Dissertation

zur Erlangung des akademischen Grades

des Doktors der Naturwissenschaften

(Dr. rer. nat)

A Rule-Based Agent-Oriented Framework for Weakly-Structured Scientific Workflows

eingereicht
am Institut für Informatik
des Fachbereichs Mathematik und Informatik
der Freien Universität Berlin
von

Zhili Zhao

Berlin, 2014

Gutachter:

Prof. Dr. Adrian Paschke
Department of Computer Science
Freie Universität Berlin

Prof. Dr. Hans Weigand
Department of Information Systems and Management
Tilburg University

Tag der Disputation: 12. September 2014

ERKLÄRUNG/DECLARATION

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Berlin, 29 July, 2014

Zhili Zhao

Abstract

Existing solutions for business workflows as well as scientific workflows mainly focus on the orchestrated and pre-structured execution of compute-intensive and data-oriented tasks. On the contrary, this thesis explicitly considers Weakly-structured Scientific Workflows (WsSWFs), which contain goal-oriented tasks that have to make agile runtime decisions. They may involve interactions between multiple participants or have complicated logic to express scientific policies and cater to dynamic execution environments. In general, such WsSWFs not only need a rich process and (domain-specific) decision logic specification, but also require a flexible execution and human interaction.

The main research problem addressed in this thesis is the combination of the rule-based knowledge representation with the agent technology for the purpose of supporting the WsSWF execution from a technical perspective, and a Rule-based Agent-oriented Framework (RbAF) is proposed.

The first challenge is to describe workflows by declarative rules. This thesis employs messaging reaction rules, which go beyond global Event-Condition-Action (ECA) rules and support performing complex actions locally within certain contexts. Based on messaging reaction rules, the RbAF offers an event-driven architecture and models complex workflow patterns with the rule-based Complex Event Processing (CEP) technologies. In addition, a Concurrent Transaction Logic (CTR)-based formal semantics which precisely defines the rule-based workflow language is presented.

The second challenge is the description of (domain-specific) decision logic in workflows. This thesis addresses the problem by exploiting benefits of both Logic Programming (LP) and Description Logic (DL). LP with derivation rules is more expressive than typical Boolean expressions and also more understandable for domain experts. Moreover, the RbAF provides three ways to access domain-specific data encoded by Semantic Web technologies.

The third challenge is to support the flexibility required by the WsSWFs. Besides the rule-based process and decision logic specification, the RbAF employs distributed rule-based agents as the workflow execution environment and supports asynchronous interaction between distributed agents. Moreover, the RbAF combines two ways of the workflow composition: orchestration and choreography. Another flexible mechanism is to handle workflow exceptions at runtime based on a workflow ontology structuring workflow resources.

One further challenge addressed in this thesis is to integrate human users into the workflow execution. This thesis uses a human agent, which manages the life cycle of human tasks and provides a Web interface for domain experts to operate on human tasks. Human interaction also helps in handling exceptions that cannot be automatically handled by the rule-based agents.

This thesis evaluates the RbAF from different perspectives. In contrast to three prominent scientific workflow systems, the rule-based workflow specification of this thesis shows higher expressive power with respect to the workflow patterns that are important for scientific workflows. With respect to domain knowledge representation, the analysis results indicate that general (domain-specific) decision logic in the WsSWFs can be represented by normal logic programs, which support negation and are more expressive than propositional and finite logic programs. An expressive query language for DL is employed and different reasoners can be easily configured in the RbAF to reason domain ontologies with different expressivity. In terms of an empirical evaluation, the RbAF supports most of the typical properties of computational models, including different forms of execution cycles, non-deterministic execution branches, parallel and concurrent execution, distributed computation and asynchronous communication. Moreover, an experimental evaluation based on three real-world WsSWF use cases is also given to analyze the performance and demonstrate the expressive power of the domain knowledge representation in the RbAF. This thesis concludes that the RbAF provides both an expressive workflow description and a flexible workflow execution environment, and meets requirements of the WsSWFs (except provenance).

Keywords: scientific workflows; weakly-structured processes; multi-agent systems; semantic web; logic programming; event-driven execution; user interaction

Acknowledgments

I would like to express my sincere gratitude to everyone who contributed to the completion of this thesis. All of you made my study in Berlin so wonderful!

First and foremost, I would like to thank Prof. Dr. Adrian Paschke for supervising me in the past four years. Thanks for all your suggestions for my presentations and papers. Without your advice and patience, this thesis would have never been possible. I really appreciate everything you have done for me.

I am pleased to have Prof. Dr. Hans Weigand as the second examiner of my thesis. I was impressed by your sincerity and amiability when I saw you at the VMBO workshop in 2014 for the first time. I also would like to express my gratitude to Prof. Dr. Ruisheng Zhang. Although I left your group after my master study, you still gave me a lot of valuable advice during my PhD study.

I am thankful to my present and past group members from all over the world for interesting discussions and for sharing with me PhD student experience; Kia Teymourian, Ralph Schäfermeier, Alexandru Todor, Shashishekar Ramakrishna, Mohammed Almashraee, Marko Harasic, Mario Rothe, Gökhan Coskun, Ralf Heese, Markus Luczak-Rösch and Olga Streibel. In particular, I thank Shashishekar Ramakrishna for numerous hours of technical discussions on weekends.

Furthermore, I am thankful to Jialu Hu and Hui Yu for providing significant use cases to evaluate my work. Without your patient explanation I could not identify these use cases from your research domains and use them in my work. I really appreciate it.

I also would like to thank my friends who I knew before and during my PhD study; Dr. Lili Jiang, Zhen Dong, Dr. Jiazao Lin, Dr. Yi Yang, Kai Zhang, Hong Zhang, Bin Zhang, Ting He, Tao Liao, Miaomiao Zhu, Yubin Zhao, Yuan Yang, Dr. Rongjing Hu, Fan Ding, Jiaxuan Wei and Dr. Xiaoliang Fan. It is amazing to know you and our friendship is the most precious wealth in my life! In particular, I thank Dr. Jiazao Lin for encouraging me a lot in the most difficult time of my PhD study.

I also would like to give my gratitude to the financial support of China Scholarship Council (CSC) and Corporate Semantic Web (CSW) work group.

I am grateful to my family, especially to my parents, for always supporting and encouraging me.

Finally, I deep gratitude my girlfriend Ying Li for always being there, although we were not together most of the time.

Zhili Zhao, in Berlin

Contents

List of Figures xiii

List of Tables xv

List of Abbreviations xvii

1 Introduction 1
- 1.1 Problem Statement 1
- 1.2 Research Questions 2
- 1.3 Research Methodology 3
- 1.4 Thesis Contributions 4
- 1.5 Literature Connections 6
- 1.6 Thesis Outline 7

I Background and State-of-the-Art 9

2 Background 11
- 2.1 Scientific Workflows 11
- 2.2 Scientific Workflow Life Cycle 13
- 2.3 Workflow Management Systems 14
- 2.4 Scientific vs. Business Workflows 15
- 2.5 Weakly-Structured Scientific Workflows 18
  - 2.5.1 Structured vs. Unstructured Processes 18
  - 2.5.2 WsSWF Examples 19
  - 2.5.3 WsSWF Features 26
  - 2.5.4 WsSWF Main Requirements 27
- 2.6 Workflow Description-Related Technologies 27
- 2.7 Imperative vs. Declarative Programming 29
- 2.8 Logic Program Overview 31
- 2.9 Deductive, Abductive and Inductive Reasoning 35
- 2.10 Summary 36

3 Flexible Workflow Compositions 39
- 3.1 Classic Workflow Languages 39
- 3.2 Agent-Oriented Workflow Compositions 41
- 3.3 Rule-Based Workflow Languages 44
- 3.4 Main Scientific Workflow Languages 46
- 3.5 Efforts on Weakly-Structured Workflows 48
- 3.6 Semantic-Based Workflow Compositions 49
  - 3.6.1 Semantic Web Services 50

3.6.2 Ontology-Based Workflow Specifications 52
3.7 Summary 55

II Conceptual Framework 57

4 Rule-Based Agent-Oriented Framework 59
4.1 Hierarchy of the Rule-Based Workflow Specification 62
4.2 Upper-Level Workflow Ontology 62
4.3 Declarative Workflow Specification 64
4.3.1 Reaction Rules 64
4.3.2 Event-Driven Workflow Execution 67
4.3.3 CEP-Based Workflow Pattern Modeling 68
4.4 Domain Decision-Centric Logic Description 72
4.4.1 Derivation Rules 72
4.4.2 Semantic Web Data Query 75
4.5 Integrating Orchestration with Choreography 78
4.6 Human Interaction 80
4.7 Exception Handling 82
4.8 Summary 83

5 Formal Workflow Representation 85
5.1 Workflow Model 85
5.2 CTR Overview 87
5.3 Workflow Representation Using CTR 91
5.3.1 Workflow Representation 91
5.3.2 Multiple Instances 94
5.3.3 Reactive Logic Representation 94
5.4 Communication between Processes 96
5.5 Complex Event Processing 97
5.6 Exception Handling 100
5.7 Summary 102

III Evaluation 103

6 Proof-of-Concepts 105
6.1 Prova 105
6.2 The Workflow Ontology 106
6.3 Mapping the CTR-Based Workflow Logic to Prova 108
6.4 Domain Logic Expression in Prova 113
6.5 Enterprise Service Bus Mule 116
6.5.1 Prova Agent Deployment 117
6.5.2 Mule ESB as Communication Middleware 119
6.5.3 Translations between Reaction RuleML and Prova 120

6.6 Exception Handling . . . 122
6.7 User Client . . . 124
6.7.1 Workflow Submission . . . 124
6.7.2 Exception Management . . . 125
6.7.3 Human Task Management . . . 126
6.7.4 RDF Data Management . . . 127
6.8 Summary . . . 127

7 Evaluation **129**
7.1 Workflow Pattern-Based Expressiveness Evaluation . . . 129
7.1.1 Control-Flow Patterns . . . 130
7.1.2 Data Patterns . . . 143
7.1.3 Scientific Workflow Patterns . . . 148
7.2 Evaluation of the Domain Knowledge Representation . . . 150
7.2.1 LP-based Knowledge Representation Evaluation . . . 150
7.2.2 DL-based Knowledge Representation Evaluation . . . 152
7.3 Computational Model-Based Empirical Evaluation . . . 156
7.4 Use Case-Based Experimental Evaluation . . . 158
7.4.1 Protein Prediction Result Analysis . . . 159
7.4.2 Snow Depth Data Screening . . . 163
7.4.3 Ant Identification and Treatment . . . 165
7.5 System Performance Evaluation . . . 169
7.5.1 Message Passing Overhead . . . 169
7.5.2 System Concurrency . . . 170
7.6 Summary . . . 171

IV Conclusion **173**

8 Conclusion and Outlook **175**
8.1 Summary . . . 175
8.2 Outlook . . . 176

V Appendix **179**

A Zusammenfassung **181**

B About the Author **183**

Bibliography **185**

List of Figures

1.1 The General Methodology of Design Research 4

2.1 Scientific Workflows 12
2.2 Scientific Workflow Life Cycle 13
2.3 Workflow Management System 15
2.4 Control Flow vs. Data Flow 17
2.5 Structured vs. Unstructured Processes 18
2.6 Weakly-Structured Scientific Workflows 20
2.7 The Atrophy Computation Method Stages for Multiple Sclerosis . . . 20
2.8 Process of Treating a Newly Discovered Ant 21
2.9 Building Snow Depth Forecast Model from Remote Sensing Data . . 22
2.10 Ant Identification 23
2.11 Protein Prediction Result Analysis 24
2.12 GO Term Ancestor Chart 25
2.13 Workflow Description-Related Technologies 28
2.14 Classes of Logic Programs 32
2.15 Logic Program Negations 33

3.1 Agent-Based Scientific Workflow Composition 42
3.2 Subjects and Communications in Holiday Application Process 43
3.3 The Upper Ontology of AWDL 53
3.4 Top Level of OWL-S Process Ontology 54

4.1 Rule-Based Agent-Oriented Scientific Workflow Framework 60
4.2 Hierarchy of Rule-Based Workflow Description 62
4.3 Upper-Level Workflow Ontology 63
4.4 Multiple Workflow Instances 66
4.5 Event-Driven Workflow Execution 67
4.6 Process of Implementing the AND Join Connector 69
4.7 Example of an AND Join Connector Implementation 71
4.8 Process of Implementing the OR Join Connector 72
4.9 Domain Knowledge-Intensive Decision with Derivation Rules 73
4.10 Semantic Web Data Query 76
4.11 Integrating Orchestration with Choreography 79
4.12 Integrating Humans into Scientific Workflows 80
4.13 Exception Handling in Event-Driven Scientific Workflows 83

5.1 A General Workflow Process Model 86

6.1 Protein Prediction Analysis Workflow Ontology 107
6.2 Domain Ontology: UniProt Core Vocabulary 114

6.3 SPARQL-DL Query Engine 115
6.4 Mule Application Flow 117
6.5 Class Diagram of ProvaUMOImpl 118
6.6 Mule-Based Workflow Architecture 120
6.7 Human Interaction Client 124
6.8 Human Agent Proxy 127

7.1 Expressiveness and Complexity of OWL Family 154
7.2 Complexity of Gene Ontology 161
7.3 Gene Ontology Reasoning Analysis 162
7.4 Concurrency with Increasing Number of Workflow Requests 171

List of Tables

2.1 Structured vs. Unstructured Processes 19

3.1 Main Scientific Workflow Languages 48
3.2 Comparison of Flexible Workflow Composition Solutions 56

6.1 Mapping CTR-Based Workflow Representation to Prova Rules . . . 113

7.1 Control-Flow Pattern-Based Comparison 142
7.2 Data Pattern-Based Comparison 147
7.3 Scientific Workflow Pattern-Based Comparison 149
7.4 Comparison of Reasoners Implementing OWL API 155
7.5 Data Sets of Communication Overhead Evaluation 170

List of Abbreviations

Abbreviations

ACL Agent Communication Language

AGWL Abstract Grid Workflow Language

AOP Aspect-Oriented Programming

API Application Programming Interface

ASP Answer Set Programming

BPEL Business Process Execution Language for Web Services

BPM Business Process Management

BPMN Business Process Model and Notation

CA Condition Action

CEP Complex Event Processing

CPAL Common Public Attribution License

CTR Concurrent Transaction Logic

DAX Directed Acyclic Graph in XML Format

DL Description Logic

ECA Event-Condition-Action

EDA Event-Driven Architecture

EDIT European Distributed Institute of Taxonomy

EHA Exception Handling Agent

EPC Event-driven Process Chain

ESB Enterprise Service Bus

GIS Geographic Information System

GO Gene Ontology

GUI Graphical User Interface

HA Human Agent

IDE Integrated Development Environment

IDL Interface Description Language

JMS Java Message Service

LAN Local Area Network

LP Logic Programming

LSST Large Synoptic Survey Telescope

LTL Linear Temporal Logic

MAP Multi-Agent Protocol

MAS Multi-Agent System

MoML XML-based Modeling Markup Language

NaF Negation as Failure

OGSA Open Grid Services Architecture

OMG Object Management Group

OPM Open Provenance Model

OWL Web Ontology Language

OWL-S OWL for Services

OWL-WS OWL for Workflows and Services

PSL Process Specification Language

QoS Quality of Service

RAWLS Rule-based Agent Workflow System

RbAF Rule-based Agent-oriented Framework

RBSLA Rule-Based Service Level Agreement

RDF Resource Description Framework

RDFS Resource Description Framework Schema

SAWSDL Semantic Annotations for WSDL and XML Schema

S-BPM Subject-oriented Business Process Management

SCUFL Simple Conceptual Unified Flow Language

SEDA Staged Event-Driven Architecture

SLA Service Level Agreements

SMS Stable Model Semantics

SOA Service-Oriented Architecture

SOAP Simple Object Access Protocol

SPARQL SPARQL Protocol and RDF Query Language

SWFMS Scientific Workflow Management System

SWRL Semantic Web Rule Language

SWSF Semantic Web Services Framework

SWSL Semantic Web Services Language

SWSO Semantic Web Services Ontology

TR Transaction Logic

UDDI Universal Description Discovery and Integration

UML Unified Modeling Language

URI Uniform Resource Identifier

WAN Wide Area Network

WDO Workflow-Driven Ontology

WfMC Workflow Management Coalition

WFMS Workflow Management System

WFS Well-Founded Semantics

WSDL Web Services Description Language

WSFL Web Services Flow Language

WSMF Web Service Modeling Framework

WSMO Web Service Modeling Ontology

WsSWF Weakly-structured Scientific Workflow

XPDL XML Process Definition Language

YAWL Yet Another Workflow Language

Chapter 1

Introduction

Contents

1.1 Problem Statement 1
1.2 Research Questions 2
1.3 Research Methodology 3
1.4 Thesis Contributions 4
1.5 Literature Connections 6
1.6 Thesis Outline 7

1.1 Problem Statement

Scientific workflows have attracted more and more interest in recent years, as science becomes increasingly reliant on the analysis of massive data sets and the use of distributed resources [1]. They assist scientists to perform data management, analysis and simulation of *in silico experiments* [2]. Compared with business workflows which are already supported by competing specifications and Business Process Management (BPM) standards, scientific workflows have not been widely adopted and supported yet. One significant reason is that scientific workflows have extra requirements over their counterparts in the business domain, such as explicit data/information flow, exact reproducibility, agility to quickly adapt to changed knowledge and human/machine decisions, team cooperation for distributed problem solving [3]. To address such requirements, existing business workflow technologies need to be thoroughly adapted and extended. Furthermore, existing solutions for business workflows as well as scientific workflows mainly focus on structured compute-intensive and data-oriented tasks, instead of decision-centric tasks that need the cooperation of scientists or computer agents as a team supported by weakly-structured workflows.

A WsSWF is a process, in which there are complex decision-centric tasks that require agile runtime decisions during their execution; they may involve interactions between multiple participants or have complicated logic to express scientific policies and cater to dynamic execution environments; they could be modeled at a high abstract level with standard graphical workflow representation tools (e.g., Business Process Model and Notation (BPMN)), but the inherent complex and flexible behavior during the task execution cannot be easily implemented. In the current

state-of-the-art, there are partial solutions that have been proposed for some of the aforementioned issues, such as increasing the flexibility of service composition [4, 5], incorporating knowledge tasks and objects into workflow models [6]. Nevertheless, some core issues of the WsSWFs are still unsolved. Compared with the structured computational scientific workflows, the WsSWFs focus on knowledge-intensive tasks and require:

(i) *Rich Process Specification*: the WsSWFs contain complex decision-centric tasks, which require processes to handle new and exceptional situations. Besides simple control flow descriptions (e.g., a task is enabled after the completion of a preceding task), it is also necessary to describe advanced process logic, which needs dynamic recognition of operational as well as knowledge-based states to implement intelligent routings at runtime.

(ii) *Expressing Domain-Specific Policies*: the WsSWFs often involve complex domain-specific policies, which regulate the behavior of scientific applications. In order to automate the WsSWFs, it is necessary to express such scientific policies and enable machines to deal with them automatically.

(iii) *Flexibility*: the structured processes suffer from limitations with respect to dynamic evolution and adaptation at runtime. In order to provide high flexibility, the WsSWFs should be allowed to be easily modified according to individual situations.

(iv) *Human Interaction*: scientific workflow systems are often designed to automate scientific processes and improve their operational efficiency. However, human users still need to conduct manual tasks and steer the workflow execution to deal with unforeseen problems at runtime.

(v) *Exact Reproducibility*: provenance plays an important role in verification, explanation, reproduction and informed reuse of data used and produced by scientific workflows, especially by the WsSWFs, which have non-deterministic decision logic (However, provenance is a broad standalone topic in itself and is out of the scope of this work).

1.2 Research Questions

This thesis mainly focuses on the execution phase of the scientific workflow life cycle and proposes a rule-based, agent-oriented framework, called RbAF, with a purpose of explicitly supporting the WsSWF execution.

On one hand, an agent-based framework can provide a flexible execution environment. On the other hand, declarative rules provide a declarative programming style to specify the agent behavior. The combination of them offers a promising approach to support the WsSWFs. In this thesis, four research questions are answered:

(i) **How to specify the WsSWF process logic?** ECA rules are the most common rules used to specify workflows. ECA rules react on occurred events by executing actions and are usually defined with a global scope in the knowledge base of a reactive system (e.g., in active databases). However, scientific workflows are usually executed in certain cooperation contexts rather than in global event occurrences. Moreover, it is known that reaction rules, especially ECA rules can specify basic workflow processes, but, are they expressive enough to specify the WsSWF process logic?

(ii) **How to express (domain-specific) decision logic and integrate it into the process logic?** Most of the rule-based workflow languages mainly focus on the process logic but ignore the expression of the decision points determining the execution paths at runtime. Moreover, the WsSWFs often involve domain policies regulating such decisions, which are made in terms of knowledge-intensive decision criteria and may involve multiple sub-decisions. Therefore, the (domain-specific) decision logic needs to be expressed and integrated to the process logic.

(iii) **How to support an adaptive workflow execution?** The adaptability denotes to which extent workflow processes are allowed to be automatically or manually modified according to changed situations at runtime [7]. To achieve this, two sub-questions need to be addressed: (1) How to dynamically discover appropriate resources used to perform a task according to current circumstances at runtime? (2) How to implement flexible mechanisms to handle exceptions at runtime, such as dynamic replacement of an exceptional resource?

(iv) **How to support asynchronous communication between human users and the workflow system?** Although some scientific workflow systems (e.g., Taverna [8]) can invoke Web services and hence it would be possible to wrap human behavior by Web services, none of existing scientific workflow systems provides features to specify human tasks in workflows [3]. To support user interaction, firstly it is necessary to provide a well-defined human task specification. Moreover, what users need are not only integrating them into the workflow execution, but also asynchronous interaction with the workflow system, especially when performing long running activities, such as discussions and exhaustive knowledge searches.

1.3 Research Methodology

This thesis follows a general design science research methodology [9], which offers specific guidelines for building and evaluating the utility of information system research artifacts. Figure 1.1 shows a general cycle of design science research adapted from [10]. Every design starts with an interesting problem (*Awareness of Problem*), which may come from developments in industry or in a reference discipline. The

output of this phase is a (formal or informal) **Proposal** for a new research effort. The *Suggestion* phase is an essentially creative step wherein a **Tentative Design** is envisioned based on a novel configuration of either existing or new and existing elements. It is intimately connected with **proposal** as the dotted line indicates. During the *Development* phase, an **Artifact** is implemented in terms of the suggestions in the previous phase. The artifact could be seen as an implementation of **Tentative Design**. The implementation is then evaluated (in the *Evaluation* phase) according to **Measures** implicitly or explicitly made in the *Suggestion* phase. The *Conclusion* phase is the last step of a specific research effort and gives **Results** of the research. As shown in Figure 1.1, *Development*, *Evaluation* and *Suggestion* are iteratively performed during the course of a research effort. The basis of the iteration, i.e., the flow from partial completion of the cycle back to *Awareness of the Problem*, is indicated by the Circumscription arrow.

After identifying the problems of the WsSWFs (see Sections 1.1 and 1.2), and this thesis further proposes an overall conceptual workflow framework based on the combination of the declarative programming using rules with the agent technology (see Chapter 4 and 5). Afterwards, this thesis presents a design artifact in Chapter 6 to support the WsSWF execution. The evaluation and conclusion are then given in Chapters 7 and 8, respectively.

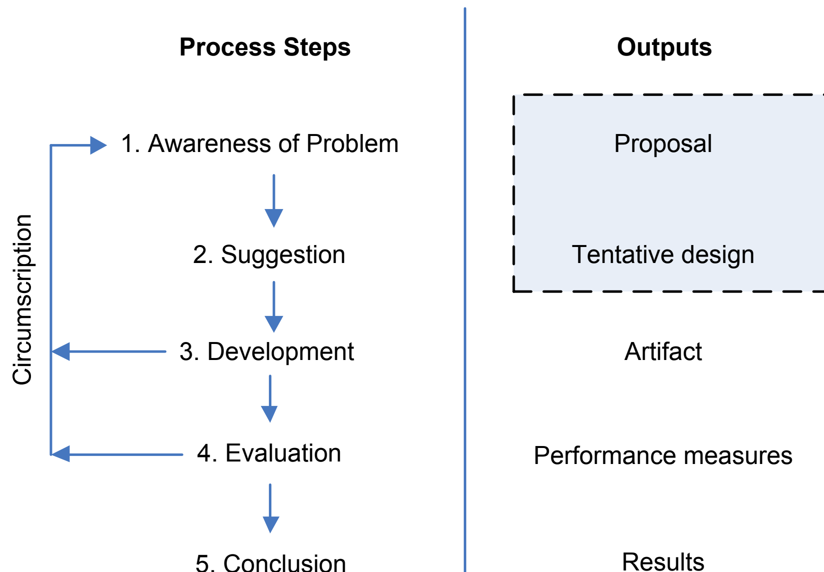


Figure 1.1: The General Methodology of Design Research

1.4 Thesis Contributions

For the purpose of supporting the WsSWF execution, this thesis proposes the RbAF, which exploits the benefits of both the declarative programming using rules and the

agent technology. In particular, the major contributions of this thesis are as follows:

(i) *An expressive rule-based language for describing the WsSWFs.* From a technical perspective, this thesis provides an expressive rule-based workflow specification, which combines reaction and derivation rules to describe complex reactive and decision logic of the WsSWFs. Moreover, a CTR-based formal semantics which precisely defines the rule-based workflow language is presented. According to the workflow pattern-based evaluation (see Section 7.1), the rule-based workflow specification of this work can not only describe basic workflow patterns, but also support advanced workflow patterns that are not fully supported by other scientific workflow systems.

(ii) *Domain-specific decision logic expression combining logic programs with description logic.* This thesis gives a hybrid approach, which exploits the benefits of both DL and LP to express complicated (domain-specific) decision logic. The decision logic encoded as declarative derivation rules also can integrate external domain-specific Semantic Web data, which gives domain semantics or even pragmatic meanings to the concepts involved.

(iii) *Adaptive workflow execution.* The declarative rules specify the agent behavior and make it possible to dynamically replace exceptional resources by reasoning the workflow ontology. Moreover, the RbAF combines two ways of the workflow composition: orchestration and choreography. The agents can be process-agnostic and are employed to execute part of a complex workflow, and they also can execute choreography workflows via conversation-based messaging reaction rules.

(iv) *Asynchronous user interaction.* The RbAF supports two types of activities that require user interaction: the human tasks which are performed manually, and the unexpected exceptions that cannot be automatically handled by the workflow systems. To support the asynchronous user interaction, a human agent manages the life cycle of user interaction and provides a Web interface for users to operate on manual activities asynchronously.

Compared to the existing workflow solutions, this thesis explicitly considers the WsSWFs from a technical perspective, and highlights the following major aspects:

(i) *Abstraction* via a distributed multi-agent model reflecting the semiotic structure of scientific teams in a distributed choreography style of the workflow execution and distributed problem-solving.

(ii) *Complex decision logic* via derivation rules and logical inference deductions beyond the typical restricted expressiveness of simple gateways in process execution models.

(iii) *Situation-awareness and behavioral dynamic reactions* via reaction rules leading to dynamic and agile workflow reaction patterns.

(iv) *Semantic workflow execution* via domain models and information models represented as ontologies which are integrated into the workflow execution semantically.

(v) *Decoupled* via event messages enabling asynchronous communication and parallel processing, non-deterministic execution branches of problem solving tasks in distributed agents.

(vi) *Extending the range of workflow applications* via combining the benefits of both orchestration and choreography, i.e., maintaining the overall workflow execution in a centralized way, while complex decision-centric tasks can be performed by a group of collaborative agents sharing the same goal.

(vii) *Asynchronous user interaction* via the asynchronous communication enabling users to operate on manual tasks or handle unexpected exceptions.

1.5 Literature Connections

Several publications were achieved in the course of accomplishing this thesis. A general overview of this thesis was accepted by the PhD Symposium co-located with 16th International Conference on Business Information Systems in 2013.

(i) Zhili Zhao and Adrian Paschke, "A Rule-Based Agent Framework for Weakly-Structured Scientific Workflows", in *Proceedings of Business Information Systems Workshops*, pp. 290-301, 2013

The following publications are mainly about the rule-based, agent-oriented framework, RbAF, presented in Chapter 4 of this thesis. The RbAF was introduced from two different perspectives: *event-driven workflow execution* and *rule-based, agent-oriented execution*. Several real-world WsSWF use cases are demonstrated in such publications.

(ii) Zhili Zhao and Adrian Paschke, "Rule Agent-Oriented Scientific Workflow Execution", in *Proceedings of the 5th International Conference on Subject-Oriented Business Process Management*, pp. 109-122, 2013

(iii) Zhili Zhao and Adrian Paschke, "Event-Driven Scientific Workflow Execution", in *Proceedings of Business Process Management Workshops*, pp. 390-401, 2012

(iv) Zhili Zhao and Adrian Paschke, "A Semantic Multi-Agent System for Intelligent and Adaptive Scientific Workflows", in *Proceedings of the 4th International Workshop on Semantic Web Applications and Tools for the Life Sciences*, pp. 123-124, 2011

The formal semantics of the rule-based workflow language presented in Chapter 5 was published in proceedings of the SWAT4LS workshop in 2013.

(v) Zhili Zhao and Adrian Paschke, "A Formal Model for Weakly-Structured Scientific Workflows", in *Proceedings of the 6th International Workshop on Semantic Web Applications and Tools for the Life Sciences*, 2013

The following publications are about *SymposiumPlanner*—a series instances of *Rule Responder*. *Rule Responder* is a rule-based agent framework for specifying virtual organizations and provides a preliminary architecture of the RbAF in this thesis.

(vi) Zhili Zhao, Adrian Paschke, Chaudhry Usman Ali and Harold Boley, "SymposiumPlanner: Querying Two Virtual Organization Committees", in *Proceedings of RuleML 2011@BRF Challenge*, pp. 125-132, 2011

(vii) Zhili Zhao, Adrian Paschke, Chaudhry Usman Ali and Harold Boley, "Principles of The SymposiumPlanner Instantiations of Rule Responder", in *Proceedings of the 5th International Conference on Rule-based Modeling and Computing on the Semantic Web*, pp. 97-111, 2011

(viii) Zhili Zhao, Kia Teymourian, Adrian Paschke, Harold Boley and Tara Athan, "Loosely-Coupled and Event-Messaged Interactions with Reaction RuleML 1.0 in Rule Responder", in *Proceedings of RuleML 2012 Challenge*, 2012

Another related publication is about *Reaction RuleML* , which is used as an interchangeable rule format between the RbAF and its client.

(ix) Adrian Paschke, Harold Boley, Zhili Zhao, Kia Teymourian and Tara Athan, "Reaction RuleML 1.0: Standardized Semantic Reaction Rules", in *Proceedings of the 6th International Symposium on Rules on the Web: Research and Applications*, pp. 100-119, 2012

1.6 Thesis Outline

After identifying the challenges of scientific workflows, this chapter presented the research questions, methodology and contributions of this thesis. Further chapters of this thesis are organized as follows:

Chapter 2 introduces the background information of scientific workflows. In particular, the WsSWF features and requirements are detailed through specific use cases. Moreover, this chapter presents the basic information of declarative programming and classes of logic programs.

Chapter 3 presents the state-of-the-art on different solutions with a purpose of improving the flexibility of both business workflows and scientific workflows, such as classical workflow languages, agent-oriented workflow compositions, rule-based languages, existing scientific workflow languages and using Semantic Web technologies in the workflow composition. Most of these efforts focus on orchestrated and pre-structured workflows with a purpose of providing efficient and reliable processes to users, rather than the WsSWFs considered in this thesis.

Chapter 4 presents the design of the conceptual workflow framework, the RbAF. This chapter describes a declarative rule-based workflow language to specify complex reactive and decision logic at a lower level. With the benefits of the declarative rules and the agent technology, the framework supports the dynamic and adaptive workflow execution from different aspects. In addition, the RbAF also integrates human behavior into the workflow execution.

Chapter 5 details a formal semantics of the rule-based workflow language of the RbAF based on CTR. The formal semantics provides a mathematical workflow representation and focuses on conversation-based reactive logic representation, (data) event-driven CEP and the communication between processes.

Chapter 6 introduces the implementation of the RbAF—a Rule-based Agent Workflow System (RAWLS). As a proof-of-concept, a Web rule language Prova is adapted to represent knowledge-intensive scientific logic as semantic rules, wrapped in the agents, and to support message-driven conversation-based interactions between the rule-based agents.

Chapter 7 evaluates the RbAF from different perspectives. First, based on control-flow and data patterns proposed by Van der Aalst et al., the rule-based workflow language is evaluated for the level of solving different types of tasks. Then the domain knowledge representation is evaluated from both LP and DL perspectives. Afterwards, an empirical evaluation of the RAWLS based on typical properties of computational models is presented. A system performance is also given at the end of the chapter.

Chapter 8 summarizes the achievements and highlights of this thesis. Finally, this chapter outlines directions for future work.

Part I

Background and State-of-the-Art

Chapter 2

Background

Contents

2.1 Scientific Workflows 11
2.2 Scientific Workflow Life Cycle 13
2.3 Workflow Management Systems 14
2.4 Scientific vs. Business Workflows 15
2.5 Weakly-Structured Scientific Workflows 18
2.5.1 Structured vs. Unstructured Processes 18
2.5.2 WsSWF Examples 19
2.5.3 WsSWF Features 26
2.5.4 WsSWF Main Requirements 27
2.6 Workflow Description-Related Technologies 27
2.7 Imperative vs. Declarative Programming 29
2.8 Logic Program Overview 31
2.9 Deductive, Abductive and Inductive Reasoning 35
2.10 Summary 36

2.1 Scientific Workflows

The workflow technology has been widely recognized to streamline a group of services to accomplish large and sophisticated goals. In the business domain, workflows automate and optimize an organization's processes fulfilled by human or computer agents in an administrative context [11], also known as business workflows. Nowadays, they have been used in numerous business domains, such as finance and banking, healthcare, telecommunications and office automation. For example, an e-business process of placing an order includes activities, such as order placing, processing, payment and arrangement of shipment. In the healthcare domain, the workflow technology manages care-providing tasks that involve direct interactions between healthcare specialists and customers (e.g., adjusting a client's drip or medication), and administrative tasks that are more related to the financial part of a customer's situation, like the registration of a customer's personal details and insurance related data [12]. They control business processes with an aim of providing better services for customers and reducing costs for business owners at the same time.

Scientific workflows can be regarded as an application of the workflow technology in the scientific domain. In recent years, science has experienced a step change in problem-solving ability brought about by the increasing digitization and automation of scientific instruments and practice, leading to a new era of science, also known as *e-Science* [13]. On one hand, scientific research activities have become to rely more and more on advanced information, computational and software techniques; there is a growing demand not only for computational tools and resources, but also for collaborations between scientists around the world. On the other hand, the Web has a significant impact on facilitating the practice of science and supports wide-scale information discovery and sharing, facilitating collaboration and enabling widespread participation in digital science [13]. The development of distributed computing technologies, especially the advent of Service-Oriented Architecture (SOA) and Web services, makes it possible to access and integrate resources on demand simply and transparently. Over the years, the distributed computing technologies have experienced an evolution from *Cluster Computing* closely connecting a group of loosely computers in local networks, *Grid Computing* providing a transparent and pervasive computing infrastructure, which integrates the resources (e.g., supercomputers, storage systems and data sources) over a Local Area Network (LAN), metropolitan or Wide Area Network (WAN), to the latest buzzing paradigm *Cloud Computing* driven by economies of scale, in which a pool of abstract, virtualized, dynamically scalable, managed computing power, storage, platforms, and services are delivered on demand to external customers over the Internet [14].

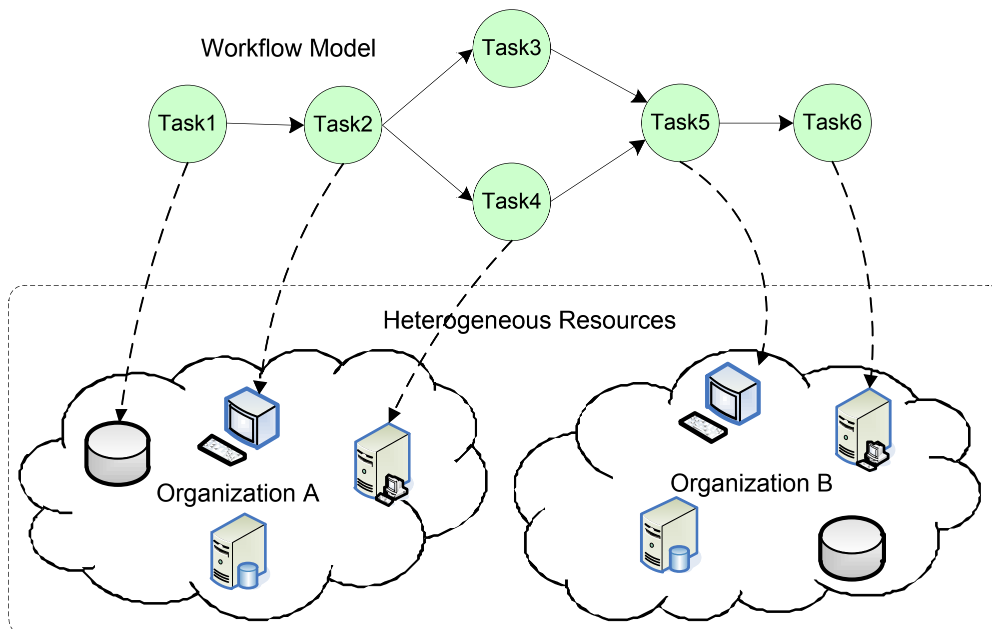


Figure 2.1: Scientific Workflows

Workflows represent a main programming model for the development of scientific applications on the distributed systems [15]. Scientific workflows can support scientists to perform large-scale and complex *e-Science* processes, which usually involve a group of small tasks, such as data management, calculation, analysis and representation. They enable scientists to streamline a group of small tasks into sophisticated ones and execute them systematically on distributed resources, as shown in Figure 2.1. In addition, scientific workflow systems provide the ability of automatically recording the provenance (or lineage) of intermediate and final data products generated during the workflow execution to support reproducibility, validity, and re-use of scientific experiments [16].

It should be noted that, although scientific workflows have been used in many domains, it does not mean that they are useful for all scientific experiments. Scientific workflows are usually employed for *in silico* experiments [2], where research is conducted via computer simulations with models closely reflecting the real world. In such simulations, an object is always the representation of a target system [17]. However, for some experiments that have to operate directly on target systems, scientific workflows may not be always helpful. In such cases, scientific workflows are often used as auxiliary tools in, such as data extraction, conversion and analysis.

2.2 Scientific Workflow Life Cycle

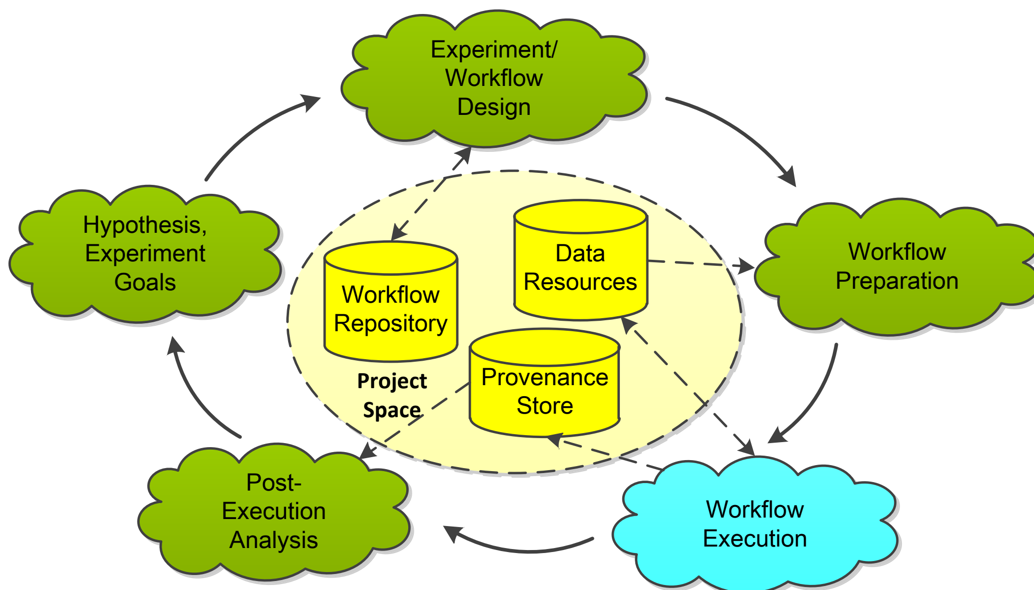


Figure 2.2: Scientific Workflow Life Cycle

Scientific workflows are exploratory in nature. Before converging on suitable parameters to define an experiment, scientific workflows are usually executed in *what-if* manners and involve an exploration of variants, and manipulation of different

workflow configurations [18]. In other words, the definition of a new workflow is usually based on existing ones, also known as scientific workflow reuse.

Figure 2.2 gives the scientific workflow life cycle adapted from [19]. A scientific workflow usually starts from a *scientific hypothesis or a specific goal*. During *the workflow design phase*, scientists often reuse existing workflows or templates from a public repository and refine them to meet their requirements. Required data sources are selected and parameters are set by scientists during *workflow preparation*. During the *workflow execution*, the processing history (i.e., provenance information) is simultaneously recorded. Scientists often evaluate workflow results in a *post-execution analysis phase*; they inspect the provenance information or compare the results with previous ones. Depending on the analysis results, the original goal may be revised and a new iteration of the cycle begins.

In essence, scientific workflows are modified as many times as possible until they produce expected outcomes. Therefore, it is crucial to provide a flexible design and allow the workflow logic to be revised easily. Moreover, the analysis of workflow results in terms of completed provenance information also plays an important role in the scientific workflow life cycle.

2.3 Workflow Management Systems

A Workflow Management System (WFMS) is a generic information system that supports modeling, execution, management and monitoring of workflows [20]. In 1995, the Workflow Management Coalition (WfMC) proposed a reference architecture (model) for business workflows, as shown in Figure 2.3.

The WfMC's workflow reference model identifies interfaces within the structure and contains a number of generic components which interact in a defined set of ways. Figure 2.3 shows its main features and the relationships between main functions. The top level of a WFMS consists of *build time* and *runtime*. At *build time*, workflows are defined via a text or graphical editor. A workflow definition specifies involved tasks and their dependencies. At *runtime*, a workflow engine executes it in terms of the workflow specification defined at *build time*. In general, the definition of a workflow determines the behavior of a workflow engine at *runtime*, and adaptive mechanisms of the workflow engine in turn require a flexible design at *build time*.

