Web Service tag is created and the description of the goal object is loaded into it. In line 4, the pre-condition of the goal (*goal.pre*) is inserted into the created module. Then in line 5, by calling `%applyWebService(?WS)` the Flora-2 reasoner attempts to prove the Web Service functionality specified in the form of an *if-then-else* statement in the knowledge base module. If `%applyWebService(?WS)` is proven, this means that the pre-condition of the Web Service is logically implied by the pre-condition of the goal, and moreover the actions specified in the post-condition of the Web Service have been carried out. In line 6, the variable `?gPost` is assigned to the goal post-condition, and in line 7 its validity is checked against the current knowledge base. If the check succeeds, this means that the goal post-condition is logically implied by the Web Service post-condition. It should be clear that the `%match` predicate indeed

verifies the validity of the formula (4.2) that we defined as the meaning of a successful match between a goal and a Web Service.

Listing 4.1: The `%match` and `%matcher` predicates

```
1  %match(?goal,?WS) :-
2      ?module=?WS.tag,
3      %loadGoal(?goal,?module),
4      %insertGoalPre(?goal,?module),
5      %applyWebService(?WS),
6      ?goal[post -> ?gPost]@?module,
7      ?gPost.
8
9  %applyWebService(?ws) :-
10     ?X = ?ws.def, ?X.
11
12 %matcher(?goal,?WS) :-
13     \if %match(?goal,?WS)
14     \then writeln(['Goal',?goal,'matches',?WS,'.'])@\prolog
15     \else writeln(['Goal',?goal,'does not match',?WS,'.'])@\
           prolog.
```

4.2 Use case: medical appointment finder

In this section we show how our approach can be used to describe a scenario of automatic medical appointment simply. The use case scenario is as follow: A patient named *Philip* wants to make an appointment with a specialist doctor (*ophthalmologist*) in *Montpellier* hospital located in a city of *France*. His preferred dates for this appointment are the days either before 19th or after 23rd (excluded) of the month. The patient should provide some basic information about himself, as well as a description of what he desires. Listing 4.2 shows a sample goal for this scenario rewritten in our specification format.

The patient provides the specialty *ophthalmology*, his name *Philip*, the hospital name he wishes to get the appointment from (i.e. *Montpellier*), and his age (which is requested by the Web Service) in the form of an appointment request. What he wants

is an appointment date either before the 19th or after the 23rd and an available specialist doctor's name.

For the Web Service side, Web Service pre and post-conditions have been given in Listing 4.3. The Web Service uses a local database of Doctor instances containing information about doctors (i.e. *doctor1* and *doctor2*) and some general facts (i.e. *Montpellier* hospital is in *Paris*), and these are also depicted in Listing 4.3.

Listing 4.2: Goal specification for the appointment use case

```
1  o_G01:c_Goal.
2  o_G01[
3      pre -> {RequestAppointment[specialty->Opthalmology,
4              patientName -> Philip,
5              appointmentDate ->?Date,
6              hospitalName -> MontpellierHospital,
7              age -> 22],
8              livesIn(Philip,Paris)},
9
10     post -> ${Appointment[appointmentDate -> ?Date,
11             doctorName -> ?DN,
12             patientName -> Philip,
13             hospitalName -> MontpellierHospital],
14             ((?Date < 19); (?Date > 23))}
15 ].
```

The Web Service provides some placeholders for its inputs while checking them over some predefined criteria (like, the patient must be at least 19 years old). Moreover, it checks whether the patient lives in the same city as the location of candidate hospital. After successful unification of inputs, the Web Service inserts all the possible appointments into the specified module (in this case `@WS01`) which is the common knowledge base between the Web Service and the goal. In this example, just the doctors with the specialty of *ophthalmology* who are working in *Montpellier* hospital are inserted into this module.

By referring to Listing 4.1 again, we can see that all these actions take place through the call to `%applyWebService(?WS)` at line 5 in the `%match` predicate. At line 7, the Flora-2 reasoning engine attempts to prove the goal post-condition. At this point, the appointment date is checked to verify that it conforms to the constraints specified by the patient in the post-condition of the goal (i.e. either before the 19th or after the 23rd). This checking filters out those doctors who are not available during the requested dates.

Listing 4.3: Web Service specification for the appointment use case

```
 1  \doctor1[
 2          doctorName -> Robert,
 3          specialty -> Neurology,
 4          hospitalName -> MontpellierHospital,
 5          availableDate -> 22
 6  ]:Doctor.
 7
 8  doctor2[
 9          doctorName -> Green,
10          specialty -> Opthalmology,
11          hospitalName -> MontpellierHospital,
12          availableDate -> 10
13  ]:Doctor.
14
15  hospital(MontpellierHospital,Paris).
16
17  o_WS01:c_WebService[
18          tag -> WS01,
19          def ->
20              {\ if (RequestAppointment[specialty -> ?DS]@WS01,
21              RequestAppointment[patientName -> ?PN]@WS01,
22              RequestAppointment[appointmentDate->?Date]@WS01,
23              RequestAppointment[hospitalName -> ?HN]@WS01,
24              RequestAppointment[age -> ?X]@WS01,
25              (?X > 18),
26              livesIn(?PN,?city)@WS01,
27              hospital(?HN,?city),
28              ?doctor:Doctor[doctorName -> ?DN,
29              specialty -> ?DS,
30              hospitalName -> ?HN,
31              availableDate -> ?Date])
32          \then (
33              ?post = {Appointment[appointmentDate->?Date,
34              doctorName -> ?DN,
```

```
35              patientName -> ?PN,
36              hospitalName -> ?HN]@WS01},
37              %insert{?post}}].
```

If we changed the line 15 in Listing 4.3 to `hospital(MontpellierHospital, Berlin)`, the match would fail since the Web Service pre-condition would not be satisfied. Similarly, if we changed line 12 in listing 4.3 to `availableDate -> 21` , again the match would fail, but this time due to the fact that the goal's post-condition would not be satisfied.

4.3 Summary

In this chapter, we explained how Flora-2 can be used as a convenient and expressive way to model semantic Web Services matching conforming to WSMO. We showed that matching can be logically expressed and the algorithm needed to do reasoning over this logical expression can be easily implemented by utilizing on the underlying Flora-2 reasoning engine and its meta-level capabilities. Moreover, we proposed a sub-language of Flora-2 for defining the specifications of WSMO goals and Web Services capability components and demonstrated the matching process through a real life example.

Chapter 5

ASM-BASED CHOREOGRAPHY OF SEMANTIC WEB SERVICES WITH F-LOGIC

In the previous chapter, we used F-logic [60] for the WSMO *capability* specification of Web Services and goals. A capability involves *pre* and *post-conditions* of Web Services and goals. Capabilities are used in the *service discovery* stage. In the this chapter however, we focus on WSMO *interfaces*, which mainly include choreography specification, and are used in the *service interaction* stage. Therefore, this chapter is organized as follows.

At first, in Section 5.1, we explain the concept of semantic Web Service choreography and show how it defers from the definition known in another research community. In Section 5.2, the concepts related WSMO choreography including specifications, modes, states, and transitions rules are reviewed in short. In Section 5.3, we give the existing WSMO choreography execution algorithm, point out its weaknesses in Section 5.4, and present a rectified version in Section 5.5. In Section 5.6, details of the implemented choreography matcher engine are given including the used architecture, realization of ASM parallelism, support of concepts' modes and contradiction detection. In Section 5.7, we provide a number of realistic choreography specification examples, representing different challenging situations, which can be choreographed by the implemented engine successfully. More examples are available in Appendix F. In Section 5.8, we discuss how Flora-2 inherently resolves granularity mismatch

problem [13] [42] in semantic Web Service choreography. Lastly, in Section 5.9, we formally prove that Evolving Ontology, used in WSMO, can be mapped to Evolving Algebra and vice versa that was missing in the literature before.

5.1 Service composition and semantic Web Service choreography

The idea of using SOA to form an IT infrastructure for carrying out B2B interactions has gained a lot of attention in the last 15 years. In this context, service composition has been studied and analyzed in many researches. Two important and complementary aspects of service composition are *service orchestration* and *service choreography*.

Service orchestration is the process of coordinating two or more services for the purpose directing them toward the accomplishment of a specific task in a centralized way. Service choreography however (as pointed out in [52] [38] [11] as well) does not have a unique understanding among researchers. In the Business Process Modeling (BPM) [91] community, choreography is known as a general predefined collaboration scenario that should be agreed upon and adhered to by two or more Web Services in order to accomplish a business goal, without the presence of a central coordinator (unlike orchestration). The choreography engine checks whether the participants are passing proper messages at the right time and in the correct order specified by the choreography designers [77]. In this paradigm, choreography is considered as a global collaboration, rather than bi-directional interaction between a service requester and a service provider.

The global collaboration view forms the basis of modeling languages such as WSCI [7], WS-CDL [30], BPEL4Chor [32] [33], Let's Dance [92], Multi-Agent Protocol (MAP) [23] and BPEL^{gold} [41]. The dominant common features of this type of

choreography languages are:

- being process-driven: a service is modeled as a process composed of series of milestones.
- having no role for goals: there is no concept of a service requester.
- staticity: the overall sequence of events is specified at design time.
- being non-semantic: no ontology is used, which is the main feature of a semantic system [21].
- not having any inferencing capability

In contrast, the concept of choreography among *Semantic Web Service* developers is understood as the behavioral interface of a *single* Web Service when it is interacting with its client (so-called *goal*), which results in an automatic, flexible conversation (dialog) between the two. In other words, it is the implicit communication protocol between two (and only two) counterparts that should be dynamically carried out in order to realize a conversation. The role of choreography engine is to dynamically control the conversation and see whether it is successful or not. This concept has been named *choreography interface* in [90].

In the rest of this dissertation, we use the term *choreography* only in the semantic web sense. Our work also falls strictly in the semantic web view of choreography, and consequently is not directly comparable to choreography languages adapted by the BPM community.

To fully automate service choreography, there is the need for unambiguous, computer processable semantics that can be used for automated reasoning [90]. A well-known semantic Web Service framework is Web Service Modeling Ontology (WSMO) [26]. In WSMO, the specification and behavior of the service provider (Web Service)

and the service requester (goal) are described using a rich semantic notation. WSMO *choreography* is a component of WSMO *interface* that deals with choreographing of WSMO-based Web Services and goals.

Although WSMO based its choreography algorithm on the well-founded theory of *Abstract State Machines* (ASM) [47][48], the algorithm is less than perfectly suited for the job at hand. In fact, our literature search has failed to reveal any choreography engine that implements it exactly as specified. For example, current implementations (such as WSMX [52][13][50] and IRS-III [38][27]), do not fully adhere to the ASM theory. Our own investigation into the algorithm has revealed certain important shortcomings which make it unsuitable for driving the correct interaction between goal and Web Service choreographies, and helps explain its lack of proper adaptation in existing choreography engine implementations.

In the following sections, our main contributions can be summarized as (i) rectifying the original ASM-based choreography algorithm, (ii) proposing an F-logic specification of WSMO goal and Web Service choreographies as an effective alternative to the current specifications in WSML [31] and OCML [40], (iii) implementing the rectified choreography algorithm in Flora-2 [56][17] with novel technics that adhere to theory of ASMs (missing in other implementations), and (iv) validating the implemented Flora-2 engine through several realistic scenarios.

5.2 ASM-based choreography in WSMO

The state-based model of WSMO choreography is inspired by ASM theory. Choreography working group has chosen ASM because of the following features [84]:

- Minimality: ASMs are based on a small assortment of modeling primitives.
- Expressivity: ASMs can model arbitrary computations.

- Formality: ASMs provide a formal framework to express dynamics.

Moreover, steps of evolution in ASM match the step-wise nature of interaction between a Web Service and its users.

To apply ASM theory in practice, WSMO choreography authors modified Basic-ASM concepts in several aspects. The concept of signature in ASM has been replaced by the concept of WSMO ontology, which involves *concepts*, *attributes*, *relations* and *axioms*. The concept of dynamic functions in ASM has been replaced by dynamic changes of instances and their attribute values, effectively, replacing the concept of *Evolving Algebra* [47] by the concept of *Evolving Ontology* [44]. This replacement, however, has not been formally justified so far, so in Appendix E, we prove the equivalence of *evolving algebras* and *evolving ontologies*, filling this gap.

5.2.1 Specification of choreographies in WSMO

In WSMO, the choreography concept has four components: *nonFunctionalProperties*, *stateSignature*, *state*, and *transitionRules*. *nonFunctionalProperties* refers the non-functional properties of the choreography, such as its author, date of creation and other meta information about the choreography, described in detail in [43].

5.2.1.1 The modes of concepts

Modes are used to define precise access rights on instances of concepts to be exercised by the environment (the client in this context) and the machine (the Web Service in this context).

Table 5.1: The modes of concept in WSMO choreography

If the concept is	static	controlled	in	shared	out
Web service can	Read	Read/Write	Read	Read/Write	Read/Write
Goal can	Read	-	Read/Write	Read/Write	Read

Five modes are defined in WSMO choreography, namely *static*, *controlled*, *in*,

shared, and *out* which control reading and writing access of the machine and the environment. Table 5.1 summarizes the encapsulation effect of each mode [26].

5.2.1.2 State

State component of WSMO choreography represents the dynamic part of the ongoing choreography instance, and consists of actual objects, i.e. instances of concepts, as well as instances of relations. The state is changed through the insertion of new instances, deletion of instances, or the update of attribute values in concept instances.

5.2.1.3 Transition rules

In WSMO, transition rules are in the form of the following expressions:

(i) **If** guard **then do** rules

(ii) **Forall** variables **with** guard **do** rules

(iii) **Choose** variables **with** guard **do** rules

In (i), guard should be an arbitrary logic formula without free variables; if the guard is true, then the rules on its right-hand side are executed. In (ii), the list of variables after *Forall* should be free in the guard and the scope of these variables extend to the rules on the right-hand side. For every value of the variables such that the guard becomes true, the actions on the right-hand side are executed in parallel. In (iii), for only one instantiation of the free variables in the guard (chosen at random), the actions on the right-hand side are executed [84].

In accordance with the original ASM definition, all rules at the top level, as well as rules on the right hand side are meant to be executed in parallel. Note that in the case of a *Forall* rule, an extra level of parallelism is introduced through the different instantiations of the variables listed after the Forall keyword.

Rules on the right-hand side, also called *actions*, are categorized into three basic update functions as follows:

(1) Adding a fact

(2) Deleting a fact

(3) Updating a fact (changing the values of the attributes)

5.3 Choreography matching algorithm in WSMO

The original algorithm of WSMO choreography is about validation of a choreography interface run. As such, it provides only an indirect operational semantics of how a choreography engine should run. For the sake of completeness, it is given in Algorithm 5.1, exactly as it is in [84].

Algorithm 5.1: The original WSMO choreography algorithm

A choreography interface run ρ is defined as a sequence of states $(S_0,...,S_n)$.
Given a choreography interface $CI = (O, T, S)$ such that S is consistent with O, a choreography interface run $\rho = (S_0,...,S_n)$ is valid for CI iff:

- $S_0 = S$
- for $0 \leqslant i \leqslant n-1$
  - $S_i \neq S_{i+1}$
  - $U = \{\text{add}(a) \mid a \in S_{i+1} \setminus S_i\} \cup \{\text{delete}(a) \mid a \in S_i \setminus S_{i+1}\}$ is an update set associated with S_i, O and T
  - S_{i+1} is consistent with O, and
- the run is *terminated*.

CI: Choreography Interface O: Ontology
T: Set of Transition-rules S: Original State-signature

In the algorithm, an update set is *not consistent* if it contains an insertion and a deletion of the same data simultaneously; otherwise it is *consistent*. A run is *terminated* if either (a) there is an update set U associated with S_n, O and T such that U is not consistent with S_n w.r.t. O, or (b) there is an update set U associated with S_n, O, and T such that $S_n = U(S_n)$, where $U(S_n)$ denotes the state obtained after the actions specified in the update set U, are implemented.

5.4 Problems with the WSMO choreography algorithm

The algorithm presented in the previous section verifies whether the choreography run is valid without addressing the role of the client of the Web Service. Indeed, by definition, choreography should model the interactive behavior of a Web Service from the client's point of view, which is not truly addressed by the algorithm.

This algorithm has three major missing ingredients that make it an incomplete way of specifying client interaction with the Web Service.

1. Choreography specification of the goal is completely ignored.
2. It is not clear what the initial state (S_0) should be. In the context of choreography, it should be state of the world, together with what the client can provide as input through its precondition component.
3. Its termination condition happens either when there are no more valid actions to take, or when it reaches a stable state, i.e. no more changes are possible to the current state. A terminated run, however, does not allow one to draw any conclusions regarding the suitability of the Web Services for a given client, since it is possible that the client requirements are not satisfied by the state in which the run terminated (whatever the reason for termination is). On the other hand, suppose a run is infinite, but at the same time an intermediate state reached in the execution is satisfactory from the client's point of view, at which point the execution should actually stop. This highlights the fact that the present algorithm overlooks the obvious link between the capability required by the goal and Web Service choreography.

Inability of the present algorithm to determine whether the client and Web Service are compatible can be summarized with the observation that no formal relationship

is established between what the client provides and the initial state of the choreography, as well as the requirements of the client and the termination condition.

These deficiencies in the algorithm are rectified in the next section by (i) taking into account the client choreography specification in addition to the Web Service choreography specification, (ii) establishing the connection between the initial state of the world plus the input provided by the client and initial state, and (iii) defining what a *successful run* is by taking into account the requirements of the client.

5.5 Improved choreography execution algorithm

In this section a modified algorithm for choreography execution is presented that rectifies the deficiencies identified in the original algorithm given in Algorithm 5.1. The rectified algorithm takes into account the pre and post conditions of the goal, making it stop when the goal can be satisfied with the current state. The original implicit style has been kept in order to highlight the differences better.

Our algorithm (Algorithm 5.2) starts with an initial state consisting of the facts and instances implied by the goal pre-condition, facts, instances and axioms contributed by the local state of the Web Service, as well as the facts, instances and axioms contained in the common ontology. Significantly, we note that the concept of a *valid* choreography interface run is replaced by a *successful* choreography interface run. At each iteration, the update set is computed, and provided that it is consistent , the next state of the system is obtained through the application of the actions in the update set. The execution of the choreography engine terminates successfully at the earliest state which logically implies the goal post-condition. Any other termination signifies failure, and can happen if (i) the execution engine reaches a stable state (i.e. the state remains unchanged by the application of the transition rules) which does not logically

imply the goal post-condition, or (ii) the update set is not consistent at any stage, or (iii) a state is reached that is not consistent with the common ontology.

Algorithm 5.2: Rectified choreography matching algorithm

A choreography interface run ρ is defined as a sequence of states ($S_0,...,S_n$). Given a choreography interface $CI = (O_{common}, T_{webservice}, T_{goal}, S_{webservice}, S_{goal})$ such that both $S_{webservice}$ and S_{goal} is consistent with O_{common}, a choreography interface run ρ = ($S_0,...,S_n$) is successful for *CI* iff:

- $S_0 = S_{webservice} \cup S_{goal} \cup O_{common}$
- for $0 \leqslant i \leqslant n-1$
  - $S_i \neq S_{i+1}$
  - $U = \{\text{add}(a) \mid a \in S_{i+1} \backslash S_i\} \cup \{\text{delete}(a) \mid a \in S_i \backslash S_{i+1}\}$ is a consistent update set associated with S_i, O_{common} and $T_{webservice} \cup T_{goal}$
  - S_{i+1} is consistent with O_{common}
  - $S_n \models goal.post$
- For all $k < n$, $\neg(S_k \models goal.post)$

CI: Choreography Interface

O_{common}: Common Ontology, possibly containing concept definitions, instances and axioms

$T_{webservice}$: Web Service Choreography Transition rules

T_{goal}: Goal Choreography Transition-rules

$S_{webservice}$: Local state of the web service, possibly consisting of instances and axioms contributed to the common working memory of the choreography execution engine

S_{goal}: Instances implied by the goal pre-condition that are contributed to the common working memory of the choreography execution engine

---

The differences between the improved algorithm and the original algorithm stand out: the concept of a *valid* choreography interface run, which says nothing about the actual suitability of the Web Service to satisfy the goal demand, is abandoned in favor of a *successful* choreography interface run, which does give useful information regarding such suitability. The initial state of the system is linked directly to the input provided by the goal pre-condition, reflecting the actual state of affairs in the real world, and the final state is linked directly to the post-condition of the goal. Consequently, it becomes possible to not only show that a Web Service can satisfy the demands of the goal, but also that there is an actual interaction sequence between the client (i.e. goal) and the Web Service which results in such satisfaction. Furthermore, both goal and Web

Service choreography specifications participate in the choreography execution run.

Given the semantic specifications of a goal and Web Service, as well as imported ontologies, the job of the choreography engine is to determine if a *successful* run is possible.

5.6 Implementing the improved choreography algorithm in Flora-2

In this section we present the specification of semantic choreographies and implementation of the improved choreography algorithm in Flora-2. It has been tested on Flora-2 Reasoner 1.2, which is available since 2017-01-30 (rev: 1258b) [20], running on Microsoft Windows 7 (64 bit).

The terms below are used in explanations in the following sections:

- **Working Memory (WM)** is the main place for storing the state of the choreography. It keeps the whole knowledge produced by the choreography run in real-time. The knowledge can be shrunk, expanded, and altered.
- **Choreography round** is the sequence of actions: (i) starting with the current state of WM, (ii) determining which rules are applicable to this current state, (iii) determining the changes to WM that the application of these rules will cause, and (iv) in case there is no contradiction in the changes (to be explained later), actually implementing those changes in the current WM, leading to a new WM.
- **Delta Working Memory (ΔWM)** is a temporary storage place for actions to be carried out on WM at each choreography round.

5.6.1 Semantic specification of goal and Web Service in Flora-2

As in WSMO, our Flora-2 specifications for goal and Web Service are composed of the elements *ontology*, *capability* and *choreography*. *Ontology* contains frames and relations that represent knowledge used by the Web Service and goal. *Capability* ele-

ment encloses two sub elements, *pre* and *post*, which represent pre and postconditions. Precondition of the goal can contain conjunctions of non-negated frames and relations. Post-condition of the goal can be an F-logic expression (including all the logical connectives). Post-condition of the Web Service can contain conjunctions of non-negated frames and relations. Precondition of the Web Service can be an F-logic expression (including all the logical connectives). Note the similarity between the precondition of the goal and the post-condition of the Web Service, as well as the similarity between the post-condition of the goal and the precondition of the Web Service. *Choreography* element is modeled by a set of transition rules. Each rule is specified by `ruleId:ruleType -> ruleBody`. Goal's `ruleId` is in form of `gRule(OID)` and Web Service's `ruleId` is in form of `wsRule(OID)`, where `OID` is any Flora-2 object identifier and is used as a label for the rule. `ruleType` can be either `ForallRule` or `ChooseRule`. `ruleBody` is a reified Flora-2 implication shown by an *if*-*then*(-*else*) statement. The implication antecedent (we refer to it as left-hand side) can be an F-logic expression, and the implication consequent (we refer to it as right-hand side) contains a set of update functions (actions). Pseudocodes 5.1 and 5.2 depict the general form of a Web Service and a goal specifications respectively. Appendix B contains the EBNF grammar of goal and Web Service specifications.

Pseudocode 5.1: General form of Web Service choreography specification

```
1  wsName:Goal.
2  wslName[
3      importOntology -> address of local ontology,
4      capability -> ${
5          pre -> ${ Conjunction of frames and predicates },
6          post -> ${ F-logic expression }
7      },
8      wsRule(R01):ForallRule -> ${
9          \if ( F-logic expression )
10         \then ( Actions ) },
11     wsRule(R02):ChooseRule -> ${
```

```
12          \if ( F-logic expression )
13          \then ( Actions ) },
14      ...
15  ].
```

Listing 5.2: General form of Goal choreography specification

```
1   goalName:Goal.
2   goalName[
3       importOntology -> address of local ontology,
4       capability -> ${
5           pre -> ${ Conjunction of frames and predicates },
6           post -> ${ F-logic expression }
7       },
8       gRule(R01):ForallRule -> ${
9           \if ( F-logic expression )
10          \then ( Actions ) },
11      gRule(R02):ChooseRule -> ${
12          \if ( F-logic expression )
13          \then ( Actions ) },
14      ...
15  ].
```

5.6.2 Proposed architecture

In this section, we describe the architecture of our choreography engine.

5.6.2.1 Modules of the system

Our Flora-2 solution for implementing the choreography algorithm utilizes the *main* module and two Flora-2 user modules: *WM* for keeping the current state signature of the choreography, and *DeltaWM* for keeping the actions for modifying the current state into a new state. The *main* module contains the declaration of concepts, instances and their modes provided by Web Service and goal, as well as the code for the choreography engine. These three modules (i.e. *main*, *WM* and *DeltaWM*) are interconnected, as shown in Figure 5.1.

5.6.2.2 Delta Working Memory (ΔWM): Realizing ASM Parallelism

An auxiliary and transient user module named Delta Working Memory (ΔWM) is used to temporarily keep single choreography round updating actions that should

be applied to the main knowledge-base (WM). In each choreography round, all the updates are aggregated into ΔWM and then ΔWM is checked whether there exists any contradiction among the requested actions (explained below). If no contradiction is detected, then all the updates are carried out on the WM, evolving it into a new conflict-free state.

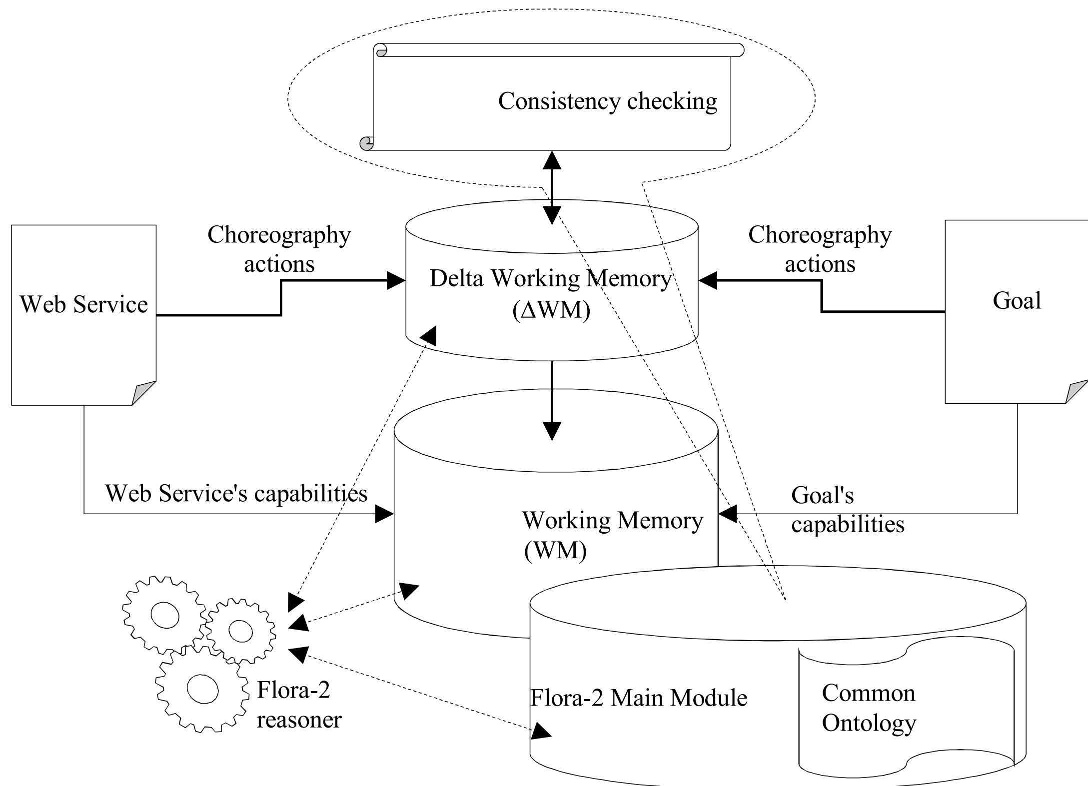

Figure 5.1: Architectural view of the choreography engine

5.6.2.3 Deterministic Choreography and Contradiction

As mentioned before, the transition rules must be applied in parallel. In the absence of any consistency checking, the rules can do contradictory actions which should be prevented.

The three kinds of contradiction that can occur in a choreography round are [52]:

1. Inserting and deleting/updating same knowledge simultaneously
2. Updating knowledge which does not exist

3. Deleting knowledge which does not exist

It is clear that such contradictory actions must be detected and the choreography execution must be stopped. In the case of non-existent knowledge, it makes no sense to remove or modify it. In the case of simultaneous insertion/deletion of the same knowledge, if the choreography run is continued, then it becomes nondeterministic in an unintended way: even though theoretically updates are done in parallel, in reality they have to be done in a serial fashion, and the order in which they are carried out leads to different WM states.

In each choreography round, in addition to testing for the above mentioned issues, checking for any violation on the modes of concepts (explained in section 5.2.1.1) are performed as well. If one of the participants wants to do an action on a concept which violates the concept's access mode, that action is prevented, leading to choreography execution failure.

5.6.3 Major predicates of the choreography engine

The main process is started through a call to `%start` predicate which is shown in Listing 5.1.

Listing 5.1: The top-level predicate of the choreography engine

```
 1  %start(?goal,?WS) :-
 2     %initializations,
 3     %preProcessCheckings(?goal,?WS),
 4
 5     %prepareModule(WM),
 6     %prepareModule(DeltaWM),
 7
 8     %importOntology(?goal,WM),
 9     %importOntology(?WS,WM),
10
11     %insertGoalPre(?goal,WM),
12     %runChoreography(?goal,?WS).
```

The choreography engine execution begins with a call to the `%initializations`

predicate on line 2, which currently resets the seed of the random number generator that is used to process the choose rule type. On line 3, goal and Web Service rules are checked for mode violations before the choreography rounds start. On lines 5 and 6, the modules *WM* and *DeltaWM* are created if they do not already exist. On lines 8 and 9, local ontologies of the goal and Web Service are loaded into WM. On line 11, the goal precondition is loaded into WM, becoming part of the initial state of the choreography run, followed by a call to the predicate `%runChoreogarphy` on line 12 which tries to satisfy the goal post-condition through repeated application of goal and Web Service transition rules.

`%runChoreography` implements the improved choreography algorithm given in Section 5.5, employing recursion instead of iteration. The definition of the `%runChoreography` predicate is given in Listing 5.2. It attempts first to prove the goal

Listing 5.2: `%runChoreography` predicate

```
1  %runChoreography(?goal, ?WS) :-
2     %proveGoalPost(?goal), !,
3     %watchln(['Success! '-?goal-'and'-?WS-'are'-'choreographed!'])
         .
4
5     %runChoreography(?goal, ?WS) :-
6     %eraseModule(DeltaWM),
7
8     %runWsRules(?WS),
9     %runGoalRules(?goal),
10
11    ( ( %contradictory(WM,DeltaWM), !,
12        %watchln('Choreography failed due to CONTRADICTION.') )
13        \or
14
15      ( \+ %deltaMakesAChange(WM,DeltaWM), !,
16        %watchln('Choreography failed due to NO CHANGE.') )
17        \or
18
19      ( %mergeDeltaIntoWM,
20        %runChoreography(?goal,?WS) )
21    ).
```

post-condition with the current state of WM in lines 1-3; the cut (!) operator on line 2 prevents backtracking and a success message is shown to the user on line 3. If the goal post-condition is not satisfied, then the second definition of %runChoreography is called: ΔWM is emptied on line 6, Web Service and goal rules are applied, populating ΔWM with pending actions to be performed on the WM (lines 8 and 9). On lines 11-13, ΔWM is checked for consistency (line 11) and whether pending actions result in a new state of WM (line 15). In the case of inconsistent changes or no change to WM, execution is stopped to prevent infinite recursion and a failure message is reported (lines 12 and 16). Otherwise, the actions in ΔWM are applied to WM to obtain an updated WM (line 19), and the process is repeated through a recursive call to %runChoreography (line 20).

5.6.4 Running (firing) the rules

One of the key features of ASMs is that rules should be fired in parallel. We realize this by placing all the insertion, deletion, and update actions on the right-hand sides of the rules that match the current WM into ΔWM, checking them for consistency, and then applying them to the previous WM to get an updated WM. The predicates %deltaInsert, %deltaDelete, and %deltaUpdate represent tentative changes to WM, not actual ones, until they are verified to not cause any conflicts.

Listing 5.3 contains the predicates for running goal and Web Service rules. The Flora-2 setof operator is used to iterate over all rules in the choreography specification (lines 2, 6, 11, and 15). In the case of **forall** rules, if the rule antecedent contains only ground facts, it models the ASM **if-then** transition rule type.

Listing 5.3: Running the goal and web service rules

```
1  %runWsRules(?WS) :-
2     ?_Temp = setof{ ?ruleID |
```

```
3     ?WS:WebService[wsRule(?ruleID):ForallRule -> ?ruleBody],
4     %invoke(WEBSERVICE,?ruleBody)},
5
6     ?_Temp2 = setof{ ?ruleID |
7     ?WS:WebService[wsRule(?ruleID):ChooseRule -> ?ruleBody],
8     %invokeChoose(WEBSERVICE,?ruleBody)}.
9  /*-----------------------------------------------------*/
10 %runGoalRules(?goal) :-
11    ?_Temp = setof{ ?ruleID |
12    ?goal:Goal[gRule(?ruleID):ForallRule -> ?ruleBody],
13    %invoke(GOAL,?ruleBody)},
14
15    ?_Temp2 = setof{ ?ruleID |
16    ?goal:Goal[gRule(?ruleID):ChooseRule -> ?ruleBody],
17    %invokeChoose(GOAL,?ruleBody)}.
```

If the antecedent contains free variables, it acts like the ASM **forall** transition rule type. In the case of a **choose** rule, the predicate `%invokeChoose` randomly selects exactly one ground instance of the free variables existing in the antecedent of the rule.

5.6.5 Contradictions in applying the rules

Checks for contradictory actions are implemented by the definition of the predicate `%contradictory` in Listing 5.4.

Listing 5.4: Contradictory cases

```
1  %contradictory(?WM, ?DeltaWM) :-
2    ins_action(?A1)@?DeltaWM, del_action(?A2)@?DeltaWM,
3    %contained(?X1,?A1),%contained(?X2,?A2),?X1 = ?X2,!.
4
5  %contradictory(?WM, ?DeltaWM) :-
6    del_action(?A)@?DeltaWM,
7    %convertReifiedObjectModule(?A, ?DeltaWM, ?WM, ?A_new),
8        \+ ?A_new@?WM.
9
10   %contradictory(?WM, ?DeltaWM) :-
11   update_action(?objOld,?objNew)@?DeltaWM,\+ ?objOld@?WM.
```

In lines 1-3, simultaneous insertion and deletion of the same item is detected.; in lines 5-8, deletion of a non-existent item is detected; finally in lines 10-11, update of a non-existent item is detected.

5.6.6 Access control to objects of different types

ASMs define access control modes for object manipulation. In the implementation, this access control has been enforced by checking whether a given manipulation is legal or not. This depends on the actor of the manipulation. For example, if an object has *in* mode, then only a goal can change its attributes. While invoking the rules belonging to the goal or Web Service, the legality of the access to the object is verified before the real action.

Listing 5.5 depicts a part of the implementation of access control for a goal. Before the rule is tested against the current WM, its concepts and user predicates on its left-hand side and right-hand side are extracted through the `%extractConcepts`, `%extractPredicates`, `%filterOutPredicates` predicates (lines 4-7), and access rights of the goal are verified for those concepts via `%checkAllFramesModes` and `%checkAllPredicatesModes` (lines 9, 14, 25, and 30). If a concept is on the right-hand side of a rule, the goal must have write access to it. On the other hand, if it is on the left-hand side, only read access is enough. `%checkAllFramesModes` (lines 35 to 39) checks modes for a list of extracted frames through calls to `%checkFrameMode` (defined on lines 41 to 45). In case of failure, error messages are generated on lines 47 to 51. The complete list of predicates and their explanations are presented in Appendix D, and the full source code of the implementation is available at [19].

Listing 5.5: Checking access mode

```
1  %check(?gOrWs,?X) :-
2     ?X ~ ${\if ?Y \then ?Z}, !,
3
4     %reformatToString(?Y, ?YStr),
5     %extractConcepts(?YStr, [], ?conceptsInY),
6     %extractPredicates(?YStr, [], ?termList1),
7     %filterOutPredicates(?termList1, [], ?predicatesInY),
8
9     \if (\+ %checkAllFramesModes(?gOrWs,READ,?conceptsInY))
```

```
10      \then
11          (writeln(['Error: Illegal access mode in '-?gOrWs])@\
               prolog,!,
12          \false),
13
14      \if (\+ %checkAllPredicatesModes(?gOrWs,READ,?predicatesInY))
15      \then
16          (writeln(['Error: Illegal access mode in '-?gOrWs])@\
               prolog,!,
17          \false),
18
19      %decomposeRHS(?Z, [], ?allFsOrPs),
20      %reformatToString(?allFsOrPs, ?allFsOrPsStr),
21      %extractConcepts(?allFsOrPsStr, [], ?conceptsInZ),
22      %extractPredicates(?allFsOrPsStr, [], ?temp),
23      %filterOutPredicates(?temp, [], ?predicatesInZ),
24
25      \if (\+ %checkAllFramesModes(?gOrWs,WRITE,?conceptsInZ))
26      \then
27          (writeln(['Error: Illegal access mode in '-?gOrWs])@\
               prolog,!,
28          \false),
29
30      \if (\+ %checkAllPredicatesModes(?gOrWs,WRITE,?predicatesInZ))
31      \then
32          (writeln(['Error: Illegal access in '-?gOrWs])@\prolog,!,
33          \false).
34  /*--------------------------------------------------------------*/
35  %checkAllFramesModes(?gOrWS, ?reOrWr,[]).
36
37  %checkAllFramesModes(?gOrWS, ?reOrWr,[?F|?R]):-
38      %checkFrameMode(?gOrWS, ?reOrWr,?F),
39      %checkAllFramesModes(?gOrWS, ?reOrWr,?R).
40  /*--------------------------------------------------------------*/
41  %checkFrameMode(GOAL,READ,?F):-
42      (?F:In \or ?F:Out \or ?F:Shared \or ?F:Static), !.
43
44  %checkFrameMode(GOAL,WRITE,?F):-
45      (?F:In \or ?F:Shared), !.
46
47  %checkFrameMode (GOAL,READ,?F):-
48      writeln(['Illegal GOAL READ action for',?F])@\prolog, !, \
           false.
49
50  %checkFrameMode(GOAL,WRITE,?F):-
51      writeln(['Illegal GOAL WRITE action for',?F])@\prolog, !, \
           false.
```

5.7 Semantic choreography specification examples

5.7.1 Flight ticket reservation

In this example, we give the choreography specification for interacting with an online flight reservation service. An autonomous software agent, acting on behalf of a human, is used to make the purchase. The behavior of the agent is described semantically in the form of a goal, with a choreography component. Similarly, the behavior of the online reservation service is described semantically as a Web Service, with its own choreography component. A person who wants to use the service can just provide the required information to the agent and leave the scene. The agent then interacts with the Web Service in accordance to its choreography, provided that it is compatible with the Web Service's choreography. Figure 5.2 depicts the UML sequence diagram [45] of this process. The essence of this scenario has been inspired by the online ticket reservation web site of an actual airline, similar to Virtual Travel Agency used in [8].

The actual scenario between a human and a web site providing flight reservation service is as follows. After opening the airline website, the user is able to set six items: departure city/airport, arrival city/airport, whether the trip is roundtrip or one-way (only roundtrips are considered in this case), departure date, return date, and the number of passengers. After the user submits this information, the reservation website offers some candidate flight numbers and their details, including date, time, airport, and price. The user has to choose one of the candidates. In the next step, the online ticket reservation site asks for passenger data, such as the full name, gender, and date of birth. After these items are provided by the user, the system asks for credit-card information, including the holder's name, credit-card number and its CVV code. After the user provides the card specifications, the online ticket service queries the bank to

validate the card. Depending on the outcome of the bank query, the flight reservation service completes the transaction and issues a ticket to the user.

Listings 5.6 and 5.7 are Flora-2 specifications of the user (goal), and the Web Services (reservation system) respectively. We simulate the conversation which should take place between the reservation service and the bank because it is a third party and not directly involved in the choreography.

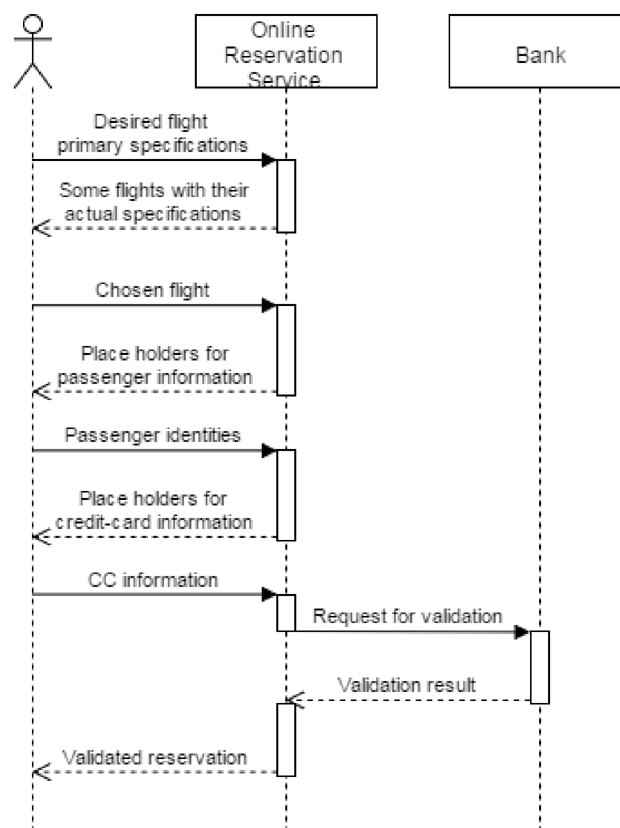

Figure 5.2: The UML sequence diagram for online ticket reservation

Listing 5.6: Goal choreography specification for online flight ticket reservation

```
1  /* Local ontology stored in a separate file
2  Name('Peter').
3  DateOfBirth(19830622).
4  Gender('Male').
5  CreditCardNo('1234432156788765').
6  CreditCardHolder('PETER JACKSON').
7  CreditCardCVV(123).*/
8  myGoal:Goal[
9     importOntology -> '../Flight/GoalsOntology.flr',
```

```
10
11    capability -> ${
12            pre -> ${ myRequest:RequestFlight[
13                      From->'Paris',
14                      To->'Chicago',
15                      Departure->23,
16                      Return->30]},
17
18            post -> ${ (?R:Reservation[?X->?Y])} },
19
20    gRule(R01):ChooseRule -> ${
21       \if ( tripChoice(?fl_dep,?fl_ret,?P),
22            (\+ trip:Trip) )@WM
23       \then (
24           %deltaInsert(${trip:Trip[
25                              Dep->?fl_dep,
26                              Ret->?fl_ret]}) ) },
27    gRule(R02):ForallRule -> ${
28       \if ( (?Q:QuestionByWS[
29                    Name->?X,
30                    DateOfBirth->?Y,
31                    Gender->?Z])@WM,
32                   (Name(?N), DateOfBirth(?DoB), Gender(?G))@WM )
33       \then (
34          %deltaInsert(${answer:AnswerByGoal[
35                             Name->?N,
36                             DateOfBirth->?DoB,
37                             Gender->?G]}) ) },
38    gRule(R03):ForallRule -> ${
39       \if ( (?Q:QuestionByWS[
40                    CreditCardNo->?X,
41                    CreditCardHolder->?Y,
42                    CreditCardCVV->?Z])@WM,
43                           ( CreditCardNo(?CCN),
44                             CreditCardHolder(?CCH),
45                             CreditCardCVV(?CCCVV) )@WM
46           )
47       \then (
48          %deltaInsert(${answer:AnswerByGoal[
49                             CreditCardNo->?CCN,
50                             CreditCardHolder->?CCH,
51                             CreditCardCVV->(?CCCVV)]}) ) }
52 ].
```

Listing 5.7: Web Services specification for online flight ticket reservation

```
1  /* Local ontology stored in a separate file
2  flight(F100,Paris,Chicago,23,250).
3  flight(F101,Paris,Chicago,23,350).
4  flight(F102,Paris,Chicago,25,400).
5  flight(F103,Chicago,Paris,29,150).
6  flight(F104,Chicago,Paris,30,200).
7  flight(F105,Chicago,Paris,30,150).
8  */
9  FlightReservationService:WebService[
10    importOntology -> '../Flight/WebServicesOntology.flr',
```

```
11    capability -> ${
12            pre -> ${ ?Req:RequestFlight[?X1->?Y1] },
13
14            post -> ${ (?Res:Reservation[?X2->?Y2]) } },
15    wsRule(R01):ForallRule -> ${
16       \if ( (?R:RequestFlight[
17                   From->?X,
18                   To->?Y,
19                   Departure->?Z,
20                   Return->?W])@WM,
21                  (flight(?fl_dep,?X,?Y,?Z,?priceDep))@WM,
22                  (flight(?fl_ret,?Y,?X,?W,?priceRet))@WM,
23                  (%sum(?priceDep,?priceRet,?priceTot)) )
24       \then (
25          %deltaInsert(${tripChoice(?fl_dep,?fl_ret,?priceTot)} ))
             },
26    wsRule(R02):ForallRule -> ${
27       \if ( ?T:Trip[
28                 Dep->?fl_dep,
29                 Ret->?fl_ret] )@WM
30       \then (
31          %deltaInsert(${question:QuestionByWS[
32                            Name->?X,
33                            DateOfBirth->?Y,
34                            Gender->?Z]}) ) },
35    wsRule(R03):ForallRule -> ${
36       \if ( ?A:AnswerByGoal[
37                 Name->?X,
38                 DateOfBirth->?Y,
39                 Gender->?Z] )@WM
40       \then (
41          %deltaInsert(${question:QuestionByWS[
42                            CreditCardNo->?XX,
43                            CreditCardHolder->?YY,
44                            CreditCardCVV->?ZZ]}) ) },
45    wsRule(R04):ForallRule -> ${
46       \if ( ?A:AnswerByGoal[
47                 CreditCardNo->?X,
48                 CreditCardHolder->?Y,
49                 CreditCardCVV->?Z] )@WM
50       \then (
51          %deltaInsert(${validation:CreditCardValidation[
52                            Number->?X,
53                            Holder->?Y,
54                            CVV->?Z]}) ) },
55    wsRule(R05):ForallRule -> ${
56       \if ( (BankYesNoAnswer('Yes'))@WM,
57             (trip:Trip[
58                 Dep->?fl_dep,
59                 Ret->?fl_ret])@WM )
60       \then (
61          %deltaInsert(${reservation:Reservation[
62                           Number->11100,
63                           Flight1->?fl_dep,
64                           Flight2->?fl_ret]}) ) },
65    wsRule(Bank_R01):ForallRule -> ${
66       \if ( (?R:CreditCardValidation[
67                   Number->?X, Holder->?Y, CVV->?Z])@WM,
```

```
68                    (DB_CreditCard(?X,?Y,?Z) )@WM
69             )
70        \then (
71           %deltaInsert(${BankYesNoAnswer('Yes')}) ) }].
```

In the pre-condition of the goal, the departure and arrival cities and days have been specified. The post-condition states that a reservation instance is demanded. On the flight Web Service side, the first rule of choreography `wsRule(R01)` at line 16 checks if there is a request for a flight consisting of all the necessary items, searches for a roundtrip on the specified days and if this search is successful, inserts a new triple into the knowledge-base containing two flight numbers and their total price. On the goal side, the rule `gRule(R01)`, which is of rule type **choose**, is responsible for checking the existence of any choice on the knowledge-base. As soon as some flight choices become available in the knowledge-base, this rule selects just one of them randomly and inserts this selection into the knowledge-base. Note the condition (`\+ trip:Trip`) which prevents the rule from being fired again.

The rest of the rules in the goal cover the answers to the general questions such as name, date of birth, credit-card information, etc. On the flight Web Service side, rule `wsRule(R02)` at line 27 checks the knowledge-base for any trip choice by the user; as soon as this choice becomes available, it asks for all the passenger identities. After receiving the answers from the goal, it then asks for credit-card information and checks its validity by querying the bank. If it receives a positive reply from the bank, it puts a reservation into the knowledge-base which satisfies the goal post-condition and the choreography terminates successfully; otherwise it fails. Table 5.2 shows what new knowledge is added to WM in each choreography round of a successful choreography, effectively tracing the execution of the choreography engine.

Table 5.2: Items added to WM at each choreography round

Round	Added knowledge
0	Name('Peter'). DateOfBirth(19830622). Gender('Male'). CreditCardNo('1234432156788765'). CreditCardHolder('PETER JACKSON'). CreditCardCVV(123). flight(F100,Paris,Chicago,23,250). flight(F101,Paris,Chicago,23,350). flight(F102,Paris,Chicago,25,400). flight(F103,Chicago,Paris,29,150). flight(F104,Chicago,Paris,30,200). flight(F105,Chicago,Paris,30,150). DB_CreditCard('876543212345678','PAUL BROWN,123). DB_CreditCard('1234432156788765','PETER JACKSON',123).
1	myRequest:RequestFlight[From->Paris] myRequest:RequestFlight[To->Chicago] myRequest:RequestFlight[Departure->23] myRequest:RequestFlight[Return->30]
2	tripChoice(F100,F104,450) tripChoice(F100,F105,400) tripChoice(F101,F104,550) tripChoice(F101,F105,500)
3	trip:Trip[Dep->F101] trip:Trip[Ret->F105]
4	question:QuestionByWS[Name->_h592309] question:QuestionByWS[DateOfBirth->_h592309] question:QuestionByWS[Gender->_h592309]
5	answer:AnswerByGoal[Name->Peter] answer:AnswerByGoal[DateOfBirth->19830622] answer:AnswerByGoal[Gender->Male]
6	question:QuestionByWS[CreditCardNo->_h592309] question:QuestionByWS[CreditCardHolder->_h592309] question:QuestionByWS[CreditCardCVV->_h592309]
7	answer:AnswerByGoal[CreditCardNo->1234432156788765] answer:AnswerByGoal[CreditCardHolder->PETER JACKSON] answer:AnswerByGoal[CreditCardCVV->123]
8	validation:CreditCardValidation[Number- >1234432156788765] validation:CreditCardValidation[Holder->PETER JACKSON] validation:CreditCardValidation[CVV->123]
9	BankYesNoAnswer(Yes)
10	reservation:Reservation[Number->11100] reservation:Reservation[Flight1->F101] reservation:Reservation[Flight2->F105]

5.7.2 Flight ticket reservation with data granularity mismatch

Here we show that the choreography specifications of the goal and Web Service used in Flight Reservation Service can have different data granularity. As Flora-2 keeps the frames in the form of separated `attribute->value` pairs, this mismatch is handled by the choreography engine. Listings 5.8 and 5.9 contain the specifications of the goal and Web Service respectively.

Listing 5.8: Goal choreography specification for flight reservation (granularity mismatch problem)

```
1  // Local ontology (will be stored on a separate file after
       deployed)
2  /*
3  Name('Peter').
4  DateOfBirth(19830622).
5  Gender('Male').
6  CreditCardNo('1234432156788765').
7  CreditCardHolder('PETER JACKSON').
8  CreditCardCVV(123).
9  */
10 myGoal:Goal.
11 myGoal[
12    importOntology -> '../Flight/GoalsOntology.flr',
13    capability -> ${
14        pre -> ${
15           myRequest:RequestFlight[
16                 From->'Paris',
17                 To->'Chicago',
18                 Departure->23,
19                 Return->30]
20        },
21
22        post -> ${
23           (?R:Reservation[?X->?Y])
24        }
25    },
26    gRule(R01):ChooseRule -> ${
27       \if
28          (tripChoice(?fl_dep,?fl_ret,?P), (\+ trip:Trip))@WM
29       \then ( %deltaInsert(${trip:Trip[Dep->?fl_dep,Ret->?fl_ret
            ]}) )
30    },
31
32    gRule(R02):ForallRule -> ${
33       \if (
34          (?Q:QuestionByWS[Name->?X], Name(?N))@WM
35       )
36       \then ( %deltaInsert(${answer:AnswerByGoal[Name->?N]}) )
37    },
38
```

```
39     gRule(R03):ForallRule -> ${
40        \if (
41           (?Q:QuestionByWS[DateOfBirth->?Y], DateOfBirth(?DoB))@WM
42        )
43        \then ( %deltaInsert(${answer:AnswerByGoal[DateOfBirth->?
              DoB]}) )
44     },
45
46     gRule(R04):ForallRule -> ${
47        \if (
48           (?Q:QuestionByWS[Gender->?Z], Gender(?G))@WM
49        )
50        \then ( %deltaInsert(${answer:AnswerByGoal[Gender->?G]}) )
51     },
52
53     gRule(R05):ForallRule -> ${
54        \if (
55           (?Q:QuestionByWS[
56                  CreditCardNo->?X,
57                  CreditCardHolder->?Y,
58                  CreditCardCVV->?Z])@WM,
59                 ( CreditCardNo(?CCN),
60                    CreditCardHolder(?CCH),
61                    CreditCardCVV(?CCCVV))@WM
62        )
63        \then (
64           %deltaInsert(${answer:AnswerByGoal[
65                                CreditCardNo->?CCN,
66                                CreditCardHolder->?CCH,
67                                CreditCardCVV->(?CCCVV)]})
68        )
69     }
70  ].
```

Listing 5.9: Web Service choreography specification for flight reservation (granularity mismatch problem)

```
 1  // Local ontology (will be stored on a separate file after
       deployed)
 2  /*
 3  flight(F100,Paris,Chicago,23,250).
 4  flight(F101,Paris,Chicago,23,350).
 5  flight(F102,Paris,Chicago,25,400).
 6  flight(F103,Chicago,Paris,29,150).
 7  flight(F104,Chicago,Paris,30,200).
 8  flight(F105,Chicago,Paris,30,150).
 9  */
10  FlightReservationService:WebService.
11  FlightReservationService[
12     importOntology -> '../Flight/WebServicesOntology.flr',
13     capability -> ${
14         pre -> ${ ?Req:RequestFlight[?X1->?Y1] },
15         post -> ${ (?Res:Reservation[?X2->?Y2]) }
16     },
17     wsRule(R01):ForallRule -> ${
18        \if (
```

```
19            (?R:RequestFlight[From->?X,To->?Y,Departure->?Z,Return
                 ->?W])@WM,
20           (flight(?fl_dep,?X,?Y,?Z,?priceDep))@WM,
21           (flight(?fl_ret,?Y,?X,?W,?priceRet))@WM,
22           (%sum(?priceDep,?priceRet,?priceTot))
23        )
24        \then (
25           %deltaInsert(${tripChoice(?fl_dep,?fl_ret,?priceTot)}))
                }},
26     wsRule(R02):ForallRule -> ${
27        \if
28           (?T:Trip[Dep->?fl_dep,Ret->?fl_ret])@WM
29        \then (
30           %deltaInsert(${question:QuestionByWS[
31                                 Name->?X,
32                                 DateOfBirth->?Y,
33                                 Gender->?Z]})) },
34     wsRule(R03):ForallRule -> ${
35        \if
36           (?A:AnswerByGoal[
37                  Name->?X,
38                  DateOfBirth->?Y,
39                  Gender->?Z])@WM
40        \then (
41           %deltaInsert(${question:QuestionByWS[
42                                 CreditCardNo->?XX,
43                                 CreditCardHolder->?YY,
44                                 CreditCardCVV->?ZZ]}) ) },
45     wsRule(R04):ForallRule -> ${
46        \if
47           (?A:AnswerByGoal[
48                 CreditCardNo->?X,
49                 CreditCardHolder->?Y,
50                 CreditCardCVV->?Z])@WM
51        \then (
52           %deltaInsert(${validation:CreditCardValidation[
53                                 Number->?X,
54                                 Holder->?Y,
55                                 CVV->?Z]}) ) },
56     wsRule(R05):ForallRule -> ${
57        \if (
58             (YesNoAnswer('Yes'))@WM,
59            (trip:Trip[Dep->?fl_dep,Ret->?fl_ret])@WM )
60        \then (
61           %deltaInsert(${reservation:Reservation[
62                                 Number->11100,
63                                 Flight1->?fl_dep,
64                                 Flight2->?fl_ret]}) ) },
65     wsRule(Bank_R01):ForallRule -> ${
66        \if (
67           (?R:CreditCardValidation[Number->?X,Holder->?Y,CVV->?Z])
                @WM,
68           (DB_CreditCard(?X,?Y,?Z))@WM)
69        \then (
70           %deltaInsert(${YesNoAnswer('Yes')}) ) }
71  ].
```

Rule `wsRule(R02)` at line 27 in the Web Service specification poses a question asking for passenger name, date of birth and gender in the form of a frame. In the goal specification, the answer to each of these items are provided by rules `gRule(R02)` (line 32), `gRule(R03)` (line 39), and `gRule(R04)` (line 46). However, the left-hand side of rule `wsRule(R03)` (line 35) in the Web Service choreography specification still can be proven when all items are inserted by the goal and the rule can fire.

5.7.3 Choosing a branch by the goal or Web Service

There are situations where either the goal or Web Service wants to continue the choreography in the specific branch they choose. For example, in an online payment scenario a goal can either choose to do the payment via credit-card or PayPal. Depending on the goal choice, the choreography moves on. Here, we show this possibility in an abstract example. Listings 5.10 and 5.11 depict the semantic goal and Web Service choreography specifications.

Listing 5.10: Goal choreography specification for online payment (choose case)

```
1  // Local ontology (will be stored on a separate file after
       deployed)
2  /*
3  CreditCardNo('1234432156788765').
4  CreditCardHolder('PETER JACKSON').
5  CreditCardCVV(123).
6  PayPalThreshold(1000).
7  PayPal(UserName,'PetJack').
8  PayPal(Password,'1234').
9  */
10 myGoal:Goal.
11 myGoal[
12     importOntology -> '../PayPal/GoalsOntology.flr',
13 
14     capability -> ${
15         pre -> ${ myRequest:RequestPurchase[Item->'ABC'] },
16         post -> ${ ?R:PurchaseReceipt[?X->?Y] }
17     },
18     gRule(R01):ForallRule -> ${
19        \if (
20             (?Q:QuestionByWS[
21                    price-> ?P,
22                    paymentMethod->?PM])@WM,
23                   (PayPalThreshold(?T))@WM , (?P > ?T)
```

```
24          )
25          \then (
26            %deltaInsert(${answer:AnswerByGoal[
27                                     paymentMethod->'CreditCard']}) ) },
28       gRule(R02):ForallRule -> ${
29          \if (
30               (?Q:QuestionByWS[
31                     price-> ?P,
32                     paymentMethod->?PM])@WM,
33                    (PayPalThreshold(?T))@WM, (?P =< ?T)
34          )
35          \then (
36            %deltaInsert(${answer:AnswerByGoal[
37                                     paymentMethod->'PayPal']}) ) },
38      gRule(R03):ForallRule -> ${
39          \if (
40               (?Q:QuestionByWS[
41                     CreditCardNo->?X,
42                     CreditCardHolder->?Y,
43                     CreditCardCVV->?Z])@WM,
44                    (CreditCardNo(?CCN),
45                         CreditCardHolder(?CCH),
46                         CreditCardCVV(?CCCVV))@WM
47          )
48          \then (
49             %deltaInsert(${answer:AnswerByGoal[
50                                      CreditCardNo->?CCN,
51                                      CreditCardHolder->?CCH,
52                                      CreditCardCVV->(?CCCVV)]}) ) },
53      gRule(R04):ForallRule -> ${
54          \if (
55               (?Q:QuestionByWS[
56                    PayPalUserName->?UN,
57                    PayPalPassword->?PW])@WM,
58                   (PayPal(UserName,?myUN),PayPal(Password,?myPW))@WM
59          )
60          \then (
61             %deltaInsert(${answer:AnswerByGoal[
62                                      PayPalUserName->?myUN,
63                                      PayPalPassword->?myPW]}) ) }
64   ].
```

Listing 5.11: Web Service choreography specification for online payment (choose case)

```
1   // Local ontology (will be stored on a separate file after
        deployed)
2   /*
3   */
4   OnlinePayment:WebService.
5   OnlinePayment[
6      importOntology -> '../PayPal/WebServicesOntology.flr',
7
8      capability -> ${
9          pre -> ${ ?Req:RequestPurchase[?X1->?Y1] },
10         post -> ${ ?Rec:PurchaseReceipt[?X2->?Y2] }
```

```
11    },
12    wsRule(R01):ForallRule -> ${
13       \if (
14          (?R:RequestPurchase[Item->?X])@WM
15       )
16       \then (
17          %deltaInsert(${Q:QuestionByWS[
18                            price-> 1700,
19                            paymentMethod->_]}) ) },
20    wsRule(R02):ForallRule -> ${
21       \if
22          (?A:AnswerByGoal[paymentMethod->'PayPal'])@WM
23       \then (
24          %deltaInsert(${Q:QuestionByWS[
25                            PayPalUserName->_,
26                            PayPalPassword->_]}) ) },
27    wsRule(R03):ForallRule -> ${
28       \if
29          (?A:AnswerByGoal[paymentMethod->'CreditCard'])@WM
30       \then (
31          %deltaInsert(${Q:QuestionByWS[
32                            CreditCardNo->_,
33                            CreditCardHolder->_,
34                            CreditCardCVV->_]}) ) },
35    wsRule(R04):ForallRule -> ${
36       \if
37          (?A:AnswerByGoal[
38                CreditCardNo->?X,
39                CreditCardHolder->?Y,
40                CreditCardCVV->?Z])@WM
41       \then (
42          %deltaInsert(${Rec1:PurchaseReceipt[Method->'CreditCard
               ']}) ) },
43    wsRule(R05):ForallRule -> ${
44       \if
45          (?A:AnswerByGoal[
46                PayPalUserName->?myUN,
47                PayPalPassword->?myPW])@WM
48       \then (
49          %deltaInsert(${Rec2:PurchaseReceipt[Method->'PayPal']})
               ) }
50 ].
```

In this example, the goal specification contains two rules which have complementary left-hand side conditions: `gRule(R01)` at line 18 checks if the price of the requested item is more than the specified threshold (in this case 1000), and `gRule(R02)` at line 28 checks if the price is less than or equal to the threshold. Based on the price, one of the answers `paymentMethod->'CreditCard'` or `paymentMethod->'PayPal'`

is provided by the goal and the Web Service continues the choreography based on it. As it can be seen, the Flora-2 relational operators such as > or =< are available for the choreography specification.