A Scientific Workflow Management System (SWFMS) is a specialized WFMS designed for scientific workflows. Since the WfMC's workflow reference model was proposed in 1995, the reference model and its variants have been widely adopted in the development of different business WFMSs, but none of these reference architectures is suitable for SWFMSs [21]. Such architectures are mainly used to simplify access, control and orchestration of remote resources (e.g., Web services), and they lack support for additional requirements of scientific workflows (see a comprehensive comparison between scientific and business workflows in Section 2.4).

There are different SWFMSs that have been developed during the past few years, some prominent ones, such as Kepler [22], Triana [23], Taverna [8], Trails

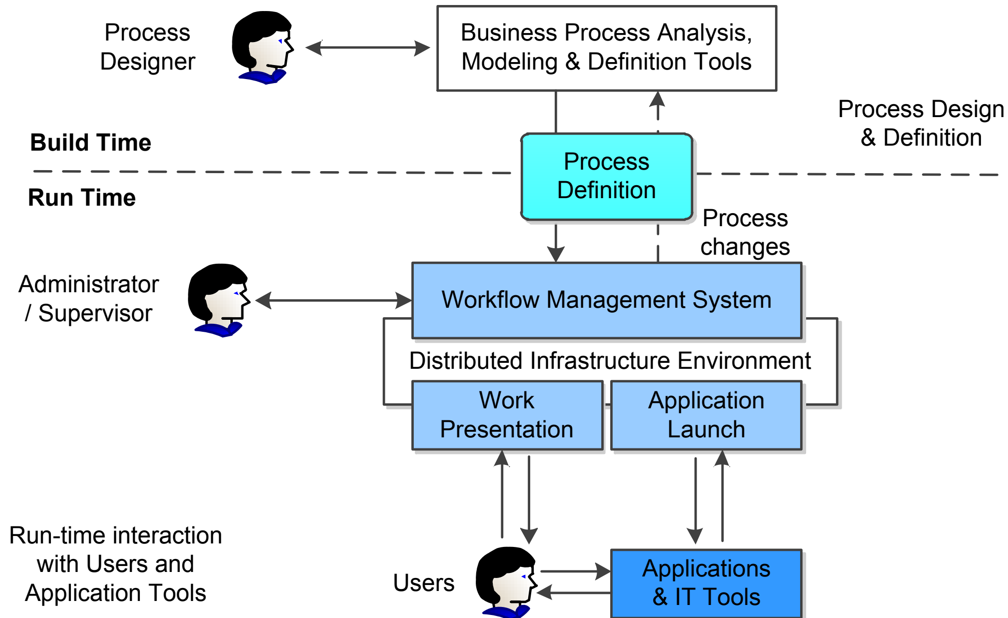


Figure 2.3: Workflow Management System

[24], Pegasus [25] and Swift [26]. However, these systems usually have their own proprietary frameworks, and an architecture which can be used as a reference model for future research and development is still missing [21]. It is worth noticing that proposing a scientific workflow reference model goes beyond the scope of this thesis. The purpose of this section is mainly to illustrate the interactive relationship between *build time* and *runtime* of a WFMS.

2.4 Scientific vs. Business Workflows

The formal concept of *workflow* has been around in the business domain for several decades. In 1996, the WfMC defined a workflow as:

"*The automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules.*"

Over the years, many competing specifications and standards of business workflows were proposed, some of which have been broadly accepted and used, superseding others [27]. In 1995, the WfMC firstly published a workflow reference model, which defines a WFMS and identifies the most important system interfaces (see above section). The WfMC also creates XML Process Definition Language (XPDL), which is an interchange business process definition between different workflow products. With the combination of IBM's Web Services Flow Language (WSFL) with Mi-

crosoft's XLANG, BEA Systems, IBM, Microsoft and other companies created Business Process Execution Language for Web Services (BPEL) and submitted BPEL 1.1 to OASIS for standardization in April 2003, followed by BPEL 2.0. BPEL [28] is a standard way of orchestrating Web service execution in the business domain. Compared with other business process languages, BPEL is supported by a great deal of well-designed tools, such as ActiveBPEL Designer, Oracle BPEL Process Manager, ActiveBPEL Engine, the BPWS4J Engine and Twister. In addition, the Object Management Group (OMG) develops BPMN, which is a popular graphical representation for specifying business processes in a business process model. As mentioned before, the existing business workflow tools cannot be directly reused to capture scientific workflows, but it is still valuable to compare them and obtain experience from the development of business workflows.

Scientific workflows have different goals with business workflows. Business workflows aim to automate and optimize an organization's processes fulfilled by human or computer agents in an administrative context. The main purpose of building a workflow for companies is to enable customers to make better use of its services and gain profits from them. Their main concerns are the workflow integrity and security. For example, the service providers of an electronic business platform must ensure that they provide comprehensive, powerful and customer-friendly transaction processes, and every transaction is conducted in a secure environment; a seller will not be notified to ship products if a customer's payment is unsuccessful. On the other hand, scientific workflows usually assist scientists in streamlining (domain-specific) knowledge-intensive activities in their experiments, especially in proving scientific goals or hypotheses. Such scientific processes are often exploratory in nature, with new analysis methods being rapidly evolved from initial ideas and preliminary workflow designs [19]. They are often executed in multiple *what-if* (aka. *trial-and-error*) manners, which may involve an exploration of variants, and manipulation of different workflow configurations—often leading to significant changes to a workflow as the experiment evolves to some useful outcomes [18, 29].

Also, the degree of flexibility that scientists have in their work is usually much higher than in the business domain, where business processes are usually predefined and executed in a routine fashion [29]. Driven by customers' demand, business workflows have to rely on fixed resources, and companies are willing to use their own resources to ensure their workflows running in a robust environment, even if some resources are expensive. However, scientific workflows are usually executed in distributed environments, where integrated resources are heterogeneous and evolving in nature. It is necessary to provide flexible policies to react to dynamic changes at runtime. For example, it is necessary to define a policy to find alternative resources if one requested resource is not available. Moreover, business workflows are usually constructed by professional business workflow engineers. However, scientists are workflow authors, who are experts in their specific domains and are not necessarily experts in information technology. It is necessary to hide the complexity of underlying distributed environments and provide users with friendly interfaces. All these factors demand a separate workflow definition independent with concrete

implementation details.

Business workflows are typically control flow-oriented and a task starts when its precedent tasks are completed (see Figure 2.4 (a)). However, scientific computations are usually more data flow-oriented and a task is executed when required operands are produced by its precedent task(s). In other words, what passes between workflow steps is not just control, but also data that flows between and through the connections from one task to another and that drives the computation [19] (see Figure 2.4 (b)).

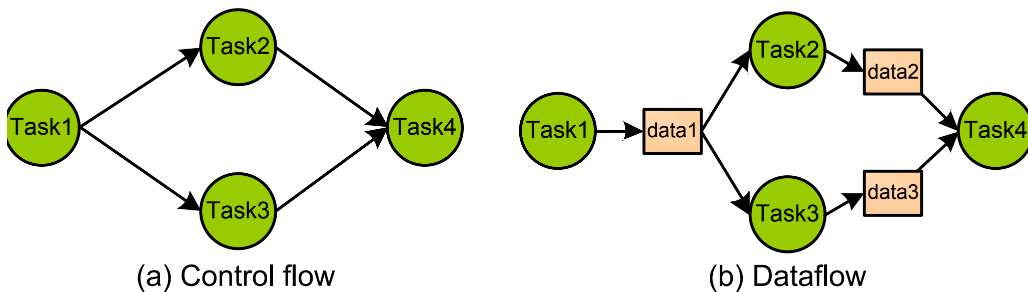


Figure 2.4: Control Flow vs. Data Flow

In business WFMSs, it is important that a service keeps promised functional and no-functional properties, and it is not important that how a service achieved its aims in terms of used resources, software [30]. On the contrary, after a scientific workflow completes, scientists often want to know the derivation history of a workflow in order to prove the hypothesis set up at the outset of the workflow. They may execute the workflow again to inspect if the same result could be produced once more, also known as scientific workflow reproduction. In other words, scientific workflows should be reproducible, and specific data products and tools used to generate workflow results have to be recorded. Provenance information provides scientists with an explanation of workflow execution and ensures that processes can be reproduced and extended.

As it can be noticed from the above comparison, such additional requirements of scientific workflows are the real challenges that prevent the development of scientific workflows. Currently there are two main ways to build a SWFMS: one solution is to reuse the workflow technologies in the business domain, such as [31, 32]. However, although a great number of commercial business workflow systems exist, the support for scientific experimenting is still rudimentary and the commercially available tools do not cover the whole life cycle of scientific experiments [33]. The other solution is to build SWFMSs from scratch, such as Kepler, Triana, Taverna, Trails, Pegasus and Swift. But these systems usually have their own proprietary frameworks and limited application domains.

2.5 Weakly-Structured Scientific Workflows

2.5.1 Structured vs. Unstructured Processes

From the process structure perspective, workflows could be categorized into *structured* and *unstructured* processes, as shown in Figure 2.5.

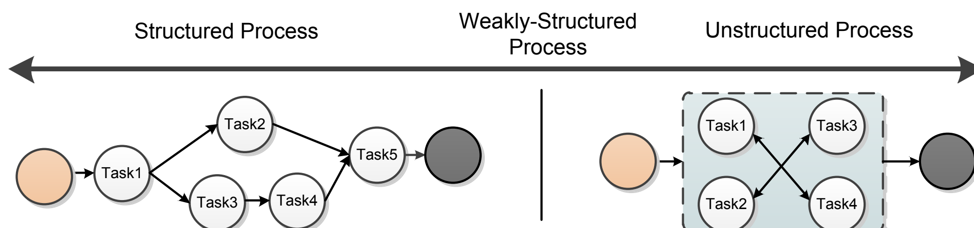


Figure 2.5: Structured vs. Unstructured Processes

The structured processes usually refer to a series of activities with a high degree of organization to improve operational efficiency and rein in costs through automation. They have routine processes, and activity dependencies are rigorously pre-defined in advance for the purpose of taking into account all process instance permutations at runtime. Once established, the structured processes undergo far fewer changes and are executed frequently with newly acquired datasets or varying parameter settings [19]. Moreover, they are traditionally executed in a centralized manner, which employs an efficient central engine to coordinate and schedule tasks of the processes. Scientific workflows which are *compute/data-intensive* fall into the structured ones, and their values lie in integrating distributed computational resources to process large volumes of data. For example, Virtual Screening is a promising approach to accelerate the drug development process, which is both data-intensive and compute-intensive; it analyses a large number of chemical compounds in order to identify those structures which are most likely to bind to a drug target [34]. Screening and further simulating each compound, depending on structural complexity, can take from one to a few minutes on a standard PC, which means screening a database with millions of chemical compounds can take years of computation time [35]. With the benefits of the workflow technology, it is possible to integrate distributed resources to perform parallel computations and make more efficient drug discovery process. Another type of scientific workflows in the scope of the structured processes is *administrative scientific workflows*, which refer to routine activities, such as managing data coming from instrument streams [29].

Whereas the unstructured processes are often goal-oriented and done with uncertainty. They have variable logic and need a flexible design to support dynamic adaptation at runtime. Also, human users are often involved to guide the workflow execution at runtime, and execution flows inside these processes might be modified by human users as needed. Moreover, compared with the structured processes, the unstructured processes often involve knowledge-intensive activities as well as interactions between multiple participants. Table 2.1 summarizes the differences between

the structured and unstructured processes.

However, it is difficult to find a real-world scientific process, which is completely structured or unstructured. The common ones are the processes that fall in between the structured and unstructured cases. Such processes contain activities that are either knowledge-intensive or exploratory but are vital to correct results. In this thesis, these processes are referred to as Weakly-structured Scientific Workflows (WsSWFs).

Table 2.1: Structured vs. Unstructured Processes

	Structured processes	**Unstructured Processes**
Goal	Improving efficiency	Goal oriented
Process logic	Routine process with static logic	Variable logic
Adaptability	Predefined in advance and undergone fewer changes	Dynamic modification
Service Composition	Centralized workflow execution	Collaboration between participants
Human Interaction	Fewer user interaction	Rich user interaction and workflow steering
Applications	Compute or data-intensive	Knowledge intensive/decision-centric

2.5.2 WsSWF Examples

The WsSWFs are intermediate ones in between the structured and unstructured processes. At the heart of these workflows there are tasks that are often done with uncertainties, but they are crucial parts of the whole workflows. These tasks may be decision-centric and involve exploration of variants or manipulation of different workflow configurations; they may be collaborative and involve interactions between multiple participants in different places; they could be knowledge-intensive and involve complex logic to express messy scientific policies or deal with unpredictable exceptions (or unforeseen scenarios) at runtime; moreover, human users might be involved to perform manual tasks, e.g., making decisions at runtime and steering the workflow execution by determining the order of workflow activities.

Therefore, the WsSWF is not one single workflow but an umbrella term used to generalize more specific workflows: exploratory workflows, collaborative workflows, knowledge-intensive workflows, event-driven workflows, interactive workflows and ad hoc workflows, as shown in Figure 2.6. In what follows, real-world use cases from different domains are employed to introduce these workflows.

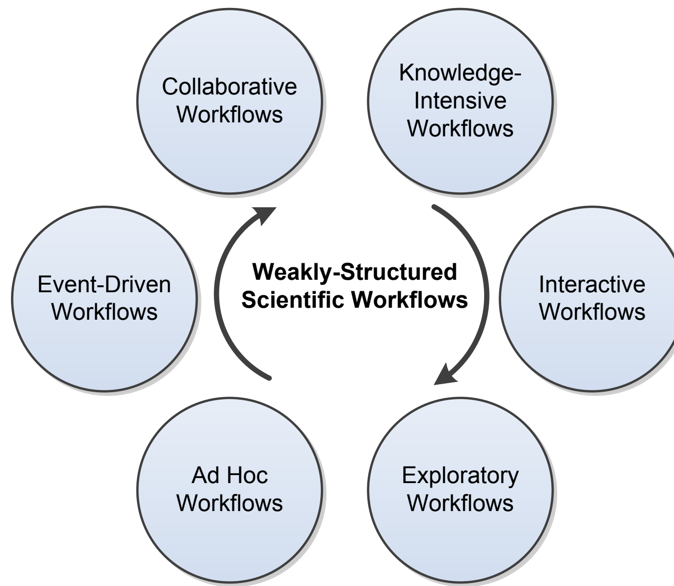


Figure 2.6: Weakly-Structured Scientific Workflows

2.5.2.1 Exploratory Scientific Workflows

Scientific workflows are traditionally used to weave the steps of scientific experiments. In a number of new applications, however, workflows are assembled for exploratory, in other words, scientists are interested in undertaking *one-of-a-kind* [36] or *what-if* [18] scenarios. Instead of designing a single workflow that will be run thousands of times, a user (or set of users) manipulates ensembles of workflows that are iteratively refined as he/she formulates and tests hypotheses [36]. Analysis methods or tools in these *exploratory scientific workflows* are evolved frequently from initial ideas and preliminary workflow definitions. With the formulation of experiments as scientific workflows, one could imagine creating a first workflow description of an experiment and subsequently testing with different combinations of parameters and process adaptations until a suitable solution is found [37].

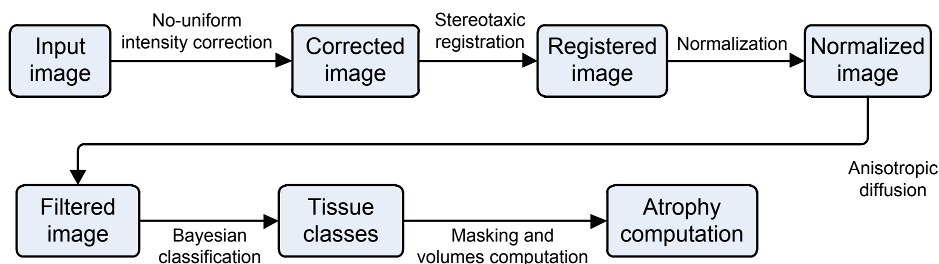


Figure 2.7: The Atrophy Computation Method Stages for Multiple Sclerosis

Caeiro-Rodriguez et al. [37] present an exploratory process in medical image analysis, which is about the identification of brain tissue loses in order to diagnose

Multiple Sclerosis. For the purpose of designing an automatic image-based method to quantify brain atrophy, an iterative exploration on different stages and datasets is involved. The process of the atrophy computation methods adapted from [38] is shown in Figure 2.7. Depending on the raw image features, some stages may be required or not. For instance, the steps of *non-uniformity intensity correction*, *normalization* and *anisotropic diffusion* are dependent on the hardware used to generate the input image. The *stereotaxic registration* is employed to rectify discrepancies in the raw images if necessary.

2.5.2.2 Collaborative Scientific Workflows

Existing SWFMSs mainly allow single scientist to compose and manipulate workflows [39]. As the nature of research questions becomes more and more complex, many scientific research projects have become collaborative to find answers, and there is a compelling need of a proper IT infrastructure and online services to support collaborative scientific workflows on the Internet [39]. *Collaborative scientific workflows* not only integrate distributed heterogeneous data and computational resources, but also allow researchers from different organizations collaborating in a large scientific experiment.

Collaborations supported by the scientific workflows could take place in a variety of ways, including data or computational resource sharing, delegating tasks to other parties, working together on complex tasks, etc. Figure 2.8 presents a fictional but realistic process of identifying a newly discovered *ant* (scientific name: *formicidae*). It is taken from European Distributed Institute of Taxonomy (EDIT), which is a network of excellence gathering 28 major institutions devoted to knowing the living world better with the support of the European Commission. The process involves the collaboration of three participants: *fieldworker*, *taxonomist* and *curator*.

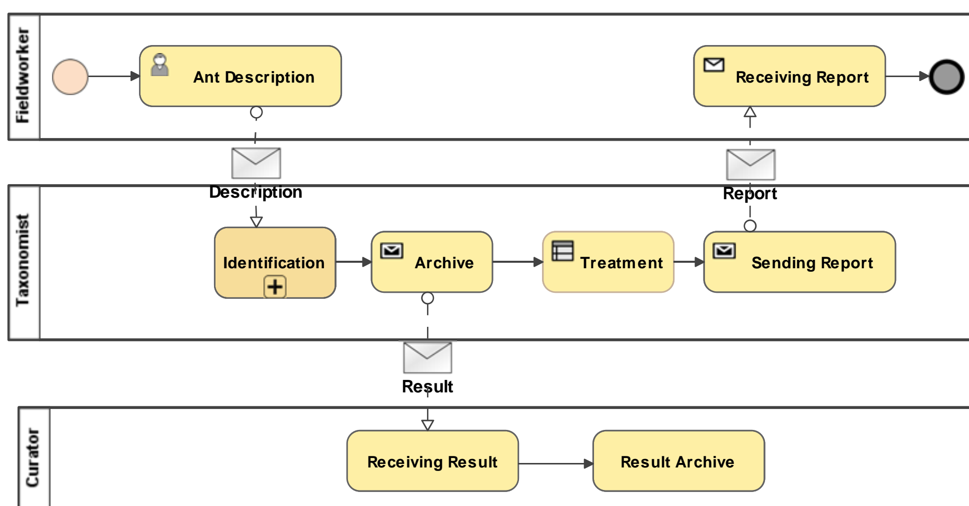


Figure 2.8: Process of Treating a Newly Discovered Ant

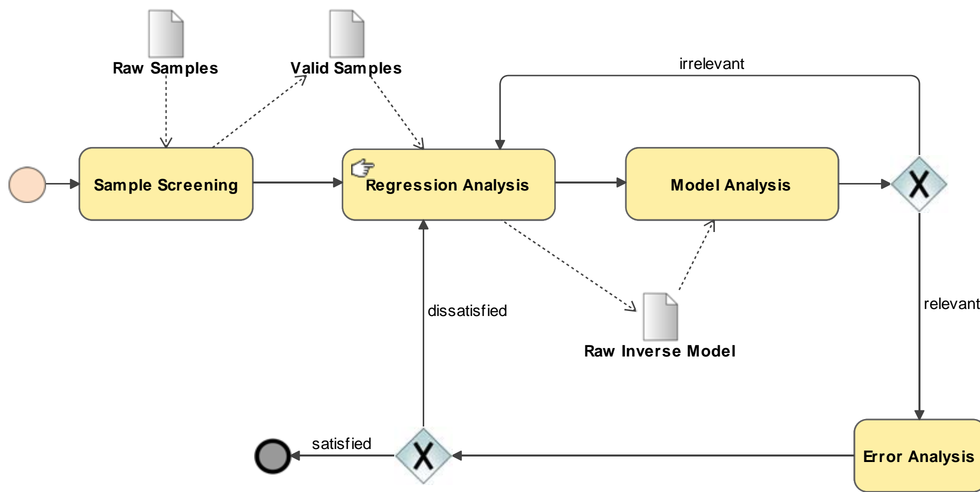


Figure 2.9: Building Snow Depth Forecast Model from Remote Sensing Data

The process is organized as follows. First, a *fieldworker* who often works in the countryside triggers the identification process. He/She describes a newly discovered ant and then sends the description to a *taxonomist*, who has experience and expertise to perform the identification and treat it. Afterwards, a *curator* archives the identification result. Finally, the corresponding treatment schemes are then provided to the *fieldworker*. These participants are often in different locations and collaborate on the ant identification.

2.5.2.3 Knowledge-Intensive Scientific Workflows

Knowledge-intensive scientific workflows often involve knowledge-intensive activities regulated by complicated scientific policies. Such activities are *decision-centric*, and the decisions are often made based on domain-specific knowledge, require multiple sub-decisions, use a complex situation based on decision technique and conclude one or more results [40].

Figure 2.9 shows a workflow taken from the remote sensing and Geographic Information System (GIS) domain with a goal of establishing a snow depth model for pastoral areas without field measurements. The workflow is based on an iterative regression analysis to estimate the relationship between satellite data (horizontal and vertical polarization brightness temperature differences, to be more precisely) and the snow depth. To build a precise prediction model, the process involves a task of screening valid snow depth samples, which is decision-centric with complex criteria. For example, in an experiment of establishing a snow depth model for the pastoral area of northern Xinjiang (in China) [41], screening snow depth samples is performed in terms of the criteria: (1) the snow depth must be thicker than 3.0 centimeters; (2) the meteorological station should be at an elevation lower than 2000 meters; (3) the snow must not be thaw; (4) the snow must not be wet snow that contains much liquid; (5) the snow must not be covered by deep frost; etc. The judgment of the

wet snow and the frost layer further involve sub-criteria, such as the thaw happens in March if the local temperature is higher than 6 ℃.

In other experiments of building the snow depth model, the criteria can be reused with different parameters. For example, in another similar experiment of building a snow depth model in Qinghai Province of China [42], the thaw happens when the local temperature is higher than 9 ℃. Moreover, there is no limit the elevation of meteorological stations.

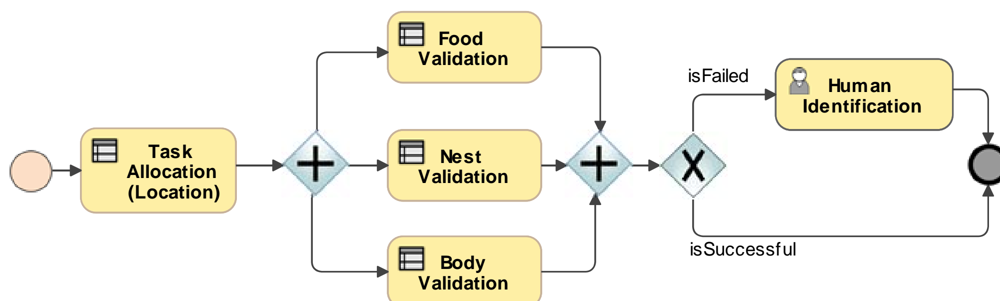


Figure 2.10: Ant Identification

In addition, the ant identification task itself involves complicated logic to distinguish an ant from its homogeneous groups; it is represented as a sub-process (with a "+" mark in the notation) in Figure 2.8. The details of the identification are shown as a process in Figure 2.10. The process starts with assigning the task to an inference service acting on the *taxonomist* behalf in terms of the location, where the ant is discovered. Afterwards, the ant is identified in terms of the domain knowledge. Ants can differ widely in their food requirements and behaviors, some pests even can cause a serious impact on crops. According to the Bayer's ant identification guide [43], the policies used to identify an ant include body features, nest structure and habits (e.g., food preference). Likewise, the task assignment and the ant treatment also need to be encoded with domain knowledge. In Figure 2.10, these knowledge-intensive decision-centric tasks are represented as rounded rectangles with small table notations in them. Moreover, if a discovered ant is unusual, it might involve domain experts to identify it manually. From a technical perspective, it is difficult to implement this kind of knowledge-intensive decision-centric process by traditional WFMSs.

Figure 2.11 shows another process used to analyze the precision of protein prediction algorithms. Proteins perform most important tasks in organisms, such as catalysis of biochemical reactions, transport of nutrients, recognition and transmission of signals [44]. In general, protein function can be thought of as, "anything that happens to or through a protein" [45]. For the purpose of describing protein functions, the Gene Ontology Consortium [46] provides an ontology of protein functions based on a dictionary of well-defined terms, also known as Gene Ontology (GO) terms. Each GO term defines gene product properties as well as the relationships

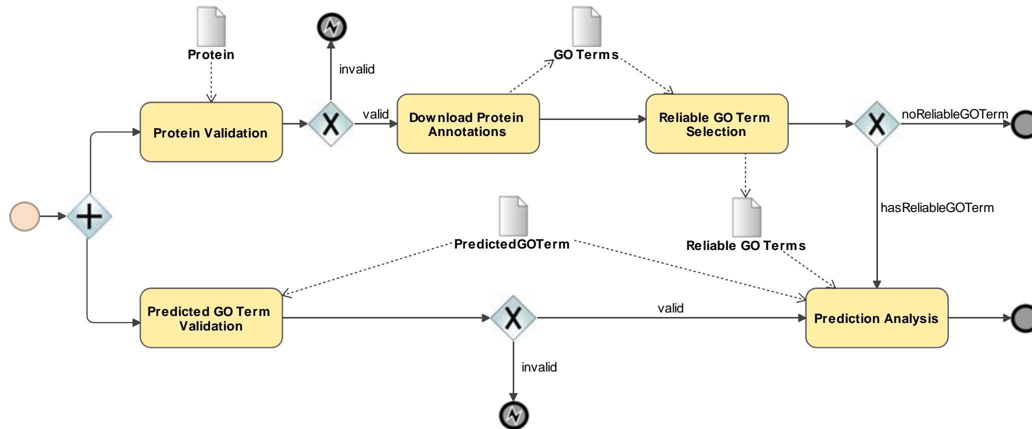


Figure 2.11: Protein Prediction Result Analysis

with other terms. The protein prediction is often conducted by computational algorithms that generate one or more GO terms indicating the functions that a protein may have. The prediction is considered correct if the protein has some true annotations (i.e., GO terms) that lie on a path in the gene ontology tree from the root to a leaf that visits the predicted annotation (i.e., GO term) [47]. For example, Figure 2.12 [48] shows an ancestor chart (gene ontology tree) of GO term *GO:0007167* in the context of its related terms. The GO term *GO:0007167* is generated by a protein prediction algorithm NetCoffee [49] to predict protein *Q15653* of human. The prediction is considered correct since the protein *Q15653* has a reliable GO term *GO:0007165* (i.e., cell death) that lies on a path in the ancestor chart of the GO term *GO:0007167* from the root to a leaf.

The process of Figure 2.11 has two inputs: a protein (represented by protein product ID) and a predicted GO term generated by a protein prediction algorithm. It starts by retrieving GO terms associated with the protein. This task is often done by querying Quick GO database [48] with a protein name. The GO terms of Quick GO are suggested by and solicited from members of the research and annotation communities. They have evidence codes that indicate how the description to a particular term is supported. In general, only reliable GO terms are used to analyze the precision of protein prediction algorithms. For example, in the analysis of protein prediction results generated by NetCoffee, GO terms with evidence codes IEA (inferred from electronic annotation) and ISS (inferred from sequence or structural similarity) are considered as unreliable terms [49]. The analysis process iteratively takes reliable GO terms of the protein, checks if there is a reliable GO term lying on a path in the gene ontology of the predicted GO term; the analysis process terminates when one or no required reliable GO term is found.

In order to automate the analysis task, it is necessary to analyze the gene ontology. Moreover, to improve the robustness of the process, the workflow inputs given by users need to be verified, i.e., the workflow inputs must be a valid protein

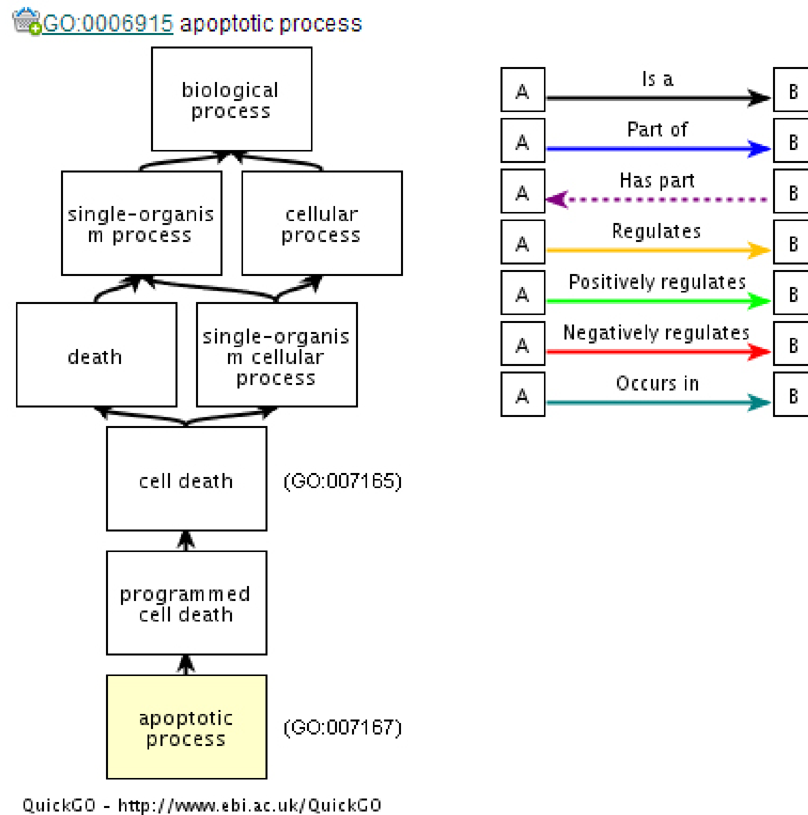


Figure 2.12: GO Term Ancestor Chart

product ID and a GO term, as shown activities in Figure 2.11. These verification activities also involve decisions based on domain knowledge and cannot be easily implemented by traditional WFMSs.

2.5.2.4 Event-Driven Scientific Workflows

The integration of content-based event notification systems with workflow management is motivated by the need for dynamic, data driven application systems which can dynamically discover, ingest data from, and interact with other application systems, including physical systems with online sensors and actuators [50]. On one hand, *event-driven workflows* are capable of providing more accurate outcomes with the use of updated or real-time data inputs. On the other hand, due to the diversity of the workflow inputs, these workflows need a flexible design to support dynamic decision-making *on-the-fly*. In other words, selecting subsequent branches is based on real-time environmental information; prior to the decision, all subsequent branches are possible to be selected.

Caeiro-Rodriguez et al. also present two event-driven scientific workflow examples: one is about producing daily forecasts of near-surface temperatures in cran-

berry bogs in Wisconsin; the other is about processing streaming observations from external sensors [37].

2.5.2.5 Interactive Scientific Workflows

Even though scientific workflows are mainly used to automate repetitive tasks, however, there are still manual tasks that are too complicated to automate, such as setting environment properties, checking intermediate results and controlling the convergence of results for subsequent branches. Such workflows requiring human interactions are also refereed as *interactive scientific workflows* in this thesis. In *ad hoc workflows* (explained in the next section), which may contain steps that need to be determined *on-the-fly*. Human users can be instrumental to make dynamic decisions and steer the workflow execution. For example, in the ant identification and treatment process mentioned above, domain experts will be involved if the intelligent inference services cannot perform the identification automatically. Moreover, user interaction also helps in handling unexpected exceptions at runtime. For example, scientists may be asked to provide missing resources.

2.5.2.6 Ad Hoc Scientific Workflows

Compared with the workflows mentioned above, *ad hoc workflows* are much closer to the unstructured ones. They have no predefined structures and the workflow execution is steered by scientists as required. There are generally two different types of ad hoc workflows: one involves a set of activities that can be pre-defined at design time, but the sequence of their performances are determined by scientists depending on latest context information; the other one can deal with unpredictable exceptions at runtime. For example, there might be no resource available to perform a scientific task at runtime, resulting in the failure of the whole process and the waste of time. With an ad hoc operation, it is possible to avoid these problems by providing missing resources or modifying the workflow logic, thereby improving workflow robustness.

2.5.3 WsSWF Features

The WsSWFs are miscellaneous and could involve two or more distinct workflows mentioned above. In general, the WsSWFs have the following features:

(i) **Variable scientific processes**: compared with the structured processes that are undergone fewer changes, the WsSWFs have variable logic in terms of individual circumstances. In other words, one instance of the process might be different from another based on different circumstances.

(ii) **Knowledge-intensive**: workflows usually contain decision points that determine the execution paths at runtime. In the WsSWFs, these decision points are not limited to simple Boolean expressions but often involve complex knowledge-intensive conditional decisions that rely upon the knowledge about a specific domain.

(iii) **More error-prone**: the WsSWFs extend the range of scientific workflow applications. However, since they are less-structured, they are more likely to subject to frequent changes and exceptions at runtime.

(iv) **Collaboration and interaction**: instead of improving the operational efficiency, the WsSWFs focus on finding solutions to problems. They usually need to interact with human experts and involve collaboration between multiple participants.

2.5.4 WsSWF Main Requirements

This section summarizes main requirements of the WsSWFs, which will be used as the benchmarks to evaluate the existing WFMSs in Chapter 3.

(i) *Rich Process Specification*: the WsSWFs contain complex decision-centric tasks, which require processes to handle new and exceptional situations. Besides simple control flow descriptions (e.g., a task is enabled after the completion of a preceding task), it is also necessary to describe advanced process logic, which needs dynamic recognition of operational as well as knowledge-based states to implement intelligent routings at runtime.

(ii) *Expressing Domain-Specific Policies*: the WsSWFs often involve complex domain-specific policies, which regulate the behavior of scientific applications. In order to automate the WsSWFs, it is necessary to express such scientific policies and enable machines to deal with them automatically.

(iii) *Flexibility*: the structured processes suffer from limitations with respect to dynamic evolution and adaptation at runtime. In order to provide high flexibility, the WsSWFs should be allowed to be easily modified according to individual situations.

(iv) *Human Interaction*: scientific workflow systems are often designed to automate scientific processes and improve their operational efficiency. However, human users still need to conduct manual tasks and steer the workflow execution to deal with unforeseen problems at runtime.

(v) *Exact Reproducibility*: provenance plays an important role in verification, explanation, reproduction and informed reuse of data used and produced by scientific workflows, especially by the WsSWFs, which have non-deterministic decision logic. However, provenance is a broad standalone topic in itself. This thesis jut provides a basic outline on it, and the problem as a whole is out of the scope of this thesis.

2.6 Workflow Description-Related Technologies

This thesis aims at describing the WsSWFs and supports their adaptive execution from a technical perspective, and this section clarifies the technologies related to a workflow programming language, as shown in Figure 2.13 adapted from [51].

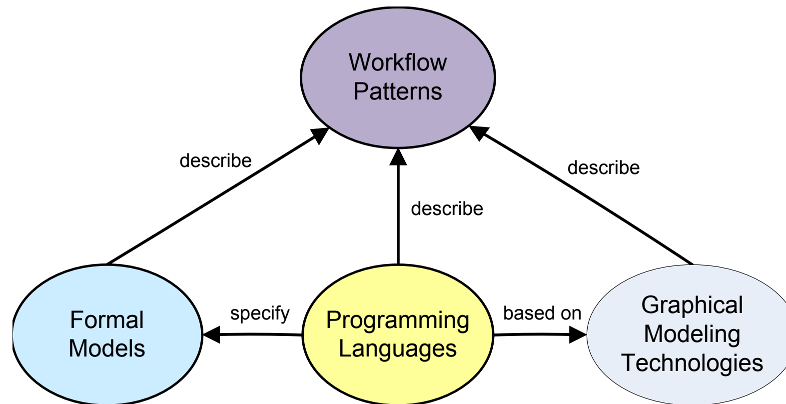


Figure 2.13: Workflow Description-Related Technologies

Workflows automate recurring activities (in whole or part) with a purpose of improving efficiency and reducing costs at the same time. Such recurring workflow processes are described by abstract workflow patterns, which are independent with specific implementation technologies. From different perspectives, the Workflow Patterns Initiative [52] has delivered four types of workflow patterns related to the development of workflow applications, i.e., control-flow patterns, data patterns, resource patterns and exception handling patterns. Workflow patterns are the formal ways of documenting the solutions to recurring problems and are reflected in different layers of the workflow description, including workflow programming languages, graphical workflow modeling technologies, and workflow formal models, as shown in Figure 2.13.

Workflow programming languages (or workflow languages) capture the relevant information of application processes with the aim of their execution controlled by a WFMS [53]. They specify both logical steps and information passing from one participant to another. The workflow specifications described by these programming languages will be used as input to a WFMS for the workflow instantiation and execution. Currently, there are a number of industrial and scientific workflow languages available, such as BPEL [28], Yet Another Workflow Language (YAWL) [54] and Simple Conceptual Unified Flow Language (SCUFL) [55]. However, since the workflow programming languages are usually verbose, graphical workflow modeling technologies are proposed to facilitate the workflow modeling at a high level. For example, BPMN is a standardized graphical representation for business processes modeling [56]; it is possible to create not only business-oriented diagrams, but also technical models for process execution. Besides BPMN, Unified Modeling Language (UML) activity diagram also can be used as a standard in the area of organizational process modeling. The evaluation results [57] show that the activity diagram of UML supports the majority of the patterns considered, including some that are typically not supported by commercial WFMSs. These graphical workflow modeling tools bring a clear division that allows workflow modeling engineers to

focus on process requirements, and lets workflow developers specify corresponding implementation details with concrete programming languages.

Workflow formal models provide a theoretical foundation to workflow programming languages and are capable of reducing ambiguity and opening possibilities for verification and analysis. Such models can be used to support the design of workflow languages and of their interpreters, compilers and optimizers as well as of debuggers, and to support the definition of verification procedures, similar to those used for verifying the correctness of complex business transactions [58]. Some prominent workflow formal models are Petri nets and calculi theory (e.g., π-calculus). Currently there are many workflow programming languages available, and some of them have syntax and semantics based on these formal models. For example, BPMN is based on Petri nets; XLANG is based on π-calculus. The thesis proposes a declarative rule-based workflow language for WsSWFs. To facilitate the evaluation, a formal semantics of the language based on CTR is given in Chapter 5.

2.7 Imperative vs. Declarative Programming

A programming language is used to create programs that control the behavior of a machine or to express algorithms precisely. There are basically two approaches of programming: *imperative* and *declarative*. Imperative programming is characterized by a state and commands which modify the state. Declarative programming, on the other hand, describes a proof that some truth holds. Imperative programming can be further divided into procedural programming (e.g., FORTRAN, C) and object oriented programming (e.g., C++, Java), and declarative programming can be divided into logic programming (e.g., Prolog, Prova) and functional programming (e.g., Haskell, Erlang).

It is worth noticing that a reasonable classification of programming languages is not the goal of this section. The goal of this section is to identify both advantages and disadvantages of two different programming paradigms, which can be used to evaluate existing solutions to support an adaptive workflow execution in Chapter 3.

The imperative programming paradigm explicitly describes the control flow of a program in terms of statements changing the program state, i.e., an imperative program prescribes the execution of a task in terms of sequences of actions to be taken. On the other hand, the declarative programming paradigm expresses what a program should accomplish without prescribing how, i.e., a declarative program expresses the logic of performing a task without explicitly describing its control flow.

In what follows, a simple example of checking the elevation of a meteorological station mentioned in the process of screening snow depth data (see Section 2.5.2) is implemented in both imperative and declarative ways. In the experiment with a purpose of establishing a snow depth model for the pastoral area of northern Xinjiang, all snow depth data must be from meteorological stations that have elevation lower than 2000 meters.

The rule is first programmed in Java:

Listing 2.1: Example of Imperative Programming

```
1 int elevationCriticalvalue = 2000;

3 boolean checkElevation(String station){
4   double stationElevation = getStationElevation(station);
5   if(stationElevation < getDepthCriticalValue())
6     return true;
7   else return false;
8 }

10 double getStationElevation(String s) {
11   if(s=='Fuhai')
12     return 500.9;
13   else if(s=='Aletai')
14     return 735.3;
15   else if(s=='Fuyun')
16     return 823.6;
17   else if(s=='Qinghe')
18     return 1218.2;
19   ...
20 }

22 double getDepthCriticalValue(){
23    return depthCriticalValue;
24 }
25 ...
```

The above program has three methods: (1) *checkElevation(String station)* checks if the elevation of a meteorological station is valid; (2) *getStationElevation(String s)* obtains the elevation of a given station; (3) *getDepthCriticalValue()* obtains the critical value of the station elevation in the experiment. Each method specifies every step to final results. For example, the programmer gives the procedure of stepping through all stations to obtain the elevation of a given station.

With declarative programming, the following programs in Prova [59] focus on declarative problem representation, instead of programming problem solutions. Prova is a declarative rule language, and it is also used in the proof-of-concept implementation of this thesis. More details will be found in Section 6.1.

Listing 2.2: Example of Declarative Programming

```
1 checkElevation(Station) :-
2    stationElevation(Station, Ele),
3    elevationCriticalvalue(ElevationCriticalvalue),
4    Ele < ElevationCriticalvalue.

6 % measure unit: meter
7 elevationCriticalvalue(2000).

9 % measure unit: meter
10 stationElevation('Fuhai', 500.9).
11 stationElevation('Aletai', 735.3).
12 stationElevation('Fuyun', 823.6).
13 stationElevation('Qinghe', 1218.2).
14 ...

16 Queries:

18 elevationCriticalValue(X)? The critical value of the station elevation.
```

```
19 stationElevation(Stations,X)? Elevations of all stations
20 stationElevation('Fuhai',X)? Elevation of 'Fuhai' station
21 checkElevation('Fuhai')? The data of station 'Fuhai' is valid?
22 ....
```

Here, the first rule describes the logic of checking if the elevation of a meteorological station is valid. The rule does not give the logic of stepping through all stations when it obtains the elevation of a given station, and matching a given station is performed by the Prova rule engine. It also uses fewer coding efforts to implement more queries than the Java program does. Besides the functions provided by the Java program, the Prova program also can query elevations of all stations. Moreover, the syntax is close to natural language and is also understandable to domain experts. If there is a necessary to change the logic, these rules can be easily modified without extending the inference machine. However, it is often difficult to modify imperative programming languages since their logic is deeply buried in source programs.