5.8 Handling granularity mismatch problem

Flora-2 has been used not only as the specification language of semantic Web Service capability and interface components, but also as the implementation language of the choreography execution engine itself. This choice gives the choreography developer a concise, frame based logical syntax to work with, as well as all the functionality of the underlying Flora-2 system in terms of its built-in predicates and reasoner. This is in contrast to WSML, the class of languages developed for WSMO, which has a verbose syntax, and must rely on external reasoners for all semantic computing activities, including choreography execution. Our choice of Flora-2 as both the specification and implementation language also helps us in dealing with the granularity mismatch problem [13] [42]. As explained in [13], data granularity can be a barrier to reach a successful choreography. Authors of [38] demonstrate the data granularity mismatch issue with an example: one Web Service requires credit-card details to be sent one at a time, whereas another requires that all details are sent in single message. Frame structures in Flora-2 intrinsically solve this type of granularity issue. For example, a credit-card can be defined as:

```
joeCreditCard:CreditCard[
              number -> "1234-5678-9012-3456",
              name -> "Joe Brown",
              CVV -> 123].
```

Internally, however, such a frame is represented as the composition of its data members, and each data member of a frame can be referred to individually, without the need to refer to other data members at the same time. Also, a frame can be built up

incrementally through the addition of its data members. Consequently, the granularity level with which frames of a certain concept are handled by the goal or Web Service becomes insignificant: the Web Service or goal can provide the constituents of a frame either in piecemeal fashion in any order or as a whole at once, and its counterpart can consume it under both conditions.

5.9 Relationship between Evolving Algebra and Evolving Ontologies

In Evolving Algebra (ASM) theory, functions can be partial and can evolve as time passes. For a function, not only can its previous range change, but also members of the domain that were not mapped to values in the co-domain under the function can be mapped to a value at a later stage. For example, if `f(a)` is `1`, `f(b)` is `2`, and `f(c)` is undefined, after a while (based on the transition rules) `f` might change in a way that `f(a)` remains `1`, but `f(b)` is mapped to `3` and `f(c)` is mapped to `4`.

Evolving Ontologies deal with objects, attributes and values of attributes, as well as relations. The state of the system at a given moment is determined by the objects that exist at that moment, the specific values of the attributes of each existing object, and instances of relations. Ontologies evolve through the insertion/deletion of objects and relations, as well as updates to the values of object attributes.

It turns out that evolving algebras and evolving ontologies are in fact equivalent to each other in the sense that through appropriate mappings, a choreography engine can simulate and ASM, and vice versa. Below, we give a brief formal definition of abstract state machines, evolving ontologies, and the mappings between the two that allow each one to simulate the other.

Definitions [87]: In an ASM state, *domains* (also called *universes*) contain data, with functions defined over the domains. The *superuniverse* is the union of all domains.

Relations are treated as Boolean valued functions, and domains are used interchangeably with characteristic functions (e.g. if b $\in$ A, then A(b)=True). A *vocabulary* Σ is a collection function names. Nullary function names (those with zero parameters) are called *constants*. The pair $(f, (a_1, ..., a_n))$, where f is a function name and $(a_1, ..., a_n)$ are parameters that the function can be applied to, is called a *location*. Every ASM vocabulary is assumed to contain the static constants *undef*, *True* and *False*. A *state* U of the vocabulary Σ consists of (i) the *superuniverse* of U (which we shall call X or $|U|$), and (ii) *interpretations* of the function names in Σ. For any n-ary function name f in Σ, its interpretation f^U is a function from X^n into X. If c is a constant of Σ, c^U is an element of X. *undef* is the default value of X and represents and undetermined object. The notions of *terms*, *formulas*, *substitutions*, *quantifiers*, *logical connectives*, and *interpretations* of terms and formulas are exactly the same as in first order logic.

Each *state* is an *algebra* in the mathematical sense of the word, with the exception that for $f \in \Sigma$, $f(v_1,...,v_n)=undef$ is permitted (i.e. functions can be partial). Furthermore, as the ASM is executing its transition rules, function interpretations can change over time, leading to the term *evolving algebras*. We should note, however, that the update rule in ASMs is of the form $f(t_1,...,t_n) := t_{n+1}$, where both the arguments and result of the function application are *terms*, not *values* in $|U|$. From a logic programming point of view, where we use terms to represent data, we can reasonably assume that $|U|$ is nothing more than H_∞, the Herbrand universe (i.e. the set of all ground terms) [46], and functions really map (tuples of) terms to other terms in H_∞.

On the ontology side, we have *concepts* (*classes* in programming language parlance), *instances* (also called *individuals*) that are members of concepts, *attributes* that are used describe properties of instances, *relations* that relate instances to one another,

and *axioms* (logic statements that say what is true in the domain of application). C is the set of concepts, T is the set of all terms, $I \subseteq T$ is the set of object identifiers denoting instances, R is the set of relation names, and A is the set of attributes. The term *evolving ontologies* comes about because update rules are used to change the values of object attributes, and add/delete relation or concept instances to/from the *working memory* to obtain a modified working memory. The complete contents of the working memory represent a state.

5.9.1 Simulation of choreography engine execution via ASM

One way to view members of A are as functions with domain I and codomain T, i.e. $a \in A : I \to T$. Since $I \subseteq T$, it is also true that $a \in A : T \to T$, with the provision that *if* $e \in (T$ - $I)$ *then A(e)=undef*, i.e. A is not defined for elements of T that are not in I. For any ontology with attribute set A, the actions of the choreography execution engine can be simulated by an ASM with vocabulary $\Sigma = R \cup A$. Table 5.3 gives the actions to be performed by an ASM that simulates the choreography engine execution. Remember that predicates in ASMs can be represented as Boolean valued functions.

Table 5.3: Simulating a move of the choreography engine with an ASM

Choreography Engine Action	**ASM Action**
Insertion of `b[a->c]`	`a(b) := c`
Deletion of `b[a->c]`	`a(b) := undef`
Update of the `a` attribute of `b` to value `c`	`a(b) := c`
Insertion of relation instance $r(a_1,\ldots,a_n)$	$r(a_1,\ldots,a_n)$:= True
Deletion of relation instance $r(a_1,\ldots,a_n)$	$r(a_1,\ldots,a_n)$:= False

Definition 3.1: An ASM state S_A is said to *correspond to* an ontological state S_C *iff*:

- whenever b $\in$ I and a $\in$ A and `b[a→c]` is true in S_C, then a $\in \Sigma$ and a(b)=c $\in S_A$.
- for any b $\in$ I and a$\in$A, if `b[a→c]` does not exist, then a(b)=undef in SA.

- whenever $r \in R$ and $r(a_1,...,a_n)$ is true in S_C, then $r \in \Sigma$ and $r(a_1,...,a_n)$=True in S_A.
- whenever $r \in R$ and $r(a_1,...,a_n)$ is false (i.e. absent) in S_C, then $r \in \Sigma$ and either $r(a_1,...,a_n)$=False or $r(a_1,...,a_n)$=undef is in S_A.

Definition 3.2 (*move of an ASM*): σ ~> ρ denotes one step move of an ASM when it goes from *abstract* state σ to *abstract* state ρ. σ ~>^{n} ρ denotes the fact that an ASM goes from *abstract* state σ to *abstract* state ρ in **n** moves.

Definition 3.3 (*move of a choreography engine*): $\mu \Rightarrow \beta$ denotes one step move of the choreography engine when it goes from *ontological* state μ to *ontological* state β. $\mu \Rightarrow^n \beta$ denotes the fact that the choreography engine goes from *ontological* state μ to *ontological* state β in **n** moves.

Theorem 3.1. (*simulation of choreography engine execution by ASM actions*). *Let* σ be a state of an ASM, and μ be an ontological state. Let $\Sigma = R \cup A$. If (σ corresponds to μ and $\mu \Rightarrow^n \beta$) then (σ ~>^{n} ρ and ρ corresponds to β), provided that each action of the choreography engine is replaced by its corresponding action as specified in Table 5.3.

Proof: By induction on the number of moves performed by the choreography engine. Let P(n) denote the statement "if (σ corresponds to μ and μ =>^{n} β) then (σ ~>^{n} ρ and ρ corresponds to β), provided that each action of the choreography engine is replaced by its corresponding action as specified in Table 5.3".

Basis: To prove the implication, we assume its antecedent and prove its consequent. By definition, $\mu \Rightarrow^0 \mu$ and σ ~>^{0} σ. By the antecedent of the implication, σ corresponds to μ. So, σ ~>^{0} σ (by definition) and σ corresponds to μ, establishing the consequent, and hence the truth of P(0).

Inductive hypothesis: Assume P(k), i.e. "(σ corresponds to μ and $\mu \Rightarrow^{k} \beta$) implies (σ ~>^{k} ρ *and* ρ corresponds to β)" is true. Consider the choreography execution sequence $\mu \Rightarrow^{k} \beta \Rightarrow \beta'$. Take an *arbitrary* action the engine performed to go from β to β'. If this action is

- an insertion of the form `b[a→c]`, according to Table 5.3, the function update action a(b):=c will be performed by the ASM engine as part of ρ ~> ρ'. Since by the inductive hypothesis ρ corresponds to β, after this insertion by the choreography engine as part of the move $\beta \Rightarrow \beta'$, and the corresponding update on the function **a** by the ASM engine as part of the move ρ ~> ρ', no violation of the correspondence relation is caused (i.e. according to Definition 3.1, if `b[a→c]` is in β', then a(b)=c should be in ρ', and it is).
- a deletion of the form `b[a→c]`, according to Table 5.3, the function update action a(b):=undef will be performed by the ASM engine as part of ρ ~> ρ'. Since by the inductive hypothesis ρ corresponds to β, after this deletion by the choreography engine as part of the move $\beta \Rightarrow \beta'$, and the corresponding update on the function **a** by the ASM engine as part of the move ρ ~> ρ', no violation of the correspondence relation is caused (i.e. according to Definition 3.1, if `b[a→c]` is not in β', then a(b)=undef should be in ρ').
- an update of the **a** attribute of **b** to value **c**, according to Table 5.3, the function update action a(b):=c will be performed by the ASM engine as part of ρ ~> ρ'. Since by the inductive hypothesis ρ corresponds to β, after this update by the choreography engine as part of the move $\beta \Rightarrow \beta'$, and the corresponding update on the function **a** by the ASM engine as part of the move ρ ~> ρ', no violation of the correspondence relation is caused (i.e. according to Definition

3.1, if b[a→c] is in β', then a(b)=c should be in ρ', and it is).

- an insertion of relation instance r(a_1, ..., a_n), according to Table 5.3, the function update action r(a_1, ..., a_n):=True will be performed by the ASM engine as part of ρ ~> ρ'. Since by the inductive hypothesis ρ corresponds to β, after this insertion by the choreography engine as part of the move $\beta \Rightarrow \beta'$, and the corresponding update on the function **r** by the ASM engine as part of the move ρ ~> ρ', no violation of the correspondence relation is caused (i.e. according to Definition 3.1, if r(a_1, ..., a_n) is in β', then r(a_1, ..., a_n):=True should be in ρ', and it is).
- a deletion of relation instance r(a_1, ..., a_n), according to Table 5.3, the function update action r(a_1, ..., a_n):=False will be performed by the ASM engine as part of ρ ~> ρ'. Since by the inductive hypothesis ρ corresponds to β, after this deletion by the choreography engine as part of the move $\beta \Rightarrow \beta'$, and the corresponding update on the function **r** by the ASM engine as part of the move ρ ~> ρ', no violation of the correspondence relation is caused (i.e. according to Definition 3.1, if r(a_1, ..., a_n) is not in β', then r(a_1, ..., a_n)=False or r(a_1, ..., a_n)=undef should be in ρ', and it is).

Since none of the actions specified in Table 5.3 cause a violation of the correspondence relation defined in Definition 3.1, and since no other actions are allowed beside those in Table 5.3, we conclude that ρ' corresponds to β', establishing the truth of P(k+1). QED

5.9.2 Simulation of ASM execution via choreography engine

The execution of any ASM can be simulated by a choreography engine using evolving ontologies. The requirement is that functions need to be represented somehow in the ontology. The reverse of the mapping given in the previous Section 5.9.1

(i.e. whenever a(b)=c in S_A then `b[a→c]` is true in S_C) does not always work, since a function may be n-ary, where n > 1. One possibility is to map functions of the ASM to relations of the ontology. Specifically, every n-ary function of the ASM can be considered an (n+1)-ary predicate of the ontology. Another possibility is to have an object called *func*, with locations as attribute names, and the value associated with the location as the value of the attribute. For example, if *f(a,b,c)=d* in the ASM, then *func[f(a,b,c)→d]* is its representation in the ontology. We use the second approach, since we can update objects in an atomic manner, but relation instances are not updatable in one step.

Definition 3.4: An ontological state S_C is said to correspond to an ASM state S_A *iff*:

- whenever $f \in \Sigma$ and $f(b_1,...,b_n) = b_{n+1}$ is in S_A ($b_{n+1} \neq$ undef) then $func \in I$ and $f(b_1,...,b_n) \in A$ and $func[f(b_1,...,b_n) \rightarrow b_{n+1}]$ exists in S_C.
- whenever $f \in \Sigma$ and $f(b_1,...,b_n)=undef$, then $func[f(b_1,...,b_n) \rightarrow b_{n+1}]$ does not exist in S_C for any value b_{n+1}.

Theorem 3.2. (*simulation of ASM execution by choreography actions*). Let μ be an ontological state and σ be a state of an ASM. Let $A = \{f(a_1,..., a_n) \mid (f, (a_1,...,a_n))$ *is a location of the ASM*$\}$. Let $I = \{func\}$. If (μ corresponds to σ and σ ~>^{n} ρ) then ($\mu \Rightarrow^n \beta$ and β corresponds to ρ), provided that each action of the ASM is replaced by its corresponding action as specified in Table 5.4.

Proof: By induction on the number of moves performed by the ASM engine. Let P(n) denote the statement "if (μ corresponds to σ and σ ~>^{n} ρ) then ($\mu \Rightarrow^n \beta$ and β corresponds to ρ), provided that each action of the choreography engine is replaced by its corresponding action as specified in Table 5.4".

Table 5.4: Simulating a move of the ASM with a choreography engine

ASM Action	Choreography Engine Action
$\texttt{f(b}_1\texttt{,...,b}_n\texttt{) := b}_{n+1}$ ($\texttt{b}_{n+1} \neq$ undef)	Update the func object's $\texttt{f(b}_1\texttt{, ..., b}_n\texttt{)}$ attribute to $\texttt{b}_{n+1}$ (if the attribute $\texttt{f(b}_1\texttt{, ..., b}_n\texttt{)}$ does not exist for func, it is created in the update procedure).
$\texttt{f(b}_1\texttt{,...,b}_n\texttt{):= undef}$	Delete $\texttt{func[f(b}_1\texttt{, ..., b}_n\texttt{)-> ?_]}$, where ?_ is a free variable(if $\texttt{f(b}_1\texttt{, ..., b}_n\texttt{)}$ does not exist in func, nothing is done in the delete procedure).

Basis: To prove the implication, we assume its antecedent and prove its consequent. By definition, $\mu \Rightarrow^0 \mu$ and σ ~>^{0} σ. By the antecedent of the implication, μ corresponds to σ. So, $\mu \Rightarrow^0 \mu$ and μ corresponds to σ, establishing the consequent, hence the truth of P(0).

Inductive hypothesis: Assume P(k), i.e. "(μ corresponds to σ and σ ~>^{k} ρ) implies ($\mu \Rightarrow^k \beta$ *and* β corresponds to ρ)" is true. Consider the ASM execution sequence σ ~>^{k} ρ ~> ρ'. Take an *arbitrary* action the ASM engine performed to go from ρ ~> ρ'. If this action is

- $f(b_1, ..., b_n) := b_n$ ($b_n \neq$ undef), according to Table 5.4, the func object's $f(b_1, ..., b_n)$ attribute will be updated by the choreography engine as part of the move $\beta \Rightarrow \beta'$. Since by the inductive hypothesis β corresponds to ρ, after this update by the ASM engine as part of the move ρ ~> ρ', and the corresponding update on the on $f(b_1, ..., b_n)$ attribute of the func object by the choreography engine as part of the move $\beta \Rightarrow \beta'$, no violation of the correspondence relation given in Definition 3.4 is caused (i.e. according to Definition 3.4, if $f(b_1, ..., b_n) = b_{n+1}$ is in ρ', func[$f(b_1, ..., b_n) \rightarrow b_{n+1}$] should be true, and it is).
- $f(b_1, ..., b_n)$:= undef, according to Table 5.4, the func object's $f(b_1, ..., b_n)$ attribute will be deleted completely by choreography engine as part of the move $\beta \Rightarrow \beta'$. Since by the inductive hypothesis β corresponds to ρ, after this update

by the ASM engine as part of the move ρ ~> ρ', and the corresponding deletion of the f(b_1, ..., b_n) attribute of the `func` object by the choreography engine as part of the move $\beta \Rightarrow \beta'$, no violation of the correspondence relation given in Definition 3.4 is caused (i.e. according to Definition 3.4, if f(b_1, ..., b_n) = undef is in ρ', func[f(b_1, ..., b_n)→b_{n+1}] should not exist in β', and it does not).

- Since none of the actions specified in Table 5.4 cause a violation of the correspondence relation defined in Definition 3.4, and since no other actions are allowed beside those in Table 5.4, we conclude that β' corresponds to ρ', establishing the truth of P(k+1). QED

5.10 Comparison with IRS-III and WSMX

In this section we show the major differences between our approach and the two available WSMO choreography implementations that are WSMX [51] [94] [80] and IRS-III [38] [27] in a comparison table (Table 5.5).

Table 5.5: Comparison of our approach with IRS-III and WSMX

Concern or Specification	**IRS-III**	**WSMX**	**Our solution**	**Comment**
Adhering to WSMO Choreography	No	Yes	Yes	WSMX Choreography project has been stopped at early stages.
Adhering to ASM concepts	No	Partially	Yes	WSMX Choreography project has been stopped at early stages.
Underlying language	OCML	WSML+ KAON2	Flora-2	No usage of KAON2 has been found in WSMX documentation.
Using peer-to-peer architecture	No	Yes	Yes	Based on WSMO principles
ASM if-then	Yes	Yes	Yes	Transition rules are fired simultaneously.
ASM Parallelism	No	No	Yes	Transition rules are fired simultaneously.
ASM Choose	No	No	Yes	One among some transition rules is chosen to be fired randomly.
WSMO modes of concepts	No	No	Yes	Controlling access modes to concepts.
Checking for inconsistent actions	No	No	Yes	In the presence of ASM parallelism, contradictory actions should be prevented.

We explained them in detail in section 3.2. Once again we emphasize that WSMX choreography project was stopped in the early stages; so, in theory it adheres to WSMO choreography, but no complete implementation is available for it.

5.11 Summary

In this chapter, we explained why the current available WSMO choreography algorithm is insufficient to be applied in practice and introduced a rectified algorithm for it. Based on the rectified algorithm we implemented the first working choreography engine conforming to the all ASM principles used in the WSMO choreography definition. We showed how the concerns such as simultaneous running transition rules in ASM, non-deterministic *choose* among transition rules, and access controls can be implemented in Flora-2. Similar to the previous chapter, we provided a new semantic choreography language over Flora-2 to define the specification of the choreography component of goals and Web Services. Choreography specifications and the capability of our choreography engine to choreograph them were demonstrated through three examples. Moreover, we explained how the granularity mismatch problem can be inherently solved by using Flora-2 as the underlying language. Finally, we formally proved the equivalence of traditional ASMs and ontological ASMs (used in WSMO) which was missing in the literature before.

Chapter 6

TIMING EVALUATION

In this chapter we evaluate the scalability of the designed choreography engine with regard to the complexity and size of goal and Web Service specifications. Time is the main concern in our evaluation. We test and compare abstract choreography specifications which are different in number of rules and different in complexity of rules' Left-Hand Sides (LHSs) and Right-Hand Sides (RHSs). We find the trend of the engine response time for each case.

One of the most important features of the ASM-based choreography engine is that the rules are checked and fired in parallel (although in the physical layer they are checked sequentially but their effects are aggregated as if they were fired in parallel). This parallelism lets rules that are checked and fired again and again unless some guards prevent them from being fired. Therefore, firstly we study the relation between the number of rules in choreography specifications of both goal and Web Service and the response time and see how the response time changes by increasing in number of total rules. For the second experiment we put some guards in the rules' LHSs and some insertions in the rules' RHSs in a way that they prevent rules from being fired again and see how the response time is improved. At last we see how the number of conjunctions - representing the complexity of LHSs can affect the response time of the choreography engine.

To compute the response time, we only consider the time needed for the pred-

icate %runChoreography to reach a conclusion. To do so, we use the XSB function epoch_milliseconds [14] to get the time of the machine once just before calling of the predicate %runChoreography and once just after it and compute the difference between them in terms of seconds and milliseconds. Table 6.1 shows the added lines to the choreography engine. The complete engine code modified for testing is available in Appendix G.

Table 6.1: Changes in the choreography engine code

Added lines to the engine code	**%Duration predicate**

`...`
`epoch_milliseconds(?S1,?MS1)`
`  @\prolog(machine),`
 	
`%runChoreography(?goal,?WS),`
 	
`epoch_milliseconds(?S2,?MS2)`
`  @\prolog(machine),`
`%duration`
`  (?S2,?MS2,?S1,?MS1,`
`   ?DS,?DMS),`
`...`
	`%duration(?S2,?MS2,?S1,?MS1,`
	`          ?DS,?DMS) :-`
	`  ?DMS_temp \is ?MS2 - ?MS1,`
	`  \if (?DMS_temp < 0)`
	`  \then`
	`    (?DS \is ?S2 - ?S1 - 1,`
	`         ?DMS \is ?DMS_temp + 1000)`
	`  \else`
	`    (?DS \is ?S2 - ?S1,`
	`         ?DMS \is ?DMS_temp).`

All the experiments were done on Windows 7 64-bit OS running on Intel® Core^{TM} i5-2410M CPU @ 2.30GHz with almost 5.7 GB of available free RAM. Reasoner is Flora-2/ErgoLite Reasoner 1.2 (*Monstera deliciosa*) of 2017-03-26 (rev: a2d10ed) and XSB engine is XSB Version 3.7.0 (*Clan MacGregor*) of 2017-03-26.

6.1 Experimental environment

Here, we briefly explain how we generate abstract goal and Web Service choreography specifications. To make comparable abstract specifications, we wrote C# code whose role is to change the size of choreography specifications at each test in terms of number transition rules as well as degree of complexity in the rules' LHSs and RHSs. The complete C# code and the tested specifications are given in Appendix G.

The goal and Web Service choreography specifications are generated in a way that the choreography begins with inserting the goal precondition (as usual) and after completely firing of all rules in both goal and Web Service, the goal post-condition becomes provable by firing of the last rule from the Web Service side. We deliberately generate choreography specifications in this way to simulate the worst possible case of a choreography run.

As it is shown in the subsequent sections (Listings 6.1, 6.2, and 6.3), both goal and Web Service have the frame(s) in form of `obj:Concept[attr_x->val_x]`, wherein `x` represents a number, in their LHSs and have `%deltaInsert($obj:Concept[attr_x->val_x])` action(s) in their RHSs. In the first two experiments the difference between test cases are only in their number of transition rules. It means the choreography engine has to check and fire different number of transition rules to accomplish the choreography run. In the last experiment, we keep the size of the specification fixed in terms of the number of transition rules at each test, but we change the size by making different complexities in LHSs and RHSs of the transition rules. This is done by making conjunctions of more frames at LHSs and making conjunctions of more insertion actions at RHSs, so the choreography engine has to prove the existence of every frame at LHSs and inserting all the frames specified at RHSs.

The overall procedure of testing is that the rule generator explained above is run with different input parameters to generate a variety of specifications for both goal and Web Service. At each generation step, we run the choreography engine over the generated specification and record the time spent for the processing of `%runChoreography` predicate given by the choreography engine. The timing results of runs are averaged and compared to each other. In the following, we explain each test separately.

6.2 Response time w.r.t. the number of rules

For this experiment, we generate abstract goal and Web Service specifications which are only different in total number of rules. Each combination is run independently from scratch to make almost same experiment environment for all tests.

As a starting point we suppose both goal and Web Service have 5 rules (including *goal.pre*), each rule has only one frame at LHS and one `%deltaInsert` at RHS. The choreography is designed in a way that all the rules in both goal and Web Service have to be fired and the last rule in Web Service provides the post-condition of the goal which makes a successful run. Listing 6.1 shows how typical goal and Web Service specifications are for this experiment. The specifications were generated by running of the C# program given in Listing 7.16 (Appendix G) with the input parameter 10. We also ran the generator with input parameters 20, 30, 40, and 50 for this experiment and took the time needed by the choreography engine to completely choreograph the specifications which are recored in Table 6.2. The raw data containing the execution times recorded by the choreography engine is given in the Appendix G. This raw data is written by Line 22 of the code given in Listing 7.17.

Listing 6.1: Goal and Web Service Choreography specifications

```
1  myGoal:Goal.
2  myGoal[
3     importOntology ->
4          '../Benchmarking/Choreography/Bench/GoalsOntology.flr',
5
6     capability -> ${
7          pre -> ${obj:Concept[attr_1->val_1]},
8          post -> ${obj:Concept[attr_10->val_10]}},
9     gRule(R1):ForallRule -> ${
10         \if (obj:Concept[attr_2->val_2]@WM)
11         \then (%deltaInsert(${obj:Concept[attr_3->val_3]})) },
12    gRule(R3):ForallRule -> ${
13         \if (obj:Concept[attr_4->val_4]@WM)
```

```
14          \then (%deltaInsert(${obj:Concept[attr_5->val_5]})) },
15     gRule(R5):ForallRule -> ${
16          \if (obj:Concept[attr_6->val_6]@WM)
17          \then (%deltaInsert(${obj:Concept[attr_7->val_7]})) },
18     gRule(R7):ForallRule -> ${
19          \if (obj:Concept[attr_8->val_8]@WM)
20          \then (%deltaInsert(${obj:Concept[attr_9->val_9]})) }
21  ].
22  ----------------------------------------------------------
23  myService:WebService.
24  myService[
25     importOntology ->
26          '../Benchmarking/Choreography/Bench/WebServicesOntology.
               flr',
27
28     capability -> ${
29          pre -> ${?OBJ:Concept[?_X1->?_Y1]},
30          post -> ${?OBJ:Concept[?_X2->?_Y2]}},
31     wsRule(R1):ForallRule -> ${
32          \if (obj:Concept[attr_1->val_1]@WM)
33          \then (%deltaInsert(${obj:Concept[attr_2->val_2]})) },
34     wsRule(R3):ForallRule -> ${
35          \if (obj:Concept[attr_3->val_3]@WM)
36          \then (%deltaInsert(${obj:Concept[attr_4->val_4]})) },
37     wsRule(R5):ForallRule -> ${
38          \if (obj:Concept[attr_5->val_5]@WM)
39          \then (%deltaInsert(${obj:Concept[attr_6->val_6]})) },
40     wsRule(R7):ForallRule -> ${
41          \if (obj:Concept[attr_7->val_7]@WM)
42          \then (%deltaInsert(${obj:Concept[attr_8->val_8]})) },
43     wsRule(R9):ForallRule -> ${
44          \if (obj:Concept[attr_9->val_9]@WM)
45          \then (%deltaInsert(${obj:Concept[attr_10->val_10]})) }
46  ].
```

As can be seen, the choreography is started by the goal precondition with the frame `obj:Concept[attr_1->val_1]` and continues to run step-by-step to R9 of the Web Service. In the last round, R9 is fired and the frame `obj:Concept[attr_10->val_10]` is put into WM, which is the post-condition of the goal. Therefore, in this way the choreography succeeds in 10 rounds (including goal precondition insertion). The sequence of firings is arranged in a way that specifications model the worst-case of a

choreography run.

We repeated the experiment for 3 times with total number of 10, 20, 30, 40, and 50 rules. The response times are shown in Table 6.2.

Figure 6.1 depicts the trend of time with respect to the number of rules. As can be seen, time complexity is *O(n^2)*. The reason is that in this experiment the number of rules does not reflect the number of firings, because at each round, the previously matched LHSs are again fired. To count the total number of firings, one can count

Table 6.2: Choreography engine response time

# of rules	# of firings	RT(ms) 1st run	RT(ms) 2nd run	RT(ms) 3rd run	Average (ms)
10	25	2528	2621	2558	2569
20	100	10717	11466	10702	10961.67
30	225	24897	24711	24274	24627.34
40	400	43150	44585	43446	43727
50	625	67501	68322	66971	67598

them in each round and then sum them up. We use the pair *(n,m)* to show the number of firings for the Web Service and goal respectively at each round; therefore, for specification with total number of 10 rules, in the 1st round we have *(1,0)*, 2nd round *(2,1)*, 3rd round *(3,2)*, 4th round *(4,3)*, and 5th round *(5,4)* of firings. The sum of all firings will be (1+2+3+4+5) for the Web Service and (0+1+2+3+4) for the goal which are 25 in total.