Accordingly, the counterpart languages used to specify workflows can also be categorized into imperative and declarative workflow languages, respectively. The imperative workflow languages explicitly describe every possible process behavior with atomic design primitives. This kind of rigid approach couples modeling with execution and does not fit flexible processes. On the contrary, declarative workflow programming eliminates the rigidness of computation by describing the outcome of a workflow, rather than specifying how to get the outcome. However, it is difficult to which one is better. They play well when they are used in their own fields.

2.8 Logic Program Overview

Generally, a logic program is a finite set of rules. Each rule r is of the form:

$$\alpha_1 \vee ... \vee \alpha_k \ :-\ \beta_1,\ ...,\ \beta_m,\ not\ \beta_{m+1},\ ...\ ,\ not\ \beta_n.$$

where α_1, ..., α_k, β_1, ..., β_n are *atoms*, and $k \geq 1, n \geq m \geq 0$. The disjunction of $\alpha_1 \vee ... \vee \alpha_k$ is the head of r, and the conjunction of $\beta_1, ..., \beta_m, not\ \beta_{m+1}, ..., not\ \beta_n$ is the body of r. Let the set of the head literals be denoted by $H(r)$ and the body of r be denoted by $B(r)$. Also, let the set of positive and negative body literals be denoted by $B^+(r)$ and $B^-(r)$, respectively.

Furthermore, an *atom* is a formula p(t_1, ..., t_n), where p is a predicate symbol of arity n (n≥0). A literal is either an atom or of the form *not* β where β is an atom. Each argument of an atom t_i is a *term*, which is either a *variable* or a *function term* with a form of f(t_1, ..., t_k). f is a function symbol of arity k (k≥0), and each t_i is a term. A functional term with arity zero is a *constant*.

Based on different classes of rules, there are different types of logic programs: propositional logic programs, Datalog logic programs, definite logic programs, stratified logic programs, normal logic programs, extended logic programs, disjunctive logic programs and combinations of classes, as shown in Figure 2.14 adapted from [60].

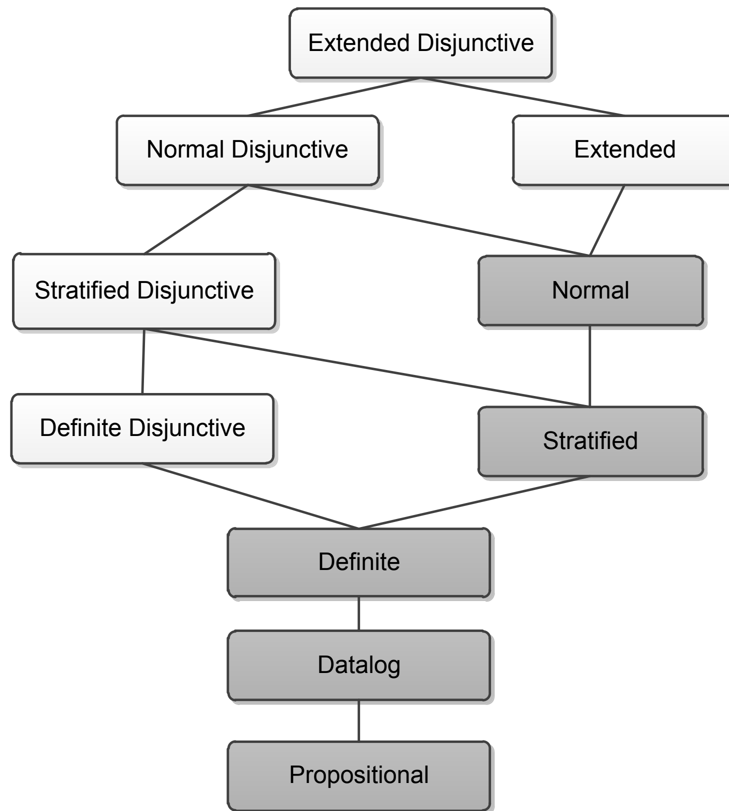


Figure 2.14: Classes of Logic Programs

A *propositional logic program* consists of simple propositional clauses. All literals of a clause are propositional ones without variables, quantifiers and functions, and thus propositional logic programs provide limited expressive power.

As a lightweight declarative logic programming language, Datalog is often used as a query language for deductive databases. Moreover, Datalog is function-free, i.e., all functional terms appearing in a Datalog program are constants. In addition, variables in the head of a rule must appear in its body. Note that Datalog has its origins as a query language in database systems, and there are extensions that have been made to Datalog [61, 62].

A *definite logic program* is a set of rules of the form:

$$\alpha : -\beta_1,\ \beta_2,\ ...,\ \beta_n. \qquad (n\ \geq\ 0)$$

where α and β_1, ..., β_n are all positive atoms, i.e., $B^-(r)\ =\ \emptyset$.

A finite logic program goes beyond Datalog programs by allowing functions, which provide the ability of handling finite sets of constants, such as encoding lists, trees and other common data structures.

However, unlimited function symbols may make common reasoning tasks undecidable even for rather simple programs. A natural decidable fragment of logic

programming with functions are *nonrecursive programs*, in which no predicate depends syntactically on itself [63]. In addition, there are decidable and expressive fragments of logic programs with function symbols that have been identified. In a survey of the decidability results for Answer Set Programming (ASP) programming with functions, Alviano et al. classify decidable ASP programs into three groups: *bottom-up commutable*, *top-down commutable* and *finitely representable stable models* [64]. Programs in the bottom-up commutable group allow for stable model computation and query answering by searching over finite ground programs. Classes of programs in this group are ω*-restricted* [65], γ*-restricted* [66], *finite domain* [67], *arguments restricted* [68] and *finitely ground programs* [67]. Programs in the second group are designed for query answering. They usually have an infinite number of answer sets and a finite number of atoms must be identified to guarantee the decidability of answering a query. Programs in this group are *FP2 programs* [69], *positive and stratified finitely recursive programs* [70, 71], and *finitary programs* [72]. Programs in the third group are *FDNC* [73] and *bidirectional programs* [74], which are characterized by infinite stable models with a finite representation in the shape of a forest of trees.

Definite logic programs in general are not expressive for general knowledge representation, which involves decision and situational logic. This is because definite logic programs exclude negation, an important feature for real knowledge representation applications.

A *normal logic program* has a similar form with definite logic programs, but each body literal can be either positive or negative:

$$\alpha \ :- \ \beta_1, \ ..., \ \beta_m, \ not \ \beta_{m+1}, \ ... \ , \ not \ \beta_n. \qquad (n \ \geq \ m \ \geq \ 0)$$

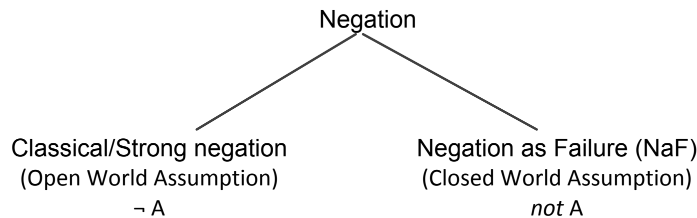


Figure 2.15: Logic Program Negations

In general, there are two types of negations: Negation as Failure (NaF) and *classical negation*, as shown in Figure 2.15. NaF is based on *closed world assumption*, which assumes something is false if it cannot be proved to be true. In other words, it transforms proving something false by proving its truth. This is useful for representing normals and exceptions. The following example presents a set of Prova rules, which describes that an animal is dead if it cannot fly. *jack* is proved to be dead since the rule engine cannot prove *fly(jack)*.

Listing 2.3: NaF Implemented in Prova

```
1  :- eval(died(jack)).

3  fly(X):- bird(X).

5  died(X):-
6      not(fly(X)),
7      println(["Yes"]),
8      !.
9  died(X):-
10     println(["No"]).

12 bird(tweety).
13 rabbit(jack).

15 % NaF as implemented in Prova internal function.
16 not(A) :-
17     derive(A),
18     !,
19     fail().
20 not([X|Args]).
```

The classical negation is based on *open world assumption*, which assumes something to be false, if it is explicitly proved to be false. In other words, the negation is in the head of a rule. However, the classical negation might lead to logical conflicts between rules.

The negation in normal logic programs is NaF, and a negative literal succeeds when all attempts to prove the literal fail in a finite amount of time.

However, although negation is needed in practical knowledge representation applications, it is a complex problem in deductive databases and logic programming, since it may result in a computation that does not fail finitely. The problem is presented by the following example taken from [75].

Listing 2.4: Recursion Through Negation

```
1  :- eval(happy("Joe")).

3  stressed(X):-
4      not(happy(X)),
5      not(works(X)).

7  works(X):-
8          normal(X).

10 normal(X):-
11     not(stressed(X)).

13 happy(X):-
14     not(works(X)),
15     !,
16     println(["Happy"]).
17 happy(X):-
18     println(["unhappy"]).

20 not([X|Args]) :-
21     derive([X|Args]),
22     !,
23     fail().
24 not([X|Args]).
```

When trying to derive *happy("Joe")*, it needs to check *not(works("Joe"))* and for this it is necessary to check *normal("Joe")* which implies to check *stressed("Joe")*. And then the derivation of *happy("Joe")* enters an infinite loop. A key feature of this kind of program is that there is a negation wrapped in a recursion, e.g., the derivation of *not(works(X))* involves checking *happy(X)* itself. For the purpose of solving this problem, there are approaches that have been proposed: *stratified semantics*, *stable model semantics*, *well-founded semantics*, etc.

Stratification is a constraint usually placed on logic programs to rule out negation wrapped inside recursion. The predicates of *stratified logic programs* are placed into strata so that one can compute over the strata [60]. A detailed definition of stratified logic programs can be found in [76].

However, not all logic programs can be stratified. Stable Model Semantics (SMS) addresses this issue by checking whether a candidate set of atoms is stable or not. Informally, an interpretation I of a normal logic program P is a stable model of P if I is the least Herbrand model of P.

If a logic program can be stratified, then it has a unique stable model, and its stratified semantics and stable semantics coincide. In general, a normal logic program may have zero, one or multiple stable models. In other words, SMS cannot provide a stable model to every logic program; moreover, the computation of stable models is *NP-complete*. To address the limitation of SMS, Well-Founded Semantics (WFS) is invented.

Roughly speaking, WFS assigns value "unknown" to an atom, if it is defined by unstratified negation [63]. In WFS, a logic program has a unique model. Note that if a logic program P can be stratified, then it has a unique minimal model, which is also both a stable model and a total well-founded model. However, the disadvantage of WFS is that it cannot distinguish between several stable models and no stable model.

Disjunctive logic programs allow rules with disjunctions at their head and provide a more natural and flexible knowledge representation. They may contain definite or indefinite information which reflects human limitation in understanding the world being modeled [77]. Logic programs which employ both types of negations are called *extended logic programs*. In addition, there are combinations of such logic programs, e.g., definite disjunctive, stratified disjunctive, normal disjunctive and extended disjunctive logic programs, as shown in Figure 2.14. On one hand, these combined logic programs increase the expressiveness to some extent, but on the other hand, their complexity increases as well.

2.9 Deductive, Abductive and Inductive Reasoning

Reasoning a declarative logic program is the process of using existing knowledge to draw conclusions, make predictions, or construct explanations [78]. There are three different types of reasoning: deduction, abduction and induction.

- *Deductive reasoning* [78] starts with the assertion of a general rule and proceeds to a guaranteed specific conclusion in terms of the rule (conclusion guaranteed). The conclusion must be true when the original assertions are true. Deductive reasoning is often used for derivations of knowledge as conclusions from given knowledge.

- *Abductive reasoning* [78] typically begins with an incomplete set of observations and proceeds to the likeliest possible explanation for the set (conclusion likely). Abduction is the opposite of deduction regarding to the reasoning direction: deduction starts from a set of premises and deduces a conclusion, while abduction begins with a conclusion and proceeds to possible premises. It is often used for goal-oriented planning, i.e., finding premises that can proceed to a given conclusion.

- *Inductive reasoning* [78] begins with observations that are specific and limited in scope, and proceeds to a generalized conclusion that is likely, but not certain, in light of accumulated evidence (educated guess). One could say that inductive reasoning moves from the specific to the general. It is often used to learn patterns and regularities with a purpose of inducing a larger set of conclusions.

This thesis focuses on describing domain-specific decision logic and conditional reactive logic of the WsSWFs by declarative rules. The execution of such rules can be regarded as drawing conclusions from given knowledge, i.e., deductive reasoning with rules based on facts. More details can be found in Section 4.4.

2.10 Summary

This chapter presented background information about scientific workflows and logic programming in detail. After giving a brief introduction to scientific workflows and their life cycle, this chapter introduced a reference architecture for WFMSs. This chapter also compared scientific workflows with business workflows, and explained why business workflow technologies can not be directly employed into scientific workflows.

Furthermore, this chapter classified workflows into structured and unstructured processes from the process structure perspective. In particular, this chapter introduced the WsSWFs by means of several real-world use cases from different domains. Such WsSWFs containing knowledge-intensive tasks are the emphasis of this thesis.

Also, this chapter introduced the technologies related to the workflow description, including workflow formal models, workflow programming languages, workflow graphical modeling technologies and workflow patterns. Two programming approaches: *imperative* and *declarative* workflow programming were detailed. Imperative programming prescribes performing a task in terms of sequences of actions to be taken, and declarative programming expresses what a program should accomplish without prescribing how. In addition, different forms of logic programs and

three different types of reasoning: deduction, abduction and induction were briefly introduced.

In the next chapter, the state-of-the-art on different solutions with a purpose of improving the flexibility of both business and scientific workflows is presented.

Chapter 3

Flexible Workflow Compositions

Contents

3.1 Classic Workflow Languages 39
3.2 Agent-Oriented Workflow Compositions 41
3.3 Rule-Based Workflow Languages 44
3.4 Main Scientific Workflow Languages 46
3.5 Efforts on Weakly-Structured Workflows 48
3.6 Semantic-Based Workflow Compositions 49
3.6.1 Semantic Web Services 50
3.6.2 Ontology-Based Workflow Specifications 52
3.7 Summary 55

With the development of the workflow technologies, more and more efforts have been put into automating business and scientific processes. However, workflow processes in practice are complex and require a flexible design to represent their process logic and decision logic. Moreover, although distributed execution environments provide a huge amount of resources, their uncertainty and heterogeneity also bring difficulties to an effective workflow execution. To deal with these challenges, it is necessary to provide not only an expressive workflow description language, but also sophisticated mechanisms to handle dynamic exceptions at runtime.

This chapter surveys existing solutions with a purpose of providing a flexible workflow composition and supporting an adaptive workflow execution. Sections 3.1, 3.2 and 3.3 introduce classical workflow languages, agent-oriented workflow compositions and rule-based workflow languages, respectively. Section 3.4 describes main scientific workflow languages used in existing scientific SWFMSs. Section 3.5 outlines some preliminary efforts on the weakly-structured workflows. Section 3.6 introduces how semantic ontologies are used to describe distributed resources and workflows, and an analysis of the state-of-the-art is given in Section 3.7.

3.1 Classic Workflow Languages

Classic workflow languages are usually process-oriented and describe an executable process involving both control and data flows between participant services. Especially with the advent of SOA, such loosely-coupled paradigm of Web services

greatly promotes the development of both business and scientific workflows using Web service computing and service-based workflow technologies [79].

BPEL [28] is an XML-based language that supports the orchestration of Web services into sophisticated services using a workflow process. BPEL is based on IBM's WSFL and Microsoft's XLANG specification. Supported by a broad range of well-known companies (e.g., Microsoft, IBM, SAP, Siebel, and Oracle), BPEL has been regarded as the de-facto standard for orchestrating Web services for business workflows. Moreover, many well-designed tools, such as ActiveBPEL Designer, Oracle BPEL Process Manager, ActiveBPEL Engine, have been developed to support it. A BPEL process connects a Web service through a partner link, which defines the relationship between the BPEL process and the Web service. BPEL supports recursive composition by describing the inbound and outbound process interfaces in Web Services Description Language (WSDL), i.e., BPEL process itself is also a Web service. However, such process-oriented workflow composition languages suffer from two main limitations with respect to modularity and flexibility [80]. On one hand, crosscutting concerns (e.g., logging, persistence and security) are spread across the workflow specification and intertwined with main process logic, making the workflow specification hard to understand, maintain, change and reuse. On the other hand, the predefined process-oriented composition languages lack support for dynamic adaptation because of unexpected situations and failures at runtime, such as the unavailability of a service, variations of the Quality of Service (QoS) properties of a partner, or changes in regulations and collaboration conditions.

Aspect-Oriented Programming (AOP) is a new programming technique that increases the modularity of a system by allowing the separation of crosscutting concerns and changes [81]. In other words, crosscutting concerns and changes are specified in a modular way using aspects and are dynamically integrated into main business logic at runtime. For the purpose of addressing the drawbacks of BPEL mentioned above, an aspect-oriented extension to BPEL, also known as AO4BPEL [80] is proposed. In the AO4BPEL, crosscutting concerns are separately modularized and dynamically weaved into the main workflow process logic to make it more flexible and adaptable.

YAWL [54] is seen as an alternative to BPEL. Based on high-level Petri nets, YAWL offers comprehensive support to the workflow patterns elaborated by Van der Aalst et al. YAWL supports dynamic adaptation of workflow models through the notion of worklets, i.e., each task is associated with a set of self-contained sub-processes (i.e., worklets), by which a dynamic runtime selection can be made depending on the context of a particular work instance. Like BPEL, the data handling in YAWL relies on XML standards, like XPath and XQuery. However, YAWL and BPEL usually coordinate activities wrapped inside a Web service and limit their use to specific context, although Web services are common in a business scenario [82].

In general, these processes created by graph-based languages are usually suited to describe the structured processes with stable logic. The biggest expectation to these workflows is to run efficiently. Most of them are XML-based and focus on service orchestration. In other words, the service composition is under control of a

central entity, and services are usually process-agnostic and can be reused in other processes. Moreover, the flow transition conditions provided by them are based on Boolean expressions or simple rules and are not adequate to express complex decision logic and conditional reaction logic.

3.2 Agent-Oriented Workflow Compositions

In computer science, an agent is a computer system that is intelligent and autonomous. Communities of agents support flexible cooperation and are usually used to solve problems that are difficult or impossible for an individual agent to solve, also known as a Multi-Agent System (MAS). In general, an MAS provides high scalability and can scale via distributed process execution. With the MAS framework, it is also possible to provide additional reasoning intelligence inside an agent, which can be invoked by the agent as decision procedures. Barker et al. [4] summarize that the MAS framework offers a number of advantages over other techniques used by existing projects focusing on the scientific workflow composition: reasoning models of individual agents; inter-operability via the communication between heterogeneous agents in a common language with agreed semantics; layered structure via filling the gap between the low level transport issues of an agent and its high level rational processes; abstraction via distributed agents acting as stubs or proxies to Web services; rapid prototyping via protocols providing executable specifications of the coordination; compatibility via no alteration on external Web services.

Buhler et al. [83] present an approach to specify a workflow as an MAS, which can intelligently adapt to changing environmental conditions. Specially, they present BPEL can be used as a specification language for the initial social order of an MAS. In other words, each agent uses a passive Web service as its external behavior. The approach makes it easier to implement agents, but the centralized agents eliminate the robustness and opportunistic behavior, which are landmark advantages of MASs. Barker et al. [4, 84] capture scientific processes with Multi-Agent Protocol (MAP), which allows typical features of scientific workflow requirements to be understood in terms of pure coordination and to be executed in an agent-based, decentralized, peer-to-peer architecture. They present a motivating scientific workflow taken from Large Synoptic Survey Telescope (LSST) and show how the agent-based approach is helpful to classify previously unknown objects. Each agent taking part in the interaction adopts a role, by which the agent references a reasoning Web service that implements all decision procedures required for that role type. Figure 3.1 adapted from [84] presents the dependencies between the agents (shown as servers) and the outside services. Wagner [85] proposes an approach that combines an agent with the workflow concepts for the purpose of improving the flexibility of the workflow execution. The basic principle behind the approach is that of hierarchical nested *subworkflows*, which are controlled by an overall *structure-workflow*. An agent called *structure-agent* is responsible for executing the *structure-workflow* and its *subworkflows*. The *subworkflows* can be dynamically selected according to current circum-

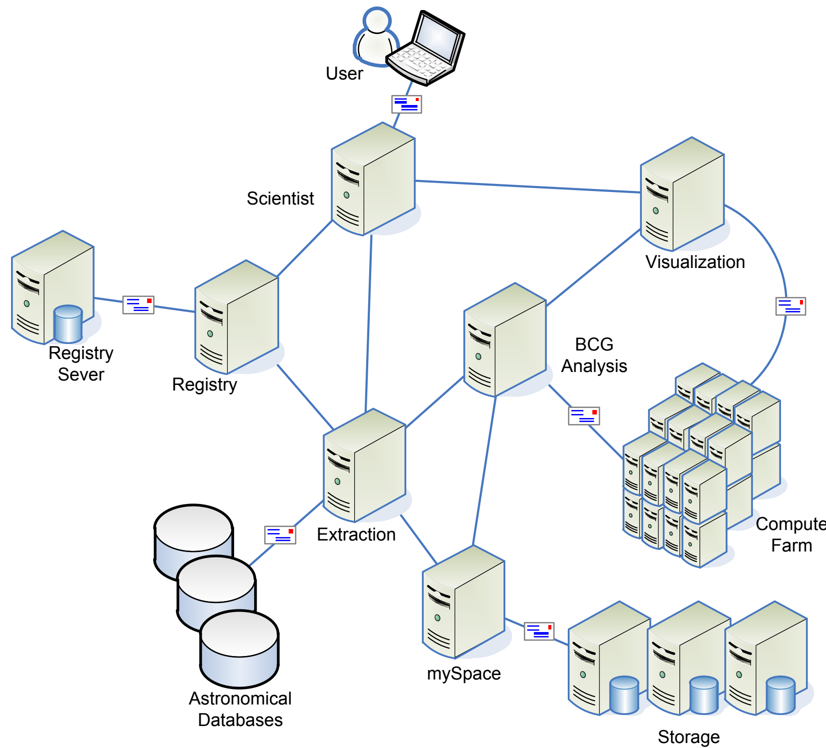


Figure 3.1: Agent-Based Scientific Workflow Composition

stances, newly modeled and integrated. However, the key technologies related to the flexibility, such as how the agents are organized, are not mentioned. Lam et al. [86] state that an MAS can be constructed to enact a set of given workflows while respecting the constraints of a given organization. They use Semantic Web languages (i.e., OWL-DL [87] and Semantic Web Rule Language (SWRL) [88]) to describe organizational and domain knowledge. Such knowledge can be used by the agents to make intelligent decisions at runtime. Moreover, they describe business organizations by defining roles, role classification and norms together. However, the norm-governed organization representation is not necessary for scientific workflows. The RbAF presented in this thesis provides a lightweight workflow ontology to describe general workflow concepts and their relationships. It focuses on describing complex decision logic and controls the agent behavior by declarative rules. The exception handling strategies employed by the RbAF is similar to the ones used in [86], but the differences can be found in the user interaction-based exception handling when certain exceptions cannot be handled by the agents automatically.

On the basis of natural language constructs (subject, predicate, object) and communication patterns between actors (subjects), a new BPM methodology Subject-oriented Business Process Management (S-BPM) is proposed to show how individual members of an organization could contribute to coherent and intelligible process specifications [89]. The S-BPM puts the subject of a process at the center of atten-

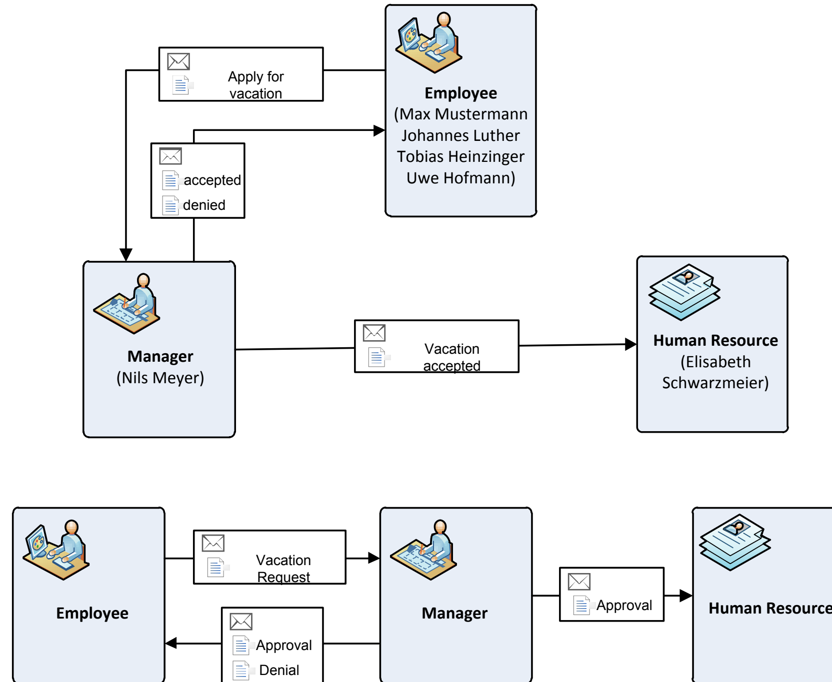


Figure 3.2: Subjects and Communications in Holiday Application Process

tion and thus deals with business processes and their organizational environment from a new perspective, meeting organizational requirements in a much better way than traditional approaches [90]. The S-BPM provides a coherent procedural framework to model an organization's business processes: its focus is the cooperation of all stakeholders involved in the strategic, tactical, and operational issues, sharing their knowledge in a networked structure [90]. A subject represents an agent performing actions in a process, which can be either technical or human (e.g., a thread in an IT system or a clerk). The process in a whole structures the internal behavior of each subject and coordinates with other subjects to accomplish a goal. An example adapted from [89] is shown in Figure 3.2. It shows a holiday application process, which involves three subjects: employee, HR department and manager. Each party exchanges messages with another party to process the application. Compared with the RbAF presented in this thesis, the *subject* of S-BPM denotes a concept, but the agent of this thesis is an instance. In other words, the RbAF can be seen as an implementation of the S-BPM approach, where the subject is a rule-based agent, which follows declarative rules to perform specific behavior. Moreover, instead of modeling organizational environment of a process, the RbAF focuses on the interactions between distributed agents from a high level and aims to provide an agent-based, flexible architecture to support the WsSWF execution.

To sum up, the aforementioned efforts demonstrate the advantages of agent-oriented architecture and support the flexible workflow composition from a workflow

structure perspective. They usually focus on, such as coordination mechanisms, organizational environment modeling and norm specifications. However, most of them do not consider how to specify the internal behavior of an agent. This is crucial if there are domain-specific knowledge-intensive activities that are involved in a workflow. Moreover, although there are benefits of the agent-oriented workflow composition, it loses the strengths of the centralized workflow execution, i.e., different services can be composed efficiently through a centralized flow.

3.3 Rule-Based Workflow Languages

The rule-based approaches can also support flexible service composition and model the process logic with declarative rule languages. They have advantages, such as allowing intuitive formal semantics by exploiting a limited set of primitives, direct support for business and science policies, flexibility by alternative execution paths in case of errors or unreachable solutions, adaptability by easy interaction and retraction of rules, and reusability by their property of being isolated from the process context, which have entitled them as alternative declarative approaches for solving complicated business logic and scientific problems [91].

As an initial attempt on this topic, some efforts simply combine existing standard business workflow languages with declarative rules, i.e., separating complex decision-centric logic from standard workflow logic in the aspect-oriented flavor. Paschke et al. [40, 92] propose a heterogeneous service-oriented integration of rules (decision rules and complex event processing reaction rules) into BPM to describe business processes. The decision-centric activities can be semantically represented by declarative rules and heterogeneously integrated into BPEL processes by invoking and executing them as semantic inference services. Such kind of approach alleviates the problem to some extent, but the adaptation is possible as long as they concern the content of pre-identified business rules that fit into the fixed workflow logic (e.g., BPEL) [91].

There are also efforts that dedicate to provide rule-based workflow compositions. Bae et al. [93] propose an ECA rule-based WFMS and execute business processes using an active database. They classify a process flow into several patterns (blocks), each of which becomes an elementary unit of representing process models and identifying ECA rules. The ECA rule-based behavior representation of each block type is based on ACTA formalism. Lin et al. [94] propose a solution to support interorganizational workflow execution based on process agents and ECA rules. ECA rules are used to control internal state transitions, and the process agents are used to control external state transitions of tasks in local workflows. Frincu et al. [5] look at scientific workflows from a distributed system perspective and argue that the rule-based workflow composition has advantages in handling issues with respect to scalability, failure tolerance, data integrity and scheduling. They give an overview of typical workflow issues and solutions, and present a simple ECA rule-based workflow language intended for self-adaptation and auto-generation. The events are viewed as

completed tasks and are placed on the left-hand-side of rules, and subsequent tasks executed after the completed tasks are placed on the right-hand-side of rules. The linking of the output of left-hand-side tasks with the input of the right-hand-side tasks is accomplished by variables. Chen et al. [95] also use ECA rules to realize the service composition. Besides an ECA rule-based workflow formalization, an automatic event composition algorithm is developed to ensure the correctness and validness of the service composition at design time. Compared with [5], the ECA-based workflow in [95] is more sophisticated and a graphical process modeling tool is given. Weigand et al. [91] propose a declarative rule-driven framework FARAO for dynamic service composition. That is, Condition Action (CA) rules describe data dependencies between services involved in the orchestration, and each CA rule corresponds to a message sent by the orchestrator to invoke a service. CA rules are also extended with business rules to steer the decisions in the orchestration. Moreover, they propose a service-oriented approach to business rule management, which allows business rules are maintained and updated outside of the operational services. Compared with FARAO, the task dependencies of scientific workflows in this thesis are described by messaging reaction rules [40, 96], and domain decision-centric policies are expressed by derivation rules that can also access external Semantic Web data. In addition, this thesis combines the agent technology with declarative rules and supports not only structured service orchestration, but also peer-to-peer conversation-based interactions between distributed participants.

Some efforts also strive to employ the benefits of the MASs into the rule-based approach. Rule Responder [97, 98, 99] is a framework for specifying virtual organizations as a semantic MAS to support collaborative teams. Human members of an organization are assisted by autonomous rule-based agents, which use Semantic Web rules to describe aspects of their owner's derivation and reaction logic. The solution presents a flexible and scalable framework to accomplish complex goals and also provides a preliminary architecture of the RbAF of this thesis. There are still some issues that need to be addressed when it comes to scientific workflows, such as service composition, user interaction and exception handling.

In general, there are different types of rules, such as derivation rules, reaction rules, integrity rules, deontic rules, transformation rules and facts [60]. The first two types usually influence the operational and decision processes, and integrity and deontic rules often act as constraints. For the purpose of providing a rule-based workflow language that involves both process structures and decision logic, it is necessary to combine them and take their strengths. Paschke et al. [100] elaborate a homogeneous integration approach, which combines derivation rules, reaction rules and other rule types, such as integrity constraints into a general framework of logical programming. The framework exploits the advantages of different rules and establishes a foundation for the declarative rule-based workflow language of this thesis.

To sum up, most existing efforts mentioned above employ ECA rules to implement the workflow composition. ECA rules following the form *On Event If Condition Do Action* provide active real-time or just-in-time reactions to events and

are right for detection and reaction to events in dynamic computing environments. Moreover, ECA rules are usually defined with a global scope [96] and are suited to represent reactive systems that actively detect or query internal and external events in a global context and then trigger reactions. In contrast to these efforts, this thesis describes scientific workflows by messaging reaction rules, which complement global ECA rules by describing reactive logic that is available in certain local contexts (e.g., a workflow, conversation protocol state or complex situation) and make it possible to employ distributed agents to perform complex tasks. Moreover, the combination of the process logic represented by the messaging reaction rules with the decision logic implemented in terms of derivation rules provides an expressive logic to describe the WsSWFs. Compared to the classic workflow languages that provide simple qualifying conditions, declarative rules are able to represent complex conditional decisions, task/goals and reactions which are needed in scientific workflows. However, it is worth noticing that, although the rule-based workflow languages have advantages in the expressive workflow composition, they usually do not provide good usability as other workflow languages supported by graphical workflow designers.

3.4 Main Scientific Workflow Languages

Scientific workflows were proposed after business workflows and have gained a lot of attention to support scientific computations and experimenting. In the early years of the scientific workflow development, most efforts focus on the workflow creation, reuse, provenance tracking, performance optimization and reliability [101]. A number of SWFMSs have been developed to facilitate scientific activities, such as Kepler [22], Pegasus [25], Triana [25], Taverna [55] and Trails [23].

Kepler [22] is a workflow system built on the Ptolemy II system for designing, executing, reusing, evolving, archiving and sharing scientific workflows. Kepler is a data flow-oriented workflow development environment with an actor-oriented modeling paradigm and uses XML-based Modeling Markup Language (MoML) as its workflow description language. The MoML allows a workflow specification that includes actors, directors, connections, ports and parameters. The actors represent workflow steps and are usually defined as place holders set prior to the workflow execution. They can communicate with other actors via input and output ports. The workflow execution is controlled by directors that orchestrate and mange actors to run workflows. They give the execution semantics of actor dependencies, i.e., they define when and how actors execute. Over 350 ready-to-use actors which act as processing components in Kepler can be easily customized, connected to perform an analysis, automate data management and integrate applications efficiently. The main focus areas of Kepler so far are ecology, geology, biology and astronomy.

Pegasus [25] is a framework that maps high-level scientific workflow descriptions onto low-level distributed resources. Pegasus abstract workflows are represented in a form of Directed Acyclic Graph in XML Format (DAX), which has syntax for expressing jobs, arguments, files, and dependencies. The DAX can be generated

using a Java, Perl and Python Application Programming Interface (API). Such DAX-based abstract workflows can be mapped to executable workflows with available cyber infrastructure resources and then be submitted to DAGMan (a workflow executor) for execution. Pegasus now can bridge existing cyber infrastructures to coordinate multiple distributed resources. Moreover, Pegasus provides a flexible structure for dealing with failures and recovering by retrying tasks. The main focus areas of Kepler so far are astronomy, bioinformatics, earthquake science, gravitational wave physics, ocean science and limnology.

Taverna [55] is an open-source, Grid-aware WFMS of myGrid project [2]. It is implemented as an SOA and provides suite of tools to design, edit and execute workflows. A Taverna workflow is a linked graph of processors, which represent Web services or other executable components, each of which transforms a set of data inputs into a set of data outputs. Taverna uses an XML-based workflow language called SCUFL for workflow representation. The SCUFL is a data flow-centric workflow composition language that defines a graph of data interactions between different processors. When combining processors into workflows, users only need to consider in terms of the data consumed and produced by logical services and connect them together. A set of control links that specify running order dependencies where direct data flow is not required. Taverna is currently targeting a wide range of areas within biology.

Triana [23] is an open source problem solving environment which is particularly good at automating repetitive tasks. There is no explicit language to support control constructs, and it employs an intuitive Graphical User Interface (GUI) for scientists to create workflows from Open Grid Services Architecture (OGSA) Grid services with a minimum of effort or even no programming. In other words, loops and execution branching are handled by components defined in XML files. Users can compose workflows by dragging and dropping these programming components. Besides the GUI, Triana has an underlying system, which consists of a collection of interfaces that bind to different types of middleware and services. The integration allows Triana can represent any services or primitives exposed by such middleware, and these tools can be interconnected to create mixed-component workflows [102]. The main focus areas of Triana so far are astronomy and physics.

In addition, Askalon [103] provides a graphical editor for the UML-based modeling of Grid workflow applications and distributed and parallel programs. It takes an XML-based language Abstract Grid Workflow Language (AGWL) to describe workflows at a high level. Swift [26] uses a scripting language called SwiftScript, which is designed for composing application programs into parallel applications that can be executed on multicore processors, clusters, grids, clouds and supercomputers [104].

In summary, these systems are mainly designed for structured compute or data-intensive applications, and each system has its own application areas. For the purpose of supporting an adaptive execution at runtime, most of them provide an abstract workflow description at a high level and hide underlying specific implementation details. They provide proprietary workflow languages and support different

models of computation. Table 3.1 shows a summary of the main workflow languages employed in the existing SWFMSs. Moreover, they mainly allow single scientist to compose and manipulate workflows [39]. For example, in Kepler the actors take their execution instructions from the director. In addition, none of these SWFMSs provides features to specify human tasks in workflows [3].

Table 3.1: Main Scientific Workflow Languages

Name	Workflow Language	Features
Kepler	MoML	Data flow-oriented; directors control actors
Pegasus	DAX	Mapping abstract workflows (DAX) to executable workflows (DAG)
Tarverna	SCUFL	Data flow-centric
Triana	No explicit language	Component programming; data flow-oriented
ASKALON	AGWL	A high level workflow abstraction
Swift	SwiftScript	Distributed parallel scripting

3.5 Efforts on Weakly-Structured Workflows

There are also efforts that try to support complex weakly-structured workflows from different perspectives. Van Elst et al. [105] state that the weakly-structured workflows are incomplete process models. Such workflows have exploratory knowledge-intensive activities and cannot be sufficiently modeled by classical, static process models and workflows. To support these workflows, they propose a process-oriented knowledge management approach that integrates process modeling and workflow enactment and facilitates active information support in dynamic changing environments. In this way, it is possible that the knowledge-intensive processes are started with a partial model and are refined later during the execution. This approach is a variant of the standard case-based reasoning with reuse and continuous adaptation (and hopefully improvement) of knowledge-intensive processes in an organization [105]. However, the knowledge-intensive tasks considered in this thesis usually involve domain-specific decision logic, and this thesis describes and executes them by providing a flexible rule-based agent-oriented framework.

Papavassiliou et al. [6] focus on the weakly-structured knowledge-intensive business process modeling and present an approach for integrating BPM and knowledge management. They state that the weakly-structured business processes are both knowledge-intensive and weakly-structured. Such processes have central decision steps that require personal judgment or access to much specific information. The sequence of processing steps of a process may vary for specific instances, and complex decisions are embedded into the process as black boxes. Also, they present a workflow modeling tool that can explicitly incorporate knowledge tasks and objects

into business process models. However, their approach gives a theoretical framework to enhance business process modeling with knowledge management activities, and does not provide a proof-of-concept implementation.

DECLARE is a prototype of a WFMS that uses a constraint-based process modeling language for the development of declarative models describing loosely-structured processes [106]. DECLARE defines constraints on the execution orders of loosely-structured processes, which can be completed in any order that accords with the constraints. Compared with an over-specify imperative model, the declarative model of DECLARE implicitly defines all possibilities that do not violate a given constraint. Based on Linear Temporal Logic (LTL), DECLARE provides constraint templates to model constraints "init", "1..*", "response", "responded existence" and "responded absence". In addition, by combining YAWL with DECLARE, it is possible to support large processes containing mixtures of loosely-structured and high-structured processes. In contrast to DECLARE, which focuses more on the constraint-based workflow structures, the rule-based workflow specification of this thesis focuses more on describing complex domain-specific decision logic and conditional reactive logic of the WsSWFs, which determine intelligent workflow routings at runtime. With the reactive event messaging between distributed agents, both structured service orchestration and peer-to-peer conversation-based interactions between distributed participants are supported. Moreover, LTL is often used for abductive planning, while this thesis focuses on the execution of decision logic of the WsSWFs, i.e. deductive reasoning with rules based on facts (The differences between abduction and deduction can be found in Section 2.9).

As it can be noticed from the aforementioned efforts, the weakly-structured processes are more complex than the structured compute/data-intensive workflows. The existing efforts mainly focus on declarative control flow constraint representation and theoretical modeling, and do not consider (domain-specific) decision logic in the weakly-structured processes. This thesis considers the weakly-structured processes from the technical perspective and provides a rule-based workflow language not only for the flexible workflow composition, but also for the expressive (domain-specific) decision logic expression. Moreover, a distributed, multi-agent-based execution environment is provided to support the workflow execution.

3.6 Semantic-Based Workflow Compositions

For the purpose of realizing an adaptable execution of scientific workflows, it is important to decouple the workflow description from execution and support a dynamic service binding at runtime. In other words, workflow engines should dynamically discover available services in terms of the requirement description of a task. There are two prerequisites to meet this goal: first, the tasks of a workflow should be described in an abstract way that is independent from underlying implementation technologies; second, the workflow engine should be capable of understanding the abstract task description and finding the most appropriate one from available heterogeneous

services. Therefore, it is crucial to give well-defined meanings to workflow-related concepts and enable the workflow engines to process them intelligently.

Currently, Semantic Web is an active community engaging extending traditional Web resources with additional metadata and semantic knowledge and allows knowledge to be shared and reused across applications, enterprises, and community boundaries. Semantic Web technologies, such as Resource Description Framework (RDF), SPARQL Protocol and RDF Query Language (SPARQL), Web Ontology Language (OWL) enable people to create data stores on the Web, build ontologies, and write rules for handling data. Ontologies are agreements about shared conceptualization, and they are capable of specifying the concepts, relationships and other distinctions that are relevant for modeling a domain of interest. They are structural frameworks of organizing information and provide a foundation upon which machine understandable knowledge can be obtained. With the help of ontology technologies, the semantic workflow description includes not only syntactic structure that reveals what type of inputs or outputs it expects to receive, but also semantic information that describes more complex validation rules for both inputs and outputs [5]. This section presents some solutions of extending the syntactic interface description of traditional Web services with semantic information as well as several ontology-based workflow representations.

3.6.1 Semantic Web Services

Web services are software functions provided over the Web and support interactions between different machines connected through a network. As an implementation of SOA, Web services are delivered over the Web using technologies such as XML, WSDL, Simple Object Access Protocol (SOAP), and Universal Description Discovery and Integration (UDDI). Moreover, Web services are platform independent. Based on these XML-based standards, it is possible to integrate heterogeneous resources over the Internet, thereby improving resource sharing. Web services are also loosely-coupled. WSDL provides a resilient relationship between systems and organizations, which brings the flexibility that a change in the underlying implementation of an application does not necessarily require a change in its definition.

However, the original Web service technology only standardizes syntactic specification and communication and makes the Web service usage and integration need to be inspected manually. With the combination of ontology and Web services, semantic Web services provide an abstract description lacking in traditional Web service description and enable users and software agents to manage Web services automatically. This section presents some prominent efforts aiming at semantically describing Web services and introduces their strengths and weaknesses, respectively.

OWL for Services (OWL-S) [107] builds on OWL and describes the properties and features of Web service in a machine-readable markup language. It provides three essential types of knowledge about a service: *ServiceProfile*, *ServiceModel* and *ServiceGroundling*. The *ServiceProfile* and *ServiceModel* are abstract representations of a Web service, while the *ServiceGrounding* deals with the mapping from

an abstract service representation to a concrete specification of the service description elements, such as WSDL. This approach is also known as a top-down approach, which begins by identifying all of the relevant concepts, organizing them into a compact, high-level taxonomy and system of axioms, and then proceeding from there to more specific concepts and axioms [108]. OWL-S is the first specification submitted to W3C in 2004 and has affected the development of semantic Web services. However, it has not been widely applied because of its complexity and the top-down approach which does not fit well with industrial developments of SOA [109].