Hence, for this experiment Formula 6.1 can be applied to count the total number of firings, wherein R is the total number of rules.

$$\sum_{i=1}^{\frac{R}{2}} i + \sum_{i=0}^{\frac{R}{2}-1} i = \left(\frac{R}{2}\right)^2 \tag{6.1}$$

Figure 6.2 shows the trend of response time w.r.t. the number of firings which is

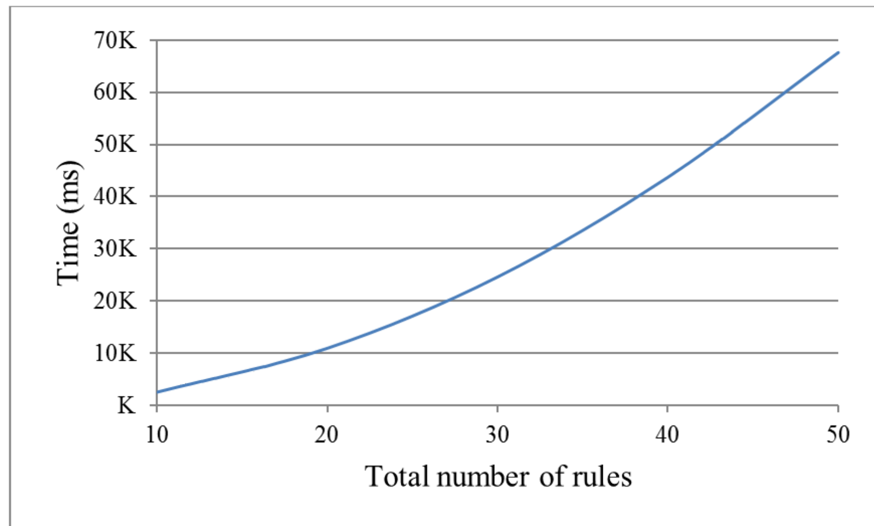

Figure 6.1: Trend of time w.r.t. the number of rules

obviously linear (i.e. *O(n)*).

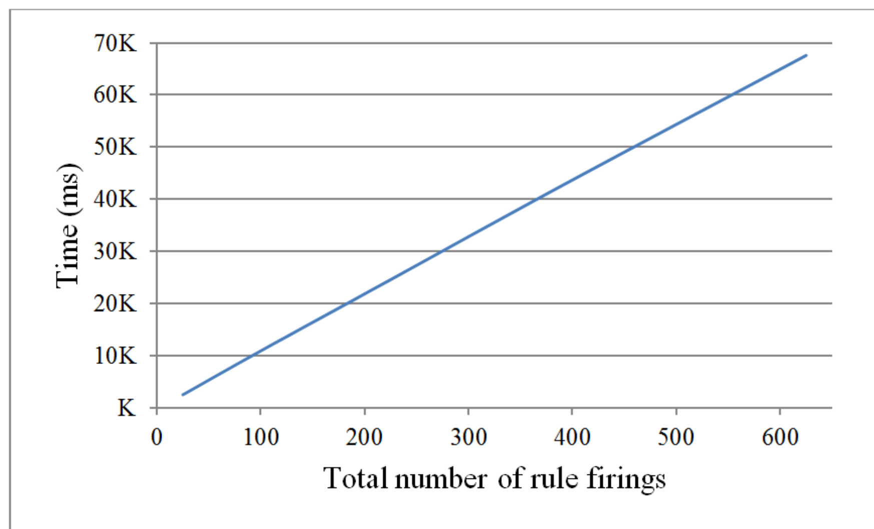

Figure 6.2: Trend of time w.r.t. the number of rule firings

6.3 Response time w.r.t. the number of rules (re-firing is prevented)

For the second experiment, we write the choreography specification in a way that by firing of a rule a guard is activated at the rule's RHS to prevent re-firing of the same rule in the next rounds. In this way each rule is only fired once however, checking of the LHSs should be done anyway. Listing 6.2 shows how the goal and Web Service choreography specifications are for this experiment.

As it can be seen, the negation of the frame `(\+ thisRule:Control[off->N])@WM` is checked at LHS of every rule; if the whole condition is true (i.e. the frame `thisRule:Control[off->N])@WM` is not present in WM) then the rule is fired.

Listing 6.2: Goal and Web Service Choreography specifications

```
1  myGoal:Goal.
2  myGoal[
3     importOntology ->
4        '../Benchmarking/Choreography/Bench/GoalsOntology.flr',
5
6     capability -> ${
7        pre -> ${obj:Concept[attr_1->val_1]  },
8        post -> ${obj:Concept[attr_10->val_10]  }},
9     gRule(R1):ForallRule -> ${
10       \if ((\+ thisRule:Control[off->2])@WM ,
11             obj:Concept[attr_2->val_2]@WM)
12       \then (%deltaInsert(${thisRule:Control[off->2]}) ,
13              %deltaInsert(${obj:Concept[attr_3->val_3]})  )},
14    gRule(R3):ForallRule -> ${
15       \if ((\+ thisRule:Control[off->4])@WM ,
16             obj:Concept[attr_4->val_4]@WM)
17       \then (%deltaInsert(${thisRule:Control[off->4]}) ,
18              %deltaInsert(${obj:Concept[attr_5->val_5]})  )},
19    gRule(R5):ForallRule -> ${
20       \if ((\+ thisRule:Control[off->6])@WM ,
21             obj:Concept[attr_6->val_6]@WM)
22       \then (%deltaInsert(${thisRule:Control[off->6]}) ,
23              %deltaInsert(${obj:Concept[attr_7->val_7]})  )},
24    gRule(R7):ForallRule -> ${
25       \if ((\+ thisRule:Control[off->8])@WM ,
26             obj:Concept[attr_8->val_8]@WM)
27       \then (%deltaInsert(${thisRule:Control[off->8]}) ,
28              %deltaInsert(${obj:Concept[attr_9->val_9]})  )}
29 ].
30 -----------------------------------------------------------------

31 myService:WebService.
32 myService[
33    importOntology ->
34        './Benchmarking/Choreography/Bench/WebServicesOntology.flr
           ',
35    capability -> ${
36        pre -> ${?OBJ:Concept[?_X1->?_Y1]},
37        post -> ${?OBJ:Concept[?_X2->?_Y2]}},
38    wsRule(R1):ForallRule -> ${
```

```
39        \if ((\+ thisRule:Control[off->1])@WM ,
40             obj:Concept[attr_1->val_1]@WM )
41        \then (%deltaInsert(${thisRule:Control[off->1]}) ,
42               %deltaInsert(${obj:Concept[attr_2->val_2]})  )},
43    wsRule(R3):ForallRule -> ${
44        \if ((\+ thisRule:Control[off->3])@WM ,
45             obj:Concept[attr_3->val_3]@WM )
46        \then (%deltaInsert(${thisRule:Control[off->3]}) ,
47               %deltaInsert(${obj:Concept[attr_4->val_4]})  )},
48    wsRule(R5):ForallRule -> ${
49        \if ((\+ thisRule:Control[off->5])@WM ,
50             obj:Concept[attr_5->val_5]@WM )
51        \then (%deltaInsert(${thisRule:Control[off->5]}) ,
52               %deltaInsert(${obj:Concept[attr_6->val_6]})  )},
53    wsRule(R7):ForallRule -> ${
54        \if ((\+ thisRule:Control[off->7])@WM ,
55             obj:Concept[attr_7->val_7]@WM )
56        \then (%deltaInsert(${thisRule:Control[off->7]}) ,
57               %deltaInsert(${obj:Concept[attr_8->val_8]})  )},
58    wsRule(R9):ForallRule -> ${
59        \if ((\+ thisRule:Control[off->9])@WM ,
60             obj:Concept[attr_9->val_9]@WM )
61        \then (%deltaInsert(${thisRule:Control[off->9]}) ,
62               %deltaInsert(${obj:Concept[attr_10->val_10]}) )}
63 ].
```

By firing the rule, in addition to other frames, `thisRule:Control[off->N])@WM` is inserted as well at RHS of the rule; so, in the next rounds of choreography, the negation gets false and the rule won't be fired again. We repeat the previous experiment (Section 6.2) with this additional consideration for 10, 20, 30, 40, and 50 rules. Timing results are shown in Table 6.3.

Table 6.3: Choreography engine response time

# of rules	**# of firings**	**RT(ms) 1st run**	**RT(ms) 2nd run**	**RT(ms) 3rd run**	**Average (ms)**
10	10	1389	1393	1342	1374.67
20	20	3420	2995	3730	3381.67
30	30	4914	4992	4961	4955.67
40	40	6921	6958	6880	6919.67
50	50	9204	9173	9469	9282.00

Because at each round only one rule is fired, the total number of firings is equal to the total number of rules in each test. Figure 6.3 depicts the trend of time w.r.t. the total number of rules which is obviously linear (i.e. *O(n)*).

As expected, the time of the choreography run is dramatically reduced when unnecessary repetitive firings are prevented.

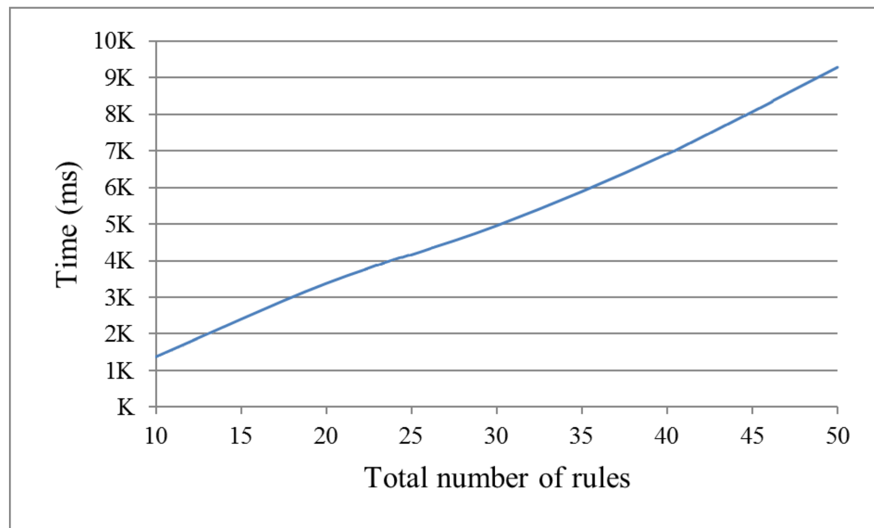

Figure 6.3: Trend of time w.r.t. the number of rules (re-firing is prevented)

6.4 Response time w.r.t. complexity of LHSs and RHSs

For the last experiment, we consider the choreography specifications which are different in the number of attributes. To do this, we generate four sets of the goal and Web Service choreography specifications with total number of 10 rules. The first set of specifications contain 1 frame in LHS and 1 insertion in RHS, the second set contains conjunctions of 2 frames in LHS and conjunctions of 2 insertions in RHS, the third set contains conjunctions of 4 frames in LHS and conjunctions of 4 insertions in RHS, and the fourth set contains conjunctions of 8 frames in LHS and conjunctions of 8 insertions in RHS. Similar to the first experiment (Section 6.2), we do not prevent re-firing of rules to see the effects of complexity differences better. Listing 6.3 shows

goal's choreography specifications with 2 and 4 attributes. Web Service's choreography specification is similar (timings of the first set is taken from the first experiment in Section 6.2).

Listing 6.3: Goal specification

```
1  myGoal:Goal.
2  myGoal[
3     importOntology ->
4        '../Benchmarking/Choreography/Bench/GoalsOntology.flr',
5
6     capability -> ${
7        pre -> ${obj:Concept[attr_A1->val_A1],
8                 obj:Concept[attr_B1->val_B1]},
9        post -> ${obj:Concept[attr_A10->val_A10],
10                obj:Concept[attr_B10->val_B10]}},
11    gRule(R1):ForallRule -> ${
12       \if (obj:Concept[attr_A2->val_A2]@WM ,
13            obj:Concept[attr_B2->val_B2]@WM)
14       \then (%deltaInsert(${obj:Concept[attr_A3->val_A3]}) ,
15              %deltaInsert(${obj:Concept[attr_B3->val_B3]})) },
16    gRule(R3):ForallRule -> ${
17       \if (obj:Concept[attr_A4->val_A4]@WM ,
18            obj:Concept[attr_B4->val_B4]@WM)
19       \then (%deltaInsert(${obj:Concept[attr_A5->val_A5]}) ,
20              %deltaInsert(${obj:Concept[attr_B5->val_B5]})) },
21    gRule(R5):ForallRule -> ${
22       \if (obj:Concept[attr_A6->val_A6]@WM ,
23            obj:Concept[attr_B6->val_B6]@WM)
24       \then (%deltaInsert(${obj:Concept[attr_A7->val_A7]}) ,
25              %deltaInsert(${obj:Concept[attr_B7->val_B7]})) },
26    gRule(R7):ForallRule -> ${
27       \if (obj:Concept[attr_A8->val_A8]@WM ,
28            obj:Concept[attr_B8->val_B8]@WM)
29       \then (%deltaInsert(${obj:Concept[attr_A9->val_A9]}) ,
30              %deltaInsert(${obj:Concept[attr_B9->val_B9]})) }
31 ].
32 -------------------------------------------------------------
33 myGoal:Goal.
34 myGoal[
35    importOntology ->
36         '../Benchmarking/Choreography/Bench/GoalsOntology.flr',
37
38    capability -> ${pre -> ${
39         obj:Concept[attr_A1->val_A1],
40         obj:Concept[attr_B1->val_B1],
```

```
41          obj:Concept[attr_C1->val_C1],
42          obj:Concept[attr_D1->val_D1]  },
43     post -> ${
44          obj:Concept[attr_A10->val_A10],
45          obj:Concept[attr_B10->val_B10],
46          obj:Concept[attr_C10->val_C10],
47          obj:Concept[attr_D10->val_D10]  }},
48     gRule(R1):ForallRule -> ${
49        \if (
50          obj:Concept[attr_A2->val_A2]@WM ,
51          obj:Concept[attr_B2->val_B2]@WM ,
52          obj:Concept[attr_C2->val_C2]@WM ,
53          obj:Concept[attr_D2->val_D2]@WM)
54        \then (
55          %deltaInsert(${obj:Concept[attr_A3->val_A3]}) ,
56          %deltaInsert(${obj:Concept[attr_B3->val_B3]}) ,
57          %deltaInsert(${obj:Concept[attr_C3->val_C3]}) ,
58          %deltaInsert(${obj:Concept[attr_D3->val_D3]}) )},
59     gRule(R3):ForallRule -> ${
60        \if (
61          obj:Concept[attr_A4->val_A4]@WM ,
62          obj:Concept[attr_B4->val_B4]@WM ,
63          obj:Concept[attr_C4->val_C4]@WM ,
64          obj:Concept[attr_D4->val_D4]@WM)
65        \then (
66          %deltaInsert(${obj:Concept[attr_A5->val_A5]}) ,
67          %deltaInsert(${obj:Concept[attr_B5->val_B5]}) ,
68          %deltaInsert(${obj:Concept[attr_C5->val_C5]}) ,
69          %deltaInsert(${obj:Concept[attr_D5->val_D5]}) )},
70     gRule(R5):ForallRule -> ${
71        \if (
72          obj:Concept[attr_A6->val_A6]@WM ,
73          obj:Concept[attr_B6->val_B6]@WM ,
74          obj:Concept[attr_C6->val_C6]@WM ,
75          obj:Concept[attr_D6->val_D6]@WM)
76        \then (
77          %deltaInsert(${obj:Concept[attr_A7->val_A7]}) ,
78          %deltaInsert(${obj:Concept[attr_B7->val_B7]}) ,
79          %deltaInsert(${obj:Concept[attr_C7->val_C7]}) ,
80          %deltaInsert(${obj:Concept[attr_D7->val_D7]}) )},
81     gRule(R7):ForallRule -> ${
82        \if (
83          obj:Concept[attr_A8->val_A8]@WM ,
84          obj:Concept[attr_B8->val_B8]@WM ,
85          obj:Concept[attr_C8->val_C8]@WM ,
86          obj:Concept[attr_D8->val_D8]@WM)
87        \then (
88          %deltaInsert(${obj:Concept[attr_A9->val_A9]}) ,
89          %deltaInsert(${obj:Concept[attr_B9->val_B9]}) ,
```

```
90          %deltaInsert(${obj:Concept[attr_C9->val_C9]}) ,
91          %deltaInsert(${obj:Concept[attr_D9->val_D9]}) )}
92  ].
```

Figure 6.4 shows the trend of time w.r.t. the number of terms (frames and insertions) in LHS and RHS. It is clearly seen that the trend is linear (i.e. *O(n)*).

Table 6.4: Choreography engine response time

# of terms in LHS and RHS	RT(ms) 1st run	RT(ms) 2nd run	RT(ms) 3rd run	Average (ms)
1	2528	2621	2558	2569
2	4870	5268	4968	5035.34
4	9128	9379	9411	9306
8	18135	17816	18450	18133.67

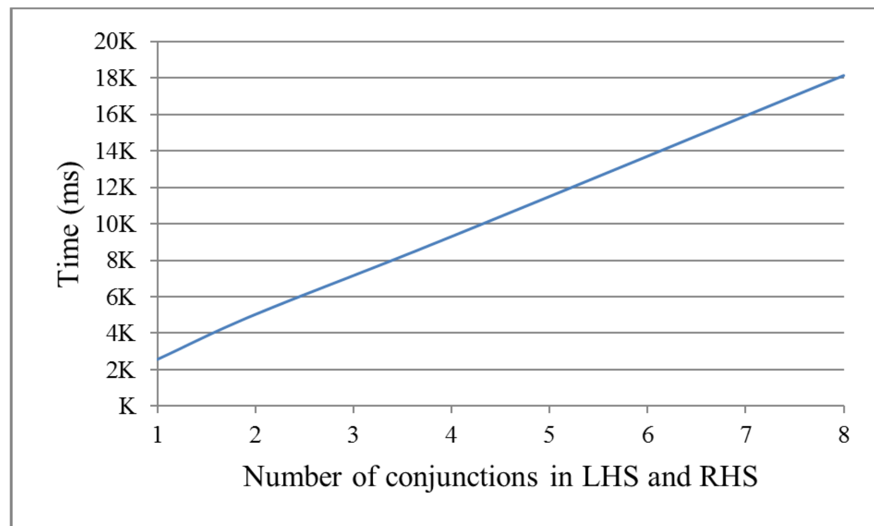

Figure 6.4: Trend of time w.r.t. the number of terms in rule LHS and RHS

6.5 Summary

In this chapter, we benchmarked our choreography engine introduced in the previous chapter. We found that in the presence of excessive transition rules firings the worst time needed for the choreography engine has the complexity of $O(n^2)$ with regard to size of the goal and web service choreography specifications i.e. total num-

ber of transition rules in them. If re-firing is prevented the complexity is reduced to *O(n)* which is an acceptable level of performance in many applications. Moreover, we showed that the complexity is *O(n)* with regard to transition rules' LHSs and RHSs complexity.

Chapter 7

CONCLUSION AND FUTURE WORK

In this work, we demonstrated how Flora-2 can be used as a convenient and expressive way to model semantic Web Services matching and semantic Web Service choreographing conforming to WSMO. We showed that semantic Web Service matching challenge can be handled very effectively by relying on the underlying Flora-2 reasoning engine and its meta-level capabilities.

For semantic Web Service choreography, we identified important weaknesses in the original ASM-based choreography execution algorithm for WSMO, which prevented it from being useful in a practical way, and improved it in order to remedy the identified weaknesses. The improved ASM-based choreography execution algorithm establishes the missing connection between the capability and interface components of WSMO. We used F-logic and Flora-2 to specify ASM-based choreographies of semantic Web Services in a concise and logical manner, and implemented a fully functional choreography execution engine based on our improved algorithm in Flora-2. The full functionality of Flora-2 and its underlying reasoning system is available for developing ontologies and writing transition rules in the choreography specification. To the best of our knowledge, this work is the first functional WSMO choreography implementation that fires rules in parallel, as required in the theory of ASMs, and models the ASM **if-then**(-**else**), **forall**, and **choose** rule types authentically, while enforcing access modes of concepts and relations. We demonstrated the workings of our algorithm

through several real-life examples, concerning different challenging scenarios, where both Web Service and goal choreographies were specified in our F-logic based syntax. We showed that our choreography engine implemented in Flora-2 can successfully choreograph the goals and Web Services specified in the examples. We also developed a visual tool that helps choreography engineers write specifications in a convenient manner, reducing the chance of mistakes in the specification.

Another important contribution of our work is that we proved for the first time the equivalence of evolving algebras (ASMs) and evolving ontologies (the basis of semantic choreography engines) through the definition of bi-directional mappings between them.

For future work, we are planning to develop our system through the addition of a grounding mechanism, as well as a mediation component. We also intend to identify and classify different types of general requests and general responses among software components and present them in the form of an ontology. Such a classification scheme will help in the development of accurate and commonly acceptable choreographic interactions.

REFERENCES

[1] Bnf and ebnf: What are they and how do they work? [Online]. Available: http://www.garshol.priv.no/download/text/bnf.html

[2] Concurrent transaction logic prototype. [Online]. Available: http://www.cs.toronto.edu/~bonner/ctr/

[3] Predicate calculus. [Online]. Available: https://www.britannica.com/topic/predicate-calculus

[4] Reference architecture foundation for service oriented architecture version 1.0. [Online]. Available: http://docs.oasis-open.org/soa-rm/soa-ra/v1.0/cs01/soa-ra-v1.0-cs01.html

[5] Shipwire: an ingram micro brand. [Online]. Available: https://www.shipwire.com/

[6] Tibco api exchange gateway. [Online]. Available: https://docs.tibco.com/pub/api-exchange-gateway/2.2.0/doc/html/index.html

[7] Web service choreography interface (wsci) 1.0. [Online]. Available: https://www.w3.org/TR/wsci/

[8] Wsmo use case "virtual travel agency" v0.1. [Online]. Available: http://www.wsmo.org/2004/d3/d3.2/b2c/20041004/

[9] “Web services glossary,” 2004. [Online]. Available: https://www.w3.org/TR/ws-gloss/

[10] “Wsml reasoning implementation,” 2004. [Online]. Available: www.wsmo.org/2004/d16/d16.2/v0.2/d16.2v0.2_20041220.pdf

[11] “Service ontologies and service description: An ontology for web service choreography, data, information and process integration with semantic web services,” 2006. [Online]. Available: sti-innsbruck.at/sites/default/files/D3.5.pdf

[12] *The Concepts of WSMO*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 63–81. [Online]. Available: https://doi.org/10.1007/978-3-540-34520-6_6

[13] “Wsmx documentation,” 2008. [Online]. Available: www.wsmx.org/papers/documentation/WSMXDocumentation.pdf

[14] “Xsb: Logic programming and deductive database system,” 2012. [Online]. Available: http://xsb.sourceforge.net

[15] (2013, 10) The json data interchange format. [Online]. Available: http://json.org/

[16] "Web services execution environment," 2013. [Online]. Available: https://sourceforge.net/projects/wsmx/

[17] "Ergo suite platform," 2014. [Online]. Available: http://coherentknowledge.com

[18] "A guide to flora-2 packages version 1.0 (cherimoya)," 2014. [Online]. Available: http://flora.sourceforge.net/documentation.html

[19] "Choreography engine implemented in flora-2," 2015. [Online]. Available: http://cmpe.emu.edu.tr/shahin/chore_engine/files.zip

[20] "Flora-2 (version 1.2.1 monstera deliciosa)," 2017. [Online]. Available: https://sourceforge.net/projects/flora/files/FLORA-2/

[21] S. Arroyo and A. Duke, "Sophie-a conceptual model for a semantic choreography framework," in *Proceedings of the Workshop on Semantic and Dynamic Web Process (SDWP 05) in conjunction with the International Conference on Web Services ICWS 05, EEUU*, 2005.

[22] S. M. Ataee and Z. Bayram, "A novel concise specification and efficient f-logic based matching of semantic web services in flora-2," in *Information Sciences and Systems 2015*. Springer, 2016, pp. 191–198.

[23] A. Barker, C. D. Walton, and D. Robertson, "Choreographing web services," *IEEE Transactions on Services Computing*, vol. 2, no. 2, pp. 152–166, 2009.

[24] A. J. Bonner and M. Kifer, “An overview of transaction logic,” *Theoretical Computer Science*, vol. 133, no. 2, pp. 205 – 265, 1994.

[25] E. Borger and R. F. Stark, *Abstract State Machines: A Method for High-Level System Design and Analysis*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2003.

[26] J. d. Bruijn, C. Bussler, J. Domingue, and D. Fensel, “Web service modeling ontology (wsmo),” 2016. [Online]. Available: http://www.w3.org/Submission/WSMO/

[27] L. Cabral, J. Domingue, S. Galizia, A. Gugliotta, V. Tanasescu, C. Pedrinaci, and B. Norton, *IRS-III: A Broker for Semantic Web Services Based Applications*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 201–214. [Online]. Available: http://dx.doi.org/10.1007/11926078_15

[28] W. Chen, M. Kifer, and D. S. Warren, *HiLog: A first order semantics for higher-order logic programming*. Morgan Kaufmann, 1989, pp. 1090–1115.

[29] W. Chen, M. Kifer, and D. S. Warren, “Hilog: A foundation for higher-order logic programming,” *The Journal of Logic Programming*, vol. 15, no. 3, pp. 187–230, 1993.

[30] E. Christensen, “Web services choreography description,” online, Nov. 2005, latest version. [Online]. Available: http://www.w3.org/TR/ws-cdl-10/

[31] J. de Bruijn, H. Lausen, A. Polleres, and D. Fensel, “The web service modeling language wsml: An overview,” in *Proceedings of the 3rd European Conference on The Semantic Web: Research and Applications*, ser. ESWC’06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 590–604. [Online]. Available: http://dx.doi.org/10.1007/11762256_43

[32] G. Decker, O. Kopp, F. Leymann, K. Pfitzner, and M. Weske, “Modeling service choreographies using bpmn and bpel4chor,” in *Proceedings of the 20th International Conference on Advanced Information Systems Engineering*, ser. CAiSE ’08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 79–93. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-69534-9_6

[33] G. Decker, O. Kopp, F. Leymann, K. Pfitzner, and M. Weske, “Modeling service choreographies using bpmn and bpel4chor,” in *Advanced Information Systems Engineering*, ser. Lecture Notes in Computer Science, Z. Bellahsène and M. Léonard, Eds. Springer Berlin Heidelberg, 2008, vol. 5074, pp. 79–93. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-69534-9_6

[34] M. Dimitrov, V. Momtchev, A. Simov, D. Ognyanoff, and M. Konstantinov, “wsmo4j programmers guide v. 2.0. 1,” *OntoText Lab.[consulted on April 24, 2012]. Available in: http://wsmo4j. sourceforge. net/doc/wsmo4j-prog-guide. pdf*, 2006.

[35] M. Dimitrov, A. Simov, V. Momtchev, and M. Konstantinov, “Wsmo studio–a

semantic web services modelling environment for wsmo," *The Semantic Web: Research and Applications*, pp. 749–758, 2007.

[36] M. Dimitrov, A. Simov, V. Momtchev, and D. Ognyanov, "Wsmo studio-an integrated service environment for wsmo," in *Proc. of the 2nd WSMO Impl. Workshop, Innsbruck, Austria*. Citeseer, 2005.

[37] J. Domingue, L. Cabral, F. Hakimpour, D. Sell, and E. Motta, "Irs iii: A platform and infrastructure for creating wsmo based semantic web services," 2004.

[38] J. Domingue, S. Galizia, and L. Cabral, *Choreography in IRS-III – Coping with Heterogeneous Interaction Patterns in Web Services*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 171–185. [Online]. Available: http://dx.doi.org/10.1007/11574620_15

[39] J. Domingue, S. Galizia, and L. Cabral, "The choreography model for irs-iii," in *System Sciences, 2006. HICSS'06. Proceedings of the 39th Annual Hawaii International Conference on*, vol. 3. IEEE, 2006, pp. 62c–62c.

[40] J. Domingue, E. Motta, and O. Corcho, "Knowledge modelling in webonto and ocml - a user guide," 11 1999.

[41] L. Engler, "Bpelgold choreography on the service bus," PhD Dissertaion, Univesity of Stuttgart, D-70569, Stuttgart, October 2009, cR-Classification H.4.1, C.2.4, D.2.11.

[42] D. Fensel, M. Kerrigan, and M. Zaremba, "Implementing semantic web services," *The SESA Framework*, 2008.

[43] D. Fensel, H. Lausen, A. Polleres, J. de Bruijn, M. Stollberg, D. Roman, and J. Domingue, *Enabling Semantic Web Services: The Web Service Modeling Ontology*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[44] G. Flouris, D. Plexousakis, and G. Antoniou, *Evolving Ontology Evolution*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 14–29. [Online]. Available: https://doi.org/10.1007/11611257_2

[45] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 3rd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[46] J. H. Gallier, *Logic for computer science: foundations of automatic theorem proving*. Courier Dover Publications, 2015.

[47] Y. Gurevich, "Specification and validation methods," E. Börger, Ed. New York, NY, USA: Oxford University Press, Inc., 1995, ch. Evolving Algebras 1993: Lipari Guide, pp. 9–36. [Online]. Available: http://dl.acm.org/citation.cfm?id=233976.233979

[48] Y. Gurevich, "Sequential abstract-state machines capture sequential algorithms," *ACM Transactions on Computational Logic*, vol. 1, no. 1, pp. 77–111, 2000.

[49] Y. Gurevich, *Abstract State Machines: An Overview of the Project*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 6–13. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-24627-5-2

[50] A. Haller, E. Cimpian, A. Mocan, E. Oren, and C. Bussler, "Wsmx - a semantic service-oriented architecture," in *In Proceedings of the International Conference on Web Service (ICWS 2005*, 2005, pp. 321–328.

[51] A. Haller, E. Cimpian, A. Mocan, E. Oren, and C. Bussler, "Wsmx-a semantic service-oriented architecture," in *Web Services, 2005. ICWS 2005. Proceedings. 2005 IEEE International Conference on*. IEEE, 2005, pp. 321–328.

[52] A. Haller, J. Scicluna, and T. Haselwanter, "D13.9v0.1 wsmx choreography," 2005. [Online]. Available: http://www.w3.org/Submission/WSMO/

[53] H. Henderson, *Finite State Machine*, revised ed. Facts on File, 2008, pp. 195–196.

[54] J. K. Huggins and C. Wallace, "An abstract state machine primer," Tech. Rep., 2002.

[55] U. Keller, R. Lara, A. Polleres, I. Toma, M. Kifer, and D. Fensel, "Wsmo web service discovery," *WSML Working Draft D*, vol. 5, 2004.

[56] M. Kiefer, *Flora-2*. Stony Brook University, 2015.

[57] M. Kifer, "Concurrency and communication in transaction logic," in *Logic Programming: Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming*. MIT Press, 1996, p. 142.

[58] M. Kifer, *Rules and Ontologies in F-Logic*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 22–34. [Online]. Available: http://dx.doi.org/10.1007/11526988-2

[59] M. Kifer, R. Lara, A. Polleres, C. Zhao, U. Keller, H. Lausen, and D. Fensel, "A logical framework for web service discovery," in *ISWC 2004 Workshop on Semantic Web Services: Preparing to Meet the World of Business Applications*, vol. 119. Hiroshima, J apan, 2004.

[60] M. Kifer and G. Lausen, "F-logic: A higher-order language for reasoning about objects, inheritance, and scheme," in *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '89. New York, NY, USA: ACM, 1989, pp. 134–146. [Online]. Available: http://doi.acm.org/10.1145/67544.66939

[61] M. Kifer, G. Lausen, and J. Wu, "Logical foundations of object-oriented and frame-based languages," *J. ACM*, vol. 42, no. 4, pp. 741–843, Jul. 1995. [Online]. Available: http://doi.acm.org/10.1145/210332.210335

[62] M. Kifer, G. Lausen, and J. Wu, "Logical foundations of object-oriented and frame-based languages," *Journal of the ACM (JACM)*, vol. 42, no. 4, pp. 741–

843, 1995.

[63] R. Krummenacher, D. Winkler, and A. Marte, “Wsml2reasoner-a comprehensive reasoning framework for the semantic web,” in *Proceedings of the 2010 International Conference on Posters & Demonstrations Track-Volume 658*. CEUR-WS. org, 2010, pp. 125–128.

[64] U. Küster, H. Lausen, and B. König-Ries, “Evaluation of semantic service discovery—a survey and directions for future research,” *Emerging Web Services Technology, Volume II*, pp. 41–58, 2008.

[65] N. Lanzanasto, S. Komazec, and I. Toma, “Reasoning over real time data streams,” *ENVISION Deliverable D*, vol. 4, pp. 8–1, 2012.

[66] R. Lara, A. Polleres, H. Lausen, D. Roman, J. de Bruijn, and D. Fensel, “A conceptual comparison between wsmo and owl-s,” *WSMO Final Draft D*, vol. 4, p. 44, 2005.

[67] M. Malaimalavathani and R. Gowri, “A survey on semantic web service discovery,” in *2013 International Conference on Information Communication and Embedded Systems (ICICES)*, Feb 2013, pp. 222–225.

[68] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne *et al.*, “Owl-s: Semantic markup for web services,” *W3C member submission*, vol. 22, pp. 2007–04, 2004.

[69] J. McCarthy and M. I. Levin, *LISP 1.5 programmer's manual*. MIT press, 1965.

[70] S. A. McIlraith, T. C. Son, and H. Zeng, "Semantic web services," *IEEE Intelligent Systems*, vol. 16, no. 2, pp. 46–53, Mar. 2001. [Online]. Available: http://dx.doi.org/10.1109/5254.920599

[71] S. Mehdipour Ataee and Z. Bayram, "An improved abstract state machine based choreography specification and execution algorithm for semantic web services," *Scientific Programming*.

[72] B. Motik and R. Studer, "Kaon2–a scalable reasoning tool for the semantic web," in *Proceedings of the 2nd European Semantic Web Conference (ESWC'05), Heraklion, Greece*, vol. 17, 2005.

[73] A. G. Neiat, M. Mohsenzadeh, R. Forsati, and A. M. Rahmani, "An agent-based semantic web service discovery framework," in *Computer Modeling and Simulation, 2009. ICCMS'09. International Conference on*. IEEE, 2009, pp. 194–198.

[74] L. D. Ngan, M. Kirchberg, and R. Kanagasabai, "Review of semantic web service discovery methods," in *2010 6th World Congress on Services*, July 2010, pp. 176–177.

[75] U. Nilsson and J. Maluszynski, *Logic, Programming, and PROLOG*, 2nd ed. New York, NY, USA: John Wiley & Sons, Inc., 1995.

[76] ontoprise GmbH, *How to write F-Logic Programs*. Ontoprise, 2007.

[77] C. Peltz, "Web services orchestration and choreography," *Computer*, vol. 36, no. 10, pp. 46–52, 2003.

[78] S. R. Ponnekanti and A. Fox, "Sword: A developer toolkit for web service composition," in *Proc. of the Eleventh International World Wide Web Conference, Honolulu, HI*, vol. 45, 2002.

[79] L. Richardson and S. Ruby, *Restful Web Services*, 1st ed. O'Reilly, 2007.

[80] D. Roman, J. De Bruijn, A. Mocan, H. Lausen, J. Domingue, C. Bussler, and D. Fensel, "Www: Wsmo, wsml, and wsmx in a nutshell," *ASWC*, vol. 4185, pp. 516–522, 2006.

[81] D. Roman and M. Kifer, "Reasoning about the behavior of semantic web services with concurrent transaction logic," in *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 2007, pp. 627–638.

[82] D. Roman and M. Kifer, "Semantic web service choreography: Contracting and enactment," in *International semantic web conference*. Springer, 2008, pp. 550–566.

[83] D. Roman, J. Scicluna, and C. Feier, "D14v01. choreography in wsmo," 2016. [Online]. Available: http://www.wsmo.org/TR/d14/v0.1/

[84] D. Roman, J. Scicluna, and J. Nitzsche, “D14v0.3. ontology-based choreography of wsmo services,” 2016. [Online]. Available: http://www.wsmo.org/TR/d14/v0.4/

[85] O. Shafiq, M. Moran, E. Cimpian, A. Mocan, M. Zaremba, and D. Fensel, “Investigating semantic web service execution environments: a comparison between wsmx and owl-s tools,” in *Internet and Web Applications and Services, 2007. ICIW’07. Second International Conference on*. IEEE, 2007, pp. 31–31.

[86] A. Sirbu, I. Toma, and D. Roman, “A logic-based approach for service discovery with composition support,” *Emerging Web Services Technology*, pp. 101–116, 2007.

[87] R. F. Stärk, J. Schmid, and E. Börger, *Java and the Java virtual machine: definition, verification, validation*. Springer Science & Business Media, 2012.

[88] G. Steele, *Common LISP: the language*. Elsevier, 1990.

[89] M. Stollberg, “Reasoning tasks and mediation on choreography and orchestration in wsmo,” in *Proceedings of the 2nd International WSMO Implementation Workshop (WIW 2005), Innsbruck, Austria*, 2005.

[90] M. Stollberg, “Reasoning tasks and mediation on choreography and orchestration in wsmo,” in *In Proceedings of the 2nd International WSMO Implementation Workshop (WIW 2005*, 2005.

[91] A. Vaisman, *An Introduction to Business Process Modeling*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 29–61. [Online]. Available: https://doi.org/10.1007/978-3-642-36318-4_2

[92] J. M. Zaha, A. Barros, M. Dumas, and A. ter Hofstede, "Let's dance: A language for service behavior modeling," in *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, ser. Lecture Notes in Computer Science, R. Meersman and Z. Tari, Eds. Springer Berlin Heidelberg, 2006, vol. 4275, pp. 145–162. [Online]. Available: http://dx.doi.org/10.1007/11914853_10

[93] R. Zaharia, L. Vasiliu, and C. Bădică, "Semi-automatic composition of geospatial web services using jboss rules," in *International Workshop on Rules and Rule Markup Languages for the Semantic Web*. Springer, 2008, pp. 166–173.

[94] M. Zaremba, M. Moran, T. Haselwanter, and H.-K. Lee, "Wsmx architecture," *D13. 4v0*, vol. 2, 2005.

APPENDICES

Appendix A: Visual editor for Flora-2 based SWS specifications (VSChor)

Visual Semantic Choreography (VSChor) is a visual software environment which we developed to facilitate designing and deploying goals and web services with Flora-2 specifications. Choreography designers can define concepts, frames, predicates and specify the structures of goals and web services by filling-out the prepared forms and automatically generate Flora-2 specifications that are ready to run on the developed choreography engine. The software also embeds the engine code and related libraries in a deployment folder. The user can test the choreography by just running a single batch file, provided the Flora-2 system is already installed on the local platform. The

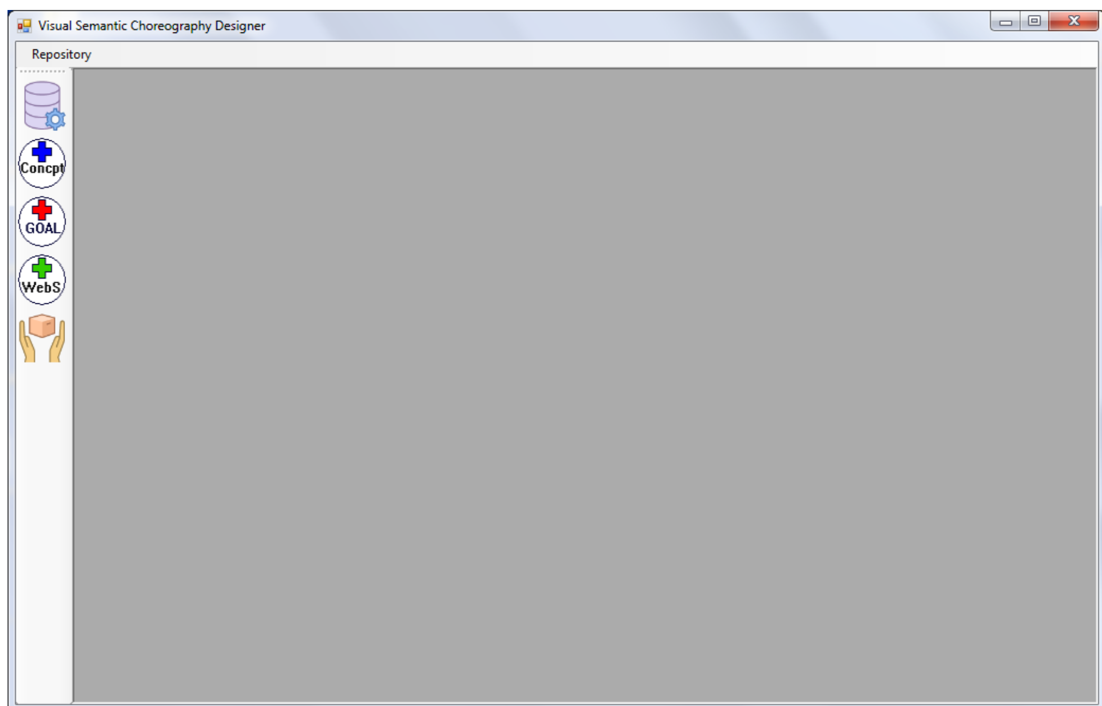

Figure 7.1: Main entrance form of VSChor

main window of the program is shown in Figure 7.1. The left tool box contains five icons: from top to down, Repository, Add Concept, Add Goal, Add Web Service, and Deploy, respectively. The repository button is used to show the currently registered concepts, frames, predicates, goals, and Web Services (Figure 7.2). It also allows the user to change the definitions of goals and Web Services. Repositories can be

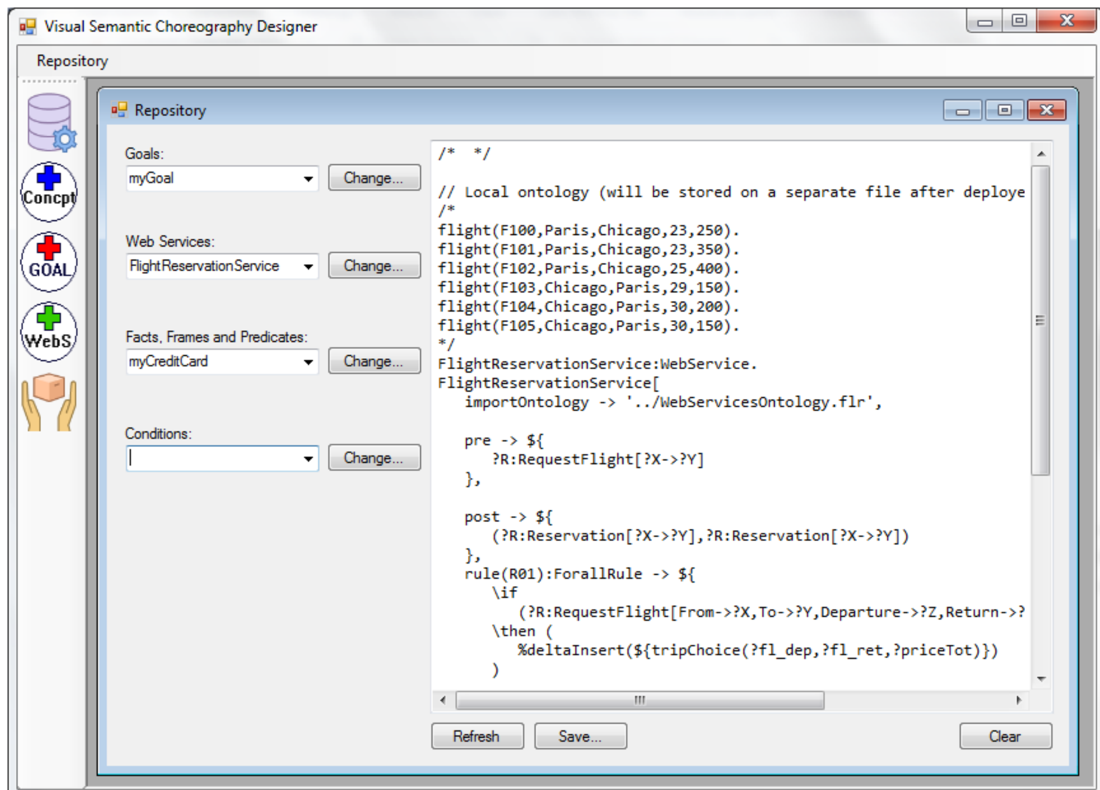

Figure 7.2: Repository form

saved and loaded by the Repository menu located at the top-left corner of the main form (Figure 7.3). Concepts can be defined by Add Concept form (Figure 7.4). The user can enter concept attributes and their types by filling out the available grid and determine the proper mode type by using the available combobox.

Goals and Web Services can use instances of the concepts in their specifications. The Add Goal and Add Service forms are essentially the same, so here we only illustrate Add Goal form, depicted in Figure 7.5.

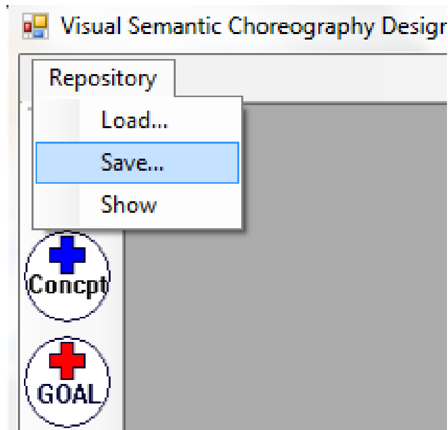

Figure 7.3: Loading and saving repositories

The Add Goal form contains places for the name, description, local ontology, capabilities, pre and post conditions, and transition rules. The Semantic description box shows the current textual description of the goal in Flora-2. The Add relation and Add frame buttons let the user define new (or use already defined) relations and rules respectively (Figure 7.6). Post-condition of a goal may contain a complex logic expression consisting of the and, or, and not operators, as well as frames and predicates. VSChor lets the user create a logic tree representing the logic expression of the goal post-condition (Figure 7.5. Post-condition view).

Transition rules are created through another form (Figure 7.7), accessible by clicking on the Add rule button on Add Goal and Add Web Service forms. The user can specify the name, rule type (either Forall or Choose), antecedent (left-hand side), and consequent (right-hand side) of the rules. Similar to the goal post-condition, the antecedent can contain a logic expression of any complexity. The consequent can consist of a set of actions, which should be one of `%deltaInsert`, `%deltaDelete`, or `%deltaUpdate`.

At any stage the user can modify the contents of goals (Web Services) using the change semantic goal (Web Service) form, accessible through the repository form.

After the choreography design is completed, a deployment package is generated by clicking on the deployment button (Figure 7.1. last button).

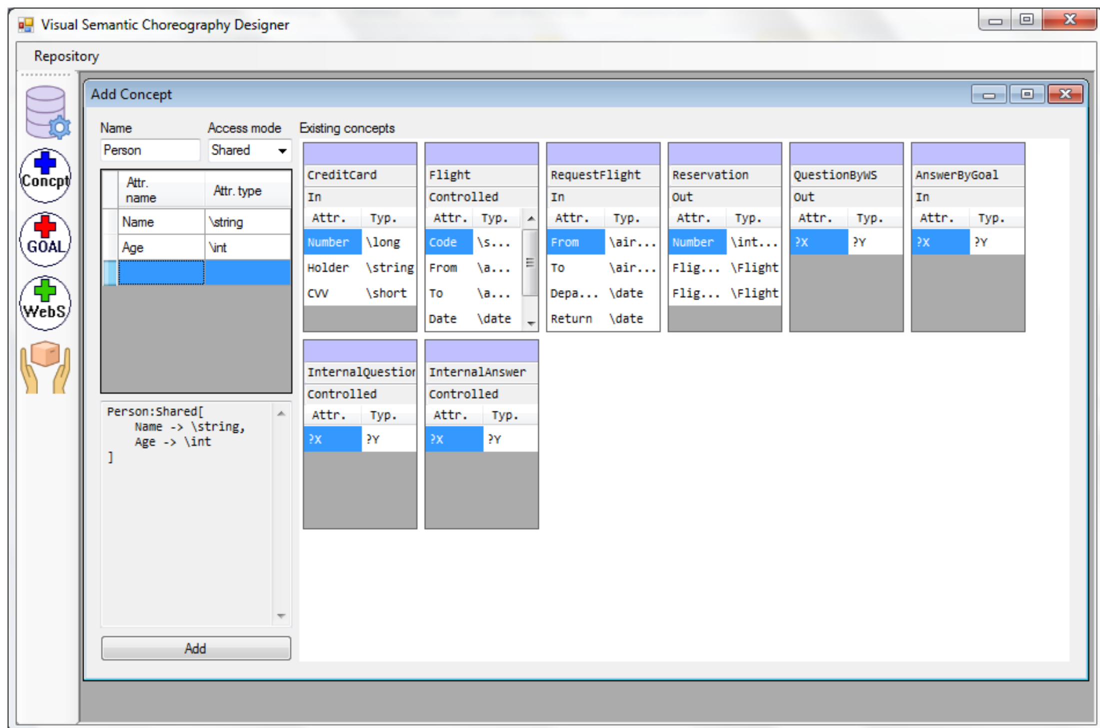

Figure 7.4: Add Concept form

The deployment package contains all the files related to goals, Web Services, common ontology, choreography engine, function libraries, as well as a batch file called “Run.bat”(Figure 7.8). The batch file allows the user to test the choreography simply by double clicking on it.

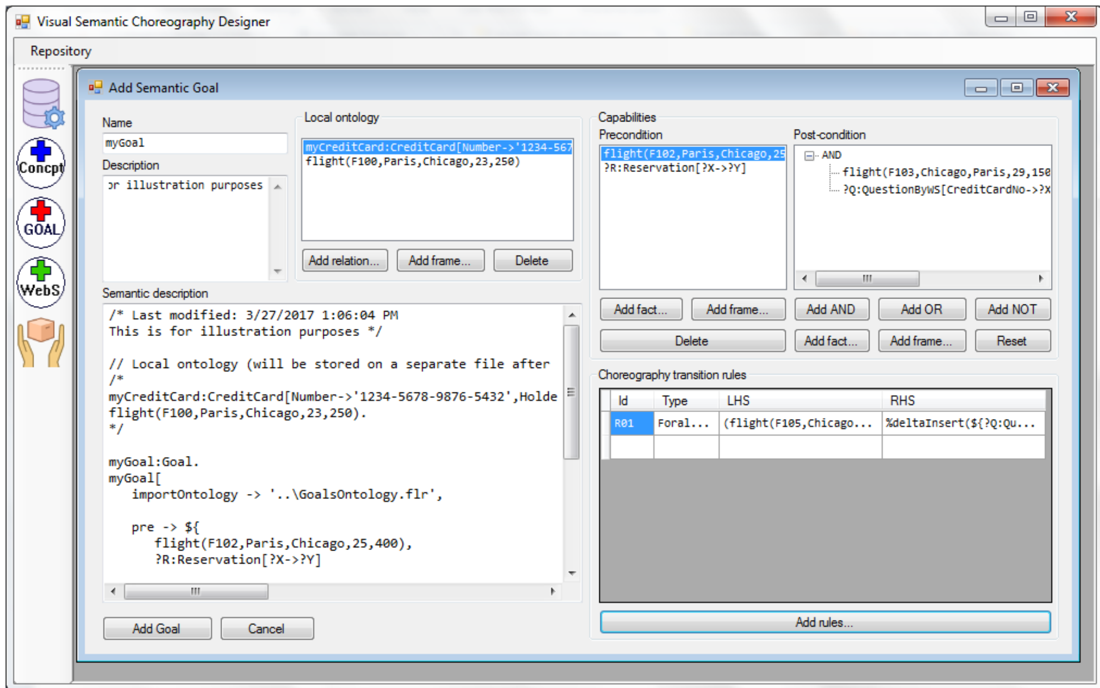

Figure 7.5: Add Goal form

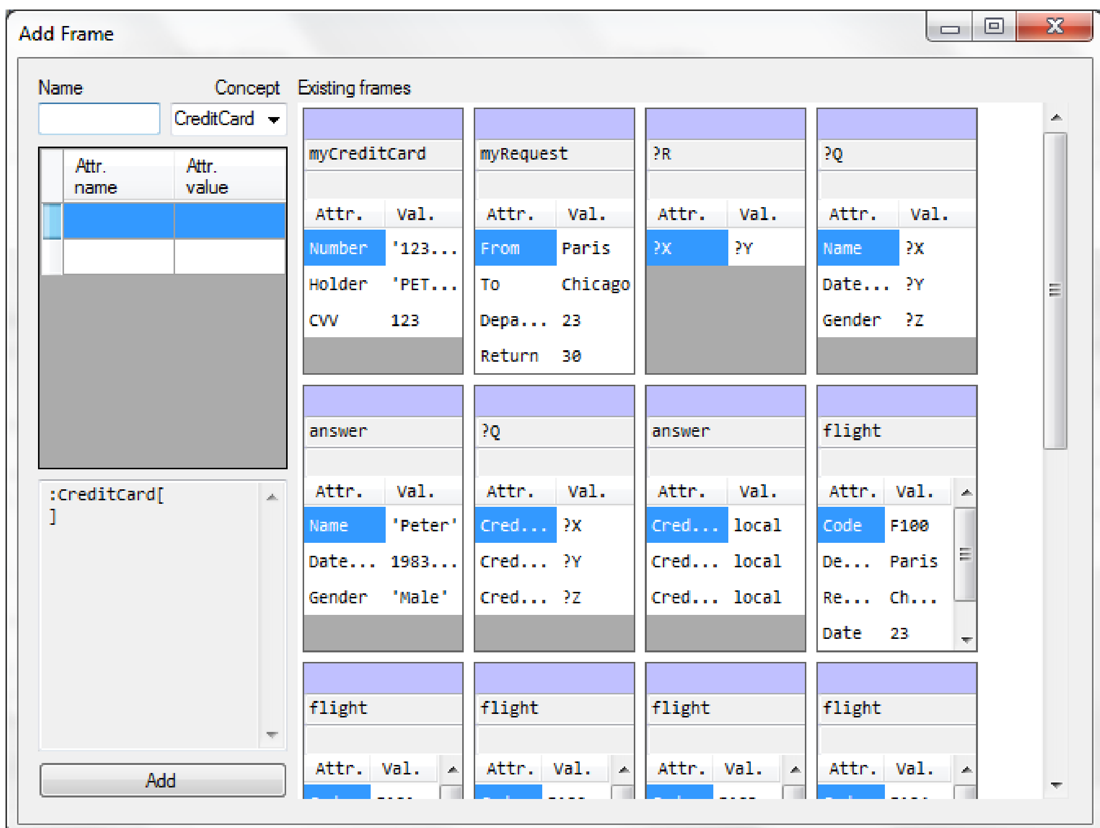

Figure 7.6: Add Frame form

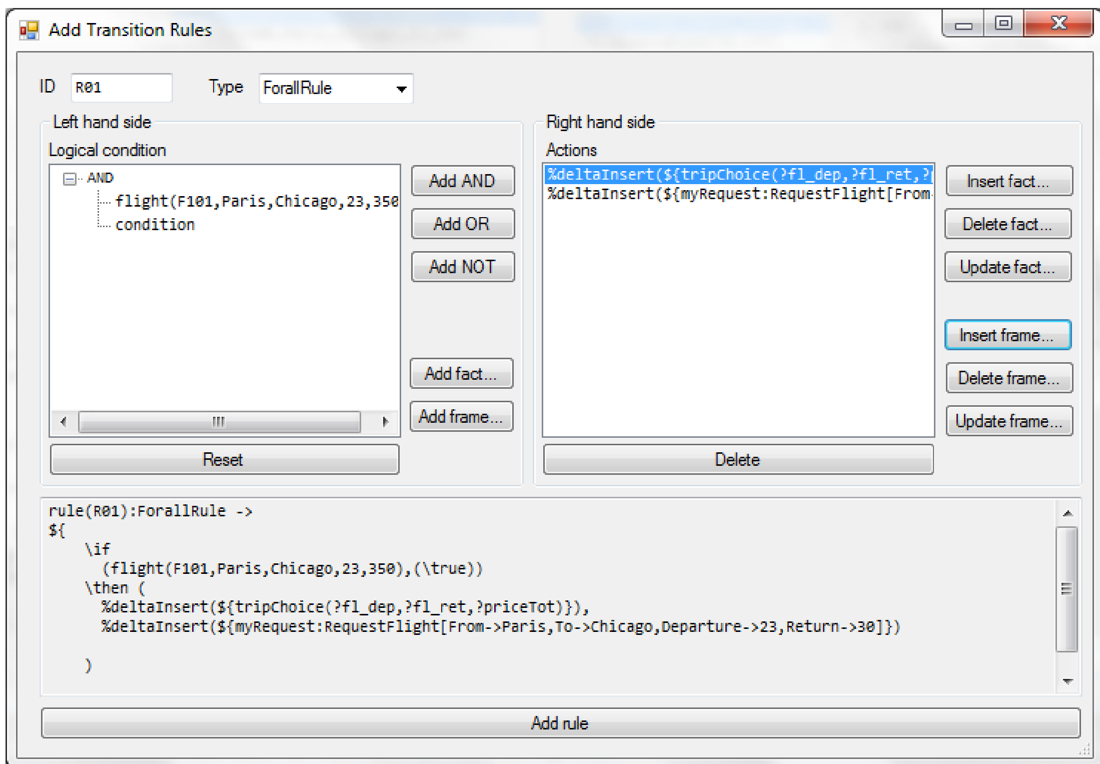

Figure 7.7: Add transition rule form

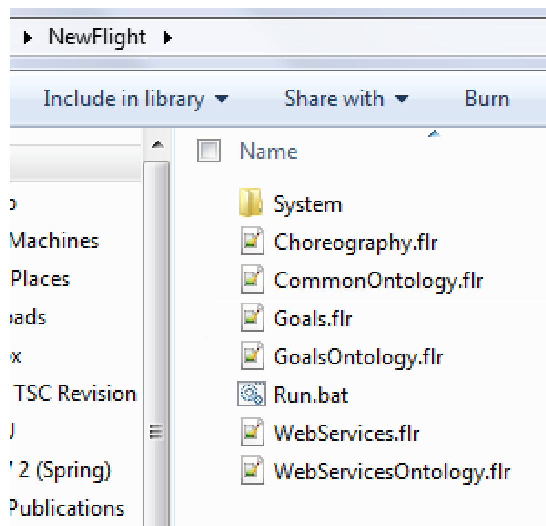

Figure 7.8: Deployment folder

Appendix B: E-BNF grammar for Flora-2 goal and Web Service specifications

Here, we present the current grammar of goal and Web Service specifications written in E-BNF [1] (Table 7.1).

Listing 7.1: E-BNF grammar for goal and Web Service specifications

```
1  <webservice>   ::= <oid> : WebService '['
2                    importOntology -> <symbolString> ,
3                    capability -> $ '{'
4                    pre -> $ '{' <condition> '}' ,
5                    post  ->  $  '{' <andcondition> '}'
6                    '}'
7                    [, <wstransitionrules> ] ']' .
8
9  <goal>         ::= <oid> : Goal '['
10                   importOntology -> <symbolString> ,
11                   capability -> $ '{'
12                   pre -> $ '{' <andcondition> '}' ,
13                   post  ->  $  '{' <condition> '}'
14                   '}'
15                   [, <gtransitionrules> ] ']' .
16
17 <andcondition> ::= <frame> | <predicate> ) {, ( <frame> | <
       predicate> ) }
18
19 <condition>    ::= <condition> ; <term>  | <term>
20
21 <term>         ::= <term> , <factor> | <factor>
22
23 <factor>       ::= <frame> | <predicate> | '(' <condition> ')' |
       '(' \+ <condition> ')'
24
25 <wstransitionrules> ::= <wsrule> {, <wsrule> }
26
27 <gtransitionrules>  ::= <grule> {, <grule> }
28
29 <wsrule> ::= wsRule ( <oid> ) : <ruletype> -> $ '{'
30              \if <condition>
31              \then '(' <actions> ')'
32              [ \else '(' <actions> ')' ] '}'
33
34 <grule> ::= gRule ( <oid> ) : <ruletype> -> $ '{'
35             \if <condition>
36             \then '(' <actions> ') '
37             [ \else '(' <actions> ')' ] '}'
38
39 <ruletype> ::= ForallRule | ChooseRule
40
41 <actions>  ::= <action> { , <action> }
42
43 <action>   ::= %deltaInsert '(' <frame> ')' |
44                %deltaInsert '(' <predicate> ')' |
```

```
45                 %deltaDelete '(' <restrictedframe> ')' |
46                 %deltaDelete '(' <predicate> ')' |
47                 %deltaUpdate '(' <objname> , <attrname> , <f_term>
                       , <f_term> ')'
48
49  <predicate> ::= <predname> [ '(' [ <f_term> {, <f_term> } ] ')' ]
50
51  <frame>      ::= <objname> : <concept> '[' [ <attribute-value-
       pairs> ] ']'
52
53  <attribute-value-pairs> ::= <attrname> -> <f_term> { , <attrname>
        -> <f_term> }
54
55  <restrictedframe> ::= <objname> '[' [ <attribute-value-pairs> ]
       ']'
56
57  <concept>          ::= <oid>
58
59  <attrname>         ::= <oid>
60
61  <objname>          ::= <oid>
62
63  <predname>         ::= <oid>
64
65  <symbolString>     ::= '''<string>'''
66
67  <string>           ::= Any sequence of characters
68
69  <oid>              ::= Any valid Flora-2 object identifier
70
71  <f_term>           ::= Any valid Flora-2 term
```

Appendix C: Converting JSON to Flora-2

Currently, REST (REpresentational State Transfer) is the most popular architecture in the development of Web Services. In RESTful architectures, message passing and data transfer is done via HTTP over the network. Clients use methods such as GET, POST, and PUT to trigger actions or to retrieve resources held by Web Services. Web Services, on the other hand, respond to these methods with HTTP messages containing formatted (and possibly annotated) information. The dominant formats in use are XML (eXtensible Markup Language) and JSON (JavaScript Object Notation). Between these two, JSON is the preferred format in REST because of its simplicity and readability. However, as it is a minimal type-free data format it does not support semantic annotations. It should be kept in mind that XML and JSON are not equivalent because the former is a language with schema, types and links but the latter is just a data presentation format. JSON-LD (JSON for Linked-Data) is a typed version of JSON and has the potential to be used in semantic based systems, but it is not integrated with the most of the popular available Web Services yet. In this appendix we present a scheme to map JSON context to an equivalent Flora-2 context. This scheme paves the way for our Flora-2 choreography solution to interact with the currently available REST APIs.