WSDL-S [110] defines a mechanism to semantically annotate WSDL documents. Unlike OWL-S, the WSDL-S extends WSDL with extra elements and attributes. This approach is also known as a bottom-up service modeling, which begins with domain specific concepts and then develops the ontology up or out from there [108]. Semantic annotations are not tied to any particular ontology representation language and can be provided with different languages, such as OWL and UML. Compared with OWL-S, WSDL-S meets the practical situation better and can be easier applied.

Based on WSDL-S, Semantic Annotations for WSDL and XML Schema (SAWSDL) [111] defines mechanisms by which semantic annotations can be added to WSDL components. SAWSDL does not specify a language for representing semantic models. Instead, it allows concepts defined either within or outside the WSDL document to be referenced by the WSDL components as annotations. Such semantic annotations expressed in formal languages can help disambiguate the description of Web services during automatic discovery and composition of Web services. Similar to WSDL-S, it also adopts the bottom-up approach.

Web Service Modeling Ontology (WSMO) [112] provides solutions to describe all relevant aspects of Semantic Web services with the top-bottom approach. Taking Web Service Modeling Framework (WSMF) [113] as a starting point, WSMO reuses its four different main elements for describing semantic Web services: ontologies providing the terminology used by other WSMO elements, Web service descriptions defining the functional and behavioral aspects of a Web service, goals representing user desires, and mediators handling interoperability problems between different WSMO elements.

Semantic Web Services Framework (SWSF) [114], which includes Semantic Web Services Language (SWSL) [115] and Semantic Web Services Ontology (SWSO) [116]. SWSL is used to specify formal characterizations of Web service concepts and descriptions of individual services. SWSO presents a conceptual model by which Web services can be described and a formal characterization of the model. In contrast to WSMO, SWSL focuses more on extending the functionality of its rule language [114].

WSMO-Lite [117] identifies a simple vocabulary for semantic descriptions of services and fills the SAWSDL annotations with concrete semantic service descriptions. A conceptual model for Web service descriptions includes: information model descriptions defining a data model for input, output, and fault messages as well as for the data relevant to the other aspects of the service description; functional descriptions define service functionality, nonfunctional descriptions, such as price, QoS;

behavioral descriptions defining external and internal behavior; and technical descriptions defining messaging details, such as message serializations, communication protocols and physical service access points.

With more and more distributed computing technologies depending on loosely coupled distributed services, dynamic service provider and service consumer relationships, there is an increasing demand for automated management and monitoring of service contracts like Service Level Agreementss (SLAs). In addition to semantic service interface descriptions, Rule-Based Service Level Agreement (RBSLA) allows representing non-functional properties of Web services for IT service management [118, 119]. Based on SLAs, RBSLA also allows for semantic mediation of information flow in cross-organizational business process models.

The specifications mentioned above have been used in real world applications, some even have been submitted to W3C and have become member submissions. However, their applications are currently limited to prototypes due to their complexities. Moreover, Web services are not the only resources that are used in scientific workflows. Due to the diversity between the scientific disciplines, different resources are often involved in scientific workflow execution, such as mathematical, data analysis tools (e.g., R, MATLAB). Also, there are tools that work with Semantic Web technologies, such as rule engines, ontology reasoners and SPARQL query engines; it is necessary to integrate these resources into workflows in order to perform domain-specific knowledge-intensive tasks.

3.6.2 Ontology-Based Workflow Specifications

Semantic Web services describe Web services with common vocabularies and facilitate efficient integration of distributed heterogeneous Web services. They enable workflows to be declaratively described instead of giving concrete implementation details and leave the workflow engines more space to select appropriate available services at runtime in terms of the requirements.

In addition, there are research efforts trying to provide flexible workflow representations based on service ontologies. Lee et al. [120] introduce a five-step process of building a ubiquitous service ontology, ranging from scenario-based ubiquitous smart space modeling, building domain ontology, defining the specification of services, building service ontology to using services. The services are abstractly described by a specification, which is further adapted to OWL-S to build a service ontology. The most appropriate service could be discovered based on the service ontology and a failed service can be replaced with an alternative available service that has the same effect. Qin et al. [121] present a semantic scientific workflow composition by separating data semantics and data representation as well as *Activity Function* and *Activity Type*, respectively. The upper ontology developed in the paper defines three main concepts: *Function*, *Data*, and *Data Representation*, as shown in Figure 3.3 [121]. The *Function* is a super class of both generic and domain-specific functions which are implemented by Web services, executables, Java classes, etc. The *Data* represents any kind of workflow data and can also be specialized into

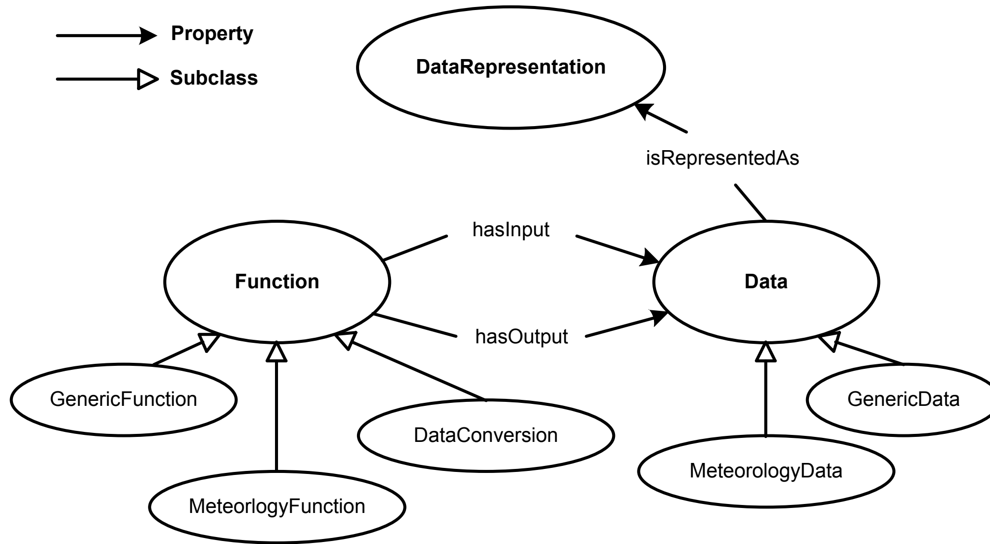


Figure 3.3: The Upper Ontology of AWDL

domain data. The *Data Representation* describes storage-related information about *Data*, that is, how data is stored in computer systems. An algorithm is also given to map semantic scientific workflow representations into syntactic workflow representations.

Pinheiro da Silva et al. [122] present a scientist-centered tool that enables scientists to encode their domain knowledge (e.g., data and methods) into a Workflow-Driven Ontology (WDO), which is then used to generate abstract workflows. Based on the WDO metamodel, scientists are allowed to identify concepts from their domains and classify them into raw data, derived data and methods (which represent the concepts related to tool functionalities, services, algorithms or anything used to access and transform data). The abstract workflows are generated via a workflow generation algorithm, which takes target data given by users and recursively searches for WDO methods that specify the target data as their outputs, until all data components used by a method become raw data. Scientists then assess the generated workflows and revise the concepts and relationships defined in the WDO ontology if necessary. The highlight of this work is that it builds workflows in terms of domain ontologies (i.e., WDOs) and is more acceptable by scientists.

Moreover, inspired by the spirit of the semantic Web services, some efforts strive to expand the existing Web service ontologies to provide a semantic declarative process-oriented representation. For instance, OWL-S also views a workflow as an *atomic process* or a *composite process* from the functional perspective [107]. An atomic process is a set of actions that a service can perform by engaging it in a single interaction. A composite process can be decomposed into other (non-composite or composite) processes in terms of control constructs, such as sequence, if-then-else, split + join, choice, condition and iterate. The composite process is a specification

of behavior, and the client executes it by sending and receiving a series of messages. For each atomic process, there must be a grounding that describes where the inputs come from and where the outputs go. A *simple process*, on the other hand, provides an abstraction view on both atomic and composite processes. The top level of OWL-S process ontology is shown in Figure 3.4 [107]. Besides the control flows, the OWL-S also supports data flow in terms of a consumer-pull conversion as well as parameter bindings.

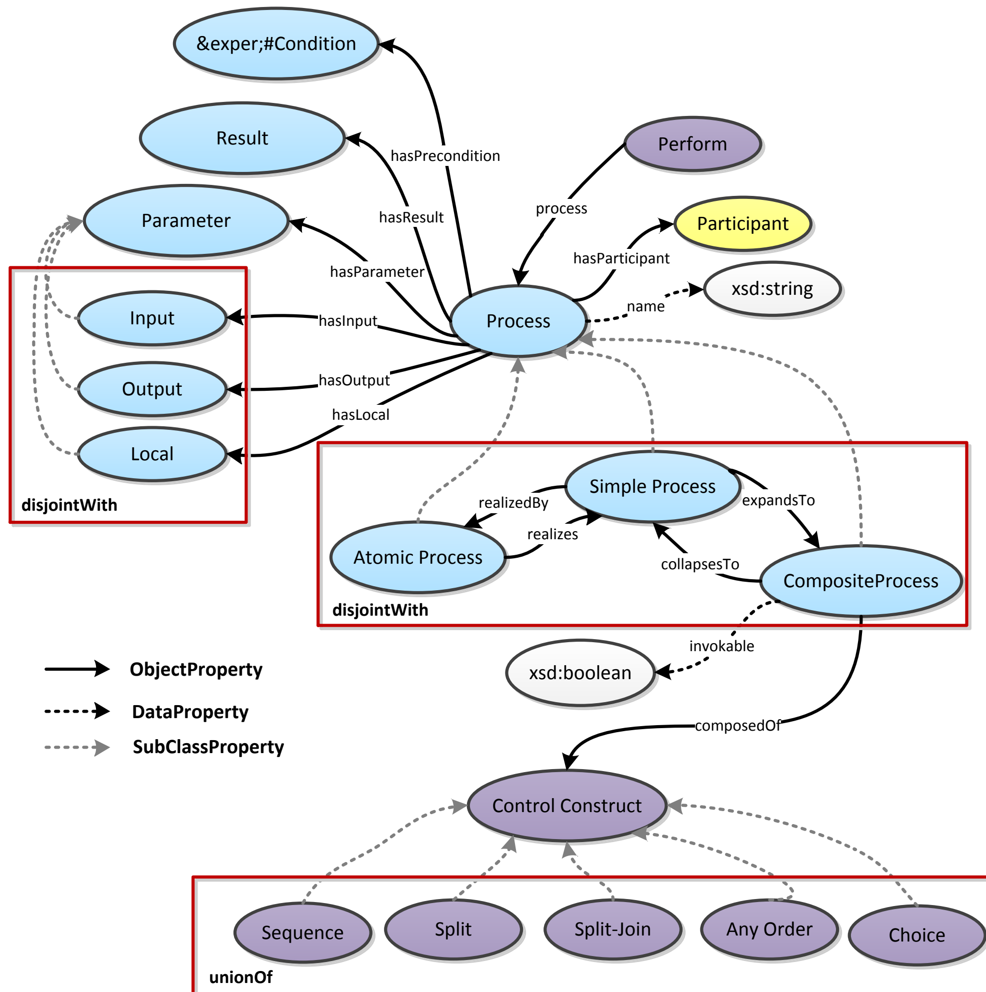


Figure 3.4: Top Level of OWL-S Process Ontology

However, OWL-S focuses on modeling a workflow that is internal to a single service, i.e., the steps specified in a process always interact with a service [123]. OWL for Workflows and Services (OWL-WS) [124] is an extended version of OWL-S invented by the NextGrid project [125]. It enforces OWL-S and allows different steps in a process to interact with different services, i.e., a grounding can be composed of components referring different services to describe more realistic and complicated

processes. However, the application of both OWL-S and OWL-WS have not been prevalent because of lacking of tools to support the development [126].

Additionally, Process Specification Language (PSL) can be used for the representation of manufacturing, engineering and business processes [127]. The process representation is based on a set of logic terms specified in an ontology, which describes the process components and their relations. The ontology has been published as an ISO standard 18629. Furthermore, it has a formal logic-based semantics using temporal situation calculus which supports both projection (i.e. deduction) as well as planning. The whole layering of PSL is highly expressive. However, PSL is a representation language that provides semantic and computational information for each workflow element, rather than a programming language that supports the workflow execution. Moreover, it offers advanced features that are not necessary for the WsSWFs, such as process planning.

To sum up, the semantic Web technologies bring workflows with semantic support for the abstract workflow composition and dynamic service discovery. However, the ontology-based workflow modeling is a kind of process representation, and it cannot specify complex decision logic and flexible mechanisms to cope with uncertain situations at runtime. In general, they are mainly designed for automatic service invocation and abstract workflow composition based on their semantic description, and it is complicated to employ ontologies to express workflow tasks and their dependencies.

3.7 Summary

This chapter presented a review of different solutions for flexible workflow composition. As it stated above, most efforts focus on business processes and aim to provide efficient processes to customers. The classical workflow languages are effective for process-oriented processes, but they have static logic and are mainly executed in a central manner. The existing SWFMSs mainly focus on structured compute/data-intensive processes; their workflow execution is controlled by a centralized engine to meet the requirements of scientific data management; they have proprietary workflow languages and cannot specify the weakly-structured knowledge-intensive workflows. The ontology-based workflow representations enrich the workflow elements with semantic information and support dynamic service discovery and the workflow composition at a high level; but they are not expressive to specify complex decision logic and flexible mechanisms to cope with uncertain situations. Moreover, although there are efforts that have been made to support the weakly-structured processes, they mainly focus on declarative control flow constraint representation and theoretical modeling, and do not provide an expressive (domain-specific) decision logic description for the workflows.

Table 3.2 summarizes the strengths and weaknesses of different flexible workflow composition approaches mentioned in this chapter. The rule-based approaches encode policies as declarative rules and have advantages in describing scientific

Table 3.2: Comparison of Flexible Workflow Composition Solutions

Name	Advantages	Disadvantages
Classic Workflow Languages	Efficient service composition; agnostic and reusable services	Block-structured; rigid logic
Agent-oriented Composition	Flexible cooperation, coordination, and negotiation	Limited agent behavior description; the decentralized execution
Rule-based Languages	Intuitive formal semantics; direct support for business and science policies; flexibility; adaptability; reusability	Bad usability
Main scientific workflow languages	Focus on structured compute/data-intensive processes	Proprietary workflow languages; cannot specify human tasks
Weakly-structured Workflows	Focus more on the workflow structure	Domain-specific decision logic is not considered
Semantic-based Composition	High level resource description	Limited process representation

processes. Besides the advantages mentioned in Section 3.3, the logic-based rule languages also have strengths, such as automated rule chaining by resolution and variable unification, a compact and comprehensive human and machine-oriented representation and high levels of automation of flexibility to adapt to rapidly changing requirements, relatively easy for end users to write rules and hence rapidly engineer and maintain rule-based decision and reactive logic. Moreover, compared with the centralized workflow execution, the agent-based frameworks have demonstrated their powerful flexibility in workflows requiring interactions between different participants.

This thesis proposes a rule-based workflow language to describe the WsSWFs and implements a distributed agent-based system, RAWLS, as the workflow execution environment. The combination of declarative rules with the agents not only provides an expressive declarative rule-based workflow specification, but also supports an adaptive workflow execution.

In what follows, Chapter 4 presents the conceptual framework, RbAF, to support the WsSWFs. Chapter 5 introduces a formal semantics of the declarative rule-based workflow language, and the prototype system, RAWLS, is given in Chapter 6.

Part II

Conceptual Framework

Chapter 4

Rule-Based Agent-Oriented Framework

Contents

4.1 Hierarchy of the Rule-Based Workflow Specification 62

4.2 Upper-Level Workflow Ontology 62

4.3 Declarative Workflow Specification 64

4.3.1 Reaction Rules 64

4.3.2 Event-Driven Workflow Execution 67

4.3.3 CEP-Based Workflow Pattern Modeling 68

4.4 Domain Decision-Centric Logic Description 72

4.4.1 Derivation Rules 72

4.4.2 Semantic Web Data Query 75

4.5 Integrating Orchestration with Choreography 78

4.6 Human Interaction 80

4.7 Exception Handling 82

4.8 Summary 83

More and more efforts have reached a consensus that scientific workflows require a flexible design for the purpose of representing complicated process logic and adapting to dynamic changes at runtime. In the current state-of-the-art, partial solutions for the issues mentioned in Chapter 3 have been proposed. However, some core issues related to the WsSWF execution (see Section 1.2) are still unsolved. This chapter presents a conceptual workflow framework, called RbAF, which not only provides a declarative rule-based workflow language combining messaging reaction rules and derivation rules, but also employs multiple inference agents as the workflow execution environment.

On one hand, an agent in computer science is defined by Wooldridge [128] as follows:

> "*An agent is a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives.*"

Besides the autonomy, an agent also has abilities, like perceiving their environments and automatically responding to changes (*reactivity*), starting new actions

on their own to pursue their own or given objects (*proactivity*), and engaging in conversations with other agents in cooperative ways (*social ability*) [128].

On the other hand, the rule-based programming can not only describe the process behavior in a declarative manner, but also support a compact and comprehensive logical domain knowledge representation with automated reasoning. One interesting application of the rule-based programming is that it can be directly used as the basis for describing the agent behavior.

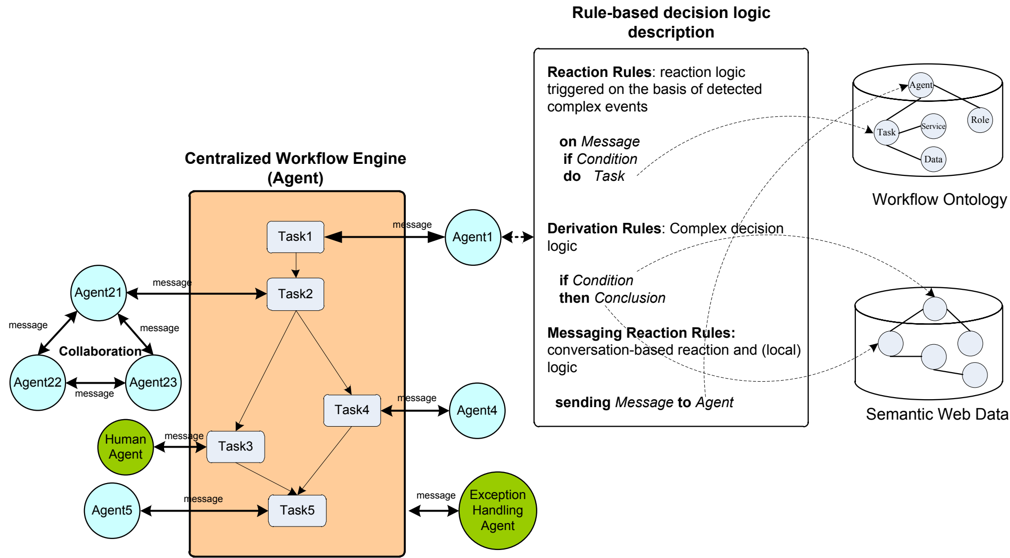


Figure 4.1: Rule-Based Agent-Oriented Scientific Workflow Framework

Figure 4.1 shows the overall architecture of the RbAF, which is characterized as providing support to the WsSWFs in two levels:

- the workflow *definition*: a workflow is described via the composition of a group of abstract tasks, each of which describes certain scientific goal and is independent of any specific implementation. A task can be either a *primitive* task or a *composite* task that defines the execution order of a set of sub-tasks (aka. a sub-workflow or sub-process).
- the workflow *execution*: during the workflow execution, each abstract task is allocated to an agent, which decides on its own to execute the task. The task execution can lead to one or more results, and the agent may return the task results to the original requester or send the results to other agents to perform subsequent tasks. That is, such agents can negotiate and collaborate with each other on performing complex tasks.

The RbAF employs two main rule types to represent WsSWFs: *messaging reaction rules* and *derivation rules*. More precisely, the workflow composition is described by *messaging reaction rules*, which describe (abstract) processes in terms of

message-driven conversations between agents. In other words, the task dependencies are described by the order of sending and receiving messages between agents. *Derivation rules*, which are often used for derivations of knowledge as conclusions from given knowledge, are mainly used to describe knowledge-intensive decision-centric steps in workflows. With the combination of these declarative rules, the RbAF provides a declarative rule-based approach to describe the WsSWFs.

For the purpose of enriching the workflow description with semantics, an upper-level ontological workflow metadata model (workflow ontology) is given to define general workflow concepts and their relationships. The upper-level workflow ontology provides a well-modularized schema and allows the general workflow concepts to be further specialized with domain-specific ontologies. With the workflow ontology, it is possible to automatically find alternative resources (e.g., agents), thereby allowing the workflow execution to resume if a resource is unavailable at runtime. In addition, the RbAF provides access to external Semantic Web data (e.g., domain vocabularies and ontologies) and reduces the effort required to achieve the similar logic by declarative rules.

The RbAF combines two ways of the workflow execution: *orchestration* and *choreography*. That is, a centralized workflow engine (also an agent) takes control of the execution of a scientific workflow and completes it via the composition of distributed agents, which are not aware of the whole complex workflow (aka. *orchestration*). Moreover, messaging reaction rules describe interactions between multiple participants, making it possible to build *choreography* workflows which focus on collaboration and message exchange between multiple participants.

The RbAF integrates human users into the WsSWFs. For the tasks that need to be performed by human users, the RbAF can transfer the workflow control to human users who are responsible for them. To do so, a Human Agent (HA) manages the life cycle of the human tasks and provides a Web interface for scientists to operate on the tasks, thereby supporting user interaction with the workflow system.

The RbAF also provides an escalated aspect-oriented, event-driven exception handling at runtime. That is, the exceptions are usually handled separately by an Exception Handling Agent (EHA) by replacing the failed sub-process dynamically. Once an exception cannot be handled automatically by the EHA, the exception will be escalated to human users to make a decision or provide the required resources. These exceptions handled by the EHA and human users are also known as *expected* and *unexpected* exceptions, respectively.

Note that this thesis focuses on specifying scientific workflows with declarative rules and providing a flexible agent-oriented workflow execution environment. Typical MAS topics, such as coordination mechanisms, organizational environment modeling and norms in MASs mentioned in Section 3.2 are out of the scope of this thesis.

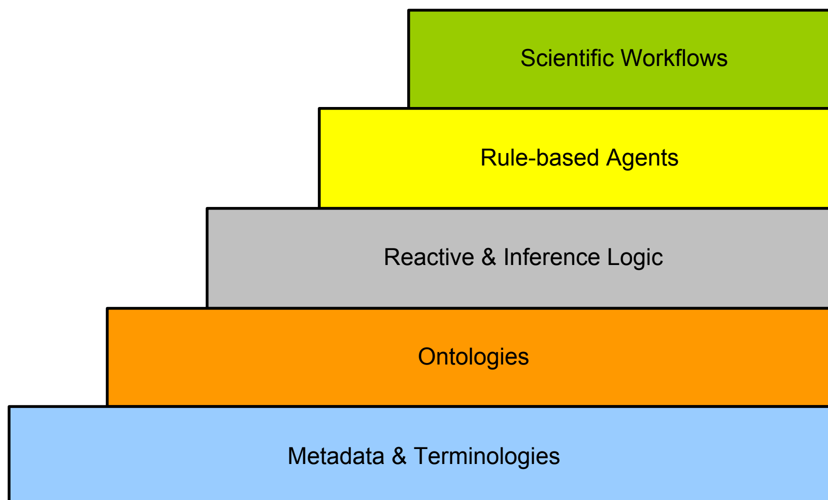


Figure 4.2: Hierarchy of Rule-Based Workflow Description

4.1 Hierarchy of the Rule-Based Workflow Specification

Following the spirit of the Semantic Web stack [129], the rule-based workflow language presents a similar hierarchical knowledge structure, as shown in Figure 4.2.

The RbAF builds scientific workflows hierarchically, and each layer is built on top of another. *Metadata and terminologies* (concepts) at the bottom layer define general concepts in scientific workflows. *Ontologies* describe logical relationships between the concepts, and they can be further specialized with domain-specific ontologies (see Section 4.2). On top of the ontologies, declarative rules describe the *reactive and decision logic* of a workflow. The simple terms of rules can be assigned with concepts defined in these ontologies, thereby providing a lightweight combination between rules and ontologies. *Scientific workflows* are further described by the *agents*, whose behavior is programmed with declarative rules.

4.2 Upper-Level Workflow Ontology

The upper-level workflow ontology defines a hierarchy of semantically linked general workflow concepts and their logical relationships, as shown in Figure 4.3. The following are the general concepts involved in the workflow ontology:

- *Data* is a super class of any kind of workflow data, which can be used or generated by tasks in scientific workflows.
- *Agent* is an entity that is considered reactive and social. In the RbAF, an agent is either a software component or a user.
- *Task* is an abstract activity performed by agents to reach a certain goal. A task could be either primitive or composite. Each *primitive* task is an elemen-

tary unit, and a composite task modularizes the execution order of a set of (primitive or composite) tasks, also known as a sub-process.

- *Role* is a meaningful collection of tasks performed by one or more agents. A role is responsible for the tasks it encompasses. The roles can be hierarchically composed and are assigned to the agents.

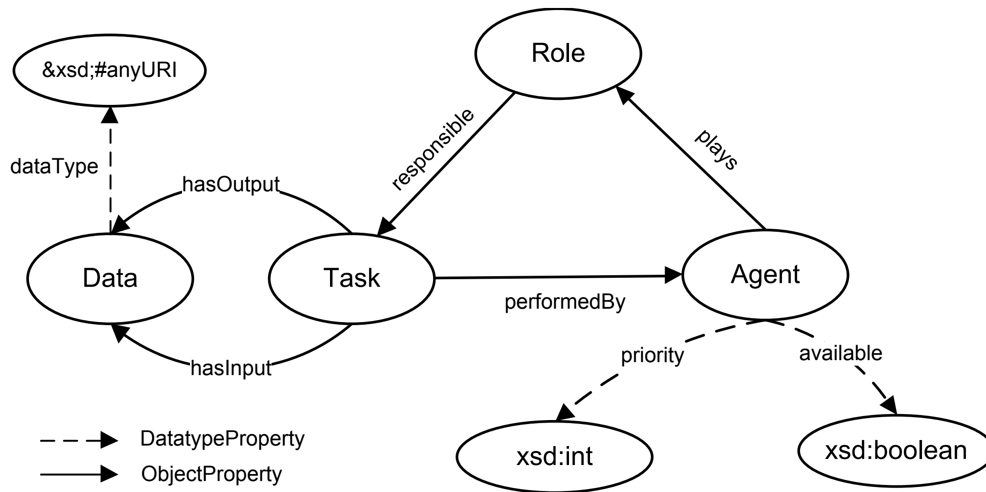


Figure 4.3: Upper-Level Workflow Ontology

Besides the workflow concepts, the properties of these concepts are defined as follows:

- *hasInput* and *hasOutput*. A *task* has inputs and outputs, which are *data*, i.e., *hasInput(Task, Data)* and *hasOutput(Task, Data)*.
- *dataType*. Every data has a type specified using a Uniform Resource Identifier (URI) that is a specification of a class (or an XML Schema datatype) that the data value belongs to.
- *plays*. Each agent always plays certain *roles*, which describe the responsibility of the agent, i.e., *plays(Agent, Role)*.
- *responsible*. The relationship *responsible(Role, Task)* specifies a task for which the role is responsible.
- *performedBy*. A *task* is performed by one or more *agents*, i.e., *performedBy(Task, Agent)*.
- *available*. The agent status is denoted by a property *available*, which can be available and unavailable. In the RbAF, only available agents can be employed to perform a task. Moreover, during the workflow execution, the agent status can be modified by the agents automatically.
- *priority*. Each agent has a property *priority* that controls if it can be selected first. During the workflow execution, one agent that is available and has the highest priority is selected.

In specific applications, these general concepts and relationships defined in the upper-level workflow ontology can be further specialized with domain-specific ontologies. For example, in the process of protein prediction result analysis (see Section 2.5), the process inputs *up:Protein* and *up:Concept* can be defined by the concepts of UniProt core vocabulary [130].

During workflow execution, the workflow ontology can be accessed by any of the agents as the shared common knowledge. One task can be dynamically allocated to a responsible agent in terms of the workflow ontology. Moreover, it is also possible to automatically find alternative resources (e.g., agents), thereby allowing the workflow execution to resume if a resource is unavailable at runtime. More details about exception handling can be found in Section 4.7.

Compared with the related efforts mentioned in Section 3.6.2, the workflow ontology of this thesis simply defines the basic information of a task (e.g., task name), instead of describing the composition of a composite task. That is, a composite task (sub-process) is defined the same as the primitive tasks in the workflow ontology, which makes it possible to hierarchically integrate the composite tasks into other processes. In this thesis, the RbAF describes the workflow composition by more flexible and understandable messaging reaction rules, and more details can be found in the following section.

4.3 Declarative Workflow Specification

Existing efforts have shown that the control-flow dependencies of a workflow can be described by rule-based languages, e.g., ECA rules, which strictly follow the *On Event If Condition Do Action* paradigm. They are usually defined in a global scope and react on internal events of the reactive system, such as changes (updates) in an active database [131, 40]. However, in a distributed environment with independent system nodes, event processing not only requires notification and communication mechanisms, but also needs to be done in a local context, e.g., in a conversation or a workflow. In this thesis, the RbAF describes the workflow composition by messaging reaction rules, which define (abstract) processes in terms of message-driven conversations between parties and describe their associated interactions via asynchronously sending and receiving event messages.

4.3.1 Reaction Rules

Messaging reaction rules involve both receiving and sending messages between distributed agents. This section describes how *reaction rules* are used to detect external event messages.

Reaction rules are (behavioral or action) rules that react to occurred events (external events or changed conditions) by executing actions [132]. In the RbAF, an agent follows the *sense-reason-act* pattern of reaction rules and interacts with external environment via conversation-based messaging reaction rules. A reaction

rule defines an event template, some of whose parameters are variables, and the event template matches single events by replacing the variables with values.

Reaction rules are the collection of reactive rules, which specify and program reactive systems in a declarative manner, and the most general form of a reaction rule consists of the following parts [132]:

define reaction rule *reaction_rule*
on [*event*]
if [*condition*]
then [*conclusion*]
do [*action*]
after [$post - condition$]
else [*elseconclusion*]
elseDo [*else/alternativeaction*]

A reaction rule consists of parts of the event or situation processing (e.g., detection, computation), condition verification, action invocation and post-condition verification, where the condition and (especially) the post-condition parts are optional [132]. Depending on the parts of the general syntax, reaction rules can be specialized into different types:

- *Derivation (Deduction) rules (if-then).* Derivation rules can derive new knowledge from other knowledge by an inference or mathematical calculation. They are often used to answer a query or search to accomplish a goal (e.g., in a decision process).
- *Production rules (if-do).* A production rule performs actions if a stated condition is true. Production rules are often used to perform actions in certain situations, which are often considered as a set of conditional tests.
- *Trigger rules (on-do).* A trigger rule performs actions whenever a stated event occurs. Trigger rules are rarely used, since they can act as *production rules* by implementing the event detection as the condition restrictions.
- *ECA rules (on-if-do).* ECA rules have an explicit event part compared with *production rules.* They can be extended with post-conditions after the condition part or be condensed to *trigger rules.*

In the RbAF, the messages (events) passing between distributed agents are associated with a conversation identifier to reflect the process execution. This is crucial to keep all tasks of a process instance running in one conversation, especially for processes that involve the synchronization of tasks running in parallel. For example, Figure 4.4 shows a simple workflow *A* that begins with two parallel tasks: *add* and *minus*, which perform addition and subtraction operations, respectively. They are synchronized by a *multiply* task that multiplies the results of *add* and *minus*. The synchronization of *add* and *minus* can only occur when both of them have completed. Suppose that two instances of the workflow *A instance1* and *instance2*

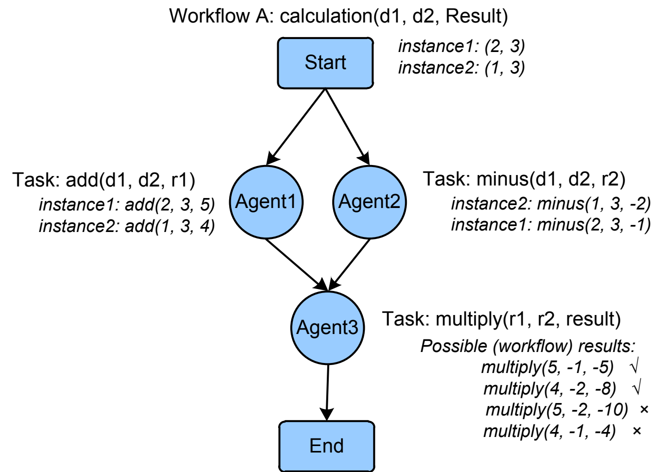


Figure 4.4: Multiple Workflow Instances

are started concurrently, i.e., there is a possibility that two *add* instances and two *minus* instances are running at the same time. If the messages are not associated with a conversation identifier, the synchronizer may use the results from different workflow instances for the multiplication. For example, the synchronizer may take the addition result of the *instance1* and the subtraction result of the *instance2* for the subsequent multiplication.

The conversation identifier is also helpful to implement advanced synchronization workflow patterns, which need to distinguish different workflow instances. More details can be found in Section 7.1. Besides the conversation identifier, a message passing two agents also includes the following information:

- *Protocol* defines the protocol of message passing.
- *Sender* and *Receiver* denote the source and destination of the message, respectively.
- *Performative* describes the pragmatic context in which the message is sent, e.g., FIPA Agent Communication Language (ACL) [133]. The message context gives meaning to the message.
- *Content* denotes the payload of the message.

The action part of a reaction rules describes the procedure of processing events (or performing tasks). The actions of performing a task could be adding/retracting knowledge, variable assignment, messaging activities (i.e., message sending and receiving), and execution of (external) functions. In contrast to global ECA rules, messaging reaction rules support performing complex actions locally within certain contexts. In other words, message sending and receiving activities can be embedded into the action part of reaction rules to employ distributed agents to perform com-

plex tasks, making it possible to implement a rule-based branching workflow logic and support the distributed workflow execution.

4.3.2 Event-Driven Workflow Execution

In the RbAF, the agents act as hosts to distributed resources, and the interactions between distributed agents are represented by messaging reaction rules. That is, an agent receives a request of performing a task, processes the request by using internal or external resources, then returns the results to the requester or sends the results to other agents. In other words, the workflow execution is driven by sending and receiving messages between agents, and thus the RbAF can also be referred to as an Event-Driven Architecture (EDA).

The event-driven RbAF is in line with the *data-driven* scientific workflow execution. This is because the EDA provides an appropriate model for active data sharing based on the production and consumption of events [134]. In the RbAF, the data passing between tasks is regarded as event messaging between agents. In other words, the event messages carry data sharing between tasks. Figure 4.5 presents the event-driven workflow execution of an abstract workflow. The first task *Task1* of the workflow is triggered by an event *e1*, which carries the data required by *Task1*. After *Task1* completes or generates required data, events *e2* and *e3* are sent to trigger subsequent tasks *Task2* and *Task3*, respectively. The tasks *Task2* and *Task3* run in parallel, which generate *e4* and *e5* to announce their completions. The workflow completes when both *Task2* and *Task3* are completed.

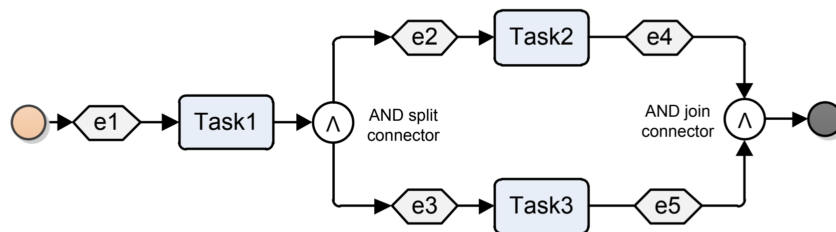


Figure 4.5: Event-Driven Workflow Execution

In general, an event carries a list of primitive data types, such as string, int and double. If the data passing between tasks is large, a *logical pointer* indicating the way to access the data can be delivered as an event. Before processing the data, the data receiver needs to dereference the *logical pointer* and retrieve the actual data identified by the *logical pointer*. This solution is also known as a handle-oriented approach [22], which avoids unnecessary transfers, especially when the workflow execution is controlled by a third-party agent (e.g., the execution of two or more tasks is controlled by a centralized workflow engine).

4.3.3 CEP-Based Workflow Pattern Modeling

As mentioned in Section 2.6, workflow patterns refer to recurring workflow processes. From different perspectives, the Workflow Patterns Initiative [52] has delivered four types of workflow patterns related to the development of workflow applications, i.e., control-flow patterns, data patterns, resource patterns and exception handling patterns. This section presents how the rule-based CEP technologies are used to represent the control-flow patterns.

In this thesis, the RbAF presents an event-driven workflow execution and models the workflow composition by messaging reaction rules, which specify and program reactive systems in a declarative manner, and in particular, they provide the ability to reason over events, actions and their effects, and allow detecting events and responding to them automatically. The workflow modeling based on the event-driven execution is also known as Event-driven Process Chains (EPCs), which is a business process model language for the representation of temporal and logical dependencies between activities in a business process [135].

Following the EPC, the RbAF defines a process with a set of activities that comprises three different types of elements connected by control flow edges: *tasks*, *event messages* and *connectors*. See Figure 4.5 as an example. The *tasks* represent activities in a process. The *event messages* are generated to trigger a task execution and the *connectors* that control the flow of a process (aka. gateways). There are three kinds of connectors: *AND*, *XOR* and *OR*. They can be used as either *split* (one incoming, multiple outgoing branches) or *join* (multiple incoming, one outgoing branch) connectors. Figure 4.5 shows an *AND* split connector, which means after *Task1* all subsequent tasks *Task2* and *Task3* are triggered to be executed concurrently, and an *AND* join connector, which means both *Task2* and *Task3* have to be completed. More details about such connectors can also be found in Section 5.1.

As aforementioned, the detection of an event corresponds to a reaction rule. An *AND* split connector can be implemented by sending parallel messages and detecting them by corresponding reaction rules later. Other complex split connectors can be implemented by imposing conditions on reaction rules triggering subsequent tasks. For example, the following example implements the *XOR* split connector after the ant identification (see Figure 2.10).

define reactive rule *identDone*
on $identDone(Cid, AntDesc, Result)$
if $not(isIdentFailed(AntDesc, Result))$
do **nothing**.

define reactive rule *identDone*
on $identDone(Cid, AntDesc, Result)$
if $isIdentFailed(AntDesc, Result)$
do $humanIdent(Cid, AntDesc)$.

Here, *not* denotes NaF, i.e., *not(isIdentFailed(AntDesc, Result))* succeeds when all attempts to prove *isIdentFailed(AntDesc, Result)* fail. Depending on the identification result, only one subsequent branch is activated. The implementation of the *XOR* split connector can be further used to implement the *Exclusive Choice* pattern, which is one of the control-flow patterns delivered by the Workflow Patterns Initiative. Similarly for the implementation of the *Multi-Choice* pattern [136].

The RbAF employs the rule-based CEP technologies to implement the join connectors. The rule-based CEP exploits the reaction rule technologies for event processing and often supports situation detection, pre- and post-conditions, and (transactional) action logic (complex actions) [92, 137]. In general, CEP aims at achieving actionable, situational knowledge from distributed systems in *real-time* or *quasi-real-time* [138, 139]. Instead of supporting a comprehensive CEP operators of event algebras, the RbAF does not impose real-time constraint on reaction time (aka. an *any-time* reaction rule system), and only supports a part of necessary operators involved in scientific workflows.

Each composite event consisting of multiple base events is usually described by an event pattern, which contains event templates, relational operators and variables [140]. The following are the relational operators used by the RbAF to define complex event patterns:

- $(e_1 \Delta e_2)(t)$. The composite event defined by this operator occurs when both e_1 and e_2 are detected; this pattern is also known as a *conjunction event pattern*.
- $(e_1 \nabla e_2)(t)$ The composite event defined by this operator occurs either e_1 or e_2 is detected; this pattern is also known as a *disjunction event pattern*.
- $ANY(m, e_1, e_2, ..., e_n)(t)(1 < m < n)$. The composite event defined by this operator occurs when m events out of n are detected.

Note that each base event e_i in the complex event patterns mentioned above could be either atomic event or composite event.

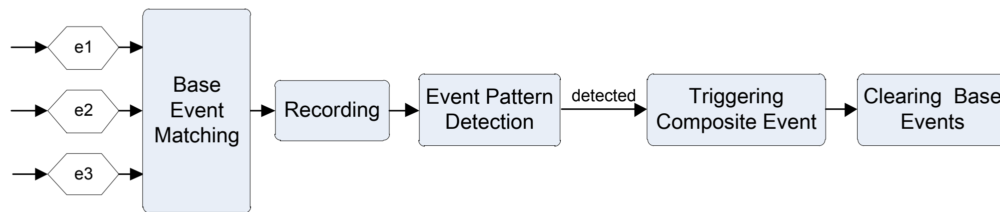


Figure 4.6: Process of Implementing the AND Join Connector

The detection of the conjunction event pattern implements the *AND* join connector, which requires all incoming branches to be completed. The agent responsible for a subsequent task needs to receive all base events indicating that the incoming tasks are successfully executed. Figure 4.6 shows the process of implementing the

AND join connector. Whenever a base event indicating the completion of an incoming task occurs, it is firstly recorded as an event fact in the knowledge base and then checks if the conjunction event pattern is detected or not. The reasoning if the conjunction event pattern is detected is implemented by a derivation rule, which is proved to be true when all base events are in the knowledge base. As soon as the complex event pattern is detected, the complex event is triggered. After that, the base facts are removed from the knowledge base, as shown in Figure 4.6.