JSON supports only two types of structure, namely objects and arrays. Objects are anonymous and contain comma separated key-value pairs. Keys are always strings and values can be other objects, string/Boolean/numerical literals, or arrays. Arrays contain series of objects and can be nested as well.

In Flora-2, frames model objects. Unlike JSON, frames have names in the form

of object identifiers and can be instantiated from a concept (class). Frames are composed of attribute-value pairs which are similar to JSON key-value pairs. But because objects are anonymous in JSON it is not straightforward to directly map key-value pairs in JSON to attribute-value pairs in Flora-2. Another consideration is that in JSON, objects are referenced by their relative position and path in the whole structure and there is no direct pointer to a specific data, but in Flora-2 frames are decomposed and stored in form of predicates. Data in Flora-2 is retrieved by unification, so accessing data is fundamentally different from JSON. Taking into account the above mentioned issues, below we present a mapping scheme between JSON and Flora-2.

The top-level entity in JSON can be either an object or an array. We can show this by frames `json[content->object(obj_id)]` or `json[content->array(arr_id)]`.

In JSON, key-value pairs belonging to the same object are wrapped together inside the object. We can show this binding by combining predicate and frame notations in Flora-2. For example, the attribute-value pairs `object(id_1,a)->1` and `object(id_1,b)->2` tell that both attributes a and b belong to the object with id `id_1` and their values are 1 and 2 respectively. In this way, the query `object(id_1,?X)->?Y` returns all attribute-value pairs belonging to the object with id `id_1`. Moreover, `id_1` can be used as a direct reference to the object; in contrast to JSON wherein objects are anonymous.

The elements of an array in JSON are referenced by their position in a comma separated list. To show JSON lists in Flora-2 we can use again predicate and frame notations. For example, the attribute-value pair `array(id_1, 2)->a` says that the second element in array `id_1` has the value `a`.

An array which itself is a value of an attribute can be represented in Flora-2

as the predicate array whose parameter is the (generated) id of the array. For example the JSON object `"someArray": [10,20]` can be represented in Flora-2 by the facts `object(id_1, "someArray")->array(id_2)` , `array(id_2, 1)->10`, and `array(id_2, 2)->20` where `id_1` and `id_2` are automatically generated Flora-2 object identifiers. Using the Flora-2 reasoner, one can query such composite structures in a simple manner. For example, a query such as `someFrame:someConcept[object(id_1, a)->array(id_2), array(id_2, ?X)->?Y]` returns all the elements' index-value pairs of array `id_2` which is in turn the value of attribute a of object `id_1`.

Table 7.2: An example of JSON to Flora-2 conversion

JSON

```
[
 {
  "firstName": "John",
  "lastName" : "doe",
  "age"      : 26,
  "address"  :
   {
    "street" : "Naist",
    "city"   : "Nara",
    "zipCode": "0192"
   },
  "phoneNumbers": [
   {
    "type"  : "iPhone",
    "number":
        "1-567-8888"
   },
   {
    "type"  : "home",
    "number":
        "1-567-8910"
   }
  ]
 }
]
```

Flora-2

```
json[content -> array(arr_01)].

array(arr_01,1) -> object(obj_01).
object(obj_01,"firstName") -> "John".
object(obj_01,"lastName") -> "doe".
object(obj_01,"age") -> 26.
object(obj_01,"address") ->
    object(obj_02).

object(obj_02, "street") -> "Naist".
object(obj_02, "city") -> "Nara".
object(obj_02, "postalCode") -> "0192".

array(arr_01,2) -> object(obj_03).
object(obj_03,"phoneNumbers") ->
    array(arr_02).

array(arr_02,1) -> object(obj_04).
object(obj_04,"type") -> "iPhone".
object(obj_04,"number") ->
    "1-567-8888".

array(arr_02,2) -> object(obj_05).
object(obj_05,"type") -> "home".
object(obj_05,"number") ->
    "1-567-8910".
```

Table 7.2 depicts an example of REST response message, both in JSON and its

equivalent Flora-2 notations. It is obvious that data presented in Flora-2 can be simply queried and reasoned about, and these are not possible with JSON.

Appendix D: Choreography engine predicates list

Predicate	Defined in	Functionality
%checkAllRHS (?gOrWS,?List)	Library.flr	Picks each of the collected RHSs in the ?List and passes it to %checkRHS. It is used to see if there is any usage of membership operators in deltaDelete actions.
%checkExistanceOfColon (?gOrWS,?Action)	Library.flr	If the specified action (?Action) is deltaDelete, then it checks whether there is any membership operator used in the action subject.
%checkMembershipOperator (?Obj)	Library.flr	Converts the reified Flora-2 object (?Obj) into string and then passes its characters one-by-one to %noColonExists.
%checkModule (?M)	Utility.flr	Checks existence of module ?M, if not creates it.
%checkRHS (?gOrWS,?P)	Library.flr	If ?P is a pair it decomposes it, otherwise it sends ?P to %checkExistanceOfColon.
%checkUsageOfMembership OperatorInDeltaDelete (?gOrWS)	Library.flr	Prepares the list of all RHSs belonging to ?gOrWs which can be either a goal or a web service.
%contained (?X, ?A)	Utility.flr	Checks whether ?X exists in ?A. ?A is conjunction of literals.
%contradictory (?WM, ?DeltaWM)	Choreography.flr	Checks whether there are any contradictory actions who are pending in Delta Working Memory. WM and DeltaWM are in ?WM and ?DeltaWM respectively.
%convertReifiedObject Module(?obj1,?M1,?M2,?obj2)	Library.flr	Copies the reified object ?obj1 into object ?obj2 with the module name changed from ?M1 to ?M2.
%copyReifiedObjectInto Module(?obj1,?M1,?M2,?obj2)	Library.flr	Makes a copy of the reified object ?obj1 in module ?M1 into ?obj2 and inserts it in ?M2.
%debug (?X)	Utility.flr	Toggles the debug mode.
%deltaDelete (?obj)	Library.flr	Inserts deletion of an object into delta working memory.
%deltaInsert (?obj)	Library.flr	Inserts insertion of an object into delta working memory
%deltaMakesAChange (?WM, ?DeltaWM)	Choreography.flr	Checks whether the pending actions in delta working memory make a new state for choreography

Predicate	Defined in	Functionality
%deltaUpdate (?objOld, ?objNew)	Library.flr	Inserts updating of an object into delta working memory
%eraseModule (?M)	Utility.flr	Deletes the content of the specified module.
%extractConcepts (?Str, ?LstIn, ?LstOut)	Library.flr	Finds and extracts the concept terms in the string ?Str.
%extractPredicates (?Str, ?LstIn, ?LstOut)	Library.flr	Finds and extracts the terms which could be object names or predicate names in the string ?Str
%filterOutPredicates (?List, ?LstIn, ?LstOut)	Library.flr	Among list of object names and predicate names represented by ?List ,it extracts the predicate names.
%giveElementAt (?L,?n,?elementAt)	Utility.flr	Returns the ?nth element of the list ?L in ?elementAt.
%importOntology (?X,?module)	Library.flr	Loads the specified ontology into the specified module.
%initializations	Utility.flr	Initializes the choreography engine parameters (currently initialize the RNG seed).
%insertGoalPre	Library.flr	Inserts the precondition of goal ?goal into the ?M module.
%invoke (WEBSERVICE,?X)	Choreography.flr	Fires the forall rules of a web service or a goal.
%invokeChoose (WEBSERVICE,?X)	Choreography.flr	Fires the choose rules of a web service or a goal.
%isNotFrame ([?H|?T])	Library.flr	Determines if a string represents a Flora-2 frame.
%makeFileAddress (?dir,?name,?fileAddress)	Utility.flr	Concatenates the directory path and the file name to make a complete file address.
%mergeDeltaIntoWM	Choreography.flr	Does actual actions which were pending in delta working memory and makes a new state of working memory.
%noColonExists ([])	Library.flr	Checks whether there is a ':' in the passed string.
%pause()	Utility.flr	Stops the reasoner to let the user to enter something.
%prepareModule (?module)	Utility.flr	Creates a new module if it does not exist.
%preProcessCheckings (?goal,?WS)	Library.flr	Checks the access modes in the goal precondition and the transition rules.

Predicate	Defined in	Functionality
%prove (?X)	Library.flr	Tries to prove any logical expression.
%proveGoalPost (?goal)	Library.flr	Tries to prove the goal post condition.
%rand (?L,?U,?R)	Utility.flr	Generates a random integer number in the specified range.
%readTheTerm(?Str,?termIn, ?termOut, ?remainder)	Utility.flr	Extracts terms after specific characters in the string ?Str one at a time.
%reformatToString (?In, ?Str)	Utility.flr	Converts a Flora-2 object definition into a string.
%removeParenthesis (?Str, ?In, ?Out)	Utility.flr	Scans a string and reads the characters until reaching an opening parenthesis.
%replaceAll (?Str,?Pattern, ?Replace,?NewStr)	Utility.flr	Replaces all the substrings matching ?Pattern with the string ?Replace and returns the new string by ?NewStr.
%runChoreography (?goal, ?WS)	Choreography.flr	The main predicate of choreography execution.
%runGoalRules (?goal)	Choreography.flr	Runs the transition rules of the goal.
%runWsRules (?WS)	Choreography.flr	Runs the transition rules of the web service.
%showModule (?M)	Utility.flr	Shows the content of a module in the output.
%start (?goal,?WS)	Choreography.flr	The predicate which should be called to begin the choreography execution engine.
%watch (?X)	Utility.flr	Prints the parameter represented by ?X in the output.
%watchln (?X)	Utility.flr	Prints the parameter represented by ?X and a new line character in the output.
%check (?gOrWs,?X)	ModeChecking.flr	Checks if the LHS and RHS of a rule (belonging to goal or web service) are using correct access modes.
%checkAll (?gOrWs,?L)	ModeChecking.flr	Extracts the rules of goal and web service in the list ?L and passes them to %check one-by-one.
%checkAllFramesModes (?gOrWS, ?reOrWr,?L)	ModeChecking.flr	Extracts the concept names from the list ?L and passes them to %checkFrameMode one-by-one.

Appendix E: Choreography engine source codes

Main Predicates (Choreography.flr)

Listing 7.2: Main Predicates

```
1  /*** Generated by Visual Semantic Choreography on 3/30/2017
      8:17:15 PM ***/
2
3  #include "Utility.flr"
4  #include "Library.flr"
5  #include "ModeChecking.flr"
6  #include "../CommonOntology.flr"
7  #include "../WebServices.flr"
8  #include "../Goals.flr"
9
10 /*---------------------------------------------------*/
11 // (1) start
12 // (2) runChoreography
13 // (3) runWsRules
14 // (4) runGoalRules
15 // (5) invoke
16 // (6) invokeChoose
17 // (7) mergeDeltaIntoWM
18 // (8) deltaMakesAChange
19 // (9) contradictory
20 /*---------------------------------------------------*/
21
22 /* (1) start*/
23 %start(?goal,?WS, ?Result) :-
24         %debug(on),
25         %initializations,
26         // %preProcessCheckings(?goal,?WS),
27
28         %prepareModule(WM),
29         %prepareModule(DeltaWM),
30         %prepareModule(reportM),
31
32         %importOntology(?goal,WM),
33         %importOntology(?WS,WM),
34
35         %insertGoalPre(?goal,WM),
36     %runChoreography(?goal,?WS, ?Result).
37 /*---------------------------------------------------*/
38 /* (2) runChoreography */
39 /* The base case */
40 %runChoreography(?goal, ?WS, ?Result) :-
41         \+ stopEngine,
42         %proveGoalPost(?goal), !,
43         %showModule(WM),
44         %watchln(['Success! '-?goal-'and'-?WS-'are'-'
             choreographed!']),
45         ?Result = 'Yes',
46         insert{stopEngine}.
47         // %showModule(reportM).
48 /* The general case */
```

```
49 %runChoreography(?goal, ?WS, ?Result) :-
50         \+ stopEngine,
51
52         %eraseModule(DeltaWM),
53
54         %runWsRules(?WS),
55         %runGoalRules(?goal),
56
57         %showModule(WM),
58
59         ( ( %contradictory(WM,DeltaWM),!, writeln('Choreography
               failed due to CONTRADICTORY ACTIONS')@\prolog, ?Result
               = 'No');
60           ( \+ %deltaMakesAChange(WM,DeltaWM), !, writeln('
                 Choreography failed due to NO CHANGE')@\prolog, ?
                 Result = 'No') ;
61       ( %mergeDeltaIntoWM,
62                 %runChoreography(?goal,?WS, ?Result) ) ).
63
64 /*      \if (( %contradictory(WM,DeltaWM),?Reason=contradictory);
       ((\+ %deltaMakesAChange(WM,DeltaWM)), ?Reason=NoChange) )
65         \then (writeln('Choreography failed Due to'-?Reason)@\
               prolog,!, insert{stopEngine})
66         \else (
67                 %mergeDeltaIntoWM,
68                 %runChoreography(?goal,?WS, Result)).
                       */
69 /*-----------------------------------------------------*/
70
71 /* (3) runWsRules */
72 %runWsRules(?WS) :-
73         ?_Temp =
74                 setof{ ?ruleID | ?WS:WebService[wsRule(?ruleID):
                     ForallRule -> ?ruleBody],
75                         %invoke(WEBSERVICE,?ruleBody)},
76         ?_Temp2 =
77                 setof{ ?ruleID | ?WS:WebService[wsRule(?ruleID):
                     ChooseRule -> ?ruleBody],
78                         %invokeChoose(WEBSERVICE,?ruleBody)}.
79 /*-----------------------------------------------------*/
80
81 /* (4) runGoalRules */
82 %runGoalRules(?goal) :-
83         ?_Temp =
84                 setof{ ?ruleID | ?goal:Goal[gRule(?ruleID):
                     ForallRule -> ?ruleBody],
85                         %invoke(GOAL,?ruleBody)},
86         ?_Temp2 =
87                 setof{ ?ruleID | ?goal:Goal[gRule(?ruleID):
                     ChooseRule -> ?ruleBody],
88                         %invokeChoose(GOAL,?ruleBody)}.
89 /*-----------------------------------------------------*/
90 /* (5) invoke */
91 /* WEB SERVICE IF-THEN */
92 %invoke(WEBSERVICE,?X) :-
93         ?X ~ ${\if ?Y \then ?Z}, !,
94         %checkModeOfDeltaDeleteObjects(WEBSERVICE,?Z),
95         %prove(?X).
```

```
96  /* WEB SERVICE IF-THEN-ELSE */
97  %invoke(WEBSERVICE,?X) :-
98          ?X ~ ${\if ?Y \then ?Z \else ?W}, !,
99          %checkModeOfDeltaDeleteObjects(WEBSERVICE,?Z),
100         %checkModeOfDeltaDeleteObjects(WEBSERVICE,?W),
101         %prove(?X).
102
103 /* GOAL IF-THEN */
104 %invoke(GOAL,?X) :-
105         ?X ~ ${\if ?Y \then ?Z}, !,
106         %checkModeOfDeltaDeleteObjects(GOAL,?Z),
107         %prove(?X).
108 /* GOAL IF-THEN-ELSE */
109 %invoke(GOAL,?X) :-
110         ?X ~ ${\if ?Y \then ?Z \else ?W}, !,
111         %checkModeOfDeltaDeleteObjects(GOAL,?Z),
112         %checkModeOfDeltaDeleteObjects(GOAL,?W),
113         %prove(?X).
114 /*--------------------------------------------------*/
115
116 /* (6) invokeChoose */
117 /* WEB SERVICE */
118 /* IF-THEN */
119 %invokeChoose(WEBSERVICE,?X) :-
120         ?X ~ ${\if ?Y \then ?Z},!,
121         %checkModeOfDeltaDeleteObjects(WEBSERVICE,?Z),
122
123         ?ifThenList = setof{?T | %prove(?Y), ?T=(?Y,?Z)},
124         ?ifThenList[length -> ?len]@\btp,
125         %rand(1,?len,?chosenNum),
126         %giveElementAt(?ifThenList,?chosenNum,(?Y2,?Z2)),
127         %prove(?Z2).
128 /* IF-THEN-ELSE */
129 %invokeChoose(WEBSERVICE,?X) :-
130         ?X ~ ${\if ?Y \then ?Z \else ?W},!,
131         %checkModeOfDeltaDeleteObjects(WEBSERVICE,?Z),
132         %checkModeOfDeltaDeleteObjects(WEBSERVICE,?W),
133
134         \if (?Y)
135         \then (
136                 ?ifThenList = setof{?T | %prove(?Y), ?T=(?Y,?Z)},
137                 ?ifThenList[length -> ?len]@\btp,
138                 %rand(1,?len,?chosenNum),
139                 %giveElementAt(?ifThenList,?chosenNum,(?Y2,?Z2)),
140                 %prove(?Z2))
141         \else (
142                 ?ifThenList = setof{?T | %prove(?Y), ?T=(?Y,?W)},
143                 ?ifThenList[length -> ?len]@\btp,
144                 %rand(1,?len,?chosenNum),
145                 %giveElementAt(?ifThenList,?chosenNum,(?Y2,?W2)),
146                 %prove(?W2)).
147 /* GOAL */
148 /* IF-THEN */
149 %invokeChoose(GOAL,?X) :-
150         ?X ~ ${\if ?Y \then ?Z}, !,
151         %checkModeOfDeltaDeleteObjects(GOAL,?Z),
152
153         ?ifThenList = setof{?T | %prove(?Y), ?T=(?Y,?Z)},
```

```
154         ?ifThenList[length -> ?len]@\btp,
155         %rand(1,?len,?chosenNum),
156         %giveElementAt(?ifThenList,?chosenNum,(?_Y2,?Z2)),
157         %prove(?Z2).
158 /* IF-THEN-ELSE */
159 %invokeChoose(GOAL,?X) :-
160         ?X ~ ${\if ?Y \then ?Z \else ?W}, !,
161         %checkModeOfDeltaDeleteObjects(GOAL,?Z),
162         %checkModeOfDeltaDeleteObjects(GOAL,?W),
163 
164         \if (?Y)
165         \then (
166                 ?ifThenList = setof{?T | %prove(?Y), ?T=(?Y,?Z)},
167                 ?ifThenList[length -> ?len]@\btp,
168                 %rand(1,?len,?chosenNum),
169                 %giveElementAt(?ifThenList,?chosenNum,(?_Y2,?Z2))
                        ,
170                 %prove(?Z2))
171         \else(
172                 ?ifThenList = setof{?T | %prove(?Y), ?T=(?Y,?W)},
173                 ?ifThenList[length -> ?len]@\btp,
174                 %rand(1,?len,?chosenNum),
175                 %giveElementAt(?ifThenList,?chosenNum,(?_Y2,?W2))
                        ,
176                 %prove(?W2)).
177 /*----------------------------------------------------*/
178 
179 /* (7) mergeDeltaIntoWM */
180 %mergeDeltaIntoWM :-
181         ?_T1 =
182                 setof{
183                         ?A | ins_action(?A)@DeltaWM,
184                         %copyReifiedObjectIntoModule(?A, DeltaWM,
                                WM)},
185         ?_T2 =
186                 setof{
187                         ?A | del_action(?A)@DeltaWM,
188                         %convertReifiedObjectModule(?A, DeltaWM,
                            WM, ?A_new),
189                         deleteall{?A_new@WM}},
190         ?_T3 =
191                 setof{
192                         ?objOld | update_action(?objOld, ?objNew)
                            @DeltaWM,
193                         deleteall{?objOld@WM},
194                         %convertReifiedObjectModule(?objNew,
                            DeltaWM, WM, ?newObject),
195                         insert{?newObject@WM}}.
196 /*----------------------------------------------------*/
197 
198 /* (8) deltaMakesAChange */
199 /* case 1 */
200 %deltaMakesAChange(?WM, ?DeltaWM) :-
201         ins_action(?A)@?DeltaWM,
202         %convertReifiedObjectModule(?A, ?DeltaWM, ?WM, ?A_new),
203         \+ ?A_new@?WM.
204 /* case 2 */
205 %deltaMakesAChange(?WM, ?DeltaWM) :-
```

```
206         del_action(?A)@?DeltaWM,
207         %convertReifiedObjectModule(?A, ?DeltaWM, ?WM, ?A_new),
208         ?A_new@?WM.
209 /* case 3 */
210 %deltaMakesAChange(?WM, ?DeltaWM) :-
211         update_action(?objOld,?objNew)@?DeltaWM,
212         %convertReifiedObjectModule(?objNew, ?DeltaWM, ?WM, ?
            newObject),
213         \+ ?newObject@?WM.
214 /*----------------------------------------------------*/
215
216 /* (9) contradictory */
217 /* case 1 */
218 %contradictory(?_WM, ?DeltaWM) :-
219         ins_action(?A1)@?DeltaWM,
220         del_action(?A2)@?DeltaWM,
221         %contained(?X1,?A1),
222         %contained(?X2,?A2),
223         ?X1 = ?X2, !.
224 /* case 2 */
225 %contradictory(?WM, ?DeltaWM) :-
226         del_action(?A)@?DeltaWM,
227         %convertReifiedObjectModule(?A, ?DeltaWM, ?WM, ?A_new),
228         \+ ?A_new@?WM.
229 /* case 3 */
230 %contradictory(?WM, ?DeltaWM) :-
231         update_action(?objOld,?objNew)@?DeltaWM,
232         \+ ?objOld@?WM.
233 // We should also check if two insertion of a same object/value
       are done.
234 /*----------------------------------------------------*/
```

Library Predicates

Listing 7.3: Library Predicates

```
1  /*** Generated by Visual Semantic Choreography on 3/30/2017
       8:17:15 PM ***/
2
3  /*--------------------------------------------------*/
4  // (1) preProcessCheckings
5  // (2) importOntology
6  // (3) insertGoalPre
7  // (4) proveGoalPost
8  // (5) prove
9  // (6) delta Actions
10 // (7) convertReifiedObjectModule
11 // (8) copyReifiedObjectIntoModule
12 // (9) extractConcepts
13 // (10) extractPredicates
14 // (11) filterOutPredicates
15 // (12) isNotFrame
16 // (13) checkUsageOfMembershipOperatorInDeltaDelete
17 /*--------------------------------------------------*/
18
19 /* (1) preProcessCheckings */
20 %preProcessCheckings(?goal,?WS) :-
21         %checkUsageOfMembershipOperatorInDeltaDelete(?goal),
22         %checkUsageOfMembershipOperatorInDeltaDelete(?WS),
23         %preProcessCheckModesForGoalPre(?goal),
24         %preProcessCheckModes(GOAL,?goal),
25         %preProcessCheckModes(WEBSERVICE,?WS).
26 /*--------------------------------------------------*/
27
28 /* (2) importOntology */
29 %importOntology(?X,?module) :-
30         ?fileLocation = ?X.importOntology,
31         add{?fileLocation >> ?module}.
32 /*--------------------------------------------------*/
33
34 /* (3) insertGoalPre */
35 %insertGoalPre(?goal,?newModuleName) :-
36         ?_C = ?goal.capability, ?_C = (?_pre, ?_post), ?_pre ~ $
               {'->'(pre,(?GoalPre))@main},
37         %copyReifiedObjectIntoModule(?GoalPre, main, ?
               newModuleName).
38 /*--------------------------------------------------*/
39
40 /* (4) proveGoalPost */
41 %proveGoalPost(?goal) :-
42         ?_C = ?goal.capability, ?_C = (?_pre, ?_post), ?_post ~ $
               {'->'(post,(?GoalPost))@main},
43         //?goal[post -> ?GoalPost],
44         %convertReifiedObjectModule(?GoalPost, main, WM, ?
               newGoalPost),
45         %prove(?newGoalPost).
46 /*--------------------------------------------------*/
47
48 /* (5) prove */
49 /* AND */
50 %prove(?X) :-
```

```
51          ?X = (?Left, ?Right), !,
52          %prove(?Left),%prove(?Right).
53  /* OR */
54  %prove(?X) :-
55          ?X = (?Left; ?Right), !,
56          (%prove(?Left); %prove(?Right)).
57  /* NOT */
58  %prove(?X) :-
59          ?X = (\+ ?G), !,
60          \+ %prove(?G).
61  /* ATOM */
62  %prove(?X) :- ?X.
63  /*-----------------------------------------------------*/
64
65  /* (6) delta Actions */
66  /* deltaInsert */
67  %deltaInsert(?obj) :-
68          ?action = ${ins_action(?obj)},
69          %copyReifiedObjectIntoModule(?action, main, DeltaWM).
70  /* deltaDelete */
71  %deltaDelete(?obj) :-
72          ?action = ${del_action(?obj)},
73          %copyReifiedObjectIntoModule(?action, main, DeltaWM).
74  /* deltaUpdate: type A */
75  %deltaUpdate(?objOld, ?objNew) :-
76          %convertReifiedObjectModule(?objOld, main, WM, ?oldObject
              ),
77          \if (?oldObject) // if the specified object already
              exists in WM
78          \then (
79                  %convertReifiedObjectModule(?objNew, main,
                      DeltaWM, ?newObject),
80                  ?action = ${update_action(?oldObject, ?newObject)
                      },
81                  %convertReifiedObjectModule(?action, main,
                      DeltaWM, ?newAction),
82                  insert{?newAction@DeltaWM}) // then put update
                      action into DeltaWM
83          \else \false.
84  /* deltaUpdate: type B */
85  %deltaUpdate(?obj, ?attr, ?oldVal, ?newVal) :-
86          %convertReifiedObjectModule(${?obj[?attr->?oldVal]}, main
              , WM, ?ObjectOld),
87          %convertReifiedObjectModule(${?obj[?attr->?newVal]}, main
              , WM, ?ObjectNew),
88          ?action = ${update_action(?ObjectOld, ?ObjectNew)},
89          %convertReifiedObjectModule(?action, main, DeltaWM, ?
              newAction),
90          insert{?newAction@DeltaWM}.
91  /*-----------------------------------------------------*/
92
93  /* (7) convertReifiedObjectModule */
94  %convertReifiedObjectModule(?reifiedObject, ?sourceModule, ?
      targetModule, ?convertedReifiedObject) :-
95          ?S = ?sourceModule, name(?S,?sourceModuleName)@\prolog,
96          ?T = ?targetModule, name(?T,?targetModuleName)@\prolog,
97          ?reifiedObject =.. ?B,
98          \symbol[toType(?B) -> ?C]@\btp,
```

```
99          \symbol[concat([?C,'.']) -> ?CC]@\btp,
100         name(?CC,?F)@\prolog,
101         %replaceAll(?F,?sourceModuleName,?targetModuleName,?G),
102         string(?G)[readAll(?I)]@\parse,
103         ?I[ith(1) -> ?J]@\btp,
104         //?J =.. [?_,?K,?_], in the Last testef Ergo version
105         ?J = code(?K,?_)@\prolog,
106         ?convertedReifiedObject =.. ?K.
107 /*-----------------------------------------------------*/
108
109 /* (8) copyReifiedObjectIntoModule */
110 %copyReifiedObjectIntoModule(?reifiedObject, ?sourceModule, ?
        targetModule) :-
111         ?S = ?sourceModule, name(?S,?sourceModuleName)@\prolog,
112         ?T = ?targetModule, name(?T,?targetModuleName)@\prolog,
113         ?reifiedObject =.. ?B,
114         \symbol[toType(?B) -> ?C]@\btp,
115         \symbol[concat([?C,'.']) -> ?CC]@\btp,
116         name(?CC,?F)@\prolog,
117         %replaceAll(?F,?sourceModuleName,?targetModuleName,?G),
118         string(?G)[readAll(?I)]@\parse,
119         ?I[ith(1) -> ?J]@\btp,
120         //?J =.. [?_,?K,?_], in the Last testef Ergo version
121         ?J = code(?K,?_)@\prolog,
122         ?L =.. ?K,
123         insert{?L}.
124 /*-----------------------------------------------------*/
125
126 /* (9) extractConcepts */
127 %extractConcepts([?H|?T], ?LstIn, ?LstOut) :- ?H != 58, %
        extractConcepts(?T, ?LstIn, ?LstOut).
128
129 %extractConcepts([58|?Rest], ?LstIn, ?LstOut) :-
130         %readTheTerm(?Rest, "", ?term, ?remainder),
131         \symbol[toType(?term) -> ?termSym]@\btp,
132         ?termItem = [?termSym],
133         ?LstIn[append(?termItem) -> ?LstNew]@\btp,
134         %extractConcepts(?remainder, ?LstNew, ?LstOut).
135
136 %extractConcepts([], ?LstIn, ?LstOut) :- ?LstOut = ?LstIn.
137 /*-----------------------------------------------------*/
138
139 /* (10) extractPredicates */
140 %extractPredicates([?H|?T], ?LstIn, ?LstOut) :- ?H != 123, %
        extractPredicates(?T, ?LstIn, ?LstOut).
141
142 %extractPredicates([123|?Rest], ?LstIn, ?LstOut) :-
143         %readTheTerm(?Rest, ""^^\charlist, ?term, ?remainder),
144         ?termItem = [?term],
145         ?LstIn[append(?termItem) -> ?LstNew]@\btp,
146         %extractPredicates(?remainder, ?LstNew, ?LstOut).
147
148 %extractPredicates([], ?LstIn, ?LstOut) :- ?LstOut = ?LstIn.
149 /*-----------------------------------------------------*/
150
151 /* (11) filterOutPredicates */
152 %filterOutPredicates([?H|?T], ?LstIn, ?LstOut) :-
153         %isNotFrame(?H),
```

```
154         %removeParenthesis(?H,[],?Term),
155         \symbol[toType(?Term) -> ?TermSym]@\btp,
156         ?PreName = [?TermSym],
157         ?LstIn[append(?PreName) -> ?LstNew]@\btp,
158         %filterOutPredicates(?T, ?LstNew, ?LstOut).
159
160 %filterOutPredicates([?H|?T], ?LstIn, ?LstOut) :-
161         (\+ %isNotFrame(?H)),
162         %filterOutPredicates(?T, ?LstIn, ?LstOut).
163
164 %filterOutPredicates([], ?LstIn, ?LstOut) :- ?LstOut = ?LstIn.
165 /*---------------------------------------------------*/
166
167 /* (12) isNotFrame */
168 %isNotFrame([?H|?T]) :- (?H != 58, ?H != 91), %isNotFrame(?T).
169 %isNotFrame([]).
170
171 /* (13) checkUsageOfMembershipOperatorInDeltaDelete */
172 %checkUsageOfMembershipOperatorInDeltaDelete(?X) :-
173         ?_RHSs = setof{ ?Z | ?X[rule(?_gRule):ForallRule -> ?
                _ruleBody], ?_ruleBody ~ ${\if ?_Y \then ?Z}},
174         %checkAllRHS(?X,?_RHSs).
175
176 %checkAllRHS(?gOrWS,[]).
177 %checkAllRHS(?gOrWS,[?H|?T]) :-
178         %checkRHS(?gOrWS,?H),
179         %checkAllRHS(?gOrWS,?T).
180
181 %checkRHS(?gOrWS,?Z) :-
182         \if (?Z ~ (?Z1,?Z2))
183         \then (
184                 %checkRHS(?gOrWS,?Z1),
185                 %checkRHS(?gOrWS,?Z2))
186         \else
187                 (%checkExistanceOfColon(?gOrWS,?Z)).
188
189 %checkExistanceOfColon(?gOrWS,?Z) :-
190         ?Z =.. [?X1|?X2],
191         \if (\+ ?X1 = '%hilog'(deltaDelete,?_M))
192         \then \true
193         \else (
194                 \if (\+ %checkMembershipOperator(?X2))
195                 \then (%watchln(['Illegal deltaDelete. Do not use
                     : in deltaDelete of '-?gOrWS]), \false)
196                 \else \true ).
197
198 %checkMembershipOperator([]).
199 %checkMembershipOperator([?H|?T]) :-
200         %reformatToString(?H, ?HStr),
201         %noColonExists(?HStr),
202         %checkMembershipOperator(?T).
203
204 %noColonExists([]).
205 %noColonExists([?H|?T]) :- ?H != 58, %noColonExists(?T).
206 /*---------------------------------------------------*/
```

Mode Checking Predicates

Listing 7.4: Mode Checking Predicates

```
1  /*** Generated by Visual Semantic Choreography on 3/30/2017
       8:17:15 PM ***/
2
3  /*-----------------------------------------------------*/
4
5  /* preProcessCheckModesForGoalPre */
6  %preProcessCheckModesForGoalPre(?goal) :-
7          ?_C = myGoal.capability, ?_C = (?_pre, ?_post), ?_pre ~ $
              {'->'(pre,(?GoalPre))@main},
8          ?Z = ?GoalPre,
9          //?goal[pre->?Z],
10         %reformatToString(?Z, ?ZStr),
11         %extractConcepts(?ZStr, [], ?writeList),
12
13         %extractPredicates(?ZStr, [], ?termList),
14         %filterOutPredicates(?termList, [], ?preWriteList),
15
16         \if (\+ %checkGoalPreModes(FRAME,?writeList))
17         \then (writeln('Error: Illegal access mode of a FRAME in
              '-?goal-'PRECONDITION.')@\prolog,!,\false),
18
19         \if (\+ %checkGoalPreModes(PREDICATE,?preWriteList))
20         \then (writeln('Error: Illegal access mode of a PREDICATE
               in'-?goal-'PRECONDITION.')@\prolog,!,\false).
21
22 %checkGoalPreModes(?fOrP,[]).
23
24 %checkGoalPreModes(?fOrP,[?F|?R]):-
25         \if (\+ %checkGoalPreMode(?fOrP,?F))
26         \then (writeln(['Violating'-?fOrP-'in GOAL PRECONDITION
              is'-?F])@\prolog, !, \false),
27         %checkGoalPreModes(?fOrP,?R).
28
29 %checkGoalPreMode(?fOrP,?F) :-
30         (?F:In \or
31         ?F:Shared), !.
32 /*-----------------------------------------------------*/
33
34 /* preProcessCheckModes */
35 %preProcessCheckModes(?gOrWs,?X) :-
36         ?allRuleBodies = setof{ ?ruleBody |
37                 (?X[?ruleName(?tag):ForallRule -> ?ruleBody];
38                 ?X[?ruleName(?tag):ChooseRule -> ?ruleBody])},
39         %checkAll(?gOrWs,?allRuleBodies).
40
41 %checkAll(?gOrWs,[]) :- !.
42 %checkAll(?gOrWs,[?H|?T]) :-
43         %check(?gOrWs,?H),
44         %checkAll(?gOrWs,?T).
45
46 /* IF-THEN */
47 %check(?gOrWs,?X) :-
48         ?X ~ ${\if ?Y \then ?Z}, !,
49
50         // ?Y
```

```
51          %reformatToString(?Y, ?YStr),
52          %extractConcepts(?YStr, [], ?conceptsInY),
53          %extractPredicates(?YStr, [], ?termList1),
54          %filterOutPredicates(?termList1, [], ?predicatesInY),
55
56          \if (\+ %checkAllFramesModes(?gOrWs,READ,?conceptsInY))
57          \then (writeln(['Error: Illegal access mode of a FRAME
                for a rule LHS in '-?gOrWs])@\prolog,!,\false),
58
59          \if (\+ %checkAllPredicatesModes(?gOrWs,READ,?
                predicatesInY))
60          \then (writeln(['Error: Illegal access mode of a
                PREDICATE for a rule LHS in '-?gOrWs])@\prolog,!,\
                false),
61
62          // ?Z
63          %decomposeRHS(?Z, [], ?allFsOrPs),
64          %reformatToString(?allFsOrPs, ?allFsOrPsStr),
65          %extractConcepts(?allFsOrPsStr, [], ?conceptsInZ),
66          %extractPredicates(?allFsOrPsStr, [], ?temp),
67          %filterOutPredicates(?temp, [], ?predicatesInZ),
68
69          \if (\+ %checkAllFramesModes(?gOrWs,WRITE,?conceptsInZ))
70          \then (writeln(['Error: Illegal access mode of a FRAME
                for a rule RHS in '-?gOrWs])@\prolog,!,\false),
71
72          \if (\+ %checkAllPredicatesModes(?gOrWs,WRITE,?
                predicatesInZ))
73          \then (writeln(['Error: Illegal access mode of a
                PREDICATE for a rule RHS in '-?gOrWs])@\prolog,!,\
                false).
74
75  /*---------------------------------------------------*/
76
77  /* IF-THEN-ELSE */
78  %check(?gOrWs,?X) :-
79          ?X ~ ${\if ?Y \then ?Z \else ?W}, !,
80
81          // ?Y
82          %reformatToString(?Y, ?YStr),
83          %extractConcepts(?YStr, [], ?conceptsInY),
84          %extractPredicates(?YStr, [], ?termList1),
85          %filterOutPredicates(?termList1, [], ?predicatesInY),
86
87          \if (\+ %checkAllFramesModes(?gOrWs,READ,?conceptsInY))
88          \then (writeln(['Error: Illegal access mode of a FRAME
                for a rule LHS in '-?gOrWs])@\prolog,!,\false),
89
90          \if (\+ %checkAllPredicatesModes(?gOrWs,READ,?
                predicatesInY))
91          \then (writeln(['Error: Illegal access mode of a
                PREDICATE for a rule LHS in '-?gOrWs])@\prolog,!,\
                false),
92
93          // ?Z
94          %decomposeRHS(?Z, [], ?allFsOrPs),
95          %reformatToString(?allFsOrPs, ?allFsOrPsStr),
96          %extractConcepts(?allFsOrPsStr, [], ?conceptsInZ),
```

```
97          %extractPredicates(?allFsOrPsStr, [], ?temp),
98          %filterOutPredicates(?temp, [], ?predicatesInZ),
99
100         \if (\+ %checkAllFramesModes(?gOrWs,WRITE,?conceptsInZ))
101         \then (writeln(['Error: Illegal access mode of a FRAME
               for a rule RHS in '-?gOrWs])@\prolog,!,\false),
102
103         \if (\+ %checkAllPredicatesModes(?gOrWs,WRITE,?
               predicatesInZ))
104         \then (writeln(['Error: Illegal access mode of a
               PREDICATE for a rule RHS in '-?gOrWs])@\prolog,!,\
               false),
105
106         // ?W
107         %decomposeRHS(?W, [], ?W_allFsOrPs),
108         %reformatToString(?W_allFsOrPs, ?W_allFsOrPsStr),
109         %extractConcepts(?W_allFsOrPsStr, [], ?conceptsInW),
110         %extractPredicates(?W_allFsOrPsStr, [], ?W_temp),
111         %filterOutPredicates(?W_temp, [], ?predicatesInW),
112
113         \if (\+ %checkAllFramesModes(?gOrWs,WRITE,?conceptsInW))
114         \then (writeln(['Error: Illegal access mode of a FRAME
               for a rule RHS in '-?gOrWs])@\prolog,!,\false),
115
116         \if (\+ %checkAllPredicatesModes(?gOrWs,WRITE,?
               predicatesInW))
117         \then (writeln(['Error: Illegal access mode of a
               PREDICATE for a rule RHS in '-?gOrWs])@\prolog,!,\
               false).
118
119
120 %decomposeRHS(?X, ?LstIn, ?LstOut) :-
121         \if (?X ~ (?X1,?X2))
122         \then (
123                 %decomposeRHS(?X1,?LstIn, ?LstTemp1),
124                 %decomposeRHS(?X2,?LstIn, ?LstTemp2),
125                 \list[append([?LstTemp1,?LstTemp2])->?LstOut]@\
                       btp)
126         \else
127                 (%extractFrameOrPredicateFromDeltaAction(?X,?fOrP
                       ),
128                 \list[append([?LstIn,[?fOrP]])->?LstOut]@\btp).
129
130 %extractFrameOrPredicateFromDeltaAction(?X,?FrameOrPredicate) :-
131         ((?X ~ ${%deltaInsert(?T)});
132          (?X ~ ${%deltaDelete(?T)});
133          (?X ~ ${%deltaUpdate(?T)})),
134          ?FrameOrPredicate = ?T.
135 /*-----------------------------------------------------*/
136
137 /* checkModeOfDeltaDeleteObjects */
138 %checkModeOfDeltaDeleteObjects(?gOrWs, ?Z) :-
139         \if (?Z ~ (?Z1,?Z2))
140         \then (
141                 %checkModeOfDeltaDeleteObjects(?gOrWs, ?Z1),
142                 %checkModeOfDeltaDeleteObjects(?gOrWs, ?Z2)
143         )
144         \else
```

```
145                 (%checkModeOfDeltaDeleteObject(?gOrWs, ?Z)).
146
147  %checkModeOfDeltaDeleteObject(?gOrWs, ?Z) :-
148         ?Z =.. [?X1|?X2],
149         \if (\+ ?X1 = '%hilog'(deltaDelete,?_M))
150         \then \true
151         \else (
152                 %reformatToString(?X2, ?X2Str),
153                 %extractConceptsForDeltaDelete(?X2Str, [], ?
                        writeList),
154                 \if (\+ %checkModesDeltaDelete(?gOrWs,WRITE,?
                        writeList))
155                 \then (%watchln(['Error: Illegal access mode in
                        deltaDelete of '-?gOrWs]),!,\false)
156         ).
157
158  %checkModesDeltaDelete(?gOrWs,WRITE,[]).
159  %checkModesDeltaDelete(?gOrWs,WRITE,[?H|?T]) :-
160         %checkModeDeltaDelete(?gOrWs,WRITE,?H),
161         %checkModesDeltaDelete(?gOrWs,WRITE,?T).
162
163  //Frame
164  %checkModeDeltaDelete(GOAL,WRITE,?H) :-
165         (?H:?Concept)@WM,?Concept:Mode,
166         (?Concept:In \or
167         ?Concept:Shared), !.
168  //Predicate
169  %checkModeDeltaDelete(GOAL,WRITE,?H) :-
170         ?H:UserPredicate,
171         (?H:In \or
172         ?H:Shared), !.
173
174  //Frame
175  %checkModeDeltaDelete(WEBSERVICE,WRITE,?H) :-
176         (?H:?Concept)@WM,?Concept:Mode,
177         (?Concept:Controlled \or
178         ?Concept:Out \or
179         ?Concept:Shared), !.
180  //Predicate
181  %checkModeDeltaDelete(WEBSERVICE,WRITE,?H) :-
182         ?H:UserPredicate,
183         (?H:Controlled \or
184         ?H:Out \or
185         ?H:Shared), !.
186
187  %extractConceptsForDeltaDelete([?H|?T], ?LstIn, ?LstOut) :-
188         ?H != 123,
189         %extractConceptsForDeltaDelete(?T, ?LstIn, ?LstOut).
190
191  %extractConceptsForDeltaDelete([123|?Rest], ?LstIn, ?LstOut) :-
192         %readTheTermForDeltaDelete(?Rest, ""^^\charlist, ?term, ?
                remainder),
193         \symbol[toType(?term) -> ?termSym]@\btp,
194         ?termItem = [?termSym],
195         ?LstIn[append(?termItem) -> ?LstNew]@\btp,
196         %extractConceptsForDeltaDelete(?remainder, ?LstNew, ?
                LstOut).
197
```

```
198 %extractConceptsForDeltaDelete([], ?LstIn, ?LstOut) :- ?LstOut =
        ?LstIn.
199
200 %readTheTermForDeltaDelete([?H|?Rest], ?termIn, ?termOut, ?
        remainder) :-
201         (?H != 91, ?H != 40, ?H != 64, ?H != 58), // [ ( @ :
202         ?HS = [?H],
203         ?termIn[concat(?HS) -> ?termIn2]@\btp,
204         %readTheTermForDeltaDelete(?Rest, ?termIn2, ?termOut, ?
            remainder).
205
206 %readTheTermForDeltaDelete([?H|?Rest], ?termIn, ?termOut, ?
        remainder) :-
207         (?H = 91 ; ?H = 58 ; ?H = 40 ; ?H = 64),
208         ?termOut = ?termIn, ?remainder = ?Rest.
209
210 %readTheTermForDeltaDelete([], ?termIn, ?termOut, ?remainder) :-
        ?termOut = ?termIn, ?remainder = [].
211 /*-----------------------------------------------------*/
212
213 /* checkAllFramesModes */
214 %checkAllFramesModes(?gOrWS, ?reOrWr,[]).
215
216 %checkAllFramesModes(?gOrWS, ?reOrWr,[?F|?R]):-
217         %checkFrameMode(?gOrWS, ?reOrWr,?F),
218         %checkAllFramesModes(?gOrWS, ?reOrWr,?R).
219
220 %checkFrameMode(GOAL,READ,?F):-
221         ( ?F:In \or
222           ?F:Out \or
223           ?F:Static \or
224           ?F:Shared ), !.
225
226 %checkFrameMode(GOAL,WRITE,?F):-
227         (?F:In \or
228         ?F:Shared), !.
229
230 %checkFrameMode(GOAL,READ,?F):- writeln(['Illegal GOAL READ
        action for',?F])@\prolog, !, \false.
231 %checkFrameMode(GOAL,WRITE,?F):- writeln(['Illegal GOAL WRITE
        action for',?F])@\prolog, !, \false.
232
233 %checkFrameMode(WEBSERVICE,READ,?F):-
234         (?F:Controlled \or
235         ?F:In \or
236         ?F:Out \or
237         ?F:Static \or
238         ?F:Shared), !.
239 %checkFrameMode(WEBSERVICE,WRITE,?F):-
240         (?F:Controlled \or
241         ?F:Out \or
242         ?F:Shared), !.
243
244 %checkFrameMode(WEBSERVICE,READ,?F):- writeln(['Illegal
        WEBSERVICE READ action for',?F])@\prolog, !, \false.
245 %checkFrameMode(WEBSERVICE,WRITE,?F):- writeln(['Illegal
        WEBSERVICE WRITE action for',?F])@\prolog, !, \false.
246 /*-----------------------------------------------------*/
```

```
247
248  /* checkAllPredicatesModes */
249  %checkAllPredicatesModes(?gOrWS, ?reOrWr,[]).
250
251  %checkAllPredicatesModes(?gOrWS, ?reOrWr,[?F|?R]):-
252          %checkPredicateMode(?gOrWS, ?reOrWr,?F),
253          %checkAllPredicatesModes(?gOrWS, ?reOrWr,?R).
254
255  %checkPredicateMode(GOAL,READ,?F):-
256          (?F:In \or
257          ?F:Out \or
258          ?F:Static \or
259          ?F:Shared), !.
260  %checkPredicateMode(GOAL,WRITE,?F):-
261          (?F:In \or
262          ?F:Shared), !.
263
264  %checkPredicateMode(GOAL,READ,?F):-     writeln(['Illegal GOAL
        READ action for',?F])@\prolog, !, \false.
265  %checkPredicateMode(GOAL,WRITE,?F):- writeln(['Illegal GOAL WRITE
         action for',?F])@\prolog, !, \false.
266
267  %checkPredicateMode(WEBSERVICE,READ,?F):-
268          (?F:Controlled \or
269          ?F:In \or
270          ?F:Out \or
271          ?F:Static \or
272          ?F:Shared), !.
273  %checkPredicateMode(WEBSERVICE,WRITE,?F):-
274          (?F:Controlled \or
275          ?F:Out \or
276          ?F:Shared), !.
277
278  %checkPredicateMode(WEBSERVICE,READ,?F):- writeln(['Illegal
        WEBSERVICE READ action for',?F])@\prolog, !, \false.
279  %checkPredicateMode(WEBSERVICE,WRITE,?F):- writeln(['Illegal
        WEBSERVICE WRITE action for',?F])@\prolog, !, \false.
280  /*-------------------------------------------------*/
```

Utility Predicates

Listing 7.5: Utility Predicates

```
1  /*** Generated by Visual Semantic Choreography on 3/30/2017
      8:17:15 PM ***/
2
3  dg.
4
5  /* prepareModule */
6  %prepareModule(?module) :-
7          \if (\+ isloaded{?module})
8          \then newmodule{?module},
9          //%checkModule(?module),
10         %eraseModule(?module).
11 /*-----------------------------------------------------*/
12
13 /* initializations */
14 %initializations :-     %initializeRandom.
15 /*-----------------------------------------------------*/
16
17 /*initializeRandom*/
18 %initializeRandom :- datime_setrand@\prolog(random).
19 /*-----------------------------------------------------*/
20
21 /* rand */
22 %rand(?L,?U,?R) :- ?UU \is ?U+1, random(?L,?UU,?R)@\prologall(
      random). //both inclusive
23 /*-----------------------------------------------------*/
24
25 /* giveElementAt */
26 %giveElementAt(?L,?n,?elementAt) :- nth0(?n,?L,?elementAt)@\
      prologall(lists).
27 /*-----------------------------------------------------*/
28
29 /* watch(ln) */
30 %watchln() :- \if dg \then writeln('')@\prolog.
31 %watch(?X) :- \if dg \then write(?X)@\prolog.
32 %watchln(?X) :- \if dg \then writeln(?X)@\prolog.
33 /*-----------------------------------------------------*/
34
35 /* debug */
36 %debug(?X) :- \if (?X == on) \then (insert{dg},writeln('Debug is
      ON.')@\prolog)
37                           \else \if (?X == off) \then (deleteall{
                                dg},writeln('Debug is OFF.')@\prolog
                                ).
38 /*-----------------------------------------------------*/
39
40 /* /* checkModule
41 %checkModule(?M) :- isloaded{?M}, !.
42 %checkModule(?M) :- newmodule{?M}. */
43 /*-----------------------------------------------------*/
44
45 /* eraseModule */
46 %eraseModule(?M) :- deleteall{?_(?_)@?M}, deleteall{?_(?_,?_)@?M
      }, deleteall{?_[?_ -> ?_]@?M}, deleteall{?_:?_[?_ -> ?_]@?M},
      deleteall{?_:?_@?M} .
47 /*-----------------------------------------------------*/
```

```
48
49  /* makeFileAddress */
50  %makeFileAddress(?dir,?name,?fileAddress) :- \symbol[concat([?dir
        ,?name])->?fileAddress]@\basetype.
51  /*-------------------------------------------------------*/
52
53  /* replaceAll */
54  /*Replaces a term with another in a string*/
55  %replaceAll([],?_Pattern,?_Replace,[]) :- !.
56
57  %replaceAll(?OldString,?Pattern,?Replace,?NewString):-
58    %startsWith(?OldString,?Pattern,?Rest),!,
59    %replaceAll(?Rest,?Pattern,?Replace,?TailNewString),
60    ?Replace[append(?TailNewString) -> ?NewString]@\btp.
61
62  %replaceAll([?H|?TailOldString],?Pattern,?Replace,[?H|?
        TailNewString]):-
63          %replaceAll(?TailOldString,?Pattern,?Replace,?
              TailNewString).
64  /*-------------------------------------------------------*/
65
66  /* startsWith */
67  %startsWith(?OldString,[],?OldString2):- ?OldString2 = ?OldString
        , !.
68  %startsWith([?H|?TOldString],[?H|?T],?Rest):- !, %startsWith(?
        TOldString,?T,?Rest).
69  /*-------------------------------------------------------*/
70
71  /* contained */
72  %contained(?X, ?A) :- ?X = ?A.
73  %contained(?X, ?A) :- ?A = (?A1, ?_Rest), ?X = ?A1.
74  %contained(?X, ?A) :- ?A = (?_A1, ?Rest), %contained(?X, ?Rest).
75  /*-------------------------------------------------------*/
76
77  /* showModule */
78  %showModule(?M) :-
79          %watchln('---------------------------------------'),
80          %watch('In '),
81          %watchln(?M),
82          %watchln('---------------------------------------'),
83          ?_L1 = setof{?X |
84                  ?X:?C[?Y -> ?Z]@?M,
85                  (?C != \callable),
86                  %immediateClassInModule(?X,?C,?M),
87                  %watch(?X),%watch(':'),%watch(?C),%watch('['),%
                        watch(?Y),%watch('->'),%watch(?Z),%watchln
                        (']')
88          },
89          // ?_L2 = setof{?X |
90                  // ?X@?M,%watchln(?X)
91          // },
92          ?_L3 = setof{?X |
93                  ?X(?Y)@?M,%watch(?X),%watch('('),%watch(?Y),%
                        watchln(')')
94          },
95          ?_L4 = setof{?X |
96                  ?X(?Y,?Z)@?M,%watch(?X),%watch('('),%watch(?Y),%
                        watch(','),%watch(?Z),%watchln(')')
```

```
97          },
98          ?_L5 = setof{?X |
99                  ?X()@?M,%watchln(?X)
100         },
101         ?_L6 = setof{?X |
102                 ?X(?Y,?Z,?W)@?M,%watch(?X),%watch('('),%watch(?Y)
                        ,%watch(','),%watch(?Z),%watch(','),%watch(?W)
                        ,%watchln(')')
103         } //.
104         ,
105         %pause(?_).
106 /*-----------------------------------------------------*/
107
108 /* pause */
109 %pause() :- get(?_X)@\prolog.
110 %pause(?X) :- get(?X)@\prolog.
111 /*-----------------------------------------------------*/
112
113 /* immediateClassInModule */
114 %immediateClassInModule(?obj,?class,?Mod) :-
115         ?obj:?class@?Mod,
116         ?class != \object,
117         ?class != \symbol,
118         ?class != (?_Y;?_Z).
119 /*-----------------------------------------------------*/
120
121 /* removeParenthesis */
122 // 40 is (
123 %removeParenthesis([?H|?T], ?In, ?Out) :-
124         ?H != 40,
125         ?HS = [?H],
126         ?In[concat(?HS) -> ?New]@\btp,
127         %removeParenthesis(?T, ?New, ?Out).
128
129 %removeParenthesis([], ?In, ?Out) :- ?Out = ?In.
130 %removeParenthesis([40|?T], ?In, ?Out) :- ?Out = ?In.
131 /*-----------------------------------------------------*/
132
133 /* report */
134 %report(?item) :- \true.
135 /*-----------------------------------------------------*/
136
137 %testMe([]).
138 %testMe([?H|?T]) :- ?H = 32, !, %testMe(?T).
139 %testMe(?X) :- write(?X)@\prolog.
140
141 /* readTheTerm */
142 // 92 is \, 43 is +, 64 is @, 32 is [space]
143 %readTheTerm([92|?Rest], ?termIn, ?termOut, ?remainder) :- !,%
        readTheTerm(?Rest, ?termIn, ?termOut, ?remainder).
144 %readTheTerm([43|?Rest], ?termIn, ?termOut, ?remainder) :- !,%
        readTheTerm(?Rest, ?termIn, ?termOut, ?remainder).
145 %readTheTerm([32|?Rest], ?termIn, ?termOut, ?remainder) :- !,%
        readTheTerm(?Rest, ?termIn, ?termOut, ?remainder).
146
147 %readTheTerm([?H|?Rest], ?termIn, ?termOut, ?remainder) :-
148         ?H != 64,
149         ?HS = [?H],
```

```
150         ?termIn[concat(?HS) -> ?termIn2]@\btp,
151         %readTheTerm(?Rest, ?termIn2, ?termOut, ?remainder).
152
153  %readTheTerm([64|?Rest], ?termIn, ?termOut, ?remainder) :- ?
        termOut = ?termIn, ?remainder = ?Rest.
154  /*---------------------------------------------------*/
155
156  /* reformatToString */
157  %reformatToString(?In, ?Str) :-
158         \symbol[toType(?In) -> ?A]@\btp,
159         name(?A,?Str)@\prolog.
160  /*---------------------------------------------------*/
161
162  ///////////////////////////
163  /* %readTheTerm2([?H|?Rest], ?termIn, ?termOut, ?remainder) :-
164         ?H != 64,
165         ?HS = [?H],
166         ?termIn[concat(?HS) -> ?termIn2]@\btp,
167         %readTheTerm2(?Rest, ?termIn2, ?termOut, ?remainder).
168
169  %readTheTerm2([64|?Rest], ?termIn, ?termOut, ?remainder) :- ?
        termOut = ?termIn, ?remainder = ?Rest. */
```

Appendix F: More choreography specification examples

Example 1: Semantic authentication

Service requesters usually are asked to be authenticated by online services before being able to utilize them. Authentication information can be asked at the beginning, in the middle or in last steps of the service consumption process. Here we present a semantic choreography for the authentication process. If the service requester (goal) and service provider (Web Service) use the same terminology, the requester can pass authentication information in a choreography to the Web Service. Listings 7.6 and 7.7 depict the semantic choreography specifications of the goal and the Web Service respectively.

Listing 7.6: Goal choreography specification for semantic authentication

```
1  // Local ontology (will be stored on a separate file after
       deployed)
2  /*
3  myUserName('Riccardo').
4  myPassword('98765').
5  */
6  myGoal:Goal.
7  myGoal[
8     importOntology -> '../Auth/GoalsOntology.flr',
9
10    capability -> ${
11        pre -> ${
12           ar:AuthenticationRequest[UserName->?UN, Password->?PW],
13           AuthenticationRequest(UserName,?UN),
14           AuthenticationRequest(Password,?PW)
15        },
16
17        post -> ${
18           ?AV:AuthenticationValidation[?X->?Y] \or
19           AuthenticationValidation() \or
20           AuthenticationValidation(?Z) \or
21           AuthenticationValidation(?W,?S)
22        }
23    },
24    gRule(R01):ForallRule -> ${
25       \if (
26          ?X:QuestionByWS[UserName->?Y]@WM,
27          myUserName(?UN)@WM
28       )
```

```
29        \then (
30          %deltaInsert(${A:AnswerByGoal[UserName->?UN]})
31        )
32     },
33     gRule(R02):ForallRule -> ${
34        \if (
35           ?X:QuestionByWS[Password->?Y]@WM,
36           myPassword(?PW)@WM
37        )
38        \then (
39          %deltaInsert(${A:AnswerByGoal[Password->?PW]})
40        )
41     },
42     gRule(R03):ForallRule -> ${
43        \if (
44           QuestionByWS(UserName,?X)@WM,
45           myUserName(?UN)@WM
46        )
47        \then (
48          %deltaInsert(${AnswerByGoal(UserName,?UN)})
49        )
50     },
51     gRule(R04):ForallRule -> ${
52        \if (
53           QuestionByWS(Password,?X)@WM,
54           myPassword(?PW)@WM
55        )
56        \then (
57          %deltaInsert(${AnswerByGoal(Password,?PW)})
58        )
59     }
60  ].
```

Listing 7.7: Web Service choreography specification for semantic authentication

```
1  // Local ontology (will be stored on a separate file after
       deployed)
2  /*
3  DB_UserName_Password('PeterJM','12345').
4  DB_UserName_Password('Angel83','112233').
5  DB_UserName_Password('Riccardo','98765').
6  DB_UserName_Password('Agent007','7777777').
7  */
8  AuthenticationService:WebService.
9  AuthenticationService[
10     importOntology -> '../Auth/WebServicesOntology.flr',
11
12     capability -> ${
13         pre -> ${
14            ?AR:AuthenticationRequest[?X->?Y]
15         },
16
17         post -> ${
18            ?AV:AuthenticationValidation[?X->?Y]
19         }
20     },
21     wsRule(R01):ForallRule -> ${
22        \if (
23           (?AR:AuthenticationRequest[?X->?Y])@WM
```

```
24          )
25          \then (
26             %deltaInsert(${Q:QuestionByWS[UserName->_]}),
27             %deltaInsert(${Q:QuestionByWS[Password->_]})
28          )
29       },
30       wsRule(R02):ForallRule -> ${
31          \if (
32             (?X:AnswerByGoal[UserName->?UN])@WM,
33             (?Y:AnswerByGoal[Password->?PW])@WM,
34             DB_UserName_Password(?UN,?PW)@WM
35          )
36          \then (
37             %deltaInsert(${av:AuthenticationValidation[
38                                UserName->?UN, Password->?PW]})
39          )
40       },
41       wsRule(R03):ForallRule -> ${
42          \if (
43             (?X:AnswerByGoal[UserName->?UN])@WM,
44             (?Y:AnswerByGoal[Password->?PW])@WM,
45             (\+ DB_UserName_Password(?UN,?PW))@WM
46          )
47          \then (
48             %deltaInsert(${M:Message[
49                               WS->'Incorrect username or password
                                     .']})
50          )
51       }
52    ].
```

In this example, precondition of the goal contains two presentations of the request: one in the form of the frame `ar:AuthenticationRequest[UserName->?UN, Password->?PW]` and one in the form of the predicate pair `AuthenticationRequest(UserName,?UN), AuthenticationRequest(Password,?PW)`. Also, the postcondition can be satisfied in different ways. On the other side, the Web Service only understands the requests in the form of frame. Similar to Flight Reservation Service, the choreography is progressed through a semantic conversion between the goal and Web Service.