Since the base event detection here is event-driven, this kind of complex event pattern detection is also known as *forward chaining event-driven reasoning*. As a concrete example, Figure 4.7 shows a process of detecting a composite event e, which requires all base events e_1, e_2 and e_3 to be successfully proved, i.e., e is $e_1 \Delta e_2 \Delta e_3$. The detection of base events e_1, e_2, and e_3 is described by three reaction rules, as shown as follows. The derivation rule *check(e)* describes the detection of the composite event e and is evaluated whenever a base event occurs. Note that *consequent* $\leftarrow$ *antecedent* is the basic form of derivation rules (see Section 4.4.1 for more details).

define reactive rule *detect*_e_1
on e_1
do $ins(e_1)$, $check(e)$.

define reactive rule *detect*_e_2
on e_2
do $ins(e_2)$, $check(e)$.

define reactive rule *detect*_e_3
on e_3
do $ins(e_3)$, $check(e)$.

$$check(e) \leftarrow e_1, \quad e_2, \quad e_3, \quad trigger(e), \quad del(e_1), \quad del(e_2), \quad del(e_3).$$

Suppose that e_1 is detected first, it is immediately recorded as an event fact in the knowledge base. Since e_2, and e_3 have not been detected, the execution of the derivation rule *check(e)* fails at the moment. The reasoning of the rule *check(e)* for the second time is triggered when another base event is detected. The dash line denotes that the second *check(e)* reasoning happens when any of e_2 and e_3 is detected, rather than happens immediately. The derivation rule *check(e)* can only succeed when all base events are detected, then the composite event e is triggered, and the base events are removed from the knowledge base. The implementation of *AND* join connector can be further used to implement the *Synchronization* pattern delivered by the Workflow Patterns Initiative [136].

The detection of the disjunctive event pattern implements the *XOR* join connector. The *XOR* join connector can have either local or non-local semantics [141]. The non-local *XOR* join connector expects only one incoming task to be successfully

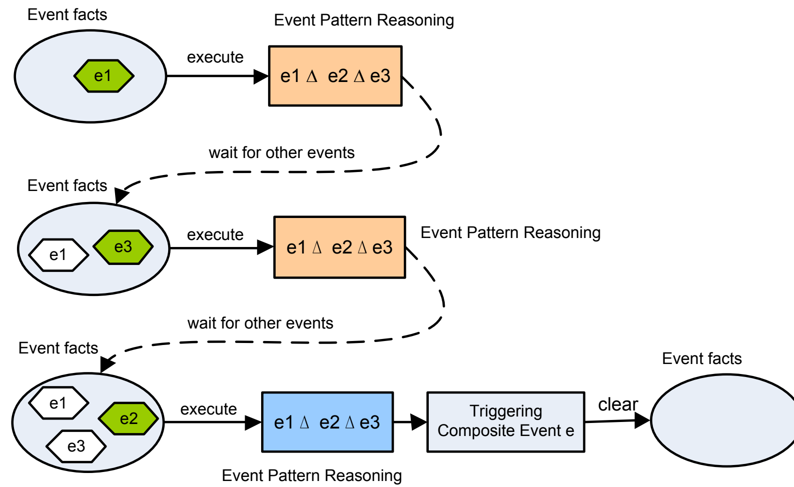


Figure 4.7: Example of an AND Join Connector Implementation

executed. The thread of control is passed to the subsequent branch when the first incoming branch has been enabled. Subsequent enablements of incoming branches do not result in the thread of control being passed on. On the contrary, the local *XOR* join connector propagates each incoming process token without ever blocking. Depending on two different situations, the non-local and local *XOR* join connectors can be used to implement the *Structured Discriminator* pattern and the *Multiple Merge* pattern delivered by the Workflow Patterns Initiative [136], respectively.

The difference between non-local and local *XOR* join connectors lies in if the subsequent incoming branches are blocked or not after the first incoming branch has been enabled. There are two solutions to implement them. One solution is to control the occurrence of the composite event. According to the occurrence times of the composite event, the composite event can be triggered once or more. The other solution is to control the consumption of the composite event. The subsequent task can consume the composite event once or multiple times. The RbAF adopts the latter to implement the *XOR* join connector. The implementation of the *XOR* join connector has the same process as the *AND* join connector, and the difference can be found in the derivation rules that are used to detect the disjunctive event pattern, i.e., there are multiple derivation rules used to describe different situations that the disjunctive event pattern can be detected, one for each base event.

The *OR* join connector usually has non-local semantics and synchronizes all incoming branches that are active. In other words, the non-local *OR* join connector needs to detect which branches are still active, and which will never be active. But the detection is usually difficult if the incoming branches involve cyclic processes that could never be completed. The RbAF addresses this issue by attaching a timeout to the event pattern detection, which waits for a pre-specified time and then consumes events that are there, as shown in Figure 4.8. The *OR* join connector is usually used

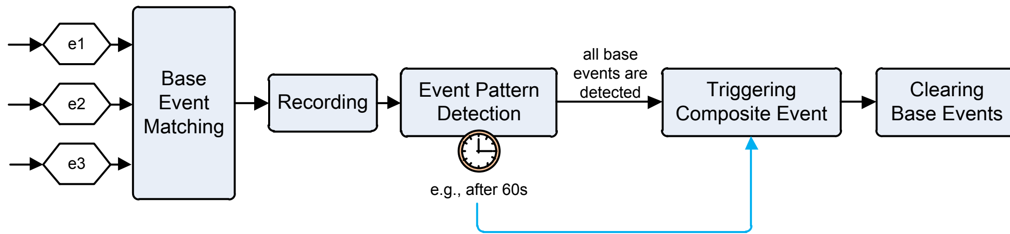


Figure 4.8: Process of Implementing the OR Join Connector

to implement the *Structured Synchronizing Merge* pattern delivered by the Workflow Patterns Initiative [136].

The *ANY* event pattern can also be implemented in the similar way as the *XOR* join connector implementation, and the only difference can be found in the derivation rules that are used to detect the *ANY* event pattern, i.e., the base events required to be detected must be explicitly specified.

To sum up, the RbAF employs the rule-based CEP technologies to implement the control-flow patterns. Moreover, declarative derivation rules provide the ability to reason over events and make it possible to implement advanced control-flow patterns. More details about a workflow pattern-based evaluation can be found in Section 7.1.

4.4 Domain Decision-Centric Logic Description

4.4.1 Derivation Rules

Knowledge representation focuses on methods for describing the world in terms of high-level, abstracted models which can be used to build intelligent applications, i.e., it provides methods to find implicit consequences of explicitly represented knowledge [60]. It is a broad research area, including language and graphical representations, ontology engineering, etc. The RbAF of this thesis does not attempt to support different ways of domain knowledge representation but focuses on representing domain decision logic by logical derivation rules (aka. deductive rules).

Derivation rules follow an *if(antecedent)-then(consequent)* style and derive new information from existing data. They are formulas of the form $q \leftarrow p$, where p is antecedent specified, and q is the conclusion deduced. The reasoning of derivation rules can be both *forward* and *backward* (aka. *forward and backward reasoning*, respectively). The *forward reasoning* starts with available data and uses inference rules to extract more data until a goal is reached. An inference engine using forward chaining searches inference rules until it finds one where the antecedent (*if* clause) is known to be true. The *backward reasoning* starts with a list of goals and works backwards to see if the data supports any of these goals available. An inference engine using backward chaining would search inference rules until it finds one which has a consequent (*then* clause) that matches a desired goal.

The common deductive computational model of logic programming uses the

backward reasoning (goal-driven) resolution to instantiate the program clauses via goals and uses unification to determine the program clauses to be selected and the variables to be substituted by terms [60]. The RbAF also uses the backward reasoning to implement domain decision-centric activities, which are usually represented as decision goals. For each decision goal, an inference engine checks if this goal is satisfied or not. This way of derivation is also known as *backward chaining goal-driven reasoning*. Different with the backward deductive derivation, the way of complex event pattern detection introduced in Section 4.3.3 is known as *forward chaining event-driven reasoning*. In other words, a complex event pattern is detected as soon as the base events required for the pattern are detected.

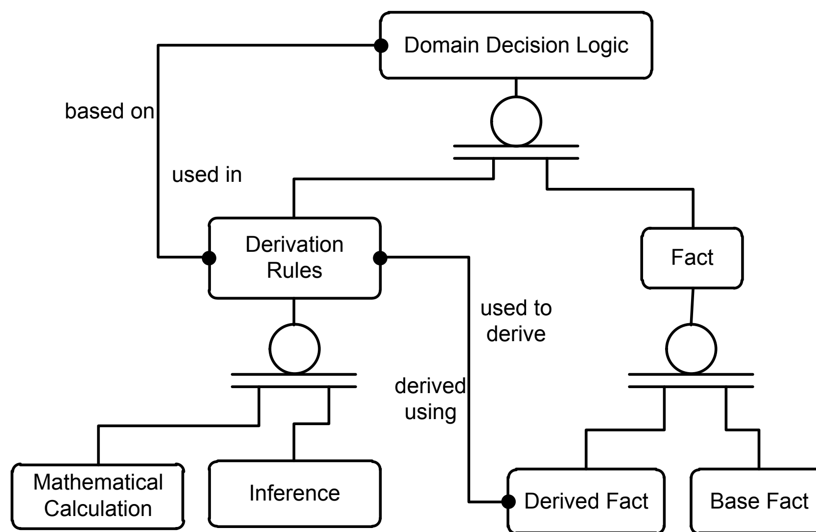


Figure 4.9: Domain Knowledge-Intensive Decision with Derivation Rules

Figure 4.9 shows the derivation model adapted from [142]. The domain logic consists of derivation rules and facts. The facts are further classified into base facts that are given in a specific domain, and derived facts that are created from existing ones. One can use derivation rules and facts to drive implicit facts (i.e., the derived facts). In the RbAF, the facts describe information, including events, (object-oriented) object instances, class individuals (of ontology classes), norms, constraints, states, conditions, actions, data (e.g., relational, XML), etc. A derivation is either a mathematical calculation that generates a derived fact according to a specified mathematical algorithm or an inference that creates the derived facts using logical induction.

Derivation rules focus on declarative problem representation and go beyond typical restricted expressiveness of simple gateways in process execution models. Moreover, they are more understandable and allow domain experts to express complex scientific rules in their own terms. For example, the following Prova program implements the policies of screening snow depth data in the experiment of building a snow depth model in the pastoral area of northern Xinjiang (in China) (see Section 2.5.2).

The first part of the program (Line 1-26) presents the rules of identifying deep frost layer, dry snow, thaw, temperature, snow depth and elevation, respectively. For example, the rule *checkDepth* specifies that the snow depth must be thicker than its critical value, i.e., *DepthCriticalvalue*. *MonthAvgTemp* < 10, *Depth* > 0.5 and *Depth* < 10 are mathematical calculations implementing logical expressions. The second part of the program (Line 29-41) presents some facts that are used in the snow data screening. For example, the fact "depthCriticalvalue(3.0)." denotes the critical snow depth of this experiment is 3.0 centimeters.

Listing 4.1: Snow Depth Sample Screening Implemented in Prova

```
1  deepFrostLayer(Station, Year, Month, Depth, MonthAvgTemp):-
2     MonthAvgTemp < 10,
3     Depth > 0.5,
4     Depth < 10.

6  drySnow(Tb36V, Tb18V):-
7     Tb36V > 195.0,
8     Tb36V < 225.0,
9     Tb18V < 255.5.

11 thaw(Month, DayMaxT):-
12    Month = 3,
13    DayMaxT >= 6.

15 checkTemperature(Temp) :-
16    tempCriticalValue(TempCriticalValue),
17    Temp < TempCriticalValue.

19 checkDepth(Depth) :-
20    depthCriticalValue(DepthCriticalValue),
21    Depth >= DepthCriticalValue.

23 checkElevation(Station) :-
24    stationElevation(Station, Ele),
25    elevationCriticalvalue(ElevationCriticalvalue),
26    Ele < ElevationCriticalvalue.

28 % measure unit: meter
29 depthCriticalValue(3.0).

31 % measure unit: celsius
32 tempCriticalValue(6).

34 % measure unit: meter
35 elevationCriticalvalue(2000).

37 % measure unit: meter
38 stationElevation('Fuhai', 500.9).
39 stationElevation('Aletai', 735.3).
40 stationElevation('Fuyun', 823.6).
41 stationElevation('Qinghe', 1218.2).
42 ...
```

Derivation rules provide a natural way of domain decision-centric logic representation and also support easy adaption, changes and extensions. Compared to imperative programs, in which the logic is deeply buried, such declarative rules can be easily adapted to other similar experiments. For example, the above rules can

be easily adapted to another similar experiment of building a snow depth model in Qinghai Province of China [42].

4.4.2 Semantic Web Data Query

Derivation rules are usually built on the facts that specify propositions taken to be true in a domain. For example, facts *stationElevation('Fuhai', 500.9)* and *monthAvgTemp('Fuhai', 2004, 01, -20.9)* indicate the elevation of Fuhai meteorological station is 500.9 meters, and its average temperature in January of 2004 is -20.9 ℃. This section presents how existing Semantic Web data can be reused in domain knowledge representation.

With the development of Semantic Web technologies, Semantic Web applications are being developed for many aspects of scientific research, from experimental data management, discovery and retrieval, to analytic workflows, hypothesis development and testing, to research publishing and dissemination [143]. Currently, there are domain specific glossaries, taxonomies and Semantic Web ontologies available on the Internet [13]. In particular, the applications of Semantic Web technologies in the life science domain have got good achievement and are one step ahead of other research domains. Supported by international workshop SWAT4LS [144], which provides a venue to present and discuss the benefits and limits of the adoption of Semantic Web technologies in the life sciences domain, there are efforts that focus on publishing scientific data using Semantic Web technologies. For example, Identifiers.org [145] is a system providing resolvable persistent URIs that can be used to identify data for the scientific community. The European Bioinfomatics Institute (EBI) [146] provides freely available data from life science experiments covering the full spectrum of molecular biology. In a bid to support Semantic Web technologies, the EBI has published a new RDF platform [147] to access bioinformatics resources in 2013. COEUS [148] is a Semantic Web application framework targeting quick creation of new biomedical applications. Such Semantic Web data provides a common, comprehensible foundation for resources of different scientific domains, and it is wise to reuse it rather than waste efforts to achieve similar logic by declarative rules from scratch.

Semantic Web data is usually in the form of vocabularies or ontologies. As W3C explained, there is no clear division between vocabularies and ontologies. The trend is to use the word "ontology" for more complex, and possibly formal collections of terms, whereas "vocabulary" is used when such strict formalism is not necessarily used or only in a loose sense [129]. An ontology is an explicit specification of a conceptualization, which represents a set of objects and their relations in a domain [149]. The vocabularies, ontologies and rules can all be employed to represent domain-specific logic. Moreover, they are expressible in each other to some extent. For example, derivation rules can express the concepts and relationships encoded by semantic vocabularies and ontologies, e.g., $A \subseteq B$ in OWL can be encoded as $A(x) \rightarrow B(x)$. However, although there are overlaps between ontologies and rules, they cannot replace each other. The ontology-based knowledge representation

depends on the expressiveness of DL. They are good at representing schema level knowledge and even asserting the existence of unknown individuals, but they cannot specify arbitrary relationships between instances (individuals). The rule-based knowledge representation takes the perspective of LP and represents domain policies in a clear logical way. However, declarative rules also have limitations, e.g., they cannot express the existence of unknown/unnamed individuals. For the purpose of exploiting the benefits of both rules and ontologies, there are efforts attempting to combine them as a unified logic, such as [150, 151, 152]. However, these efforts are still under the way. Different with these efforts, the RbAF adopts a lightweight solution to integrate existing Semantic Web data into declarative rules, rather than provides domain experts with a complex unified logic. That is, the RbAF builds declarative rules on top of ontologies and enables rules to access existing Semantic Web data as external codes. To do so, the RbAF provides three ways to access domain data encoded by Semantic Web technologies, also shown in Figure 4.10.

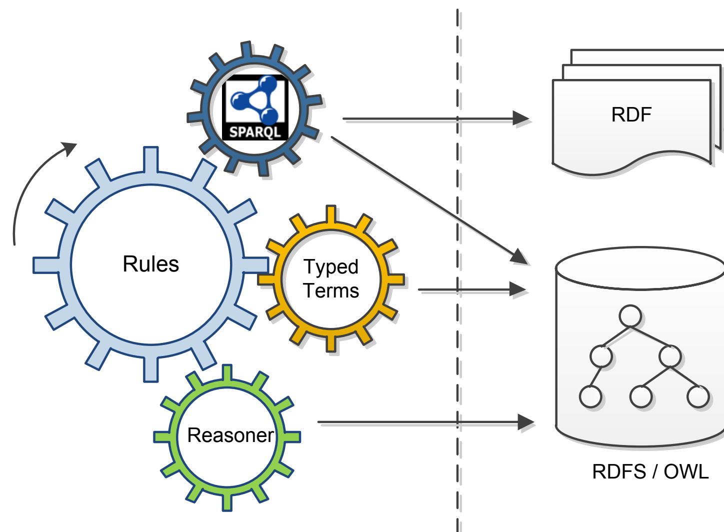


Figure 4.10: Semantic Web Data Query

Querying Vocabularies with SPARQL

RDF is a data model for the description of Web resources. The model denotes an RDF graph, where both data and relationships are represented by URIs. In other words, an RDF graph is a set of triples with a form of (subject, predicate, object). SPARQL is the standard query language for this model. For example, the following SPARQL query retrieves all proteins of fruit fly.

Listing 4.2: Example of SPARQL Query (1)

```
1 PREFIX up:<http://purl.uniprot.org/core/>
2 SELECT ?protein
3 WHERE
4 {
```

```
5   ?protein a up:Protein .
6   ?protein up:organism ?organism .
7   ?organism up:commonName ?name .
8   FILTER regex(?name, "Fruit fly", "i" )
9 }
```

Since the SPARQL draft was firstly released in 2004, it had become an official W3C Recommendation in 2008. The latest SPARQL 1.1 was released in March of 2013, which has new features, including sub-queries, value assignment, path expressions, aggregations, etc. However, SPARQL has some challenges [153]: (1) due to the open world semantics of RDF, RDF databases are inherently incomplete; (2) SPARQL cannot get complete answers when it queries the vocabularies with predefined semantics (e.g., ontologies in Resource Description Framework Schema (RDFS) or OWL) (see Listing 4.4); (3) the normative SPARQL assumes that the RDF data resides in a single repository and the queries have full access, but in reality, SPARQL has to work at Web scale to accommodate Linked Data.

Incorporating Ontologies as Typed Rules

Compared with RDF, RDFS is a simple ontology language that allows individuals sharing properties to be classified into classes. The individuals of a class are referred to as instances of that class. RDFS defines the relationship between instances and classes with a special URI *rdf:type*. For example, the triple "<uniprot:Q15653 rdf:type up:Protein>" defines that *Q15653* is a protein (Note that the namespace prefix bindings are omitted for clarity). OWL extends RDFS with more complex statements about individuals, classes and properties. Both of them can describe a set of individual objects sharing properties. However, users need to enumerate all individuals of a class when they encode the knowledge represented by the class with declarative rules. To overcome this problem, the RbAF incorporates domain ontologies (classes to be more precise) as typed rules, i.e., the variables in rules can be typed with concepts defined in external ontologies. This solution greatly reduces rules that need to be formulated and also improves the flexibility and accuracy of domain knowledge representation. For example, the inputs of the protein prediction analysis process must be a protein and a GO term (see Section 2.5.2). However, for a cell, there may be thousands of proteins and GO terms, it is troublesome to use declare rules to validate them one by one. With this solution, the only thing needs to be done is to declare the type of two inputs as *up:Protein* and *up:Concepts*, respectively, which are already defined in the UniProt core ontology [154].

Reasoning Ontologies with Reasoners

Besides incorporating ontologies as typed rules, the RbAF also can reason complex ontologies, i.e., RDFS and OWL. In addition to classes, RDFS defines restrictions on properties. For example, *rdfs:domain* and *rdfs:range* restrict the domain of subject and the range of object of a property, respectively. Especially OWL further extends RDFS with more expressiveness, including the definition of class

relations, constraints and cardinalities, equivalences between classes, properties of properties, etc. As mentioned in querying vocabularies with SPARQL, SPARQL usually aims to data in the form of RDF triples and cannot get complete answers when it queries the vocabularies with predefined semantics. For the complex queries that need to consider predefined semantics, it is necessary to employ ontology reasoners. For example, the following SPARQL-DL query based on OWL reasoner HermiT [155] retrieves the types of a genomic DNA identified by a URI: `<http://purl.uniprot.org/embl/AE014297>`.

Listing 4.3: Example of SPARQL-DL Query

```
1 PREFIX embl: <http://purl.uniprot.org/embl/>
2 SELECT ?x WHERE
3 {
4    Type(embl:AE014297 , ?x)
5 }

7 Results:
8 ?x = http://www.w3.org/2002/07/owl#Thing
9 ?x = http://purl.uniprot.org/core/Genomic_DNA
10 ?x = http://purl.uniprot.org/core/Molecule
11 ?x = http://purl.uniprot.org/core/DNA
```

The results show that *embl:AE014297* is a kind of *up:DNA*, *up:Molecule* and *owl:Thing*. The SPARQL-DL [156] query language is a distinct subset of SPARQL; it is settled on top of the OWL API and allows to mix TBox, RBox, and ABox queries. The RbAF uses the SPARQL-DL query engine to reason complex ontologies, more details can be found in Section 6.4. However, when the query is re-formulated as a SPARQL query, the result denotes that *embl:AE014297* is only a kind of *up:Genomic_DNA*, as shown in the following SPARQL query. That is because there is no triple in the ontology directly defining that *embl:AE014297* is the subclass of *up:DNA* and *owl:Thing*.

Listing 4.4: Example of SPARQL Query (2)

```
1 PREFIX embl: <http://purl.uniprot.org/embl/>
2 SELECT ?x
3 WHERE
4 {
5    embl:AE014297 a ?x .
6 }

8 Results:
9 ?x = http://purl.uniprot.org/core/Genomic_DNA
```

4.5 Integrating Orchestration with Choreography

In general, there are two ways of the workflow composition: *orchestration* and *choreography*. Typical workflows are often executed in a central manner on a single machine, where different services are composed and coordinated efficiently through a controller to accomplish a complex goal [7, 157, 86]. Such workflows can also be regarded as complex services and integrated hierarchically into other workflows. This

execution manner is also known as *orchestration*. The service *orchestration* has advantages, such as services can be freely designed to be process-agnostic and reusable; the workflow execution is managed by the centralized workflow engine. However, due to the centralized workflow execution, the service *orchestration* suffers from the weaknesses, such as the consumption of network bandwidth, degradation of performance and single-points of failure caused by redundant data transfers [158, 159]. Compared with the service *orchestration*, *choreography* is another way of service composition, which focuses on collaboration and message exchange between multiple participants [157].

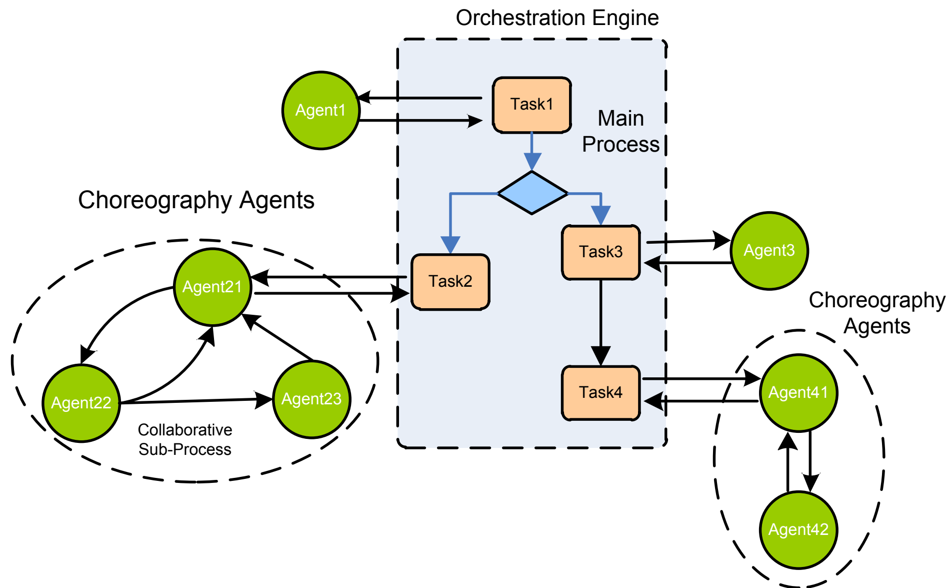


Figure 4.11: Integrating Orchestration with Choreography

The RbAF extends the range of workflow applications by combining the *orchestration* with *choreography*-style execution, as shown in Figure 4.11. That is, on one hand, a centralized workflow engine (an agent) takes control of the execution of a workflow and completes it via the composition of distributed agents (Note that the tasks here are conceptually the same as the services of the traditional workflow systems. Both of them are basic elements of a workflow). The involved agents receive task assignments from the workflow engine and return results to the workflow engine after completing them. They do not need to know in which process they are embedded, and the task assignment and process execution are managed by the centralized workflow engine.

On the other hand, distributed agents can communicate via messaging reaction rules, which enables them to build conversation-based interactions and choreography workflows. That is, the choreography interaction flows between distributed agents are defined by the order of sending and receiving messages, which are associated with conversation identifiers to reflect the process execution. Each agent involved in a choreography workflow knows exactly when to execute its operations and with

whom to interact.

The agents of the RbAF are usually loosely coupled and implemented based on their different functionalities. As a way for coordination, the RbAF uses the workflow ontology (see Section 4.2) to allocate tasks to agents. The workflow ontology describes the agent responsibilities in completing certain tasks. With the workflow ontology, one task can be dynamically allocated to a responsible agent, whose local knowledge base is deemed to be best suited for performing it.

4.6 Human Interaction

Although the vision of scientific workflows aims to automate scientific processes, in reality, scientists are still required to conduct manual tasks or make complicated decisions at runtime. Some efforts have been made to support user interaction in workflows, such as BPEL4People [160] and WS-HumanTask [161]. Both of them make it possible to wrap human behavior into Web Services, and the scientific workflow systems that can invoke Web Services also can integrate human users in workflows. However, these systems support only synchronous Web service invocations and do not allow specifying callback operations [3].

The RbAF also allows human operations in the WsSWFs. For the tasks that need to be performed by human users, a Human Agent (HA) is employed to receive human task requests, process human operations and then return the results to task requesters. Figure 4.12 shows the process of human interaction.

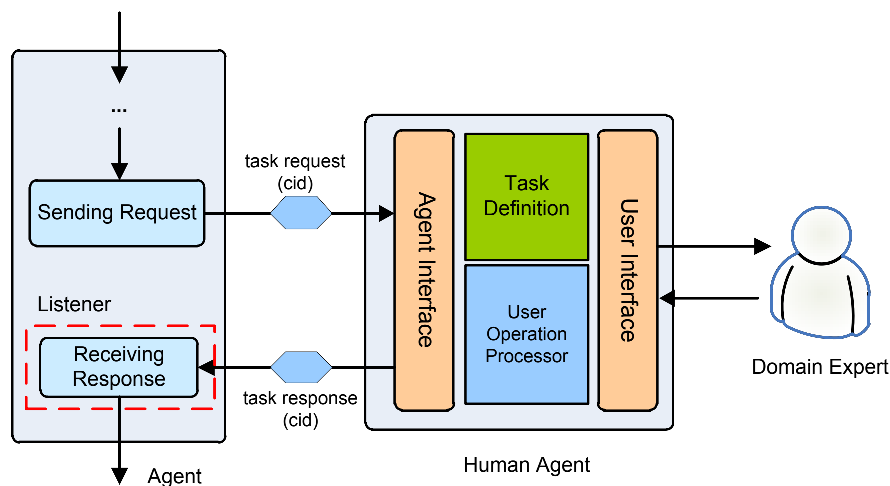


Figure 4.12: Integrating Humans into Scientific Workflows

The HA manages the life cycle of human tasks. Typically the process of performing a human task is triggered when the HA receives a human task request on its (agent) interface from other agents. The agent interface has an agent lifetime scope and it is active while the HA runs. Afterwards, the HA stores the tasks and sets their states as *pending*. Human tasks of the RbAF have two statuses, namely *pend-*

ing and *done*. They are initialized as *pending* when they are received and updated to *done* after they are completed by human users.

Following Web Services Human Task (WS-HumanTask) [161], a human task of the RbAF is described by a set of general properties, as shown in the following template.

```
<HumanTask>
   <cid><!-- conversation identifier --></cid>
   <type><!-- unexpected exceptions or human tasks --></type>
   <payload><!-- task description --></payload>
   <workflow><!-- workflow name --></workflow>
   <receivedAt><!-- task received time --></receivedAt>

   <solution><!-- task solution --></solution>
   <solver><!-- task solver --></solver>
   <solvedAt><!-- task solved time --></solvedAt>
   <status><!-- task status --></status>
</HumanTask>
```

The human task properties are conversation identifier, human task type, task description, workflow name, received time, task solution, solver, solved time and status. Note that due to the focus of this thesis is on the integration of human users into scientific workflows, the RbAF does not give a comprehensive human task specification but only defines some important properties of human tasks.

The RbAF has two types of human tasks: human tasks themselves and unexpected exceptions. Once an exception cannot be handled automatically by the rule-based agents, the exception will be escalated to human users to make a decision or provide the required resources. This kind of exception is also called an unexpected exception and considered as a human task. More details can be found in the next section.

Users operate on human tasks assigned to them via a friendly menu-based Web interface. After completing a human task (e.g., make a decision), they submit their decisions via the interface to a user operation processor, which encapsulates human operations into event messages and sends them back to the task requester to resume the workflow execution (i.e., callback). Meanwhile, the human task status is updated from *pending* to *done*.

However, what users need are not only integrating users into the workflow execution, but also asynchronous interaction with the workflow system, especially when performing long running activities, such as discussions and exhaustive knowledge searches. The RbAF addresses this problem via an asynchronous messaging style between a human task requester (agent) and the HA, as shown in Figure 4.12: the human task requester has a pair of messaging reaction rule activities, i.e., one *sending* activity which sends a human task request to the HA and the other *receiving* activity waits for the answer from HA. After the request is sent, the *receiving* activity freezes the current context of the workflow execution and creates a temporal reaction to wait for the results from the HA. This *receiving* activity is a reaction rule which has an event template that describes the desired answers from users.

The data from the received results is bound to the template variables as usual in logic programming. The binding usually includes backtracking to several variable bindings for the purpose of receiving possible human answers. Once the matching results are received, the reaction rule is triggered to activate subsequent activities. Note that the human task request and its results are associated with the same conversation identifier, which ensures that the workflow resumes as though it had never been interrupted. In other words, the conversation identifier ensures that the results are precisely returned to the original human task requester.

4.7 Exception Handling

Different strategies have been proposed to handle workflow exceptions at runtime, ranging from simple policies (e.g., retry, checkpoint/restart [162], replication [163]) to sophisticated exception handling involving human users. The RbAF provides two ways to handle workflow exceptions at runtime:

- *Dynamic exceptional activity replacement*: the dynamic replacement refers to treatments of an exception by dynamically replacing an exceptional activity with an alternative owning the same effect. The logic programming has inherent advantages in specifying alternative execution paths in case a particular execution path fails. Moreover, based on the workflow ontology, it is also possible to reallocate a task to alternative agents with the same effect, and its successors know nothing about the replacement. Since this strategy handles exceptions automatically, it is also known as *automatic exception handling* and the exceptions are referred to as *expected exceptions*.
- *Human interaction*: Besides the *expected exceptions*, there are exceptions that cannot be handled automatically by the rule-based agents. For example, no resources available (e.g., no agent is available to perform a task). Based on the asynchronous user interaction, human users are allowed to handle these exceptions by providing missing resources. Compared with the automatic exception handling, this approach is also known *manual exception handling* and the exceptions are referred to as *unexpected exceptions*.

The exception handling logic is often needed in many places of a workflow, and runs the risk of cutting across the process and making workflow maintenance more difficult. Following the AOP paradigm, the RbAF separates the exception handling logic from main workflow processes and encapsulates it in an Exception Handling Agent (EHA). More precisely, the EHA is responsible for handling the expected exceptions by finding alternative counterparts with the same effect and replacing the failed sub-process dynamically. The expected exceptions will be escalated to the unexpected exceptions and handled by the HA if they cannot be handled by the EHA automatically. Human users can make a decision or provide required resources to deal with the unexpected exceptions.

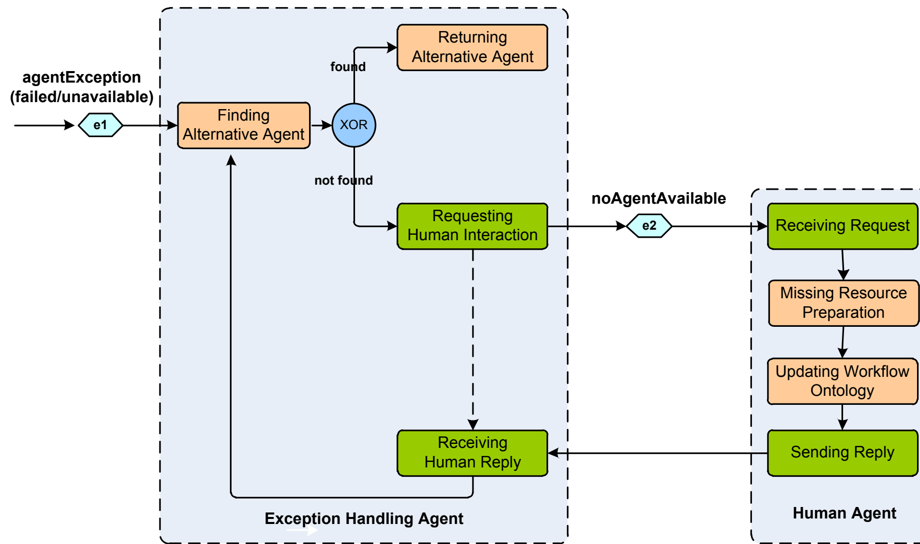


Figure 4.13: Exception Handling in Event-Driven Scientific Workflows

Figure 4.13 presents a process of dealing with the workflow exceptions at runtime. If an agent responsible for a task is *unavailable* or *failed*, then an exception message *e1* (*unavailableAgent* or *failedAgent*) is generated and sent to the EHA to find an alternative agent via reasoning the workflow ontology. The task is then allocated to the alternative agent once found. If there is no alternative agent available, then the EHA generates an escalation exception message *e2* (*noAgent* exception), and the control is passed to the HA to ask human users for help. Human users usually handle these exceptions by preparing missing resources and then notify the EHA to find a responsible agent again.

Another type of the expected exceptions is *infinite loop*, which is caused by endless communication between two or more agents. An *infinite loop* can be detected if a message is repeatedly sent more than a certain limit. More details will be found in Section 6.6.

4.8 Summary

This chapter introduced a rule-based, agent-oriented framework—RbAF, which exploits the benefits of both the declarative programming with rules and the agent technology to support the WsSWFs.

Messaging reaction rules specify workflow processes in terms of message-driven conversations between parties and describe their associated interactions via asynchronously sending and receiving event messages. They not only inherit the features of active global ECA rules, but also complement them by performing actions locally within a context. In other words, the complex subgoals of reaction rules are allowed to be proved by a group of distributed agents. Moreover, the rule-based complex

event pattern computation can model complex workflow connectors (or gateways), thereby modeling complex workflow patterns.

The domain-specific decision logic in workflows is expressed by exploiting the benefits of both LP and DL. On one hand, logic programming with derivation rules is understandable to scientists and also much easier to be modified if there are changes. On the other hand, the RbAF provides three ways to access domain data encoded by Semantic Web technologies: querying simple vocabularies with SPARQL, incorporating ontologies as typed rules and reasoning complex ontologies with reasoners. This lightweight combination of rules and ontologies not only reduces rules that need to be formulated, but also improves the flexibility and accuracy of domain knowledge representation.

Meanwhile, the RbAF extends the range of workflow applications by combining two ways of the workflow composition: *orchestration* and *choreography*. Distributed agents can be simply composed into a centralized workflow to execute part of a workflow, and also can build conversation-based interactions between multiple participants via messaging reaction rules.

The RbAF also integrates human users to perform manual operations and supports the asynchronous human user interaction. In addition, based on the workflow ontology, the RbAF supports both automatic and manual exception handling during the workflow execution.

In the next chapter, a formal semantics of the rule-based workflow language is presented.

Chapter 5

Formal Workflow Representation

Contents

5.1 Workflow Model . . . 85

5.2 CTR Overview . . . 87

5.3 Workflow Representation Using CTR . . . 91

5.3.1 Workflow Representation . . . 91

5.3.2 Multiple Instances . . . 94

5.3.3 Reactive Logic Representation . . . 94

5.4 Communication between Processes . . . 96

5.5 Complex Event Processing . . . 97

5.6 Exception Handling . . . 100

5.7 Summary . . . 102

Currently there are rule-based workflow languages that support flexible service composition and model the process logic with declarative rules. However, most of them only provide static syntactical process descriptions without precise formal semantics. The formal semantics of a workflow language helps in understanding what a workflow is doing. Without a formal semantics, workflow engines implementing a language can easily produce slightly different results. For the purpose of reducing ambiguity and opening possibilities for verification and analysis, this chapter presents a CTR-based formal semantics of the rule-based workflow language presented in this thesis.

This chapter is organized as follows: Section 5.1 introduces a general workflow model. Section 5.2 provides an overview of CTR as the main underlying formalism. Section 5.3 explains how CTR is used to represent workflow processes. Sections 5.4, 5.5 and 5.6 present the communication between processes, the complex workflow pattern modeling with the rule-based CEP technologies and the exception handling, respectively. Section 5.7 introduces the related efforts and summarizes this chapter.

5.1 Workflow Model

Before presenting the formal semantics of the logic-based workflow language, this section introduces a general workflow model to help readers understand how a workflow is composed, as shown in Figure 5.1.

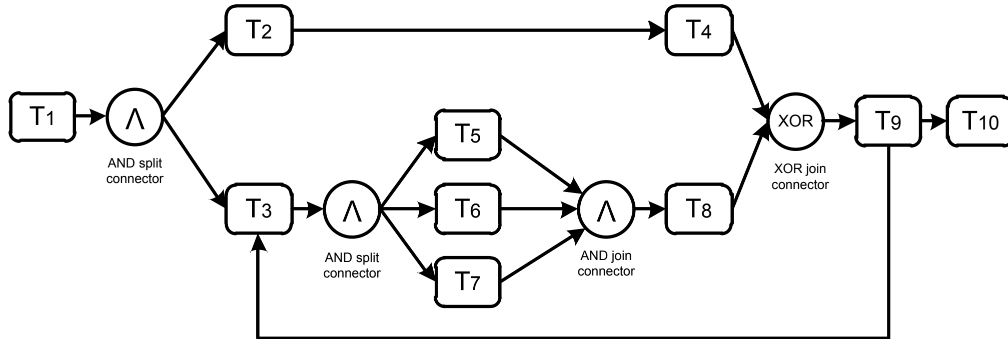


Figure 5.1: A General Workflow Process Model

Workflows can be viewed from different perspectives [164]. From a *control-flow* perspective, a workflow process consists of a group of tasks (denoted by rounded rectangles in Figure 5.1) defined in terms of business or scientific rules. The recurring features of these rules are also known as control-flow patterns. A well-known collection of control-flow patterns is proposed by the Workflow Patterns Initiative [52, 136]. Figure 5.1 contains three general control-flow patterns: *sequence*, *parallel* and *iterative*, which are common in the real-world applications. In the *sequence* pattern, a task is enabled only after the completion of a preceding task. For example, T_2 and T_4 in Figure 5.1. As for the *parallel* pattern, it refers to a point (*split connector*) where a single branch splits into multiple parallel branches. In other words, a split connector usually has one incoming and multiple outgoing branches. A split connector can be further divided into more advanced ones, such as *AND*, *XOR* and *OR* split connectors. After an *AND* split connector, all subsequent (outgoing) branches are executed concurrently. For example, after T_1, two subsequent tasks T_2 and T_3 are started to be executed concurrently. An *XOR* split connector describes a point in a process where a decision is made precisely to select one of the subsequent branches to execute. An *OR*-parallel split connector imposes conditions on each outgoing branch. Only the subsequent branches (one or more) that meet the conditions are executed. Moreover, the parallel pattern is usually associated with a *join connector*, where two or more branches are joined into a single subsequent branch. In other words, each join connector usually has multiple incoming and one outgoing branch. A join connector can also be divided into *AND*, *XOR* and *OR* join connectors. An *AND* join connector defines a synchronizer that requires all incoming branches to be completed. For example, each of T_5, T_6 and T_7 has to be completed before T_8 can be executed. An *XOR* join connector can have either *non-local* or *local* semantics [141]. A *non-local XOR* join connector expects only one incoming task to be successfully executed. The thread of control is passed to subsequent branches when the first incoming branch has been enabled. The *non-local XOR* join connector works well if incoming branches are mutually exclusive (i.e., only one incoming branch is activated). However, if more than one incoming branch

is activated, the non-local *XOR* join connector needs to block subsequent incoming branches after the thread of control is passed to the subsequent branch after the first incoming branch is enabled. The subsequent enablements of incoming branches do not result in the thread of control being passed on. For example, after T_1, two subsequent tasks T_4 and T_8 will be executed concurrently. Suppose the *XOR* join connector in Figure 5.1 is non-local, T_9 is executed when one of T_4 and T_8 has completed. Completion of the other task is ignored and does not result in executing T_9 again. A *local XOR* join connector propagates each incoming process token without ever blocking. For example, suppose the *XOR* join connector in Figure 5.1 is local, T_9 is executed after each completion of T_4 and T_8. In other words, T_9 is executed twice. Depending on different situations, they can be used to implement the *Structured Discriminator* pattern and the *Multiple Merge* pattern proposed by the Workflow Patterns Initiative [136], respectively. The *OR* join connector usually has *non-local* semantics and synchronizes all incoming branches that are active; it is usually used to implement the *Structured Synchronizing Merge* pattern [136].

Using different split and join connectors of the general parallel pattern can generate more advanced workflow patterns, such as a process starting with an *OR* split connector that first enables one or more subsequent branches and then ending with an *OR* join connector. The process is completed when all concurrently activated branches have completed.

An *iterative* pattern consists of a start node, an end node and an iteration edge that directs from the start node to the end node, such as T_3 and T_9 shown in Figure 5.1. The iterative pattern also includes a condition that specifies when the iteration is needed. Depending on the place of the condition, being either at the start or end nodes, the iterative pattern is divided into the classic *while...do* pre-test loop construct and the *repeat...until* post-test loop construct.

While the scientific workflow composition in Figure 5.1 visually emphasizes processing steps, the actual computation is often data-driven [19]. In other words, from a *data flow* perspective, what passes between workflow steps is not just control, but also data flowing from one task to another [19]. A task may require a data object as an input and produce another data object as an output, which could be used as an input by its successor. Therefore, it is also important to capture data dependencies and enable workflow engines to find suitable resources automatically for each task.

5.2 CTR Overview

Transaction Logic (TR) is a general logic that accounts for state changes in deductive databases, logic programs and arbitrary logical theories in a clean and declarative way [165]. In deductive databases with updates, each *state* represents a database, and database transactions are considered as a series of updates, which cause transitions from one state to another, thereby changing databases.

As a first-order logic language, TR has logical connectives $\wedge$ (classical conjunction), $\vee$ (classical disjunction), $\neg$ (classical negation), $\forall$ (universal quantification)

and ∃ (existential quantification). For the purpose of combining transactions sequentially, TR extends the first-order logic with a new connector of sequential composition, denoted as: ⊗ (aka. serial conjunction). The resulting logic formulas are called *transaction formulas* which are recursively defined as follows [166]:

An atomic transaction formula is an expression of the form $p(t_1, ..., t_n)$, where p is a predicate symbol, and t_1, ..., t_n are terms. A term is either a *variable* or a *function term* with a form of $\mathrm{f}(\mathrm{t}_1, ..., \mathrm{t}_k)$. A functional term with arity zero is a *constant*. Furthermore, if ψ and ϕ are transaction formulas, then so are the following expressions:

- $\phi \wedge \psi$: both ϕ and ψ must be executed along the same path;
- $\phi \vee \psi$: execute ϕ or ψ non-deterministically;
- $\neg\, \phi$: execute in any way provided that this will not be a valid execution of ϕ;
- $\phi \otimes \psi$: first execute ϕ, then execute ψ;
- (∀X) ϕ and (∃X) ψ, where X is variable.

TR accounts not only for the update orders, but also for other important features in areas, such as transaction and subroutine definition, deterministic and non-deterministic actions, static and dynamic constraints, hypothetical and retrospective transactions, and a wide class of tests and conditions on actions [167, 165]. Like procedural languages, TR enables users to combine simple actions into complex ones in a greater variety of ways. Moreover, users are allowed to specify loose constraints that a transaction must satisfy. In other words, sequences of actions and constraints that can be arbitrarily mixed, and in this way, procedural and declarative knowledge are seamlessly integrated [165]. For example, $\neg\, (a \otimes b \otimes c)$ means that the sequence $a \otimes b \otimes c$ is not allowed.

As in the first-order logic, TR includes a Horn-like fragment that has both a declarative and a procedural semantics [165]. Horn rules are formulas of the form $p \leftarrow \psi$, which can be seen as a convenient abbreviation for the formula $p \vee \neg\, \psi$. Here, p is an atomic formula, and ψ is any TR formula. The rule also has a procedural interpretation, which means that "to execute p, it is sufficient to execute ψ". The predicate symbol p here acts as the name of the procedure, and ψ acts as the procedure body or definition, which may be any TR formula (e.g., $\mathrm{a} \leftarrow \mathrm{a}_1 \otimes \mathrm{a}_2 \otimes ... \otimes \mathrm{a}_n$). Note that, for notational convenience, a rule $p(X) \leftarrow \psi$ is an abbreviation of $\forall X[p(X) \leftarrow \psi]$, $\exists\, \psi$ is an abbreviation for $\neg\forall\neg\psi$. Thus, quantifiers ∀ and ∃ are not explicitly considered in what follows.

TR has its own model theory and sound-and-complete proof theory [165]. The model-theoretic semantics of TR is based on *paths*, i.e., sequences of database states, rather than states themselves. To define truth on paths, each path is assigned a first-order formula, which specifies the formula atoms that are true on the path [165]. In other words, all formulas, atomic or complex, which are true on a path represent actions (i.e., updates or queries) that take place along the path. For example, the

path D, $D + \{a\}$, $D + \{a, b\}$ satisfies the formula ins:a $\otimes$ ins:b, since it represents an insertion of a followed by an insertion of b. Here D denotes a state. The proof-theoretic semantics of TR can verify the procedure that causes state transitions, and more details about the TR proof theory can be found in [166].

For the purpose of supporting the commutation between concurrent processes, CTR further extends TR with two logical operators: concurrent conjunction $|$ and the modality of atomicity $\odot$ for specifying atomic actions [167]. The resulting transaction formulas and their semantics are described as follows:

- $\phi \mid \psi$: ϕ and ψ are executed concurrently;
- $\odot\, \phi$: the execution of ϕ should not be interleaved with other transactions.

The concurrent processes in CTR execute in an interleaved fashion and can communicate and synchronize themselves, thereby increasing the flexibility, performance, and power of the language. Since CTR is built upon TR, the statements in what follows about TR are also hold for CTR, unless explicitly specified.

In CTR, two oracles encapsulate the elementary database operations: *data oracle* and *transition oracle*. Such oracles come with a set of database states, upon which they can operate. Each database state can be seen as a set of data items, which can be accessed by the oracles.

Both oracles are mappings: the data oracle, $\mathcal{O}^d(D)$, which maps from the database states to sets of first-order formulas, i.e., $\mathcal{O}^d(D)$ presents queries related to a particular state D; the transition oracle, $\mathcal{O}^t(D_1,\ D_2)$, which maps pairs of the states to sets of ground atomic formulas, i.e., these transition oracles represented by ground atomic formulas cause database state changes. For example, if $a \in \mathcal{O}^t(D_1,\ D_2)$, then a is an elementary update that changes state D_1 to D_2. Such oracles provide a semantics for the data items, and by using different data access primitives, CTR can accommodate different database semantics [167]. A formal semantics of the rule-based workflow language used for supporting the WsSWFs can be found in the following sections of this chapter.

CTR programs support concurrent processes, where each process produces a sequence of elementary database operations, and these concurrent processes interact and communicate via the database. For example, the following transaction base defines two processes: *processA* and *processB* [167].

$$\text{processA} \leftarrow \text{taskA}_1 \otimes \text{send}(\text{ch}_1,\ \text{startB}_2) \otimes \text{taskA}_2 \otimes \text{receive}(\text{ch}_2,\ \text{startA}_3) \otimes \text{taskA}_3$$
$$\text{processB} \leftarrow \text{taskB}_1 \otimes \text{receive}(\text{ch}_1, \text{startB}_2) \otimes \text{taskB}_2 \otimes \text{send}(\text{ch}_2,\ \text{startA}_3) \otimes \text{taskB}_3$$

The concurrent transaction *processA* | *processB* executes *processA* and *processB* concurrently while synchronizing the execution of their tasks by sending and receiving messages. More precisely, $taskB_2$ cannot execute until $taskA_1$ is finished, and $taskA_3$ cannot execute until $taskB_2$ is finished. A task in this example can be either an update or a query to the database. Both the tasks and communicating predicates *send* and *receive* are elementary database operations, which cause the transitions of database states.

The model-theoretic semantics of CTR is based on *multi-paths* or *m-paths*, which are generalized from the notion of TR *paths*. Formally, an m-path is a finite sequence of paths, where each path presents a period of continuous execution. For example, if D_1, D_2, ... D_8 are database states, then $\langle D_1D_2D_3,\ D_4D_5,\ D_6D_7D_8\rangle$ is an m-path. If the m-path represents the execution history of a process ϕ, then it means that ϕ has three periods of continuous execution and suspends twice during the whole execution. Corresponding to the logical connectives $\otimes$, $|$ and $\odot$, there are three operators on m-paths: *concatenation*, *interleaving* and *reduction*.

Suppose that m-paths $\tau = \langle \mathrm{k}_1,\ ...,\ \mathrm{k}_n \rangle$ and $\tau' = \langle \mathrm{k'}_1,\ ...,\ \mathrm{k'}_m \rangle$ represent the execution of ϕ and ϕ', respectively.

- the *concatenation* of m-paths τ and τ', i.e., $\tau \bullet \tau'$, represents the serial execution of ϕ and ϕ', i.e., $\phi \otimes \phi'$; $\tau \bullet \tau' = \langle \mathrm{k}_1,\ ...,\ \mathrm{k}_n,\ \mathrm{k'}_1,\ ...,\ \mathrm{k'}_m \rangle$.

- the *interleaving* of m-paths τ and τ', i.e., $\tau \parallel \tau'$, represents the concurrent execution of ϕ and ϕ', i.e., $\phi \mid \phi'$. If n = 2 and m = 3, then the interleaved new m-path may be $\langle \mathrm{k}_1,\ \mathrm{k'}_1,\ \mathrm{k}_2,\ \mathrm{k'}_2,\ \mathrm{k'}_3 \rangle$ or $\langle \mathrm{k}_1,\ \mathrm{k}_2,\ \mathrm{k'}_1,\ \mathrm{k'}_2,\ \mathrm{k'}_3 \rangle$. It is worth noticing that the new interleaved m-path keeps the orders of path fragments of the original m-paths.

- suppose $\tau = \langle \mathrm{k}_1,\ \mathrm{k}_2,\ \mathrm{k}_3 \rangle$ represents the execution of ϕ. If paths k_1 and k_2, k_2 and k_3 can be concatenated, then ϕ is able to execute continuously along path $\tau' = \langle \mathrm{k}_1\mathrm{k}_2\mathrm{k}_3 \rangle$, i.e., τ reduces τ'. The idea is that if ϕ is suspended and re-awakened in state k_1 or k_2, it can also execute continuously.

To define truth on m-paths, each m-path is assigned a first-order formula, which specifies the formula atoms that are true on the m-path [167]. In other words, a transaction formula, which is true on an m-path represents actions that take place along the m-path. The proof theory of CTR has an efficient SLD-style proof procedure, and more details can be found in [167].

To sum up, CTR is a formal unified logical framework that integrates concurrency, communication and updates. On one hand, CTR is capable of combining element database updates and queries into complex database transactions. On the other hand, CTR supports both synchronous and asynchronous communication between processes. The communication paradigm within CTR is inspired by π-calculus, but CTR provides a more flexible concurrency than π-calculus which requires a hand-shake before the communication [167]. Moreover, the RbAF proposed in this thesis employs *messaging reaction rules* to describe (abstract) processes in terms of message-driven conversations between agents and uses *derivation rules* to describe knowledge-intensive decision-centric steps in workflows. CTR provides the logical foundations for both state changes and interaction between concurrent processes in a logic programming language and is close to the rule-based workflow language of the RbAF. This is also the reason why logical CTR is employed as a theoretical basis for the declarative rule-based description of the WsSWFs.

5.3 Workflow Representation Using CTR

5.3.1 Workflow Representation

Définition 1 (Scientific Workflow) *A scientific workflow is a collection of coordinated tasks composed to accomplish complex goals, and it is represented as a CTR Horn goal.*

In CTR, a Horn goal that is defined recursively as follows:

- an atomic formula is a CTR Horn goal;
- if ψ and ϕ are CTR Horn goals, then so are the expressions: $\psi \otimes \phi$, $\psi \mid \phi$, $\psi \vee \phi$;
- $\odot\ \psi$, where ψ is a CTR Horn goal.

The tasks in the RbAF could be *primitive* tasks or *composite* tasks. A *primitive* task is an elementary unit of a workflow, and a *composite* task defines the execution order of a set of tasks (aka. a sub-workflow or a sub-process).

Définition 2 (Primitive Task) *The primitive task corresponds to an atomic activity of a workflow, and it is represented in CTR as an atomic formula.*

A primitive task represented by an atomic formula has the following format:

$$p(arg_1,\ ...,\ arg_n).$$

Here, predicate p denotes the name of a task, and arg_1, ..., arg_n $(n \geqslant 1)$ are task arguments that are the same as the terms defined in the first-order logic.

Since this chapter focuses on representing complex workflows using CTR, the details of task arguments are omitted. For brevity, a primitive task atom is often abbreviated as $p(\overline{X})$, where $\overline{X}$ denotes all arguments that p takes.

The states in the CTR-based workflow modeling are regarded as *datasets*. Each state is a set of data items that represent current workflow status. More precisely, if D is a state, the data oracle $\mathcal{O}^d(\mathrm{D})$ which corresponds to queries to a particular state. The transition oracle is defined as a task that consumes data to accomplish certain goals. This thesis assumes that a task must have at least one input. Formally, for a task having both input(s) and output(s), $task(\bar{i},\ \bar{o})\ \in\ \mathcal{O}^t(D_1, D_2)$ iff $D_2\ =\ D_1\ \cup\ \{\bar{o}\}\ -\ \{\bar{i}\}$. Here, $\bar{i}$ and $\bar{o}$ represent the input(s) and output(s) of the task, respectively. For a task having only input(s), $task(\bar{i})\ \in\ \mathcal{O}^t(D_1,\ D_2)$ iff $D_2\ =\ D_1\ -\ \{\bar{i}\}$.

A primitive task can change the workflow state and act as a query to present a particular workflow state. In terms of different situations, a primitive can be an *update-task* and a *query-task*, respectively. A workflow language that programs state-changing actions is helpful to pass data between tasks. For example, a query task can check if its precedent tasks are completed or the required data for its execution is available. The RbAF considers data passing as event messages, more details about the CEP-based workflow representation can be found in Section 5.5.

The execution of primitive tasks can be guarded by conditional statements, i.e., preconditions and post-conditions in workflows. For example, the following formula denotes that the protein prediction analysis can be executed if the user input is a protein.

$$isProtein(Protein) \otimes proteinAnnotaionAnalysis(Protein, GOTerms)$$

Here, the atom *isProtein(Protein)* checks if the user input is a protein or not. Different with simple Boolean expressions, such condition evaluation usually involves complex decision logic based on domain-specific knowledge.

Définition 3 (Composite Task) *A composite task (aka. a sub-workflow or sub-process) is the composition of a set of tasks, and it is defined as a CTR Horn rule with a form* p $\leftarrow$ ϕ*, where* p *is an atomic formula, and* ϕ *is a CTR Horn goal.*

Since CTR Horn rules define the composition of a workflow, they are also known as *workflow formation rules* in this thesis. The head of a workflow formation rule is a predicate, which corresponds to a composite task only. As the primitive task, the name of the predicate denotes the name of the composite task. The body of the workflow formation rule recursively gives the definition of the composite task. For example, the process of ant identification is represented as:

$$\begin{aligned} identProcess(Cid, AntDesc) \leftarrow\ & allocation(AntDesc, Agent) \otimes \\ & ident(Cid, Agent, AntDesc, Result) \otimes \\ & identDone(Cid, AntDesc, Result) \end{aligned}$$

The identification starts with a task allocation, which assigns the identification task to an agent. After the identification task *ident* is done, an event *identDone* is triggered to denote the end of the identification. The reactive logic representation in CTR can be found in Section 5.3.3.

The connectives $\otimes$, $|$ and $\vee$ used to specify a CTR Horn goal have the following semantics in this thesis:

- $\phi \otimes \psi$: execute task ψ after task ϕ. Model-theoretically, $\phi \otimes \psi$ is satisfied (or is true) on an m-path τ if and only if ϕ and ψ are true on some m-paths τ_1, τ_2 whose concatenation $\tau_1 \bullet \tau_2$ reduces to τ, i.e., $\tau_1 \bullet \tau_2 = \tau$.
- $\phi \mid \psi$: tasks ψ and ϕ are executed concurrently. Model-theoretically, $\phi \mid \psi$ is true on an m-path τ if and only if ϕ and ψ are true on some m-paths τ_1, τ_2 whose interleaving reduces to τ, i.e., $\tau_1 \parallel \tau_2 = \tau$.
- $\phi \vee \psi$: represents a nondeterministic task, which means "execute task ϕ or execute task ψ". Model-theoretically, $\phi \vee \psi$ is true on an m-path τ if and only if either ϕ or ψ is true over on τ.

$\otimes$, $|$, and $\vee$ are binary connectives, which compose two tasks into a composite one. Besides them, $\neg$ is a unary connective, and the satisfaction of $\neg\phi$ is defined

as: $\neg\phi$ is true on any m-path τ if and only if task ϕ is not true on the m-path τ. $\neg$ plays an important role to express decision logic of scientific workflows. $\odot\phi$ means the execution of task ϕ must not interleave with other concurrently running tasks. The satisfaction of $\odot\phi$ is true on any m-path τ if and only if τ is a path. Since $\odot$ is often used to specify atomic actions that are rarely used in scientific workflows, it is not considered in this thesis. In addition, it is worth noticing that, the body of CTR Horn rules does not include the classical connective $\wedge$, which is usually expressed as a constraint in TR [165].

Définition 4 (Optional Task) *Optional tasks are the ones that may not be directly needed for building workflows (since they do not affect workflow results), but they allow for necessary variation and make workflows more flexible. An optional task is presented as a composite nondeterministic task as:* $\psi \vee$ ***state***.

Here, the **state** is a special propositional constant, which is true on paths of length 1, i.e., on database states. This means that $\psi \vee$ **state** is always evaluated to be true, even if task ψ is not executed.

Définition 5 (Iteration Task) *Iteration tasks are the ones which are executed repeatedly during a workflow. In general, three distinct forms of repetition can be identified: arbitrary cycles, recursion and structured loop [136].*

Arbitrary cycles mean that the workflow tasks should have more than one entry or exit point. CTR can represent *arbitrary cycles* by reaction rules, which specify the conditions under which actions can be done. Moreover, events triggering reaction rules may from multiple sources. More details about the reactive logic representation in CTR will be found in Section 5.3.3.

A *recursive task* means that it can invoke itself during its execution or an ancestor in terms of the overall decomposition structure with which it is associated [136]. In order to ensure that the recursion does not lead to infinite self-referencing decomposition, a recursive task in this thesis is represented by a pair of CTR Horn rules: one describes a recursive task in terms of itself, the other describes a termination condition to ensure that the overall process will eventually complete normally.

$$\psi(\overline{T}) \leftarrow \varphi \otimes \psi(\overline{U}) \otimes \phi$$
$$\psi(\overline{T}) \leftarrow (\Box\neg\varphi) \otimes \mathbf{state}$$

This above example illustrates a recursive task $\psi(\overline{T})$. It contains itself ψ and means if φ is true in the current state, then another instance of $\psi(\overline{U})$ is started. Otherwise, the recursion process is completed. Here, $\Box\neg\varphi$ is interpreted as "otherwise". $\Box\varphi$ means that the execution of φ is *necessary* at the present state (*Necessity* means that φ is executable along every path leaving the present state) [165]. The negation "$\neg$" here is of the NaF variety; i.e., it is based on the perfect-model semantics and only locally-stratified logic programs are considered [165] (More details about stratified logic programs can be found in Section 2.8). In what follows, the negation $\neg$ has the same interpretation as here, unless explicitly specified.

A *structure loop* has either a pre-test or post-test condition associated with it that is either evaluated at the beginning or the end of the loop to determine whether it should continue [136]. There are two general forms of this pattern—a *while* loop which equates to the classic *while...do* pre-test loop construct used in programming languages, and a *repeat* loop which equates to the *repeat...until* post-test loop construct. Both of them can be represented as recursive tasks in a simple, declarative way. For example, the standard *while...do* construct can be represented as follows:

$$while_\varphi_\phi \leftarrow \varphi \otimes \phi(\overline{U}) \otimes while_\varphi_\phi$$
$$while_\varphi_\phi \leftarrow (\Box\neg\varphi) \otimes \mathbf{state}$$

Similarly for the representation of *repeat...until*.

5.3.2 Multiple Instances

Multiple instance patterns describe situations where there are multiple threads of execution active in a process which relate to the same activity [136]. In scientific workflows, multiple instances are usually helpful to improve efficiency if a group of data items can be processed in parallel.

With CTR, it is possible to create multiple concurrent instances of an activity or sub-process. The following example adapted from [168] demonstrates multiple instances of *processor* are running in parallel. Each item processed by a *processor* is initially in the "in basket", represented by database relation *item*. After the item is processed, it is stored in the "out basket", represented by database relation *basket*. The first rule defines a process of retrieving an item from the "in basket", creating a new task instances to process it and then inserting the item in the "out basket". The recursion of the *simulate* rule allows multiple instances of the *processor* running in parallel.

$$simulate \leftarrow getItem(W) \otimes [simulate \mid processor(W) \otimes putItem(W)]$$
$$simulate \leftarrow item.empty$$
$$getItem(W) \leftarrow \odot[item(W) \otimes del.item(W)]$$
$$putItem(W) \leftarrow ins.basket(W)$$

5.3.3 Reactive Logic Representation

To support the WsSWF execution, the RbAF employs messaging reaction rules to describe interactions between distributed agents. Messaging reaction rules concern message-driven conversations between agents, and sending and receiving event messages are associated with the conversation identifiers to reflect the process execution. This section shows how messaging reaction rules can be represented in CTR.

Bonner et al. [169] generally show how active rules can be represented as transaction bases in TR. Such active rules usually determine what actions are performed if a given event occurs and a stated condition holds, and thus such active rules are also known as ECA rules. ECA rules are typically defined with a global scope (global state) and react on internal events of the reactive system, such as changes

(updates) in the active database. Instead of active ECA rules, the RbAF employs messaging reaction rules to describe the workflow composition. Messaging reactive rules are not only capable of capturing global event occurrences as ECA rules, but also capable of performing complex actions locally in a particular context. In other words, they allow a complex process to be performed by distributed agents and are more suitable to specify the workflow logic.

In the CTR-based workflow representation, sending a message can be represented by an activity, which sends a message to itself or other agents. The sending activity can be simply represented as a task in the body of a CTR Horn rule. Receiving a message, i.e., the reactive logic, is described by reaction rules. As mentioned in Section 4.3.1, *reaction rules* are a collection of the reactive rules. Depending on the parts of the general syntax, reaction rules can be specialized into different types: *derivation rules*, *production rules* (if-do), *trigger rules* (on-do), and *ECA rules* (on-if-do). For example, the actions after the ant identification are presented by two ECA rules as follows:

$$\begin{array}{l}
\textbf{define} \quad \textbf{reactive} \quad \textbf{rule} \quad identDone \\
\textbf{on} \quad identDone(Cid, AntDesc, Result) \\
\textbf{if} \quad isIdentFailed(AntDesc, Result) \\
\textbf{do} \quad humanIdent(Cid, AntDesc).
\end{array}$$

$$\begin{array}{l}
\textbf{define} \quad \textbf{reactive} \quad \textbf{rule} \quad identDone \\
\textbf{on} \quad identDone(Cid, AntDesc, Result) \\
\textbf{if} \quad not(isIdentFailed(AntDesc, Result)) \\
\textbf{do} \quad \textbf{state}.
\end{array}$$

CTR provides a complete formalization (including a model and a proof theory) for the behavior of the system, and it also has one underlying notation and semantics, which can describe behavior procedurally in detail, or declaratively at a high level [169]. The above reaction rules can be represented in CTR as follows:

$$\begin{array}{l}
\ldots \\
identProcess(cid, AntDesc) \leftarrow allocation(AntDesc, Agent) \otimes \\
\qquad ident(cid, Agent, AntDesc, Result) \otimes \\
\qquad identDone(cid, AntDesc, Result) \\
\\
identDone(Cid, AntDesc, Result) \leftarrow identFailed(AntDesc, Result) \otimes \\
\qquad humanIdent(Cid, AntDesc) \\
identDone(Cid, AntDesc, Result) \leftarrow (\Box\neg\ identFailed(AntDesc, Result)) \otimes \textbf{state} \\
\ldots
\end{array}$$

The ant identification process ends with an event *identDone* following the identification task *ident*. The event *identDone* is defined by a pair of rules used to process the event. That is, the event *identDone* is invoked after the identification and triggers a nondeterministic action depending on the identification results: escalating to human taxonomists or ending normally.

5.4 Communication between Processes

In the RbAF, each agent not only manages a process to perform one or more tasks, but also coordinates (communicates) with other agents via sending and receiving event messages to complete certain goals. Unlike other deductive database languages, CTR integrates concurrency, communication and database updates in a logical framework [167].

CTR provides a different semantics for the communication between the agents. During the communication, a state is a set of communication channels. Each agent manages a communication channel, and the communication between the agents can be regarded as sending and receiving synchronization messages across channels. Each channel has a set of messages, i.e., a pool of messages. The data oracle defines a binary predicate, *peek*, which is used to specify queries related to a message in a channel. Formally, $peed(q, msg) \in \mathcal{O}^d(D)$ *iff* the channel in D has the message *msg*. Here, D is a state. The transition oracle defines two binary predicates: *send* and *receive*. *send(cid, receiver, msg)* denotes sending a message *msg* to an agent *receiver*, and *receive(cid, sender, msg)* denotes receiving a message *msg* from an agent *sender* (Note that some parameters of the event message are omitted for clarity). Formally, *send(cid, receiver, msg)* $\in \mathcal{O}^t(D_1, D_2)$ *iff* the message pool managed by the agent *sender* has *msg*, and D_2 is obtained from D_1 by removing *msg* from the *sender*. Likewise, *receive(cid, sender, msg)* $\in \mathcal{O}^t(\mathrm{D}_1, \mathrm{D}_2)$ *iff* D_2 is obtained from D_1 by adding *msg* to the message pool managed by the *receiver*.

In RbAF, the data passing between tasks is regarded as event messaging between agents. One agent sends a message, and another receives it. In such situations, event messages act as a "data carrier" during the communication. Returning to the ant treatment use case, the communication process between the fieldworker and the taxonomist is represented as follows:

$$\begin{aligned}
&\textbf{fieldworkerProcess} \leftarrow antDescription \otimes receive(Cid, Taxonomist, Report)\\
&antDescription \leftarrow description(AntDesc) \otimes send(Cid, Taxonomist, AntDesc)\\
\\
&\textbf{taxonomistProcess} \leftarrow receive(Cid, FieldWorker, AntDesc) \otimes\\
&\qquad\qquad identProcess(Cid, AntDesc) \otimes archive(Cid, Result) \otimes\\
&\qquad\qquad treatment(Result, Treatment) \otimes report(Cid, Result)\\
&\ldots\\
\\
&archive(Cid, Result) \leftarrow send(Cid, Curator, Result)\\
\\
&report(Cid, Result) \leftarrow createReport(Result, Report) \otimes\\
&\qquad\qquad send(Cid, FieldWorker, Report)
\end{aligned}$$

The processes *fieldworkerProcess* and *taxonomistProcess* manage the behavior of the fieldworker and the taxonomist, respectively. They run concurrently via communication with each other. The fieldworker first sends the ant description to an ant taxonomist. Thereafter, it waits for a report of ant identification and treatment

from the taxonomist. The identification task starts when the taxonomist receives the ant description. The taxonomist identifies it, finds an appropriate treatment scheme, generates a report and then sends it to the fieldworker and the two processes complete.

5.5 Complex Event Processing

Reaction rules specify under which conditions a task can execute, and these conditions determine intelligent routings at runtime. In this thesis, the RbAF considers the data passing between tasks as event messaging and employs the rule-based CEP technologies to represent composite events, thereby implementing complex workflow patterns (see 4.3.3).

A composite event is the combination of several base events. Each composite event is usually described by an event pattern, which contains event templates, relational operators and variables. However, the rule-based CEP is goal-driven and the check of a given event pattern is always performed at the time when the goal is set [170]. In order to address this limitation, this section adapts an *event-driven, backward chaining* approach proposed by Anicic et al. [170] to detect complex event patterns. The approach enables both logic-based (backward chaining) and data-driven (forward chaining) complex event detection.

Since the RbAF imposes no constraints on the reaction time with regard to the event processing (aka. an *any-time* reaction rule system), there is no need to process some particular events in real-time (e.g., sequence events, concurrent events [170]). This section only represents the composite events mentioned in Section 4.3.3 based on TR, rather than provides a comprehensive complex event pattern representation involved in the *real-time* CEP.

Conjunction of Events An event pattern based on the conjunction of events requires all base events are detected. For example, $ce_1 \leftarrow e_1 \wedge e_2 \wedge e_3$ defines that a composite event ce_1 occurs when all base events e_1, e_2 and e_3 are detected. The detection of conjunction events corresponds to the *AND* join connector. They can be directly mapped a CTR serial conjunction of base events, as shown follows:

$$\begin{array}{ll}
receive(cid, sender_1, e_1) \text{ :- } & insert(e_1) \otimes checkCE \\
receive(cid, sender_1, e_1) \text{ :- } & (\Box \neg checkCE) \otimes \mathbf{state} \\
receive(cid, sender_2, e_2) \text{ :- } & insert(e_2) \otimes checkCE \\
receive(cid, sender_2, e_2) \text{ :- } & (\Box \neg checkCE) \otimes \mathbf{state} \\
receive(cid, sender_3, e_3) \text{ :- } & insert(e_3) \otimes checkCE \\
receive(cid, sender_3, e_3) \text{ :- } & (\Box \neg checkCE) \otimes \mathbf{state}
\end{array}$$

$$\begin{array}{ll}
checkCE \leftarrow & e_1 \otimes e_2 \otimes e_3 \otimes \\
& ce_1 \otimes \\
& delete(e_1) \otimes delete(e_2) \otimes delete(e_3)
\end{array}$$

There are two types of rules in the above example: *forward chaining rules* used for detecting the base events and *backward chaining rules* used for reasoning the

complex event pattern. To make a difference, the notations :- and $\leftarrow$ in the above rules denote the *forward base event detection* and the *backward complex event pattern reasoning*, respectively. The first six *receive* rules represent the base event detection, and base events e_1, e_2 and e_3 are inserted into the database when they are detected. Then the *checkCE* rule checks if these base events already exist in the database. This is done by the subgoals of the *checkCE* rule e_1, e_2 and e_3 that act the same as the data oracle *peek*. The composite event ce_1 occurs if all base events are detected (i.e., all base events have been inserted into the database). After that, the base events e_1, e_2 and e_3 are removed from the database. Note that the base events are always successfully inserted into the database even if the execution of the *checkCE* rule fails (represented by $\Box\neg checkCE$). This is really important because the occurred base events should be in the database during a complex event pattern detection. Suppose that base events e_1 and e_2 are detected before e_3, the execution of the *checkCE* rule fails, but e_1 and e_2 have been both successfully inserted into the database. When e_3 is detected, the third rule inserts event e_3 into the database, and then triggers composite event ce_1.

Disjunction of Events An event pattern based on the disjunction of events is satisfied when any base event is detected. The detection of disjunction events corresponds to the *XOR* join connector. For example, $ce_2 \leftarrow e_1 \vee e_2 \vee e_3$ defines that composite event ce_2 occurs when any of base events e_1, e_2 and e_3 is detected. In this work, the detection of a disjunction event pattern is represented as follows:

$$
\begin{array}{ll}
receive(cid, sender_1, e_1) \text{ :- } & insert(e_1) \otimes checkCE \\
receive(cid, sender_1, e_1) \text{ :- } & (\Box\neg checkCE) \otimes \mathbf{state} \\
receive(cid, sender_2, e_2) \text{ :- } & insert(e_2) \otimes checkCE \\
receive(cid, sender_2, e_2) \text{ :- } & (\Box\neg checkCE) \otimes \mathbf{state} \\
receive(cid, sender_3, e_3) \text{ :- } & insert(e_3) \otimes checkCE \\
receive(cid, sender_3, e_3) \text{ :- } & (\Box\neg checkCE) \otimes \mathbf{state}
\end{array}
$$

$$
\begin{array}{l}
checkCE \leftarrow e_1 \otimes ce_2 \otimes delete(e_1) \\
checkCE \leftarrow e_2 \otimes ce_2 \otimes delete(e_2) \\
checkCE \leftarrow e_3 \otimes ce_2 \otimes delete(e_3)
\end{array}
$$

Compared with the conjunction event pattern mentioned above, there are three backward chaining *checkCE* rules used to detect the disjunction of events. It means that when either base event e_1, e_2 or e_3 is detected, then one rule *checkCE* succeeds and the composite event ce_2 occurs. Remember that the *XOR* join connector can have either *local* or *non-local* semantics (see Section 5.1). If base events e_1, e_2 and e_3 are mutually exclusive, composite event ce_2 precisely occurs once when one of base events e_1, e_2 or e_3 is detected. In the RbAF, the composite event consumption is controlled by its consumers. If base events e_1, e_2 and e_3 are not mutually exclusive, then composite event ce_2 may occur more than once, and its consumer will decide to consume it or not (see Section 4.3.3).

Advanced Event Patterns The *conjunction* and *disjunction* are standard operators to describe complex event patterns. They are often used to represent the

gateways acted as join connectors in workflows. With the *event-driven, backward chaining* approach, it is also possible to represent more complex join connectors in workflows. For example, the following rules represent a complex event ce_3, which occurs when base events e_1 and e_2 are detected, and event e_3 is optional.

$$
\begin{array}{ll}
receive(cid, sender_1, e_1) \text{ :- } & insert(e_1) \otimes checkCE \\
receive(cid, sender_1, e_1) \text{ :- } & (\Box\neg checkCE) \otimes \mathbf{state} \\
receive(cid, sender_2, e_2) \text{ :- } & insert(e_2) \otimes checkCE \\
receive(cid, sender_2, e_2) \text{ :- } & (\Box\neg checkCE) \otimes \mathbf{state} \\
receive(cid, sender_3, e_3) \text{ :- } & insert(e_3) \otimes checkCE \\
receive(cid, sender_3, e_3) \text{ :- } & (\Box\neg checkCE) \otimes \mathbf{state} \\
checkCE \leftarrow e_1 \otimes e_2 \otimes e_3 \otimes & \\
\quad ce_3 \otimes & \\
\quad delete(e_1) \otimes delete(e_2) \otimes delete(e_3) & \\
checkCE \leftarrow e_1 \otimes e_2 \otimes & \\
\quad ce_3 \otimes & \\
\quad delete(e_1) \otimes delete(e_2) &
\end{array}
$$

Moreover, in the aforementioned examples, the base events that need to be joined are determined in advance. However, sometimes only the number of events required to be joined is known. For example, suppose that multiple parallel instances of a task (say n) are created within a workflow. Once k ($k \leq n$) task instances have completed, the subsequent task is triggered. The following rule implements an advanced disjunction event pattern, where a complex event ce_4 occurs when any two of base events e_1, e_2 and e_3 are detected.

$$
\begin{array}{ll}
receive(cid, sender_1, e_1) \text{ :- } & insert(e_1) \otimes checkCE \\
receive(cid, sender_1, e_1) \text{ :- } & (\Box\neg checkCE) \otimes \mathbf{state} \\
receive(cid, sender_2, e_2) \text{ :- } & insert(e_2) \otimes checkCE \\
receive(cid, sender_2, e_2) \text{ :- } & (\Box\neg checkCE) \otimes \mathbf{state} \\
receive(cid, sender_3, e_3) \text{ :- } & insert(e_3) \otimes checkCE \\
receive(cid, sender_3, e_3) \text{ :- } & (\Box\neg checkCE) \otimes \mathbf{state} \\
& \\
checkCE \leftarrow e_1 \otimes e_2 \otimes ce_4 \otimes & \\
\quad delete(e_1) \otimes delete(e_2) & \\
checkCE \leftarrow e_1 \otimes e_3 \otimes ce_4 \otimes & \\
\quad delete(e_1) \otimes delete(e_3) & \\
checkCE \leftarrow e_2 \otimes e_3 \otimes ce_4 \otimes & \\
\quad delete(e_2) \otimes delete(e_3) &
\end{array}
$$

There are three *checkCE* rules used to detect any two of the events e_1, e_2 and e_3. In general, the number of *checkCE* rules used to detect this static partial join pattern depends on a combination computation:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \qquad (k \leq n)$$

Compared with the *event-driven, backward chaining* complex event detection [170], the RbAF also implements the CEP in a logical declarative way. The difference is that, the RbAF keeps the occurred base events in the database and directly detects complex event patterns whenever a base event is detected. Moreover, the rule-based CEP of the RbAF can represent more advanced event patterns, as shown in the above section. In addition, it is worth noticing that the decision conditions represented by the CEP are not limited to what have explained in this section. Since the RbAF describes workflow decision logic by derivation rules and supports access to external domain ontologies, it is possible to explore inherent semantic-based relationships between events to make sophisticated workflows.

5.6 Exception Handling

In general, CTR also supports an exception handler with an *if-then-else* structure as follows:

$$\psi \leftarrow \phi \otimes \textbf{state}$$
$$\psi \leftarrow (\Box\neg\phi) \otimes eHandler$$

This example describes task ϕ is part of a composite task ψ. If ϕ succeeds, it does nothing. Otherwise, a designated error-handling routine *eHandler* is triggered. In this way, task ψ can complete even if there is an exception.

The RbAF supports both automatic and manual exception handling (see Section 4.7). The former one is performed by the EHA (i.e., *eHanderAgent* in the following CTR formulas), which is employed whenever an exception occurs, as shown follows:

$$\textbf{eHandler} \leftarrow send(Cid, errorHandlerAgent, Ex)$$
$$\otimes receive(Cid, eHandlerAgent, Sol)$$

Here, *Ex* and *Sol* denote the exception and its solution, respectively. During the automatic exception handling, the *eHanderAgent* queries the public workflow ontology to find alternative counterparts with the same effect and replace failed ones dynamically. Once an exception cannot be handled by the *eHanderAgent*, the exception will be escalated to human users to make a decision or provide required resources, i.e., passing the exception to the HA (i.e., *humanAgent* in the following CTR formulas). In this case, the HA manages the life cycle of these exceptions and provides a friendly user interface for humans to operate on the exceptions.

Based on the messaging constructs introduced in Section 5.4, the communication between the EHA and the HA (Figure 4.13) can be represented by two concurrent processes, as shown follows:

$$\textbf{automateEHandler} \leftarrow receive(Cid, From, Ex) \otimes$$
$$automateEHProcess(Ex, Sol) \otimes send(Cid, From, Sol)$$

$$automateEHProcess(Ex, Sol) \leftarrow handleException(Ex, Sol) \otimes \textbf{state}$$
$$automateEHProcess(Ex, Sol) \leftarrow (\Box\neg handleException(Ex, Sol))\otimes human_handler(Ex)$$
$$human_handler(Ex) \leftarrow send(Cid, humanAgent, Ex) \otimes receive(Cid, humanAgent, Sol)$$

$$\textbf{humanEHandler} \leftarrow receive(Cid, eHandlerAgent, Ex) \otimes humanDecision(Ex, Sol) \otimes send(Cid, eHandlerAgent, Sol)$$

It shows that the *eHanderAgent* sends an exception to the *humanAgent* if it cannot handle it. The rule $human_handler(Ex)$ uses a pair of sending and receiving activities to communicate with the *humanAgent* (i.e., the HA). It is locally used within the rule *automateEHandler*, i.e., the rule *human_handler* is only applied if the automatic exception handling fails.

Note that exception handling of this thesis follows the spirit of aspect-oriented programming, and the workflow exceptions are externally handled in the *eHanderAgent* or *humanAgent*. In other words, the logic of exception handling is not intertwined with workflow processes but is encapsulated in *eHanderAgent* and *humanAgent*.

As a summary, a comprehensive CTR-based representation of the ant identification and treatment process is shown as follows:

$$\textbf{fieldworkerProcess} \leftarrow antDescription \otimes receive(Cid, Taxonomist, Report)$$
$$antDescription \leftarrow description(AntDesc) \otimes send(Cid, Taxonomist, AntDesc)$$

$$\textbf{taxonomistProcess} \leftarrow receive(Cid, FieldWorker, AntDesc)\otimes identProcess(Cid, AntDesc) \otimes archive(Cid, Result)\otimes treatment(Result, Treatment) \otimes report(Cid, Result)$$

$$identProcess(Cid, AntDesc) \leftarrow allocation(AntDesc, Agent)\otimes ident(Cid, Agent, AntDesc, Result)\otimes identDone(Cid, AntDesc, Result)$$

$$identDone(Cid, AntDesc, Result) \leftarrow identFailed(AntDesc, Result)\otimes humanIdent(Cid, AntDesc)$$
$$identDone(Cid, AntDesc, Result) \leftarrow (\Box\neg\, identFailed(AntDesc, Result)) \otimes \textbf{state}$$

$$archive(Cid, Result) \leftarrow send(Cid, Curator, Result)$$

$$humanIdent(Cid, AntDesc) \leftarrow send(Cid, Human, AntDesc)\otimes receive(Cid, Human, Result)$$
$$report(Cid, Result) \leftarrow createReport(Result, Report)\otimes send(Cid, FieldWorker, Report)$$

$$\textbf{humanProcess} \leftarrow receive(Cid, From, AntDesc) \otimes humanIdent \otimes send(Cid, From, Result)$$

$$\textbf{curatorProcess} \leftarrow receive(Cid, Taxonomist, Result) \otimes archiveResult(Cid, Taxonomist, Result)$$

5.7 Summary

Most workflow languages are procedural (e.g., BPEL, AO4BPEL and YAWL), as presented in Section 3.1. The workflow formal models, such as Petri Nets and calculi theory (e.g., π-calculus) are mathematical workflow representations and provide theoretical foundations to these workflow programming languages. Although there are rule-based workflow languages that support flexible service composition and model the process logic with declarative rules, most of them only provide static syntactical process descriptions without precise declarative formal semantics.

The WsSWFs considered in this thesis not only involve interactions between multiple participants, but also have complicated logic to express scientific policies and cater to dynamic execution environments. That is why CTR is employed as a theoretical basis for the rule-based workflow language of the RbAF. Some efforts also try to provide a logical framework for modeling flexible workflows: Roman et al. introduce a CTR-based logical model for process specification, contracting for services, service enactment, and reasoning [171]. With the help of CTR, the model specifies constraints as part of service contracts. Compared with this work, the model in [171] focuses on both process specification and service contracting. This thesis, however, not only explicitly considers the abstract representation of WsSWFs, but also provides a proof-of-concept implementation for it (see next chapter). DECLARE is a prototype of a WFMS that uses a constraint-based process modeling language for the development of declarative models describing loosely-structured processes [106]. Based on LTL, DECLARE provides constraint templates to model constraints, such as “init”, “1..*”, “response”, “responded existence”. However, LTL is often used for abductive planning, while this thesis focuses on the execution of decision logic of the WsSWFs, i.e. deductive reasoning with rules based on facts.

This chapter introduced a CTR-based formal semantics of the rule-based workflow language presented in this thesis and especially introduced reactive workflow logic, communications between sub-processes and domain-specific decision logic with a purpose of supporting the WsSWFs. CTR not only provides a logical framework for the workflow execution, but also accommodates underlying semantics to describe the system behavior declaratively at a high level.

Part III

Evaluation

Chapter 6

Proof-of-Concepts

Contents

6.1 Prova 105
6.2 The Workflow Ontology 106
6.3 Mapping the CTR-Based Workflow Logic to Prova 108
6.4 Domain Logic Expression in Prova 113
6.5 Enterprise Service Bus Mule 116
6.5.1 Prova Agent Deployment 117
6.5.2 Mule ESB as Communication Middleware 119
6.5.3 Translations between Reaction RuleML and Prova 120
6.6 Exception Handling 122
6.7 User Client 124
6.7.1 Workflow Submission 124
6.7.2 Exception Management 125
6.7.3 Human Task Management 126
6.7.4 RDF Data Management 127
6.8 Summary 127

Based on the conceptual framework, RbAF, presented in Chapter 4 and the formal semantics of the declarative rule-based workflow language introduced in Chapter 5, this chapter introduces a proof-of-concept implementation of them, the RAWLS.