Example 2: Using utility predicates

A Web Service can offer some utility predicates to its requesters, so requesters can utilize these predicates in their choreography specifications. The signature of the predicate should be known to the requester at the time of writing specification. These predicates can be imported into the common ontology with static access; so both the Web Service and goal can utilize them. Here, we show the Flight Reservation System example utilizing the utility predicate `%cheapestFlight` to select the cheapest roundtrip flight among the suggested ones. The predicate is defined as below in the common ontology. It uses Flora-2 *min* operator to find the minimum value among the values which are unified with `?priceTot`:

```
    %cheapestFlight(?price) :-
?price = min {?priceTot | tripChoice(?fl_dep,?fl_ret,?priceTot)@WM}.
```

Listings 7.8 and 7.9 contain the goal and Web Service choreography specifications respectively. The predicate `%cheapestFlight` is used in `gRule(R01)`.

Listing 7.8: Goal choreography specification for flight reservation (using customized utility predicates)

```
1  myGoal:Goal.
2  myGoal[
3     importOntology -> '../Flight/GoalsOntology.flr',
4     capability -> ${
5         pre -> ${
6            myRequest:RequestFlight[
7                 From->'Paris',
8                 To->'Chicago',
9                 Departure->23,
10                Return->30] },
11
12        post -> ${ ?R:Reservation[?X->?Y] }
13    },
14    gRule(R01):ForallRule -> ${
15       \if
16          ( %cheapestFlight(?P),(tripChoice(?fl_dep,?fl_ret,?P))
               @WM )
17       \then (
18          %deltaInsert(${ trip:Trip[Dep->?fl_dep,Ret->?fl_ret]}) )
                },
```

```
19     gRule(R02):ForallRule -> ${
20        \if (
21             (?Q:QuestionByWS[
22                   Name->?X,
23                   DateOfBirth->?Y,
24                   Gender->?Z])@WM,
25                  (Name(?N),DateOfBirth(?DoB),Gender(?G))@WM
26        )
27        \then (
28           %deltaInsert(${answer:AnswerByGoal[
29                                Name->?N,
30                                DateOfBirth->?DoB,
31                                Gender->?G]}) ) },
32     gRule(R03):ForallRule -> ${
33        \if (
34             (?Q:QuestionByWS[
35                   CreditCardNo->?X,
36                   CreditCardHolder->?Y,
37                   CreditCardCVV->?Z])@WM,
38                  (CreditCardNo(?CCN),
39                        CreditCardHolder(?CCH),
40                        CreditCardCVV(?CCCVV))@WM
41        )
42        \then (
43           %deltaInsert(${answer:AnswerByGoal[
44                                CreditCardNo->?CCN,
45                                CreditCardHolder->?CCH,
46                                CreditCardCVV->(?CCCVV)]}) ) }
47  ].
```

Listing 7.9: Web Service choreography specification for flight reservation (using customized utility predicates)

```
1  FlightReservationService:WebService.
2  FlightReservationService[
3     importOntology -> '../Flight/WebServicesOntology.flr',
4     capability -> ${
5         pre -> ${ ?Req:RequestFlight[?X1->?Y1] },
6         post -> ${ (?Res:Reservation[?X2->?Y2]) }
7     },
8     wsRule(R01):ForallRule -> ${
9        \if (
10          (?R:RequestFlight[From->?X,To->?Y,Departure->?Z,Return
                ->?W])@WM,
11          (flight(?fl_dep,?X,?Y,?Z,?priceDep))@WM,
12          (flight(?fl_ret,?Y,?X,?W,?priceRet))@WM,
13          (%sum(?priceDep,?priceRet,?priceTot))
14       )
15       \then (
16          %deltaInsert(${tripChoice(?fl_dep,?fl_ret,?priceTot)}) )
                 },
17     wsRule(R02):ForallRule -> ${
18        \if
19           (?T:Trip[Dep->?fl_dep,Ret->?fl_ret])@WM
20        \then (
21           %deltaInsert(${question:QuestionByWS[
22                                Name->?X,
23                                DateOfBirth->?Y,
```

```
24                              Gender->?Z]}) ) },
25     wsRule(R03):ForallRule -> ${
26        \if
27           (?A:AnswerByGoal[
28                 Name->?X,
29                 DateOfBirth->?Y,
30                 Gender->?Z])@WM
31        \then (
32           %deltaInsert(${question:QuestionByWS[
33                              CreditCardNo->?XX,
34                              CreditCardHolder->?YY,
35                              CreditCardCVV->?ZZ]}) ) },
36     wsRule(R04):ForallRule -> ${
37        \if
38           (?A:AnswerByGoal[
39                 CreditCardNo->?X,
40                 CreditCardHolder->?Y,
41                 CreditCardCVV->?Z])@WM
42        \then (
43           %deltaInsert(${validation:CreditCardValidation[
44                              Number->?X,Holder->?Y,CVV->?Z]})
45        )
46     },
47     wsRule(R05):ForallRule -> ${
48        \if (
49             (YesNoAnswer('Yes'))@WM,
50             (trip:Trip[Dep->?fl_dep,Ret->?fl_ret])@WM
51        )
52        \then (
53           %deltaInsert(${reservation:Reservation[
54                              Number->11100,
55                              Flight1->?fl_dep,
56                              Flight2->?fl_ret]}) ) },
57     wsRule(Bank_R01):ForallRule -> ${
58        \if (
59             (?R:CreditCardValidation[
60                    Number->?X,
61                    Holder->?Y,
62                    CVV->?Z])@WM,
63            (DB_CreditCard(?X,?Y,?Z))@WM
64        )
65        \then (
66           %deltaInsert(${YesNoAnswer('Yes')})
67        )
68     }
69  ].
```

Example 3: Shipwire example

Shipwire [5] is "*a Fortune 100 company helping brands expand to new markets all around the world with [its] technology platform and distribution centers in more than 45 countries*". The company has authenticated open API facilities for developers. As a tutorial of working with APIs, Shipwire provides a scenario consisting of step-by-step REST requests and responses showing how a developer can accomplish a shipment order. This scenario is semi-choreographic (because it involves human specific behavior as well) and is described below:

1. *The app checks stock availability of products and updates its catalog.*
2. *Audrey browses the catalog and picks items to fill her shopping cart.*
3. *When Audrey is ready, she proceeds to checkout, where she must select among different shipment rates and services.*
4. *After Audrey confirms checkout, the app places a shipping order with Shipwire.*
5. *Audrey realizes her order is one item short, so she modifies the original order and checks out again. The app updates the order with Shipwire.*
6. *Once Shipwire ships Audrey's packages, the app automatically sends Audrey a shipping confirmation email with her tracking number.*

Here, we show how such choreography scenario can also be modeled by our choreography specification (Listings 7.10 and 7.11). To make the presentation clearer, we removed un-essential details; however, the original scenario can be modeled in full in the same way.

Listing 7.10: Goal choreography specification for the Shipwire usecase

```
1  // Local ontology (will be stored on a separate file after
       deployed)
2  /*
3  myUserName('Audrey').
```

```
4  myAuth('TG9vayBhdCB0aGF0OyBEdWNrcy4uLm9uIGEgbGFrZSEK').
5  myAddress('6501 Railroad Avenue SE-Room 315').
6  WantToBuy('Laura-s_Lament',10).
7  */
8  myGoal:Goal.
9  myGoal[
10     importOntology -> '../Shipwire/GoalsOntology.flr',
11 
12     capability -> ${
13         pre -> ${
14              run:System[state->on]
15         },
16         post -> ${
17              RESPONSE:MESSAGE[
18                  PurchaseDone->?_X,
19                  TrackingNumber->?_Y
20              ]
21         }
22     }
23     ,
24     gRule(R01):ForallRule -> ${
25        \if (
26          (
27              (\+ g_Rule:Control[R01->off])@WM,
28              (run:System[state->on])@WM,
29              myUserName(?user)@WM,
30              myAuth(?auth)@WM,
31              WantToBuy(?sku,?_quan)@WM
32          )
33        )
34        \then (
35            %deltaInsert(${g_Rule:Control[R01->off]}),
36           %deltaInsert(${REQUEST:MESSAGE[
37                      TYPE->'Search',
38                      SKU->?sku,
39                      USER->?user,
40                      AUTH->?auth
41                  ]})
42        )
43     }
44     ,
45     gRule(R02):ForallRule -> ${
46        \if (
47          (
48              (\+ g_Rule:Control[R02->off])@WM,
49              (RESPONSE:MESSAGE[
50                  Available->'TRUE',
51                  ProductID->?PrdId
52              ])@WM,
53              myAuth(?auth)@WM,
54              WantToBuy(?_prod,?quan)@WM
55          )
56        )
57        \then (
58            %deltaInsert(${g_Rule:Control[R02->off]}),
59           %deltaInsert(${REQUEST:MESSAGE[
60                              TYPE->'Order',
61                              ORDER->?PrdId,
```

```
62                              QUANTITY->?quan,
63                              AUTH->?auth
64                          ]})
65        )
66     }
67     ,
68     gRule(R03):ForallRule -> ${
69        \if (
70           (
71              (\+ g_Rule:Control[R03->off])@WM,
72              (RESPONSE:MESSAGE[
73                  AcceptOrder->'TRUE',
74                  ProductID->?PrdId
75              ])@WM,
76              myAuth(?auth)@WM,
77              WantToBuy(?_prod,?quan)@WM
78           )
79        )
80        \then (
81           %deltaInsert(${g_Rule:Control[R03->off]}),
82           %deltaInsert(${REQUEST:MESSAGE[
83                              TYPE->'Checkout',
84                              ORDER->?PrdId,
85                              QUANTITY->?quan,
86                              AUTH->?auth
87           ]})
88        )
89     }
90     ,
91     gRule(R04):ForallRule -> ${
92        \if (
93           (
94              (\+ g_Rule:Control[R04->off])@WM,
95              (RESPONSE:MESSAGE[
96                      Question->'ADDRESS'
97              ])@WM,
98              myAddress(?add)@WM
99           )
100       )
101       \then (
102          %deltaInsert(${g_Rule:Control[R04->off]}),
103          %deltaInsert(${REQUEST:MESSAGE[
104                                 TYPE->'Answer',
105                                 Data->?add
106          ]})
107       )
108    }
109
110 ].
```

Listing 7.11: Web Service choreography specification for the Shipwire usecase

```
1  // Local ontology (will be stored on a separate file after
      deployed)
2  /*
3  Product('Laura-s_Lament','1361533',20).
4  User('Audrey', 'TG9vayBhdCB0aGF0OyBEdWNrcy4uLm9uIGEgbGFrZSEK').
5  */
6
7  Shipwire:WebService.
8  Shipwire[
9     importOntology -> '../Shipwire/WebServicesOntology.flr',
10
11    capability -> ${
12        pre -> ${
13           REQUEST:MESSAGE[?_X1->?_Y1]
14        },
15
16        post -> ${
17           RESPONSE:MESSAGE[?_X2->?_Y2]
18        }
19    }
20    ,
21    wsRule(R01):ForallRule -> ${
22       \if (
23             (\+ ws_Rule:Control[R01->off])@WM,
24             REQUEST:MESSAGE[
25                 TYPE->'Search',
26                 SKU->?sku,
27                 USER->?user,
28                 AUTH->?auth
29             ]@WM,
30             User(?user, ?auth)@WM,
31             Product(?sku,?PrdId,?_Quan)@WM
32       )
33       \then (
34          %deltaInsert(${ws_Rule:Control[R01->off]}),
35          %deltaInsert(${RESPONSE:MESSAGE[
36                                Available->'TRUE',
37                                ProductID->?PrdId
38                            ]})
39       )
40    }
41    ,
42    wsRule(R02):ForallRule -> ${
43       \if (
44             (\+ ws_Rule:Control[R02->off])@WM,
45             REQUEST:MESSAGE[
46                 TYPE->'Order',
47                 ORDER->?PrdId,
48                 QUANTITY->?quanReq,
49                 AUTH->?auth
50             ]@WM,
51             Product(?_sku,?PrdId,?Quan)@WM,
52             ?Quan >= ?quanReq
53       )
54       \then (
55          %deltaInsert(${ws_Rule:Control[R02->off]}),
56          %deltaInsert(${RESPONSE:MESSAGE[
```

```
57                               AcceptOrder->'TRUE',
58                               ProductID->?PrdId
59                           ]})
60         )
61      }
62      ,
63      wsRule(R03):ForallRule -> ${
64         \if (
65               (\+ ws_Rule:Control[R03->off])@WM,
66               REQUEST:MESSAGE[
67                   TYPE->'Checkout',
68                   ORDER->?PrdId,
69                   QUANTITY->10,
70                   AUTH->'
                         TG9vayBhdCB0aGF0OyBEdWNrcy4uLm9uIGEgbGFrZSEK'
71               ]@WM
72         )
73         \then (
74            %deltaInsert(${ws_Rule:Control[R03->off]}),
75            %deltaInsert(${RESPONSE:MESSAGE[
76                               Question->'ADDRESS'
77                           ]})
78         )
79      }
80      ,
81      wsRule(R04):ForallRule -> ${
82         \if (
83               (\+ ws_Rule:Control[R04->off])@WM,
84               REQUEST:MESSAGE[
85                   TYPE->'Answer',
86                   Data->?_address
87               ]@WM
88         )
89         \then (
90            %deltaInsert(${ws_Rule:Control[R04->off]}),
91            %deltaInsert(${RESPONSE:MESSAGE[
92                               PurchaseDone->'TRUE',
93                               TrackingNumber->'1234567890'
94                           ]})
95         )
96      }
97   ].
```

In this example, each rule can only be fired once. We enforce this restriction by putting proper flags in the rules' left-sides. If a rule gets fired, subsequent firing of the same rule is prevented since the flag will be false after the first firing. Step 5 of Shipware scenario denotes the situation where the user decides to change her/his mind. Although such decisions are normally made by a human agent, they can also be modeled by our specifications. For example, a predicate like `UserChangeMind('TRUE')`

can be inserted as a pre-condition at the beginning and by using the above mentioned flagging technique, we can give only one chance to the goal to update its choice. Another way is to define a random function that activates a proper condition, simulating the user's change of mind.

Example 4: Using well-known vocabularies (schema.org)

Flora-2 inherently supports defining the membership relation. The frame `A:C[B->1]` means that object A is a member of class C and its B attribute has value 1. In general, membership relation is defined by the `:` operator. This feature is very suitable to link Flora-2 concepts to well-known vocabularies like the one available on *schema.org*. For example, we can define the flight predicate used in the previous examples as an instance of `http://schema.org/Flight` by `flight:'http://schema.org/Flight'`. This relation can be stored in the common ontology and can be validated during the choreography runs. Here we rewrite the authentication example with *schema.org* annotations. Table 7.3 shows the concepts which can be mapped to the current version of schema.org vocabulary. Listings 7.12 and 7.13 show the authentication goal and Web Service specifications respectively using the *schema.org* vocabulary.

Table 7.3: *schema.org* mapping

Concept	*schema.org* type
QuestionByWS	http://schema.org/Question
AnswerByGoal	http://schema.org/Answer
myUserName	http://schema.org/identifier
myPassword	http://schema.org/accessCode
UserName	http://schema.org/identifier
Password	http://schema.org/accessCode
Message	http://schema.org/Message
Goal	http://schema.org/agent
WebService	http://schema.org/Service

Listing 7.12: Goal choreography specification for semantic authentication (*schema.org* annotated)

```
1  // Local ontology (will be stored on a separate file after
       deployment)
2  /*
3  myUserName('Riccardo').
4  myPassword('98765').
5  */
6  myGoal:Goal.
7  myGoal[
```

```
8      importOntology -> '../Auth/GoalsOntology.flr',
9
10     capability -> ${
11         pre -> ${
12            ar:AuthenticationRequest[UserName->?UN, Password->?PW],
13            AuthenticationRequest(UserName,?UN),
14            AuthenticationRequest(Password,?PW)
15         },
16
17         post -> ${
18            ?AV:AuthenticationValidation[?X->?Y] \or
19            AuthenticationValidation() \or
20            AuthenticationValidation(?Z) \or
21            AuthenticationValidation(?W,?S)
22         }
23     },
24     gRule(R01):ForallRule -> ${
25        \if (
26           ?X:QuestionByWS[UserName->?Y]@WM,
27           myUserName(?UN)@WM,
28           QuestionByWS:'http://schema.org/Question',
29           myUserName:'http://schema.org/identifier'
30        )
31        \then (
32          %deltaInsert(${A:AnswerByGoal[UserName->?UN]})
33        )
34     },
35     gRule(R02):ForallRule -> ${
36        \if (
37           ?X:QuestionByWS[Password->?Y]@WM,
38           myPassword(?PW)@WM,
39           QuestionByWS:'http://schema.org/Question',
40           myPassword:'http://schema.org/accessCode'
41        )
42        \then (
43          %deltaInsert(${A:AnswerByGoal[Password->?PW]})
44        )
45     },
46     gRule(R03):ForallRule -> ${
47        \if (
48           QuestionByWS(UserName,?X)@WM,
49           myUserName(?UN)@WM,
50           QuestionByWS:'http://schema.org/Question',
51           myUserName:'http://schema.org/identifier'
52        )
53        \then (
54          %deltaInsert(${AnswerByGoal(UserName,?UN)})
55        )
56     },
57     gRule(R04):ForallRule -> ${
58        \if (
59           QuestionByWS(Password,?X)@WM,
60           myPassword(?PW)@WM,
61           QuestionByWS:'http://schema.org/Question',
62           myUserName:'http://schema.org/identifier'
63        )
64        \then (
65          %deltaInsert(${AnswerByGoal(Password,?PW)})
```

```
66        )
67     }].
```

Listing 7.13: Web Service choreography specification for semantic authentication (*schema.org* annotated)

```
 1 // Local ontology (will be stored on a separate file after
      deployment)
 2 /*
 3 DB_UserName_Password('PeterJM','12345').
 4 DB_UserName_Password('Angel83','112233').
 5 DB_UserName_Password('Riccardo','98765').
 6 DB_UserName_Password('Agent007','7777777').
 7 */
 8 AuthenticationService:WebService.
 9 AuthenticationService[
10    importOntology -> '../Auth/WebServicesOntology.flr',
11
12    capability -> ${
13        pre -> ${
14           ?AR:AuthenticationRequest[?X->?Y]
15        },
16
17        post -> ${
18           ?AV:AuthenticationValidation[?X->?Y]
19        }
20    },
21    wsRule(R01):ForallRule -> ${
22       \if (
23          (?AR:AuthenticationRequest[?X->?Y])@WM
24       )
25       \then (
26          %deltaInsert(${Q:QuestionByWS[UserName->_]}),
27          %deltaInsert(${Q:QuestionByWS[Password->_]})
28       )
29    },
30    wsRule(R02):ForallRule -> ${
31       \if (
32          (?X:AnswerByGoal[UserName->?UN])@WM,
33          (?Y:AnswerByGoal[Password->?PW])@WM,
34          DB_UserName_Password(?UN,?PW)@WM,
35          AnswerByGoal:'http://schema.org/Answer',
36          UserName:'http://schema.org/identifier',
37          Password:'http://schema.org/accessCode'
38       )
39       \then (
40          %deltaInsert(${av:AuthenticationValidation[
41                                               UserName->?UN,
42                                               Password->?PW]})
43       )
44    },
45    wsRule(R03):ForallRule -> ${
46       \if (
47          (?X:AnswerByGoal[UserName->?UN])@WM,
48          (?Y:AnswerByGoal[Password->?PW])@WM,
49          (\+ DB_UserName_Password(?UN,?PW))@WM,
50          AnswerByGoal:'http://schema.org/Answer',
51          UserName:'http://schema.org/identifier',
52          Password:'http://schema.org/accessCode'
```

```
53          )
54          \then (
55             %deltaInsert(${M:Message[
56                            WS->'Incorrect username or password.']})
57          )
58       }
59    ].
```

Example 5: HTTP messages

As mentioned before REST APIs are very popular. Here, we show how HTTP messages can be encoded to Flora-2 choreography specifications. In Section 7, we demonstrated a choreographic scenario provided by the Shipwire website [5]. We abstractly showed how this scenario can be modeled with the choreography specification. At a lower level, REST APIs are called in single step request/response pairs and because HTTP is a stateless protocol, control flow is managed at higher levels.

Here, we show how a sample REST request/response pair can be modeled by the Flora-2 choreography specification. The sample GET request/response taken from TIBCO ® API Exchange Gateway [6] and is shown below:

Request	http://localhost:8090/tpmRest/v1/participants/transports/all?participantName=partner1&protocolName=EZComm
Response	{"result":[{"name":"file","type":"FILE"},{"name":"http","type":"HTTP"}]}

We use the JSON to Flora-2 conversion presented in Appendix C to specify the JSON response provided by the Web Service. Listings 7.14 and 7.15 show the goal and Web Service specifications. We use the attribute ID to make correlation between the request and response messages.

Since there is only one request/response step, the request method can be also defined in *goal.pre*. In this form, goal will be rule-free however.

Listing 7.14: Goal choreography specification for semantic GET request

```
1  myGoal:Goal.
2  myGoal[
3    importOntology -> '../Tibco/GoalsOntology.flr',
4
5    capability -> ${
6      pre -> ${ run:System[state->on] },
7      post -> ${
```

```
8          RESPONSE:'HTTP_1.1_MESSAGE'[
9            ID->123,
10           STATUS->200,
11           BODY->?format[content -> ?top(?topId)]
12         ]
13     }
14   }
15   ,
16   gRule(R01):ForallRule -> ${
17     \if (
18     (
19       (\+ g_Rule:Control[R01->off])@WM,
20       (run:System[state->on])@WM
21     )
22     )
23     \then (
24       %deltaInsert(${g_Rule:Control[R01->off]}),
25       %deltaInsert(${
26          REQUEST:'HTTP_1.1_MESSAGE'[
27              ID->123,
28              METHOD->'GET',
29              URI->'http://localhost:8090/tpmRest/v1/
30                                   participants/transports/all',
31                 PARAMS->participantName('partner1'),
32                 PARAMS->protocolName('EZComm')
33                 // , HEADER->Accept('text/json'),
34                 // BODY->Optional
35             ]})
36     )
37   }
38 ].
```

Listing 7.15: Web Service choreography specification of semantic GET response

```
1  Tibco:WebService.
2  Tibco[
3    importOntology -> '../Tibco/WebServicesOntology.flr',
4
5    capability -> ${
6      pre -> ${ REQUEST:'HTTP_1.1_MESSAGE'[?_X1->?_Y1] },
7
8      post -> ${ RESPONSE:'HTTP_1.1_MESSAGE'[?_X2->?_Y2] }
9    }
10   ,
11   wsRule(R01):ForallRule -> ${
12     \if (
13       (\+ ws_Rule:Control[R01->off])@WM,
14       REQUEST:'HTTP_1.1_MESSAGE'[
15             ID->?id,
16             METHOD->'GET',
17             URI->?uri,
18             PARAMS->?_param(?val)
19             // , HEADER->Optional,
20             // BODY->Optional
21       ]@WM
22       // , call the appropriate function with
23       // the provided parameters
24     )
25     \then (
```

```
26        %deltaInsert(${ws_Rule:Control[R01->off]}),
27        %deltaInsert(${RESPONSE:'HTTP_1.1_MESSAGE'[
28                ID->?id,
29                STATUS->200,
30                BODY->json[content->object(obj_01)],
31
32                object(obj_01,'result')-> array(arr_01),
33
34                array(arr_01,1)->object(obj_02),
35
36                object(obj_02,'name')->'file',
37                object(obj_02,'type')->'FILE',
38
39                array(arr_01,2)->object(obj_03),
40
41                object(obj_03,'name')->'http',
42                object(obj_03,'type')->'HTTP'
43              ]})
44      )
45    }
46  ].
```

Appendix G: The source code of timing evaluation and the output raw data

Choreography specification generator (C#)

Listing 7.16: Choreography specification generator

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.IO;
6
7  namespace RuleGenerator
8  {
9    class Program
10   {
11     static StreamWriter srGoal = new StreamWriter("./Benchmarking
          /Choreography/Bench/Goals.flr");
12     static StreamWriter srWS = new StreamWriter("./Benchmarking/
          Choreography/Bench/WebServices.flr");
13
14     static void Main(string[] args)
15     {
16       WriteLineGoal("myGoal:Goal.\r\nmyGoal[\r\n   importOntology
            -> \'C:/Users/ShAhin MPA/Desktop/Benchmarking/
           Choreography/Bench/GoalsOntology.flr\',\r\n");
17       WriteLineWS("myService:WebService.\r\nmyService[\r\n
           importOntology -> 'C:/Users/ShAhin MPA/Desktop/
           Benchmarking/Choreography/Bench/WebServicesOntology.flr
           ',\r\n");
18
19       int pre = 1;
20       int post = (args != null && args.Length > 0 ? Int32.Parse(
           args[0]) : 10);
21
22       int i = pre;
23
24       WriteLineGoal("   capability -> ${pre -> ${obj:Concept[
           attr_" + i + "->val_" + i + "]}, post -> ${obj:Concept[
           attr_" + post + "->val_" + post + "]}},");
25       WriteLineWS("   ${pre -> ${?OBJ:Concept[?_X1->?_Y1]}, post
           -> ${?OBJ:Concept[?_X2->?_Y2]}},");
26
27       for (i = pre; i < post; i++)
28       {
29         WriteLineWS("   wsRule(R" + i + "):ForallRule -> ${ \\if
             (obj:Concept[attr_" + i + "->val_" + i + "]@WM) \\then
              (%deltaInsert(${obj:Concept[attr_" + (i + 1) + "->
             val_" + (i + 1) + "]})) }" + ((i < (post - 1)) ? "," :
              ""));
30         i++;
31         if (i >= (post - 1) )
32         {
33           break;
```

```
34          }
35          WriteLineGoal("   gRule(R" + (i - 1) + "):ForallRule -> $
                { \\if (obj:Concept[attr_" + i + "->val_" + i + "]@WM)
                 \\then (%deltaInsert(${obj:Concept[attr_" + (i + 1) +
                 "->val_" + (i + 1) + "]})) }" + ((i < (post - 2)) ?
                "," : ""));
36        }
37        WriteLineGoal(@"].");
38        WriteLineWS(@"].");
39        srGoal.Close();
40        srWS.Close();
41      }
42
43      static void WriteLineGoal(string str)
44      {
45        //Console.WriteLine(str);
46        srGoal.WriteLine(str);
47      }
48
49      static void WriteLineWS(string str)
50      {
51        //Console.WriteLine(str);
52        srWS.WriteLine(str);
53      }
54    }
55  }
```

Changes to `Choreography.flr`

Here, the partial view of the file `Choreography.flr` (Listing 7.2) altered for benchmarking is given. Lines 22 to 36 in Listing 7.2 are replaced with Lines 1 to 23 of Listing 7.17 below.

The predicate `%duration(?S2,?MS2,?S1,?MS1,?DS,?DMS)` (Lines 29 to 33 of the Listing 7.17 below) is added to the file `Choreography.flr` to compute the time needed for the choreography engine to run the given specifications.

Listing 7.17: Partial view of `Choreography.flr`

```
1  /* (1) start*/
2  %start(?goal,?WS) :-
3    %debug(on),
4    %initializations,
5    %preProcessCheckings(?goal,?WS),
6
7    %prepareModule(WM),
8    %prepareModule(DeltaWM),
9
10   %importOntology(?goal,WM),
11   %importOntology(?WS,WM),
12
13   %insertGoalPre(?goal,WM),
14
15   //Benchmarking
16   epoch_milliseconds(?S1,?MS1)@\prolog(machine),
17
18   %runChoreography(?goal,?WS),
19
20   epoch_milliseconds(?S2,?MS2)@\prolog(machine),
21     %duration(?S2,?MS2,?S1,?MS1,?DS,?DMS),
22   ?Filename = 'C:/Users/ShAhin MPA/Desktop/Benchmarking/
       Choreography/Bench/out.txt', ?Filename[open(append,?Stream)]
       @\io, ?Stream[writeln([?DS,?DMS])]@\io, ?Stream[close]@\io.
23   //%watchln(['Duration:'-?DS-?DMS]).
24 /*--------------------------------------------------*/
25
26 /* Other predicates omitted */
27
28 // %duration(?X) returns the difference between two second-based
     times
29 %duration(?S2,?MS2,?S1,?MS1,?DS,?DMS) :-
30   ?DMS_temp \is ?MS2 - ?MS1,
31   \if (?DMS_temp < 0)
32   \then (?DS \is ?S2 - ?S1 - 1, ?DMS \is ?DMS_temp + 1000)
33   \else (?DS \is ?S2 - ?S1, ?DMS \is ?DMS_temp).
```

The raw data for the experiments done in Chapter 6

Output of the code given in Listing 7.2 for the experiment done in Section 6.2 is shown below. The numbers are written in the file out.txt; a sample snapshot of this file is shown below in Figure 7.9. The first number (2) represents seconds and the second number (528) represents milliseconds.

	1st run	2nd run	3rd run
When number of rules is 10:	[2, 528]	[2, 621]	[2, 558]
When number of rules is 20:	[10, 717]	[11, 466]	[10, 702]
When number of rules is 30:	[24, 897]	[24, 711]	[24, 274]
When number of rules is 40:	[43, 150]	[44, 585]	[43, 446]
When number of rules is 50:	[67, 501]	[68, 322]	[66, 971]

Output of the code given in Listing 7.2 for the experiment done in Section 6.3:

	1st run	2nd run	3rd run
When number of rules is 10:	[1, 389]	[1, 393]	[1, 342]
When number of rules is 20:	[3, 420]	[2, 995]	[3, 730]
When number of rules is 30:	[4, 914]	[4, 992]	[4, 961]
When number of rules is 40:	[6, 921]	[6, 958]	[6, 880]
When number of rules is 50:	[9, 204]	[9, 173]	[9, 469]

Output of the code given in Listing 7.2 for the experiment done in Section 6.4:

	1st run	2nd run	3rd run
When # of terms in LHS and RHS is 1:	[2, 528]	[2, 621]	[2, 558]
When # of terms in LHS and RHS is 2:	[4, 870]	[5, 268]	[4, 968]
When # of terms in LHS and RHS is 4:	[9, 128]	[9, 379]	[9, 411]
When # of terms in LHS and RHS is 8:	[18, 135]	[17, 816]	[18, 450]

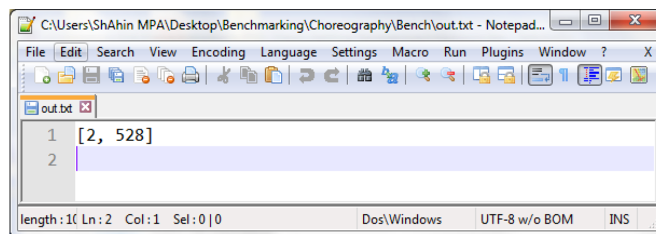

Figure 7.9: A snapshot of out.txt generated by the choreography engine