6.1 Prova

Prova [59], derived from Mandarax [172] Java-based inference system, is both a Semantic Web rule language and a highly expressive distributed rule engine. On one hand, Prova combines different rule types and provides an expressive, hybrid, declarative and compact rule programming language. Prova supports rule-based workflows and business processes by the following syntactic and semantic Prova instruments [173]:

- *reactive messaging*;
- inherent non-determinism for defining process divergences;

- *concurrency support*, including partitioned and non-partitioned thread pools;
- built-in predicate *spawn* for running tasks;
- *process join*;
- *predicate join*;
- *reaction groups* combining event processing with workflows;
- support for *dynamic event channels*;
- *guards*

Prova has a tight integration of Java and Semantic Web technologies. Prova combines imperative, declarative and functional programming styles by using a Prolog syntax that allows calling external procedural attachments (e.g., Java methods). Also, Prova follows the spirit of W3C Semantic Web initiative and allows using external ontologies as a type system for declarative rules. With these combinations, it is possible that Prova can access external data sources via query languages, such as SQL, SPARQL and XQuery. As a programming language, Prova is also provided as an Eclipse plugin and supports programming Prova rules within Eclipse Integrated Development Environment (IDE).

On the other hand, Prova is an economic and efficient, JVM-based rule engine that supports the execution of declarative decision rules, complex reaction rule-based workflows, rule-based CEP in a runtime production environment. Moreover, it is designed to work in distributed Enterprise Service Bus (ESB) and OSGi environments.

The latest version of Prova 3.2.1 was published in January of 2013. This chapter shows how Prova is adapted to specify the agent behavior in the RbAF, thereby describing the WsSWFs.

6.2 The Workflow Ontology

The upper-level workflow ontology of the RbAF provides general concepts of the workflow definition (see Section 4.2) and can be further specialized with domain-specific ontologies. This thesis designs the workflow ontology with Protege [174]—a famous ontology editor to design domain models and knowledge-based applications with ontologies.

Figure 6.1 shows a workflow ontology of the protein prediction result analysis. All tasks (i.e., *obtainReliableGOTerms* and *proteinPredictionAnalysis*) involved in the process are defined as the instances of the class *Task*. The tasks are performed by the agents. However, for the purpose of increasing system flexibility, the agents are not specified at workflow design time but are determined at runtime by the responsibilities of their roles. For example, the role *reliableGOTermProcessor* is responsible for the task *obtainReliableGOTerms*, which can be performed by both

agents *reliableGOTermAgentA* and *reliableGOTermAgentB*, which play the role *reliableGOTermProcessor*.

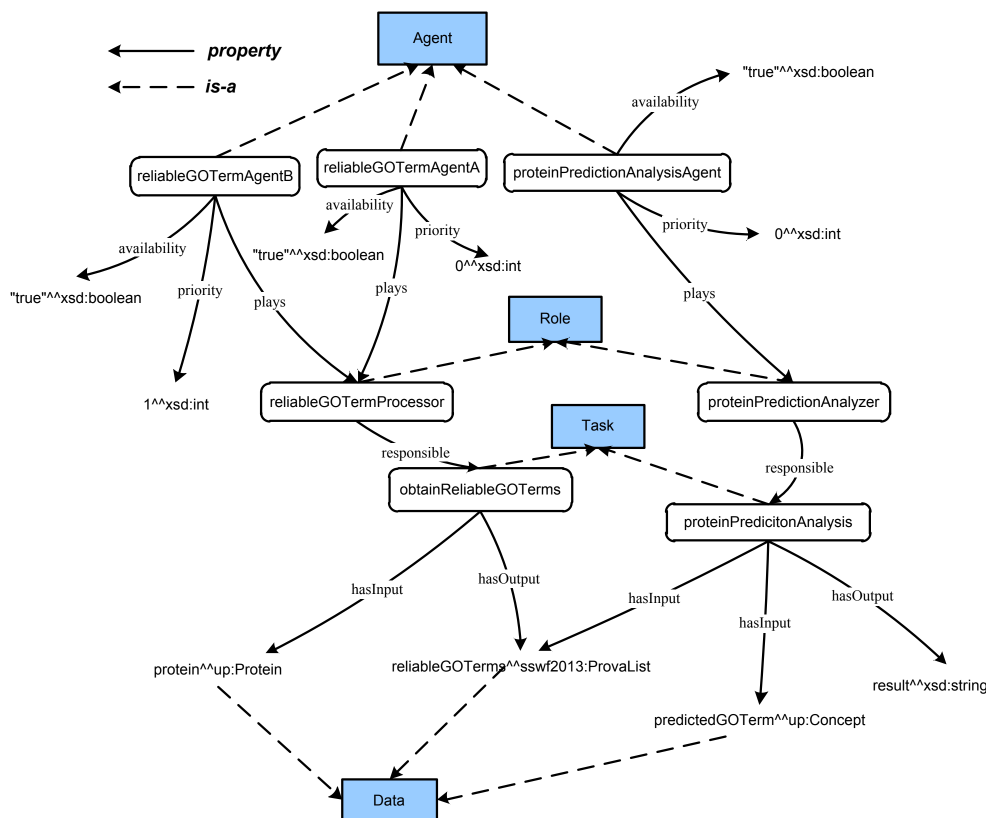


Figure 6.1: Protein Prediction Analysis Workflow Ontology

The tasks have inputs and outputs that are data. The data type is specified by a URI, which denotes a specification of a class (or an XML Schema datatype) that the data value belongs to. For example, the *obtainReliableGOTerms* task has an input data *protein*, whose type is *up:Protein* (denoted by “protein^^up:Protein” in Figure 6.1) defined in the UniProt core ontology [154].

The agent status and priority are useful for finding the most appropriate available agents. The status of an agent (i.e., the agent availability) could be *true* or *false*. Each agent has three levels of priority: 0 (the highest priority), 1 and 2 (the lowest priority). As shown in Figure 6.1, both agents *reliableGOTermAgentA* and *reliableGOTermAgentB* are available, and the former has a higher priority than the latter. Therefore, the following SPARQL query of finding the most appropriate available agent for the task *obtainReliableGOTerms* will get the agent *reliableGOTermAgentA* to perform it.

Listing 6.1: Finding an Agent for a Task

```
1 PREFIX : <http://www.corporate-semantic-web.de/sswf2013#>
```

```
2 PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
3 SELECT ?agent
4 WHERE {
5    ?role :responsible ?task .
6    ?agent :plays ?role .
7    ?agent :available "true"^^xsd:boolean .
8    ?agent :priority ?priority .
9    FILTER regex(str(?task), 'obtainReliableGOTerms')
10       }
11 ORDER BY DESC(?priority)
12 LIMIT 1
```

The agent status is also helpful to handle dynamic exceptions by finding alternative agents to replace exceptional ones. More details about the exception handling can be found in Section 6.6.

6.3 Mapping the CTR-Based Workflow Logic to Prova

Chapter 5 presents the CTR-based formal semantics of the declarative rule-based workflow language of the RbAF. This section presents its implementation by mapping the CTR-based workflow logic to Prova rules.

The CTR-based workflow logic is a set of CTR Horn rules that defines the composition of a workflow. As mentioned in Section 5.3, the body of a CTR Horn rule represents a CTR Horn goal with forms of: $\psi \otimes \phi$, $\psi \mid \phi$, $\psi \vee \phi$, where ψ and ϕ are CTR Horn goals. Prova is built upon Prolog, and a serial Horn rule $p(\overline{X}) \leftarrow \psi(\overline{Y}) \otimes \phi(\overline{Z})$ can be directly translated into a Prova rule with a form: $p(\overline{X})$:- $\psi(\overline{Y})$, $\phi(\overline{Z})$. This is simply done by replacing serial connector $\otimes$ with a comma ",", and $\psi(\overline{Y})$ and $\phi(\overline{Z})$ act as serial subgoals of the rule.

Furthermore, a nondeterministic composite task can be simply implemented by a set of Prova rules. For example, $p(\overline{X}) \leftarrow \psi(\overline{Y}) \vee \phi(\overline{Z})$ can be implemented by: (1) $p(\overline{X})$:- *cond*1, $\psi(\overline{Y})$.; (2) $p(\overline{X})$:- *cond*2, $\phi(\overline{Z})$. *cond1* and *cond2* denote the conditions under which to select $\psi(\overline{Y})$ or $\phi(\overline{Z})$.

The optional tasks and *if-then-else* statements can be translated into a set of Prova rules similarly. For an optional task, the propositional constant *state* is translated to a condition without subsequent goals, i.e., task $p(\overline{X}) \leftarrow \psi(\overline{Y}) \vee state$ can be implemented by: (1) $p(\overline{X})$:- *cond*1, $\psi(\overline{Y})$.; (2) $p(\overline{X})$:- *cond*2. The *if-then-else* statements, which are imposed by two opposite conditions, can be implemented by: (1) $p(\overline{X})$:- *cond*, $\psi(\overline{Y})$.; (2) $p(\overline{X})$:- *not*(*cond*), $\phi(\overline{Z})$. *not* denotes NaF, and *not(cond)* succeeds when all attempts to prove *cond* fail. The implementation of the concurrent tasks is based on Prova reactive event messaging and will be introduced later in this section.

The recursive tasks represented by CTR can be translated into a pair of Prova rules: one describes a recursive task in terms of itself, the other describes a termination condition to make the task complete normally. For example, the classic loop *while...do* can be implemented by two Prova rules: (1) $while_\varphi_\phi$:- φ, $\phi(\overline{U})$, $while_\varphi_\phi$.; (2) $while_\varphi_\phi$:- *not*(φ). *not*(φ) denotes the termination condition of the loop. Similarly for the *repeat...until* loop.

The communication between processes is implemented by the following Prova constructs of sending and receiving one or more context-dependent event messages:

Listing 6.2: Prova Constructs of Messaging Reaction Rules

```
1    sendMsg(XID, Protocol, Agent, Performative, Payload|Context)
2    rcvMsg(XID, Protocol, From, Performative, Payload|Context)
3    rcvMult(XID, Protocol, From, Performative, Payload|Context)
```

Here, *XID* is the conversation identifier of a message. *Protocol* defines the communication protocol. *Agent* and *From* denote the destination and source of the message, respectively. *Performative* describes the pragmatic context in which the message is sent. A standard nomenclature of the performative is, e.g., the FIPA ACL [133]. And *Payload|Context* denotes the actual content of the message.

The task dependencies of a workflow can be directly implemented by messaging reaction rules of Prova. There are two types of Prova reaction rules: *inline* and *global* reaction rules. The *inline* reaction rules usually locate in the body of a rule and act as its sub-goals. They can be restricted to accept just one message, a specified number of messages, or be limited by a timeout, which can be employed to implement the *non-local XOR* join connector, *local XOR* join connector and OR join connector, respectively (For more details about such connectors, see Sections 4.3.3 and 5.5). For example, the logic after the ant identification can be implemented by the following Prova rules. An inline reaction rule used to handle the identification result (Line 3) is waiting for an event message of pre-defined type “answer” on the *async* protocol. It can accept only one incoming message. For more details about receiving multiple messages or the message receiving limited by a timeout, see [173].

Listing 6.3: Example of Prova Inline Reaction Rules

```
1  identProcess(XID, From, AntDesc, Result) :-
2      ...,
3      rcvMsg(XID,async,Agent,answer,identDone(AntDesc, Result)),
4      bound(Result),
5      processResult(XID, AntDesc, Result).

7  processResult(XID, From, AntDesc, Result) :-
8      not(isIdentFailed(Result)).

10 processResult(XID, From, AntDesc, Result) :-
11     isIdentFailed(Result),
12     sendMsg(XID, async, humanAgent, ident(AntDesc)),
13     rcvMsg(XID, async, humanAgent, ident(Result)),
14     ....
```

The *global* reaction rules look exactly like Prova rules, but their semantics are more aligned with message (event)-driven reactive rules. A *global* reaction rule has a rule base lifetime scope, i.e., it is active while the rule base runs on a Prova engine (agent), and it is ready to receive any number of messages arriving at the agent. For example, the logic of waiting for external identification requests can be implemented by a global reaction rule as follows:

Listing 6.4: Example of Prova Global Reaction Rules

```
1 rcvMsg(XID,async,Agent,request, antIdent([inArgs|Paras], outArgs(Res))) :-
2     antIdentAllocation(XID,Paras, Res),
3     ....
```

Prova reactive messaging can be used to implement the concurrent tasks represented by CTR. Prova has four internal protocols for reactive messaging: *self*, *task*, *async*, and *swing* (Note that the *swing* protocol is not used in the RAWLS) [173]. An inline reaction rule *rcvMsg* itself is executed on the main thread, but it creates an inline reaction waiting for a reply according to its protocol. The *self* protocol indicates that the thread waiting for a reply is the main thread, and this protocol ensures sequential processing of received messages. The *task* protocol indicates that the thread waiting for a reply is randomly taken from a task thread pool, which is used for running tasks achieving maximum throughput. The *async* protocol makes good use of the conversation identifier, and the thread waiting for a reply is chosen in terms of the conversation identifier associated with event messages. This means that the messages belonging to one conversation are always processed on the same thread (see Listing 6.3). This is also a preferred way to handle long-going conversations. In addition, Prova has an *esb* protocol used for passing messages among components of an ESB, which is used as a container for Prova agents. Prova agents can be deployed locally on one ESB server or multiple ESB servers in different locations. In other words, the *esb* protocol allows one Prova agent to send a message to another Prova agent on a local or remote machine. For more details about using an ESB as the communication middleware, see Section 6.5. In Prova, all *task*, *async*, and *esb* protocols can be used to implement parallel and concurrent processes.

The following Prova example shows two tasks are executed in parallel.

Listing 6.5: Parallel Task Implemented in Prova

```
1  split_process(XID) :-
2      fork_a_b(XID).

4  fork_a_b(XID) :-
5      % Task a
6      sendMsg(XID,esb,agent1,request,a(Ins, Outs)),
7      rcvMsg(XID,esb,agent1,answer, a(Ins, Outs)).
8  fork_a_b(XID) :-
9      % Task b
10     sendMsg(XID,esb,agent2,request, b(Ins, Outs)),
11     rcvMsg(XID,esb,agent2,answer, b(Ins, Outs)).
```

In the above example, tasks *a* and *b* are started with sending messages sent to *agent1* and *agent2*, respectively. The Prova event messaging predicate *rcvMsg* (Line 7) does not block the current thread but waits for the task results asynchronously, while keeping all current data in a transparent created closure. In other words, task *b* can be immediately started after task *a* is started without waiting for the completion of task *a*. A concurrent process can also be implemented in a similar way, and an example will be found in Section 7.4.3, which presents the implementation of the ant identification and treatment involving interactions between distributed agents.

The event-driven computation of complex event patterns is also implemented by Prova reactive messaging. In Prolog, the updates are not undone during backtracking. For instance, a Prolog rule "p :- $assert(event(a))$, $fail$." asserts *event(a)* into the knowledge base even though the prove of p fails. Although it is a limitation to implement some CTR-based applications that require undoing the updates during backtracking, for the event-driven computation of complex event patterns (see Section 5.5), this is useful since occurred base events need to be stored even if a complex event pattern is not matched. Based on it, two CTR rules used to detect a base event can be implemented by one Prova rule. For example, the following Prova rules implement the conjunction of events a, b and c:

Listing 6.6: Complex Event Pattern Computation Implemented in Prova

```
1  detect(XID) :-
2      rcvMsg(XID,async,From,inform, e(a)), assert(e(a)), check(XID, ce1).
3  detect(XID) :-
4      rcvMsg(XID,async,From,inform, e(b)), assert(e(b)), check(XID, ce1).
5  detect(XID) :-
6      rcvMsg(XID,async,From,inform, e(c)), assert(e(c)), check(XID, ce1).

8  check(XID, ce1) :-
9      e(a), e(b), e(c), sendMsg(XID, async, 0, inform, ce1),
10     retract(e(a)), retract(e(b)), retract(e(c)).
```

Here, *assert(e(a))*, *assert(e(b))* and *assert(e(c))* implement the semantics of receiving messages by recording the base events into the knowledge base. They are removed from the knowledge base after the complex event *ce1* occurs (Line 10). Note that events a, b and c are simplified for clarity.

Similar to the implementation of the event-driven computation of event patterns, Prova itself is also capable of grouping more than one *inline reaction* using a logical operator and allowing detected composite events to be further processed. In other words, an *exit* channel intercepts internal messages sent by multiple event channels when a complex event pattern is successfully detected. Prova provides two logical reaction groups: *@and*, which requires *all* event channels to be successfully proved, and *@or*, which requires *either* of the event channels to be successfully proved. For example, the above example can also be implemented by the Prova logical reaction group: *@and*:

Listing 6.7: Example of Prova Reaction Group

```
1  split_process(XID) :-
2      detect(XID).

4  detect(XID) :-
5      % Task a
6      @group(g1)
7      rcvMsg(XID,async,From,inform, e(a)).
8  detect(XID) :-
9      @group(g1)
10     rcvMsg(XID,async,From,inform, e(b)).
11 detect(XID) :-
12     @group(g1)
13     rcvMsg(XID,async,From,inform, e(c)).
```

```
15 detect(XID) :-
16     @and(g1)
17     rcvMsg(XID,esb,From,and,Events),
18     println(["The evens a, b, and c are detected: ",Events," "]).
```

However, Prova does not support flexible joins that are used to implement more advanced workflow patterns directly, such as partial joins. Based on the semantics of the event-driven computation of complex event patterns (see Section 5.5), a partial join can be implemented in a similar way with the example mentioned in Listing 6.6. The following example shows that a complex event *ce* occurs when any two of base events *a*, *b* and *c* are detected. Whenever a base event is detected, the event is recorded into the knowledge base, and then the evaluation of complex event detection is started.

Listing 6.8: Example of Partial Join Implemented in Prova

```
1 partial_join_process(XID) :-
2     detect(XID),
3     println(["Two of the events a, b and c are detected."]).

5 detect(XID) :-
6     rcvMsg(XID,async,From,inform, e(a)), assert(e(a)), checkCE(XID).
7 detect(XID) :-
8     rcvMsg(XID,async,From,inform, e(b)), assert(e(b)), checkCE(XID).
9 detect(XID) :-
10    rcvMsg(XID,async,From,inform, e(c)), assert(e(c)), checkCE(XID).

12 checkCE(XID) :-
13     e(a), e(b), sendMsg(XID, async, 0, inform, ce),
14     retract(e(a)), retract(e(b))).

16 checkCE(XID) :-
17     e(a), e(c), sendMsg(XID, async, 0, inform, ce),
18     retract(e(a)), retract(e(c)).

20 checkCE(XID) :-
21     e(b), e(c), sendMsg(XID, async, 0, inform, ce),
22     retract(e(b)), retract(e(c)).
```

In this example, three rules are used to detect two of three base events. However, the rules used to detect a partial join grow rapidly as the number of base events increases (see Section 5.5).

To sum up, the translation from the CTR-based workflow logic to Prova is shown in Table 6.1. It is worth noticing that the translation does not strictly follow the semantics of the CTR-based workflow logic to maintain workflow state updates, although it is possible to accommodate it by updating the knowledge base. This is because, in the RbAF, event messages passing between agents are associated with a conversation identifier to reflect the states of workflow execution, i.e., event messages are always local to specific conversation states. Moreover, this solution reduces domain experts' efforts to program state changes manually and also brings the flexibility by implementing them if necessary.

For the purpose of seamlessly connecting distributed Prova agents, enabling them to exchange data, ESB Mule [175] is employed as the communication middleware.

For more details, see Section 6.5.

Table 6.1: Mapping CTR-Based Workflow Representation to Prova Rules

Workflow Element	CTR-based Workflow Representation	Prova
Workflow	CTR Horn goal	Prova goal
Sub-process	CTR Horn rule	Prova rule
Sequential task	Serial conjunction ($\otimes$)	Serial subgoals of a Prova rule
Nondeterministic task	Classic disjunction ($\vee$)	Prova rules with conditions
Concurrent task	Concurrent conjunction ($\mid$)	Concurrent reactive messaging
Task dependency	Messaging activities	Reactive Messaging
Complex gateway (Join connector)	The rule-based CEP	Event-driven computation of event patterns

6.4 Domain Logic Expression in Prova

As an expressive rule language, Prova is capable of backward-reasoning logic programming to formalize decision logic in terms of derivation rules. In particular, with the combination of forward-directed messaging reaction rules, it is possible to employ distributed agents to prove derivation rules (see Listing 6.3 as an example, which uses distributed agents to identify newly discovered ants). In addition, the RbAF provides different ways of accessing domain-specific data encoded by Semantic Web technologies and their implementation is presented in this section.

Reusing and integrating existing domain-specific data can avoid redundancy in the knowledge base of Prova agents and improve the flexibility and accuracy of domain knowledge representation. For example, Figure 6.2 shows *up:Protein* and *up:Concept* defined in UniProt core vocabulary [130], and they are integrated in the process of protein prediction result analysis to express the process inputs. According to the vocabulary, an agent also can know other related information about a protein, such as organism, mnemonic.

Following the spirit of the W3C Semantic Web initiative, Prova provides a tight combination with Semantic Web technologies to integrate external Semantic Web data. In order to query RDF data, Prova provides SPARQL operators based on OpenRDF Sesame [176], which is an open source Java framework for storage and querying of RDF data. Sesame offers an easy-to use API that supports the leverage of RDF data in applications. In particular, it can run in a standalone server mode with multiple applications connecting to it. Based on the tight integration with Java, Prova embeds Sesame Java API into declarative rules and outsources storage

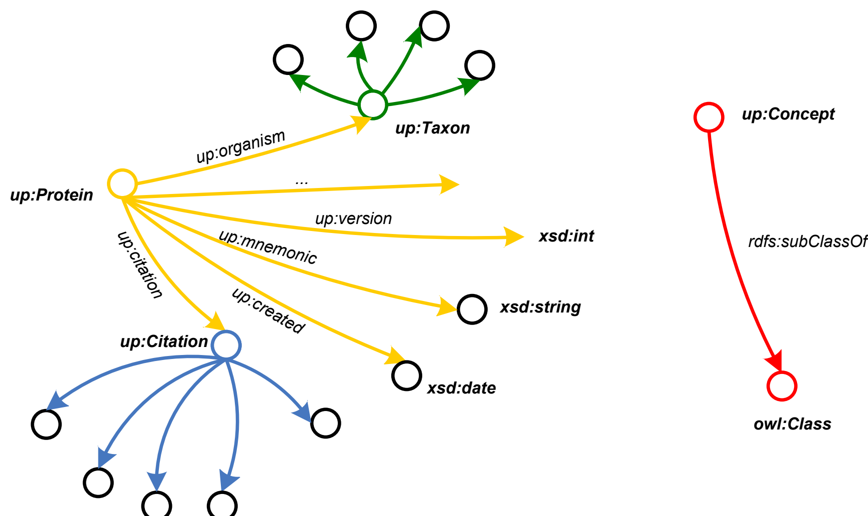


Figure 6.2: Domain Ontology: UniProt Core Vocabulary

and querying of RDF data to a Sesame server. Such solution not only reduces the complexity of Prova, but also improves the flexibility of SPARQL querying. The Sesame employed by the RAWLS is 2.7.9 published in December of 2013, which implements the SPARQL 1.1 specification.

There are four Prova built-in predicates that encapsulate the Sesame API to query RDF data: *sparql_connect/2*, *sparql_select/3*, *sparql_ask/3* and *sparql_disconnect/1*. The following Prova example shows a query of finding all available agents from the workflow ontology stored in a Sesame repository at: `http://localhost:8080/openrdf-sesame/repositories/sswf` (Line 2). The predicates *sparql_connect/2* (Line 3) and *sparql_disconnect/1* (Line 5) are used to connect and disconnect the repository, respectively. The predicate *sparql_select/3* (Line 16) performs the query and stores the results as the facts of the predicate *sparql_results* (Line 17), which are identified by an identifier *queryId*.

Listing 6.9: SPARQL Query in Prova

```
1 availableAgents(Agent):-
2      URL = "http://localhost:8080/openrdf-sesame/repositories/sswf",
3      sparql_connect(Connection, URL),
4      queryAvailableAgents(Connection),
5      sparql_disconnect(Connection).

7 queryAvailableAgents(Connection) :-
8     QueryString = '
9         PREFIX : <http://www.corporate-semantic-web.de/sswf2013#>
10        PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
11        SELECT ?agent  WHERE {
12            ?agent a :Agent .
13            ?agent :available "false"^^xsd:boolean .
14        }
15    ',
16    sparql_select(Connection, QueryString, queryId),
17    sparql_results(queryId, Agent),
```

```
18     fail().

20 queryAvailableAgents(_).
```

Besides the SPARQL query, the RAWLS also extends Prova to implement SPARQL 1.1 Update [177], which is an extension to the SPARQL query language. The Sesame update API is encapsulated into Prova built-in predicate *sparql_update/4*. SPARQL update can be used to update the status of Prova agents during the workflow execution. More details can be found in Section 6.6.

For the ontology reasoning, the RAWLS extends Prova to incorporate SPARQL-DL query engine [156]. The SPARQL-DL query language is a subset of SPARQL and is explicitly tailored to ontology-specific requests related to OWL; it uses SPARQL syntax and is more expressive than existing DL query languages by allowing mixed TBox, RBox, and ABox queries [178]. Figure 6.3 shows the architecture of the SPARQL-DL query engine adapted from *derivo GmbH* [156].

The SPARQL-DL query engine is settled on top of the OWL API. The latest implementation of SPARQL-DL API (1.0.0) developed by *derivo GmbH* in 2011 is fully compatible with OWL 2 and can be regarded as an interface to every ontology reasoner supporting OWL API [156]. In this thesis, the RAWLS employs HermiT [155], which is a Java-based OWL reasoner, to be a real reasoner behind the SPARQL-DL query engine to reason domain ontologies.

Figure 6.3: SPARQL-DL Query Engine

Similar to the implementation of SPARQL queries, the RAWLS extends Prova to provide similar built-in predicates to use the SPARQL-DL API: *sparqldl_create/2*, *sparqldl_select/3* and *sparqldl_ask/3*. For example, in the use case of protein prediction result analysis mentioned in Section 2.5.2, the prediction of a protein is considered correct if the protein has some reliable GO terms that lie on a path in the gene ontology tree from the root to a leaf that visits the predicted GO term [47]. For each GO term, the Gene Ontology Consortium provides an ontology to describe it and its relationships with other terms. Therefore, the protein prediction result analysis can be converted into a question: if the predicted GO term is the subclass of any reliable GO term of the protein. The question can be represented as a SPARQL-DL query as follows (Line 5-10).

Listing 6.10: SPARQL-DL Query in Prova

```
1 analysis(ReliableGOTerms, PredictedGOTerm, Result):-
2    Onto = de.fub.csw.protein.prediction.DataProcessor.getOnto(PredictedGOTerm),
3    sparqldl_create(Engine, Onto),
4    element(GOTerm, ReliableGOTerms),
```

```
5     QueryString = '
6        PREFIX : <http://www.geneontology.org/go#>
7        ASK {
8           SubClassOf(:$PredictedGOTerm, :$ReliableGOTerm)
9        }
10    ',
11    sparqldl_ask(Engine, QueryString, queryId),
12    Result = "yes",
13    !.

15 analysis(ReliableGOTerms, PredictedGOTerm, Result):-
16    Result = "no".

18 askQuery(Engine, QueryString, Result):-
19    sparqldl_ask(Engine, QueryString, queryId),
20    sparqldl_results(queryId),
21    Result = "yes",
22    !.

24 askQuery(Engine, QueryString, Result):-
25    Result = "no".
```

The SPARQL-DL query "SubClassOf(:$PredictedGOTerm, :$ReliableGOTerm)" asks whether class *PredictedGOTerm* is the subclass of *ReliableGOTerm* (Line 8). It is a TBox query and returns *true* if *PredictedGOTerm* is a child of *ReliableGOTerm* or equivalent to *ReliableGOTerm* in the hierarchy tree. Note that Prova can not directly concatenate strings with variables by "+" operator, the variables *Predicted-GOTerm* and *ReliableGOTerm* are attached with "$" at the beginning for clarity. In practice, Prova uses its builtin *concat/2(i, io)* to concatenate two or more strings [173].

In addition, Prova has a typed logic approach which allows using external ontologies as a type system of the rules. In other words, Prova variables can be typed with concepts defined in external ontologies. The RAWLS implements the integration of external ontologies as typed rules by the SPARQL-DL query engine. As shown in Listing 6.9, the variable *Agent* is initiated to all available agents defined in the workflow ontology. Based on the query identifier *queryId*, the available agents can be accessed everywhere in the knowledge base.

6.5 Enterprise Service Bus Mule

Mule [175] is a lightweight Java-based ESB and integration platform that allows developers to connect applications together quickly and easily, enabling them to exchange data. As an ESB, Mule acts as a transit system for carrying data between applications and enables easy integration of heterogeneous systems. In addition, Mule ESB has features, like flexible service mediation by separating business logic from protocols and message formats, message routing based on content or complex rules, heterogeneous data transformation, synchronous and asynchronous event handling, lightweight service orchestration, and service creation and hosting [175].

The default model used by Mule to process a request is based on a Staged Event-Driven Architecture (SEDA) [179], which decomposes a complex application into a

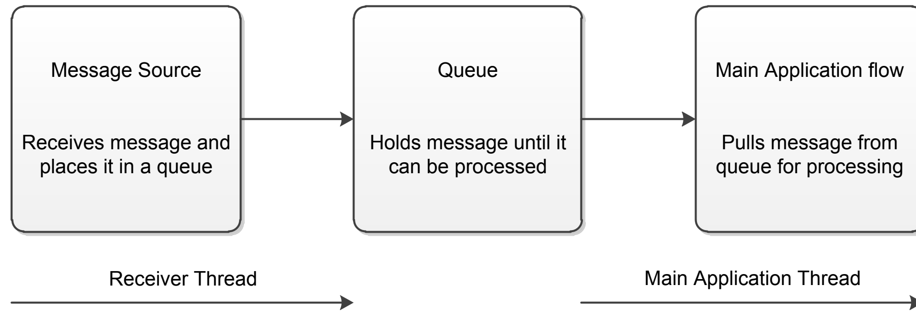


Figure 6.4: Mule Application Flow

set of stages connected by queues. Figure 6.4 adapted from [175] gives the process of a Mule application at the simplest level: a Mule application accepts one request(i.e., a message) at a time, processing received messages in the order they are received. A *message source* is often the first building block of a flow, which receives messages from one or more external sources, thus triggering a flow instance. The messages can be from multiple transport channels (such as HTTP, Java Message Service (JMS)). Incoming messages are placed into a *queue*, which holds messages until they can be processed by an *application flow*. A message is processed when it reaches the head of the queue. Typically an *inbound endpoint* serves as a message source, and it waits for messages from external sources and passes them to the rest of the flow. Also, an *outbound endpoint* is often the last building block and passes data out of a flow. A Mule application can return processing results to the source of original message (aka. a request-response pattern) or other third parties (aka. a one-way exchange pattern in which a message processor does not provide any response to the original sender), thereby forming complex Mule applications. Besides the endpoints, other message processors such as message filters, data transformers, components and flow controls can also be placed into Mule application flows to process messages [175].

Mule ESB has two versions: Community and Enterprise. Mule ESB Enterprise is the enterprise-class version of Mule ESB, with additional features and capabilities that are ideal for production deployments that have advanced requirements for performance, high availability, resiliency, or technical support. Mule ESB Community is developed under Common Public Attribution License (CPAL) [175] and is free for use in development and pre-production. In this thesis, the RAWLS employs Mule ESB Community to connect distributed Prova agents.

6.5.1 Prova Agent Deployment

As an application integration platform, Mule ESB allows deploying Prova agents as distributed rule inference services and enables their coordination and cooperation. However, for the purpose of providing an expressive, flexible workflow specification, the interactions between Prova agents in this RAWLS are actually specified by declarative rules rather than the control flows provided by Mule. In this RAWLS,

each Prova agent processes external events by a Mule flow (see Figure 6.4). As a concrete example, the following code defines the event processing flow of a Prova agent, which acts as a workflow engine.

Listing 6.11: Prova Event Processing Flow

```
<jms:activemq-connector name="jmsConnector"
                specification="1.1" brokerURL="vm://localhost" />

<jms:endpoint name="semantic_SWF_Engine" topic="ee"
  connector-ref="jmsConnector" />
<flow name="Worklfow">
  <jms:inbound-endpoint ref="semantic_SWF_Engine">
    <properties><spring:entry key="rulebase"
      value="${app.home}/rules/semantic_swf_engine/semantic_swf_engine.prova"/>
    </properties>
  </jms:inbound-endpoint>
  <component class="ws.prova.mule.impl.ProvaUMOImpl" />
</flow>
```

The message source of the flow is implemented as a global JMS endpoint *semantic_SWF_Engine* that can handle all messages arrived at this endpoint (Line 4-5). The endpoint is associated with the path of a Prova rule base, as shown in its properties (Line 8-10). The second building block of the flow is a customized Java component (ws.prova.mule.impl.ProvaUMOImpl) (Line 12) that manages the life cycle of a Prova agent. The Java component not only initializes the Prova agent, but also implements both the logic of adding messages received on the source JMS endpoints to Prova agent communicator queue and the action of dispatching messages to other Prova agents.

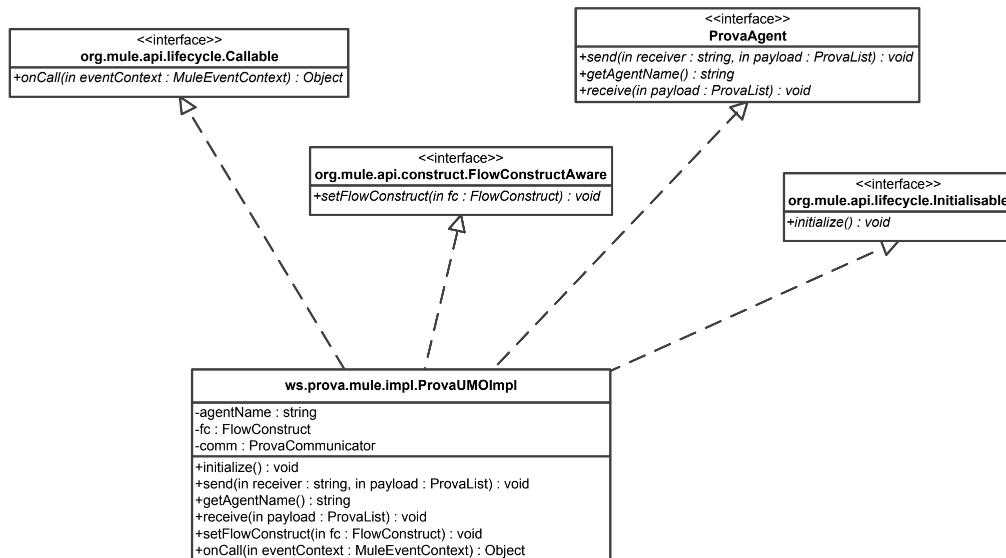


Figure 6.5: Class Diagram of ProvaUMOImpl

The Java component implements four interfaces to manage the Prova agent life cycle, as shown in Figure 6.5. The *initialize()* method is executed when the Java

component is called to initialize a Prova agent. It is customized to create an instance of *ProvaCommunicatorImpl*, which manages the communicator queue of the Prova agent. The Prova agent has a name and a rule base that specifies its behavior. They are specified in the properties of the source JMS endpoint and can be used by the Java component to initialize the Prova agent. Obtaining the properties of the rule base of a Prova agent endpoint can be done by the *setFlowConstruct(FlowConstruct fc)* method of the interface *FlowConstructAware*. The *onCall(MuleEventContext eventContext)* method of the interface *Callable* implements the logic of processing messages arriving at the source JMS endpoint. Each message passed between Prova agents is in the form of a Prova list—a compound term for holding and manipulating groups of objects [173]. In order to enable Prova to process a message, the related translations are firstly done in this method if necessary (see Section 6.5.3 for more details). Afterwards, the message is added to the Prova agent communicator queue to process. Another interface implemented by the Java component is *ProvaAgent*, whose method *send(String receiver, ProvaList payload)* is called when the Prova agent sends a message to another Prova agent. The *send* method uses a Mule client to send messages programmatically. An infinite loop detector and the translation from the Prova list to HTML (if a message will be sent to human users) are also implemented in this method, see Sections 6.5.3 and 6.6 for more details, respectively.

6.5.2 Mule ESB as Communication Middleware

Mule ESB also supports the SOA paradigm, and it connects applications and acts the same way as the classical workflow languages (see Section 3.1) from the workflow composition perspective. In the RAWLS, Mule ESB is mainly used as an integration platform for connecting distributed Prova agents. The interactions between Prova agents are controlled by declarative rules, rather than the Mule control flows that specify message routing among message processors. Figure 6.6 shows the Mule-based system architecture of the RAWLS.

Prova agents deployed on Mule ESB are distributed inference services, each of which runs a local rule base that implements reaction and decision logic of the agent. Such agents react to incoming event messages in terms of the reaction rule-based workflow logic and also provide access to external applications and data sources, such as Web services, Java object representations, Semantic Web data, relational databases. In the RAWLS, the JMS transport protocol is used for the communication between distributed Prova agents, and Apache ActiveMQ [180], which is an open source message broker, is used to manage JMS messages.

The RAWLS employs a Web-based user client to manage workflow applications. Users can inspect available workflows and invoke a workflow by sending a request to the workflow engine. The user client also hosts the HA which allows human users to perform human tasks and deal with exceptions via the communication with a HA proxy and the EHA, respectively. The HA proxy is also a Prova agent that acts as an intermediary between human task requesters and the HA. More details about the HA proxy can be found in Section 6.7.3. The *workflow engine*, the *human agent*

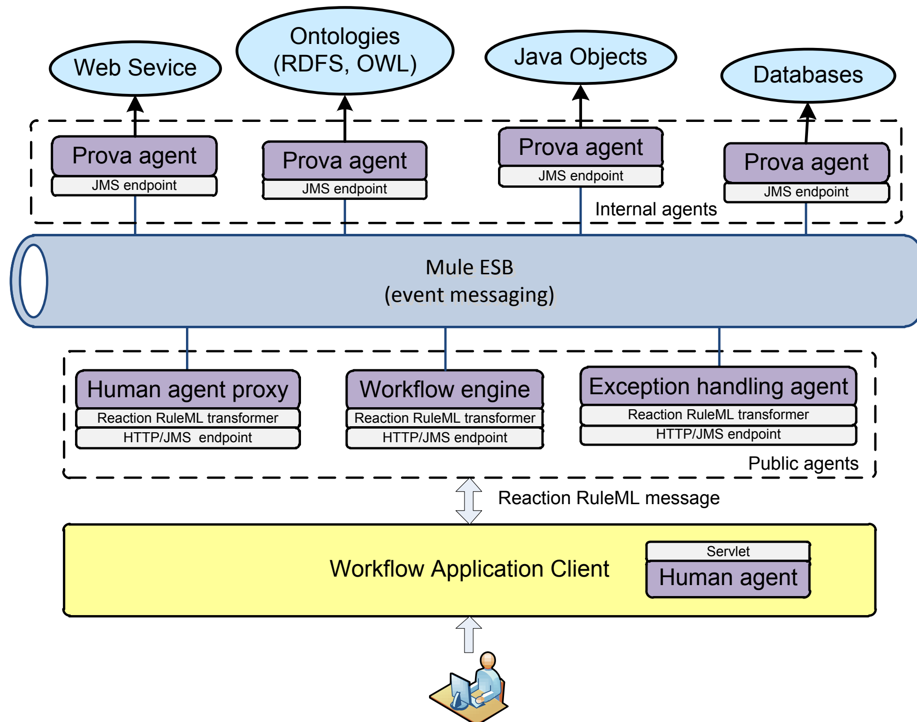


Figure 6.6: Mule-Based Workflow Architecture

(HA) proxy and the *EHA* are the agents that can communicate with the user client. Besides a JMS endpoint to communicate with other Prova agents, such agents also have an HTTP endpoint to interact with the user client. Therefore, they are also known as *public agents*, and the other agents are called *internal agents*, as shown in Figure 6.6.

6.5.3 Translations between Reaction RuleML and Prova

To communicate with the external user client, the RAWLS employs Reaction RuleML, the current de-facto standard language for reactive Web rules, to represent messages passing between the public agents and the user client. Reaction RuleML [181] is a general, practical, compact and user-friendly XML-serialized language and rule interchange format for the family of reaction rules [182, 137]. It specifies knowledge in a way that is understandable to non-computer domain experts. For the communication between distributed rule-based (agent) systems, Reaction RuleML provides a general message syntax:

```
<Message directive="<!-- pragmatic context -->">
  <oid>           <!-- conversation ID-->            </oid>
  <protocol>      <!-- transport protocol -->     </protocol>
  <sender>        <!-- sender agent/service -->     </sender>
  <receiver>      <!-- receiver agent/service --> </receiver>
  <content>       <!-- message payload -->         </content>
```

```
</Message>
```

In the context of these Reaction RuleML messages, agents can interchange events (e.g., queries and answers) as well as complete rule bases (rule set modules), e.g. for remote parallel task processing [182]. The agents can be engaged in long running asynchronous conversations and nested sub-conversations and use a *conversation identifier* to manage the conversation state. The *protocol* is used to define the message passing and coordination protocol. The *directive* attribute corresponds to the pragmatic instruction, i.e., the pragmatic characterization of the message context characterizing the meaning of the message, e.g., the FIPA ACL primitive [133]. The following example shows a request (directive="query") to invoke the process of protein prediction result analysis. It describes the *sender* (i.e., *User*) and the *receiver* (i.e., *semantic_SWF_Engine*) of the message and the message *payload* that specifies two inputs of the workflow: *Q9VAN0* and *GO:0006564*. The inputs are described as constants, which are embedded within the element <Ind></Ind>. The output of the workflow is represented by a variable *Result*, which is embedded within the element <Var></Var>. More details about Reaction RuleML see [182, 137].

Listing 6.12: A Workflow Request in Reaction RuleML

```
1  <RuleML xmlns="http://ruleml.org/spec">
2  <Message mode="outbound" directive="query">
3     <oid><Var>XID</Var></oid>
4     <protocol><Ind>esb</Ind></protocol>
5     <sender><Ind>User</Ind></sender>
6     <receiver><Ind>semantic_SWF_Engine</Ind></receiver>
7     <content>
8        <Atom>
9           <Rel>proteinPredicitonAnalysisProcess</Rel>
10             <Expr>
11                <Fun>inArgs</Fun>
12                <Ind type="java.lang.String">Q9VAN0</Ind>
13                <Ind type="java.lang.String">GO:0006564</Ind>
14             </Expr>
15             <Expr>
16                <Fun>outArgs</Fun>
17                <Var type="java.lang.String">Result</Var>
18             </Expr>
19       </Atom>
20    </content>
21 </Message>
22 </RuleML>
```

In the RAWLS, the messages passing between Prova agents are encapsulated as a compound term called Prova list. For example, the above workflow request can be represented in the form of Prova list as:

```
[httpEndpoint:1,esb,httpEndpoint,query,[proteinPredicitonAnalysisProcess,
     [inArgs,Q9VAN0,GO:0006564],[outArgs,Result]]].
```

To perform translations between Reaction RuleML and Prova messages, the RAWLS provides two translator services. The translation from the Prova message (Prova list) to Reaction RuleML is performed before sending the message to the

user client, i.e., in the method *send(String receiver, ProvaList payload)* (see Section 6.5.1). It is done by serializing simple terms (variables and constants) and complex terms (lists) of a Prova list [183]. The opposite translation is performed after a public Prova agent receives a request from the user client, i.e., in the method *onCall(MuleEventContext context)* (see Section 6.5.1). It is done via an XSLT sheet which translates Reaction RuleML to a Prova list [183].

6.6 Exception Handling

The RAWLS supports both *automatic* and *manual* exception handling. Their implementation is presented in Listing 6.13.

The rule-based languages have inherent advantages in specifying alternative exception paths by backtracking and CUT, which controls backtracking. For example, two *getResponsibleAgents* rules (Line 34-50) are used to find an available agent for a task. If the first rule is completed normally, the CUT (!) at the end of the rule inhibits backtracking to the second *getResponsibleAgents* rule. However, if no agent is found, the subgoal *sparql_results (queryID, FullAgentName)* cannot be evaluated to be true (Line 39). In this case, the second rule will be executed, and a *noAgentAvailable* exception occurs.

Similarly, the RAWLS specifies different execution paths to perform tasks. In the RAWLS, Prova agents are responsible for performing workflow tasks. The workflow engine allocates a task to an agent by sending a request to the agent and then waits for an acknowledgement sent back to it. If the agent is unavailable, the workflow engine generates an *unavailableAgent* exception and sends the exception to the EHA to deal with. The second *processMessage* rule (Line 8-11) shows a process to deal with the *unavailableAgent* exception. The process is started with updating the status of the exceptional agent to *false*. It is done by the Prova built-in predicate *sparql_update* which encapsulates the Sesame API (see Section 6.4) to update the workflow ontology stored in the Sesame repository (Line 24-31). The searching of an alternative agent is performed after the status of the exceptional agent is updated to *false*. However, if no agent is available, the *unavailableAgent* exception is escalated to a *noAgentAvailable* exception (Line 46-50), which is handled by sending the exception to the HA and asking human users for help (Line 14-21). In general, the *noAgentAvailable* exception is caused by lacking of available agents or wrong configurations. Human users can provide missing agents or correct wrong configurations and then notify the workflow engine to try to find a responsible agent again. More details about human interaction can be found in the following section. Similarly for the *failedAgent* exception, which occurs when an agent fails to perform a task (Line 2-5).

Listing 6.13: Exception Handling Implemented in Prova

```
1 % deal with the 'failedAgent' exception
2 processMessage(XID,From,Performative, failedAgent(TaskName, TaskID, Agent)):-
3      updateAgentAvailability(Agent),
4      getResponsibleAgents(XID, TaskID, TaskName, NewAgent),
```

```
5      sendMsg(XID, esb, From, "answer", [TaskID, NewAgent]).

7 % deal with the 'unavailableAgent' exception
8 processMessage(XID,From,Performative, unavailableAgent(TaskName, TaskID, Agent)):-
9      updateAgentAvailability(Agent),
10     getResponsibleAgents(XID, TaskID, TaskName, NewAgent),
11     sendMsg(XID, esb, From, "answer", [TaskID, NewAgent]).

13 % deal with the 'noAgentAvailable' exception
14 processMessage(XID,From,Performative, noAgentAvailable(TaskName, TaskID)):-
15     WorkflowName =  de.fub.csw.TaskManagementCenter.getWfName(XID),
16     sendMsg(XID, esb, humanAgent, "request",
17                     noAgentAvailable(WorkflowName, TaskName)),
18     rcvMsg(XID,esb, humanAgent, "answer", modified(ontology)),
19     println(["The workflow ontology has been updated by user."]),
20     getResponsibleAgents(XID, TaskID, TaskName, NewAgent),
21     sendMsg(XID, esb, From, "answer", [TaskID, NewAgent]).

23 % update the status of an agent
24 updateAgentAvailability(Agent):-
25     URL = de.fub.csw.constant.StringConstants.SEMANTIC_DATA_REPOSITORY_URL,
26     semanticDataConnection(URL, Connection),
27     UpdateString =  de.fub.csw.constant.StringConstants.updateAgent(Agent),
28     BASE_URL = de.fub.csw.constant.StringConstants.WF_ONTOLOGY_BASE_URL,
29     sparql_update(Connection, UpdateString, BASE_URL, updateID),
30     % remove the results from the knowledge base
31     retract(sparql_results(updateID)).

33 % find an available agent
34 getResponsibleAgents(XID, TaskID, TaskName, Agent) :-
35     URL = de.fub.csw.constant.StringConstants.SEMANTIC_DATA_REPOSITORY_URL,
36     semanticDataConnection(URL, Connection),
37     QueryString =  de.fub.csw.constant.StringConstants.queryAgentByTask(TaskName),
38     sparql_select(Connection, QueryString, queryID),
39     sparql_results(queryID, FullAgentName),
40     % remove the results from the knowledge base
41     retract(sparql_results(queryID,_)),
42     getLocalName(FullAgentName, Agent),
43     !.

45 % no agent is available
46 getResponsibleAgents(XID, TaskID, TaskName, Agent) :-
47     println(["Exception: no available agent for task: ", TaskName]),
48     sendMsg(XID,esb,exceptionHandlingAgent, "request",
49                     noAgentAvailable(TaskName, TaskID)),
50     rcvMsg(XID, esb, exceptionHandlingAgent, "answer", [TaskID, Agent]).
```

Another type of the expected exceptions is caused by endless communication between two or more agents, also known as an *infinite loop*. The RAWLS implements the communication between distributed Prova agents by passing messages and employs Mule ESB to connect such agents. In other words, all messages passing between the agents go through Mule ESB. By default, an infinite loop happens in the RAWLS if a message is repeatedly sent over 100 times in a workflow. A message counter is implemented in the method *send(String receiver, ProvaList payload)* of the Java component (see Section 6.5.1), which manages the life cycle of a Prova agent. In case an infinite loop is detected in a workflow, the workflow will be terminated and the exception will be sent to human users simultaneously. Note that the number of repeated messages used to detect an infinite loop can be configured

for specific workflow applications that have special requirements.

6.7 User Client

The RAWLS user client is a Web-based portal, which allows human users to inspect and invoke deployed workflows, perform human tasks, check exceptions occurred at workflow runtime and manage RDF data used in workflows. Moreover, it also hosts the HA, which manages the life cycle of the human tasks and the unexpected exceptions. The overall architecture of the RAWLS user client is shown in Figure 6.7.

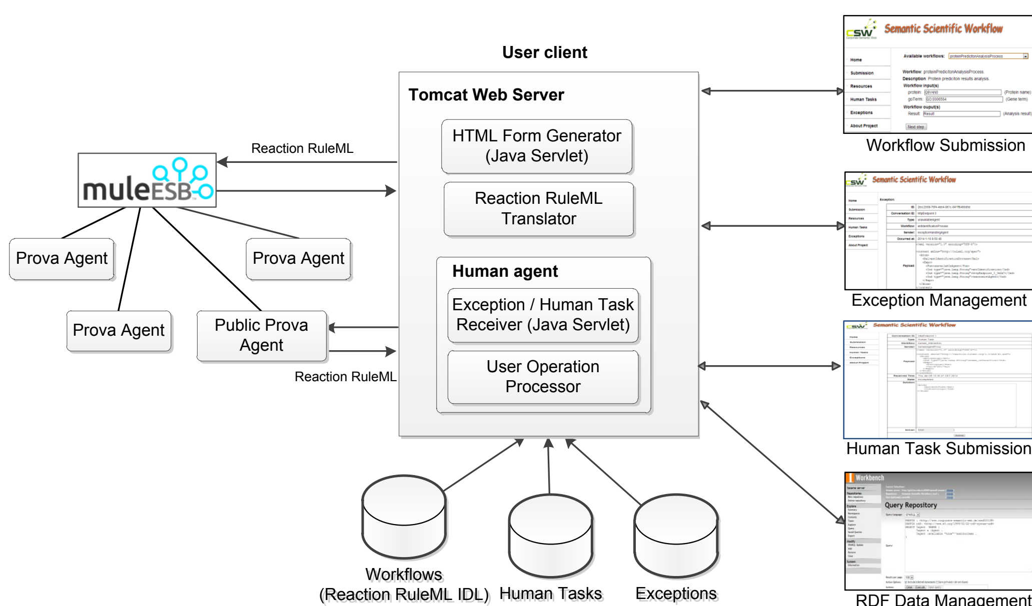


Figure 6.7: Human Interaction Client

6.7.1 Workflow Submission

The RAWLS employs Reaction RuleML Interface Description Language (IDL) [184], a sub-language of Reaction RuleML, to describe functional properties of deployed workflows. Reaction RuleML IDL describes the functions of available workflows together with their *term modes* and *type declarations*. The models are the declarations of intended input or output constellations of the predicate terms with the following semantics:

- "+" The term is intended to be input
- "−" The term is intended to be output
- "?" The term is undefined/arbitrary (input or output)

For example, the process of protein predication result analysis can be described by Reaction RuleML IDL as follows:

```
<signature>
   <label>
      <Expr>
         <Fun uri="dc:description"/>
            <Ind>Protein prediction results analysis.</Ind>
      </Expr>
   </label>
      <Fun per="value">proteinPredicitonAnalysisProcess</Fun>
         <Expr>
            <Fun meta="Workflow input(s)">inArgs</Fun>
            <Var mode="+" type = "java.lang.String" default="Q9VAN0">Protein</Var>
            <Var mode="+"  type = "java.lang.String" default="GO:0006564">GOTerm</Var>
         </Expr>
         <Expr>
            <Fun>outArgs</Fun>
            <Var mode="-" type="java://java.lang.String">Result</Var>
         </Expr>
</signature>
```

In the RAWLS user client, the deployed workflows are shown in a drop-down list by parsing a Reaction RuleML IDL document, which stores all deployed workflows, as shown in Figure 6.7. After users select a workflow from the drop-down list, an HTML form is generated in terms of the workflow description in Reaction RuleML IDL. The HTML form renders a user-friendly input form for creating a workflow request, making it easier to specify a workflow request. After human users submit the form with required values, a workflow request is sent to the workflow engine to invoke the workflow.

Note that the messages sent between the RAWLS and its user client are in a form of Reaction RuleML. In other words, a workflow request has to be packed in a Reaction RuleML message. Since Reaction RuleML IDL is a sub-language of Reaction RuleML, it can be easily translated into Reaction RuleML messages by replacing mode and type declarations with specific values given by human users.

6.7.2 Exception Management

Exceptions of the RAWLS can be either *expected* or *unexpected*. The expected exceptions are handled by the agents automatically. More precisely, the expected exceptions are *failedAgent*, *unavailableAgent* and *infiniteLoop* in this thesis. For example, the following *unavailableAgent* exception shows that agent *taxonomistAgent* responsible for task *antIdentification* is unavailable.

```
<RuleML>
 <Message mode="outbound" directive="inform">
   <oid><Ind>httpEndpoint:1</Ind></oid>
   <protocol><Ind>esb</Ind></protocol>
   <sender><Ind>exceptionHandlingAgent</Ind></sender>
   <receiver><Ind>humanAgent</Ind></receiver>
   <content>
      <Atom>
         <Rel>antIdentificationProcess</Rel>
```

```
          <Expr>
             <Fun>unavailableAgent</Fun>
             <Ind type="java.lang.String">antIdentification</Ind>
             <Ind type="java.lang.String">taxonomistAgent</Ind>
          </Expr>
      </Atom>
   </content>
</Message
</RuleML>
```

During the workflow execution, the EHA informs the HA to store such expected exceptions locally in Reaction RuleML and facilitates human users to fix them during the downtime. For the unexpected exceptions (e.g., no agent is available), they are often regarded as human tasks and manually handled by human users. More details can be found in the next section.

6.7.3 Human Task Management

In the RAWLS, the HA uses a Java servlet to receive messages from Mule ESB. The messages can be either the workflow exceptions or the human task requests. On one side, a Prova agent sends a human task request to the servlet and then waits for the task results from the HA. On the other side, the servlet stores the request into HA local repositories after receiving it.

The RAWLS also employs Reaction RuleML messages to specify human task requests and results. That is, the human task requests and results are transported in Reaction RuleML messages, and the translation between Reaction RuleML and the Prova message (Prova list) is performed before human task requests are sent and human task results are received, respectively, as shown in Section 6.5.3.

Like the asynchronous communication between distributed Prova agents, the RAWLS supports asynchronous interaction between human task requesters and the HA. That is, the Java servlet that receives human task requests is not responsible for sending the results back but stores the requests locally. Human users can inspect human task requests, provide solutions in Reaction RuleML and send them back (i.e., callback) immediately or later. The user operations are processed by a processor, which updates the task information (e.g., status, completion time, solver, solution, etc.) and then sends the results to the task requesters.

The activities of sending and receiving messages of human task requesters are processed by the threads taken from the conversation thread pool of Prova, which provides a thread according to the conversation identifier carried by the human task requests and results. After a human task request is sent, the requester agent preserves the current rule context, and when the results of a matching human task arrive, the requester resumes as though it had never been interrupted.

In the RbAF, human task requesters can be any Prova agent, and they do not interact with the HA directly, but via a HA proxy. The user client interacts with human task requesters via HTTP protocols, i.e., a requester (agent) sends a human task request to the servlet of the user client and then waits for the task results on its public HTTP endpoint. It means that each agent interacting with the user

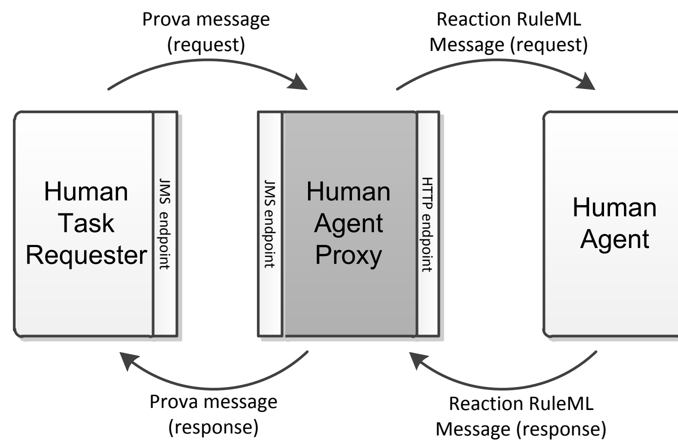


Figure 6.8: Human Agent Proxy

client should have a public HTTP endpoint. However, the number of public HTTP endpoints will grow rapidly as the number of the agents interacting with the user client increases. To deal with the problem, the HA proxy is proposed to act as an intermediary between human task requesters and the user client, as shown in Figure 6.8.

With the HA proxy, all human task requests sent by an internal Prova agent are first sent to the HA proxy, which then forwards the requests to the user client, and vice versa. The HA proxy has two inbound endpoints: JMS and HTTP. It receives the human task requests on the JMS endpoint from internal Prova agents, and waits for the task results from the HA on its public HTTP endpoint. In this way, only one public HTTP endpoint is used, and the user client only needs to know the HTTP endpoint address of the HA proxy.

6.7.4 RDF Data Management

Besides the Java API, Sesame also provides a browser client supporting RDF data management. To facilitate human users to manage RDF data, the Sesame browser client is integrated as a part of user client to manage the workflow ontology and other domain-specific ontologies, as shown in Figure 6.7. More details about the Sesame browser client can be found in [176].

6.8 Summary

This chapter presented a prototypical system, called RAWLS, to support the description and execution of the WsSWFs.

This thesis employs Prova, both a Semantic Web rule language and a distributed rule engine, to specify and execute the WsSWFs. Prova combines different rule types and is an expressive, hybrid, declarative and compact rule programming language.

This chapter first detailed the mapping from the CTR-based WsSWF representation to Prova rules, thereby implementing the reactive workflow logic. With the combination of messaging reaction rules and derivation rules, the RAWLS not only supports the implementation of distributed workflow logic, but also provides high expressive power for domain-specific decision logic. To integrate domain-specific data encoded by Semantic Web technologies, the RAWLS extends Prova to integrate OpenRDF Sesame Java API and outsources storage and querying of RDF data to an external Sesame server. In addition, the SPARQL-API query engine built on OWL API provides an expressive DL query language and allows ontology reasoners supporting OWL API to be easily configured in the RAWLS.

To support the interactions between Prova agents, Mule ESB is employed to seamlessly connect such distributed rule-based agents. The interactions between Prova agents are actually specified by declarative rules rather than the control flows provided by Mule. In addition, a Web-based client is provided to enable human users to invoke workflows and manage human tasks and workflow exceptions. The messages sent between the user client and internal Prova agents are represented by the de-facto standard reactive rule language Reaction RuleML. A detailed evaluation of the rule-based workflow specification and the efficiency of RAWLS will be presented in the next chapter.

Chapter 7

Evaluation

Contents

7.1 Workflow Pattern-Based Expressiveness Evaluation 129
- 7.1.1 Control-Flow Patterns 130
- 7.1.2 Data Patterns 143
- 7.1.3 Scientific Workflow Patterns 148

7.2 Evaluation of the Domain Knowledge Representation . . . 150
- 7.2.1 LP-based Knowledge Representation Evaluation 150
- 7.2.2 DL-based Knowledge Representation Evaluation 152

7.3 Computational Model-Based Empirical Evaluation 156

7.4 Use Case-Based Experimental Evaluation 158
- 7.4.1 Protein Prediction Result Analysis 159
- 7.4.2 Snow Depth Data Screening 163
- 7.4.3 Ant Identification and Treatment 165

7.5 System Performance Evaluation 169
- 7.5.1 Message Passing Overhead 169
- 7.5.2 System Concurrency 170

7.6 Summary 171

This chapter evaluates the RbAF and the RAWLS from different perspectives. Section 7.1 compares the rule-based workflow specification of the RAWLS with other three prominent SWFMSs in terms of both control-flow patterns and data patterns. Section 7.2 evaluates the expressive power and complexity of domain knowledge representation from both LP and DL perspectives. Section 7.3 evaluates the RAWLS in terms of typical properties of computational models. Section 7.4 experimentally analyzes the performance and demonstrates the expressive power of the domain knowledge representation in the RbAF with three real-world WsSWF use cases. Section 7.5 evaluates the performance of the RAWLS.

7.1 Workflow Pattern-Based Expressiveness Evaluation

Workflow patterns are recurrent solutions in the development of process-oriented applications. From different perspectives, the Workflow Patterns Initiative [52] have

delivered four types of workflow patterns related to the development of workflow applications, i.e., control flow patterns, data patterns, resource patterns and exception handling patterns. Such patterns are the formal ways of describing workflow recurrent solutions and provide a comprehensive benchmark to compare process modeling languages. In terms of the control-flow and data patterns, this section compares the rule-based workflow specification of the RAWLS with the workflow languages of other three prominent SWFMSs: Kepler, Taverna, and Triana to evaluate the level of solving different types of tasks. A brief introduction to the workflow languages of Kepler, Taverna, and Triana, see Section 3.4.

Note that the workflow patterns delivered by the Workflow Patterns Initiative are originally for business workflows and this section only considers the patterns that are important for scientific workflows. Additionally, this section considers four new scientific workflow patterns identified in [185].

7.1.1 Control-Flow Patterns

The Workflow Patterns Initiative [52] delivered original 20 Workflow Control-flow Patterns (WCPs) in 2003 [186]. In the latest release [136], the number of the control-flow patterns has been increased to 43 for the purpose of describing more advanced scenarios. To make a clear comparison, the numbering and definition of control-flow patterns in this section follows the Workflow Patterns Initiative.

This section does not consider the control-flow patterns that involve cancellation and force completion of a workflow activity (WCP-19, 20, 25, 26, 27, 29, 32, 35). This is because canceling a task (a process) or rolling back to original states is not supported in the RbAF. Such workflow patterns are also not supported by other three SWFMSs, i.e., Kepler, Taverna and Triana.

7.1.1.1 Basic Control-Flow Patterns

WCP-01 Sequence - A task in a process is enabled after the completion of a preceding task in the same process [136].

Like the process-oriented workflow languages, Prova implements a complex sequential task by a Prova rule, where each sub-task is interpreted as a sub-goal to evaluate the rule. Prova executes sequentially one sub-goal after another starting with the topmost one. In other words, all sub-goals of a Prova rule are executed sequentially. Moreover, based on messaging reaction rules, the sub-goals of a Prova rule can be executed by multiple distributed agents.

WCP-02 Parallel Split - A branch is diverged into two or more parallel branches, each of which execute concurrently [136].

Prova can implement a split connector by simply creating a predicate with multiple clauses (branches). Based on Prova reactive messaging, the tasks in different branches can be performed in parallel. See Listing 6.5 as an example.

WCP-03 Synchronization - The convergence of two or more branches into a single subsequent branch such that the thread of control is passed to the subsequent

branch when all input branches have been enabled [136].

The RAWLS can implement the synchronization pattern based on the event-driven computation of complex event patterns, see Listing 6.6 as an example. Prova also can implement this pattern by the logical reaction group: *@and*, which requires *all* event channels to be successfully proved. The following example shows a synchronization of two parallel threads of executing tasks *a* and *b*. The predicate *fork_ a_ b* creates three reactions. The first two reactions start tasks *a* and *b* by sending messages to agents *agent1* and *agent2*, respectively. Note that their sub-reactions (Line 8 and 13) are decorated with the *@group(g1)* and are concurrently waiting for the results from the agents *agent1* and *agent2*. The third *fork_ a_ b* reaction includes a composite sub-reaction corresponding to the *AND* join operator (*@and(g1)*, Line 15) to synchronize tasks *a* and *b*.

Listing 7.1: Synchronization Implemented in Prova

```
1  split_process(XID) :-
2      fork_a_b(XID).

4  fork_a_b(XID) :-
5      % Task a
6      sendMsg(XID,esb,agent1,request, a(Ins1, Outs1)),
7      @group(g1)
8      rcvMsg(XID,esb,agent1,answer, a(Ins1, Outs1)).
9  fork_a_b(XID) :-
10     % Task b
11     sendMsg(XID,esb,agent2,request, b(Ins2, Outs2)),
12     @group(g1)
13     rcvMsg(XID,esb,agent2,answer, b(Ins2, Outs2)).
14 fork_a_b(XID) :-
15     @and(g1)
16     rcvMsg(XID,esb,From,and, Events),
17     println(["Tasks a and b are completed: ", Events, " "]).
```

WCP-04 Exclusive Choice - The divergence of a branch into two or more branches such that when the incoming branch is enabled, the thread of control is immediately passed to precisely one of the outgoing branches based on a mechanism that can select one of the outgoing branches [136].

Prova specifies multiple branches by simply creating a predicate with a set of clauses. However, when it implements the *Exclusive Choice* pattern, each clause needs to specify a precondition as its first sub-goal to enable a branch. Moreover, Prova needs to attach a CUT to each branch to ensure that only one branch is enabled. The following example shows a process of executing task *a* or *b* exclusively. Prova reaches a CUT (!) if X =< Y succeeds, and the CUT(!) inhibits backtracking to the second rule. But if X =< Y fails, then Prova goes onto the second rule instead.

Note that this example only defines a simple Boolean expression to select subsequent branches as the classic workflow languages. The rule-based workflow description of the RAWLS not only describes multiple alternative branches, but also provides high expressive power to describe complex domain-specific decision logic in terms of derivation rules (see Section 4.4).

Listing 7.2: Exclusive Choice and Simple Merge Implemented in Prova

```
1 split_process(X,Y,XID) :-
2     fork_a_b(X,Y,XID),
3     println(["Two exclusive branches are merged here"]).

5 fork_a_b(X,Y,XID) :-
6     % Task a
7     X =< Y,
8     !,
9     execute(a).
10 fork_a_b(X,Y,XID) :-
11     % Task b
12     X>Y,
13     !,
14     execute(b).
```

WCP-05 Simple Merge - The convergence of two or more branches into a single subsequent branch such that each enablement of an incoming branch results in the thread of control being passed to the subsequent branch [136].

The *Simple Merge* pattern defines a merge of two or more branches without synchronization. Moreover, it assumes that such branches are mutually exclusive. In Prova, multiple branches created by a predicate are automatically merged after the predicate, as shown in the above example of the *Exclusive Choice* pattern (Line 3).

7.1.1.2 Advanced Branching and Synchronization Patterns

WCP-06 Multi-Choice - The divergence of a branch into two or more branches such that when the incoming branch is enabled, the thread of control is immediately passed to one or more of the outgoing branches based on a mechanism that selects one or more outgoing branches [136].

As aforementioned, Prova can specify multiple branches by simply creating a predicate with a set of clauses. To implement the *Multi-Choice* pattern, each clause has to employ a precondition as its first sub-goal. If the condition is satisfied, the corresponding branch is enabled. For example, the predicate *fork_ a_ b_ c* in following example triggers tasks *a*, *b* and *c* on the basis of evaluating conditions *condition_ 1*, *condition_ 2* and *condition_ 3*. Moreover, unlike Kepler, Triana and Taverna, Prova can represent a default branch by a *not* operator, which denotes NaF. In this example, there are two *split_process* rules, which include a pair of exclusive conditions represented by a *not* operator. If none of *fork_ a_ b_ c* clause is proved true (i.e., the first *split_process* rule cannot be proved), the second *split_process* rule specifying a default path will be executed.

Listing 7.3: Multi Choice and Structured Synchronizing Merge Implemented in Prova

```
1 split_process(X,Y,XID) :-
2     fork_a_b_c(X,Y,XID).

4 split_process(X,Y,XID) :-
5     not(fork_a_b_c(X,Y,XID)),
```

```
6      println(["The default path is enabled."]).

8  fork_a_b_c(X,Y,XID) :-
9      % Task a
10     condition_1(X,Y),
11     sendMsg(XID,esb,agent1,request,a(Ins1, Outs1)),
12     @group(g1)
13     rcvMsg(XID,esb,agent1,answer,a(Ins1, Outs1)).
14 fork_a_b_c(X,Y,XID) :-
15     % Task b
16     condition_2(X,Y),
17     sendMsg(XID,esb,agent2,request,b(Ins2, Outs2)),
18     @group(g1)
19     rcvMsg(XID,esb,agent2,answer,b(Ins2, Outs2)).
20 fork_a_b_c(X,Y,XID) :-
21     % Task c
22     condition_3(X,Y),
23     sendMsg(XID,esb,agent3,request,c(Ins3, Outs3)),
24     @group(g1)
25     rcvMsg(XID,esb,agent3,answer,c(Ins3, Outs3)).
26 fork_a_b_c(X,Y,XID) :-
27     @and(g1) @timeout(10000)
28     rcvMsg(XID,esb,From,and,Events),
29     println(["The merged branches are: ", Events, " "]).
```

WCP-07 Structured Synchronizing Merge - Unlike the *Synchronizing* pattern (WCP-03), the *Structured Synchronizing Merge* occurs in a structured context, i.e. there must be a single *Multi-Choice* construct earlier in the process model with which the *Structured Synchronizing Merge* is associated, and it must merge all of the branches emanating from *Multi-Choice* [136].

Both the *Synchronization* pattern (WCP-03) and this pattern synchronize multiple branches into a single subsequent branch. The difference is that the former requires all branches have to complete, no matter they have been activated or not. But the latter one only synchronizes the branches that have been activated, and it means that the join connector of this pattern must be aware of which branches are active. Prova implements this pattern by attaching a timeout to the composite reaction corresponding to the *AND* operator. As shown in the above example, the join waits for a pre-specified time (ten seconds) and then consumes all events detected so far in its payload (Line 29-31).

WCP-08 Multiple Merge - The convergence of two or more branches into a single subsequent branch such that each enablement of an incoming branch results in the thread of control being passed to the subsequent branch [136].

The difference between *Simple Merge* (WCP-05) and *Multiple Merge* is that in the former only one incoming branch is active, but there may be more than one incoming branch being active simultaneously in the latter. Each enablement of an active incoming branch causes the execution of the subsequent branch. Like the implementation of *Simple Merge*, *Multiple Merge* can also be implemented in the same way.

WCP-09 Structured Discriminator - The convergence of two or more branches into a single subsequent branch following a corresponding divergence earlier in the process model such that the thread of control is passed to the subsequent branch

when the first incoming branch has been enabled. Subsequent enablements of incoming branches do not result in the thread of control being passed on [136].

Prova implements the *Synchronization* pattern by the logical reaction group: *@or*, which requires *either* of the event channels to be successfully proved. The following example shows a process which waits for the results of either task *a* or *b*. The predicate *fork_a_b* creates three reactions. The first two reactions start tasks *a* and *b* by sending the task requests to agents *agent1* and *agent2*, respectively. Note that the sub-reactions of first two *fork_a_b* rules (Line 8 and 13) are decorated with the *@group(g1)*. They are concurrently waiting for the results from agents *agent1* and *agent2*. Unlike the *Synchronization* pattern (WCP-03), the third *fork_a_b* rule is a reaction corresponding to the *non-local XOR* join operator (*@or(g1)*, Line 15) to wait for the results of either task *a* or *b*. When either of them is completed, the whole group terminates, and the other task execution is ignored.

Listing 7.4: Structured Discriminator Implemented in Prova

```
1 split_process(XID) :-
2     fork_a_b(XID).

4 fork_a_b(XID) :-
5     % Task a
6     sendMsg(XID,esb,agent1,request,a(Ins1, Outs1)),
7     @group(g1)
8     rcvMsg(XID,esb,agent1,answer, a(Ins1, Outs1)).
9 fork_a_b(XID) :-
10    % Task b
11    sendMsg(XID,esb,agent2,request, b(Ins2, Outs2)),
12    @group(g1)
13    rcvMsg(XID,esb,agent2,answer, b(Ins2, Outs2)).
14 fork_a_b(XID) :-
15    @or(g1)
16    rcvMsg(XID,esb,From,or,Events),
17    println(["Task a or b is completed: ", Events, " "]).
```

WCP-28 Blocking Discriminator - Compared to *Structured Discriminator*, the *Blocking Discriminator* construct resets when all active incoming branches have been enabled once for the same process instance. Subsequent enablements of incoming branches are blocked until the discriminator construct has reset [136].

Prova does not support this pattern since it can not block a thread to process just a specific message. However, as explained in Section 4.3.1, reaction rules are associated with a conversation identifier to reflect the process execution and ensure that all tasks of a process instance running in one conversion. In other words, the join connector of this pattern can work well, even if incoming branches of different process instances are enabled.

WCP-30 Structured Partial Join - The convergence of two or more branches (say *m*) into a single subsequent branch following a corresponding divergence earlier in the process model such that the thread of control is passed to the subsequent branch when *n* of the incoming branches have been enabled where *n* is less than *m*. Subsequent enablements of incoming branches do not result in the thread of control being passed on [136].

Prova can implement the *Synchronization* pattern (WCP-03) and the *Structured Discriminator* pattern (WCP-09) by the reaction groups that implement the *AND* and *non-local XOR* join operators, respectively. However, it does not support a partial join directly. Based on the semantics of the event-driven computation of complex event patterns (see Section 5.5), a partial join can be implemented in another similar way, see the example in Listing 6.8.

WCP-31 Blocking Partial Join - The convergence of two or more branches (say m) into a single subsequent branch following one or more corresponding divergences earlier in the process model. The thread of control is passed to the subsequent branch when n of the incoming branches has been enabled (where $2 = n < m$). The join construct resets when all active incoming branches have been enabled once for the same process instance. Subsequent enablements of incoming branches are blocked until the join has reset [136].

Prova does not support this pattern since it does not block a thread to process just a specific message as the *Blocking Discriminator* pattern (WCP-28).

WCP-33 Generalized And-Join - The convergence of two or more branches into a single subsequent branch such that the thread of control is passed to the subsequent branch when all input branches have been enabled. Additional triggers received on one or more branches between firings of the join persist and are retained for future firings [136].

The difference between this pattern and the *Synchronization* pattern (WCP-03) is that the former only allows each branch to receive one trigger after activation, whereas the latter allows each branch to retain additional triggers for future firings. As explained in Section 4.3.1, reaction rules are associated with a conversation identifier to reflect the process execution and keep all tasks of a process instance running in one conversion. The join connector of this pattern can identify different branches in terms of their conversation identifiers, thereby ensuring the branches in one conversation are joined.

WCP-37 Local Synchronizing Merge - The convergence of two or more branches which diverged earlier in the process into a single subsequent branch such that the thread of control is passed to the subsequent branch when each active incoming branch has been enabled. Determination of how many branches require synchronization is made on the basis of information locally available to the merge construct [136].

This pattern is supported by the RAWLS. Based on the semantics of the event-driven computation of complex event patterns, determination of which branches require synchronization is described by a set of derivation rules (see Section 5.5). Such derivation rules can make a decision according to current available information.

WCP-38 General Synchronizing Merge - The convergence of two or more branches which diverged earlier in the process into a single subsequent branch such that the thread of control is passed to the subsequent branch when either (1) each active incoming branch has been enabled or (2) it is not possible that any branch that has not yet been enabled will be enabled at any future time [136].

Like *Structured Synchronizing Merge* (WCP-07), Prova also can implement this

pattern by attaching a timeout to the composite reaction corresponding to the *AND* join operator, and the join waits for a pre-specified time and then consumes all events recorded so far in its payload.

WCP-41 Thread Merge - At a given point in a process, a nominated number of execution threads in a single branch of the same process instance should be merged together into a single thread of execution [136].

Like *Structured Partial Join* (WCP-30), the partial join of this pattern can be implemented according to the semantics of the event-driven computation of complex event patterns.

WCP-42 Thread Split - At a given point in a process, a nominated number of execution threads can be initiated in a single branch of the same process instance [136].

Like *Parallel Split* (WCP-02), Prova can implement *Thread Split* by simply creating a predicate with multiple clauses.

7.1.1.3 Multiple Instance Patterns

WCP-12 Multiple Instances without Synchronization - Within a given process instance, multiple instances of a task can be created. These instances are independent of each other and run concurrently. There is no requirement to synchronize them upon completion [136].

The RAWLS drives the workflow execution by passing event messages: whenever an agent receives a task request (message), a new task instance is created. Multiple instances of a task can be created by sending several requests to an agent responsible for the task.

WCP-13 Multiple Instances with a Priori Design-Time Knowledge - Different with the previous pattern (WCP-12), the required number of instances of this pattern is known at design time. Moreover, it is necessary to synchronize the task instances at completion before any subsequent tasks can be triggered [136].

Like the *Synchronization* pattern (WCP-03), Prova can implement this pattern by sending multiple task requests to an agent and synchronize them at completion. The following example shows that two instances of task *a* running in agent *agent1* are synchronized by the logical reaction group: *@and*.

Listing 7.5: Multiple Instances with a Priori Design-Time Knowledge Implemented in Prova

```
1 split_process(XID) :-
2     fork_a_b(XID).

4 fork_a_b(XID) :-
5     % Task a
6     sendMsg(XID,esb,agent1,request,a(Ins1, Outs1)),
7     @group(g1)
8     rcvMsg(XID,esb,agent1,answer, a(Ins1, Outs1)).
9 fork_a_b(XID) :-
10    % Task a
11    sendMsg(XID,esb,agent1,request, a(Ins2, Outs2)),
12    @group(g1)
```

```
13      rcvMsg(XID,esb,agent1,answer, a(Ins2, Outs2)).
14  fork_a_b(XID) :-
15      @and(g1)
16      rcvMsg(XID,esb,From,and,Events),
17      println(["Tasks a and b are completed: ",Events," "]).
```

WCP-14 Multiple Instances with a Priori Runtime Knowledge - Different with the previous pattern (WCP-13), the required number of instances of this pattern may depend on a number of runtime factors, including state data, resource availability and inter-process communications, but it is known before the task instances must be created [136].

Prova does not support this pattern. Although Prova can dynamically create multiple instances by a loop facility (e.g., *for/2 (i,io)*) according to current circumstances, the instances created dynamically cannot be synchronized at completion.

WCP-15 Multiple Instances without a Priori Runtime Knowledge - Different with the previous pattern (WCP-14), this pattern further requires to add additional instances at any time [136].

Prova does not support this pattern, because the instances created dynamically cannot be synchronized at completion.

WCP-34 Static Partial Join for Multiple Instances - Similar with other partial join patterns, this pattern requires partial instances of a task to complete in order for enabling subsequent tasks [136].

Like *Structured Partial Join* (WCP-30), the partial join of this pattern can be implemented according to the semantics of the event-driven computation of complex event patterns [136].

WCP-36 Dynamic Partial Join for Multiple Instances - Different with the pattern WCP-34, the required number of instances of this pattern may depend on a number of runtime factors [136].

Prova does not support this pattern, because the instances created dynamically cannot be synchronized at completion.

7.1.1.4 State-based Patterns

WCP-16 Deferred Choice - A point in a process where one of several branches is chosen based on interaction with the operating environment. Prior to the decision, all branches represent possible future courses of execution [136].

The RAWLS supports asynchronous human interaction, which makes it possible to integrate human dynamic decision to the workflow execution, thereby supporting dynamic execution path selection at runtime. Moreover, derivation rules provide an expressive decision logic description and allow agents to make a decision according to individual circumstances. For example, there may be more than one agent available for a task, and only the available agent with the highest priority is dynamically selected.

WCP-17 Interleaved Parallel Routing - A set of tasks has a partial ordering defining the requirements with respect to the order in which they must be executed. Each task in the set must be executed once and they can be completed in any order

that accords with the partial order. However, as an additional requirement, no two tasks can be executed at the same time [136].

Unlike Taverna, Triana and Kepler, which do not impose the existence of a complete order among tasks [185], Prova needs to explicitly specify the control flow in a workflow. To implement this pattern, it is necessary to define all possible execution paths. For example, suppose that a process involves tasks a, b and c; task a must be done before the task b; task c can be performed at any time; only one of these tasks can be performed at any time during the process execution. When this process is implemented by Prova, it is necessary to design two possible execution paths of this process, i.e., $a \rightarrow b \rightarrow c$ and $c \rightarrow a \rightarrow b$. It is not difficult to specify different possibilities if there are a few tasks. However, the number of possible execution paths grows rapidly as the number of the involved tasks increases. Therefore, this pattern is not directly supported.

WCP-18 Milestone - A task is only enabled when the process instance (of which it is part) is in a specific state (typically a parallel branch). The state is assumed to be a specific execution point (also known as a milestone) in the process model [136].

None of Taverna, Triana and Kepler supports the milestone pattern. However, the RAWLS specifies control flows of a workflow by active reaction rules. A milestone can be implemented by sending a message to one agent to indicate that a particular state required to enable a task is reached.

WCP-39 Critical Section - Two or more connected subgraphs of a process model are identified as “critical sections”. At runtime for a given process instance, only tasks in one of these “critical sections” can be active at any given time. Once execution of the tasks in one “critical sections” commences, it must complete before another “critical sections” can commence [136].

This pattern is not supported by the RAWLS.

WCP-40 Interleaved Routing - Each member of a set of tasks must be executed once. They can be executed in any order, but no two tasks can be executed at the same time [136].

This pattern is not supported, because control flows of a workflow need to be explicitly specified in the RAWLS. It is possible to specify different possible execution paths if there are a few tasks in this pattern. However, the number of possible execution paths grows rapidly as the number of the involved tasks increases.

7.1.1.5 Iteration Patterns

WCP-10 Arbitrary Cycles - The ability to represent cycles in a process model that have more than one entry or exit point. It must be possible for individual entry and exit points to be associated with distinct branches [136].

In the RAWLS, an agent responsible for a task can be triggered by one or more messages from external environments (i.e., entry points). The reaction to such external messages is described by reaction rules, which specify event templates describing the required information to trigger the task. In addition, an agent can

also send messages to one or more agents (i.e., exit points). The multiple entry and exit points make it possible to implement arbitrary cycles.

WCP-22 Recursion - The ability of a task to invoke itself during its execution or an ancestor in terms of the overall decomposition structure with which it is associated [136].

Prova can implement the *Recursion* pattern with a pair of Prova rules: one describes a recursive task in terms of itself, the other describes a termination condition to make the overall process complete normally. The following example shows a recursion task *size(List, Size)* which calculates the length of a Prova list—the size of a list equals to the size of its tail plus one (Line 3, 4 and 5), and the size of an empty list is zero (Line 2).

Listing 7.6: Recursion Implemented in Prova

```
1 % size if a list
2 size([],0).
3 size([H|T],N) :-
4     size(T,N1),
5     N = N1 + 1.
```

WCP-21 Structured Loop - The ability to execute a task or sub-process repeatedly. The loop has either a pre-test or post-test condition associated with it that is either evaluated at the beginning or the end of the loop to determine whether it should continue. The looping structure has a single entry and exit point [136].

Prova relies on recursion to implement *Structured Loop*. The following Prova example implements a *while...do* loop, which prints a star each time while N is not zero. The CUT(!) in the first rule prevents the backtracking to the second rule if N is zero.

Listing 7.7: Recursion (while...do) Implemented in Prova

```
1 :- eval(print_stars(5)).
2 print_stars(0):- !.
3 print_stars(N):-
4     print(["*"]),
5     N1 = N -1,
6     print_stars(N1).
```

Similarly, the following Prova example implements a *repeat...until* loop associated with a post-test condition. It prints a star each time until N is bigger than five. Note that the CUT(!) prevents backtracking, and the predicate *fail()* forces *print_stars* to fail.

Listing 7.8: Recursion (repeat...until) Implemented in Prova

```
1 :- eval(print_stars(0)).
2 % control condition
3 print_stars(N):-
4     N > 5,
5     !,
6     fail().

8 % loop
9 print_stars(N):-
```

```
10      print(["*"]),
11      N1 = N + 1,
12      print_stars(N1).
```

7.1.1.6 Termination Patterns

WCP-11 Implicit Termination - A given process (or sub-process) instance should terminate when there are no remaining work items that are able to be done either now or at any time in the future and the process instance is not in deadlock. There is an objective means of determining that the process instance has successfully completed [136].

In the RAWLS, the workflow execution terminates after the workflow results are successfully sent to the RAWLS user client.

WCP-43 Explicit Termination - A given process (or sub-process) instance should terminate when it reaches a nominated state. Typically this is denoted by a specific end node. When this end node is reached, any remaining work in the process instance is canceled and the overall process instance is recorded as having completed successfully, regardless of whether there are any tasks in progress or remaining to be executed [136].

The RAWLS does not support *Explicit Termination*.

7.1.1.7 Trigger Patterns

WCP-23 Transient Trigger - The ability for a task instance to be triggered by a signal from another part of the process or from the external environment. These triggers are transient in nature and are lost if not acted on immediately by the receiving task. A trigger can only be utilized if there is a task instance waiting for it at the time it is received [136].

Transient Trigger requires that signals should be processed as soon as they are received. Prova can implement the *Transient Trigger* pattern by a pair of sequential reactions: the first reaction waits for a message indicating that it is ready to receive a transient trigger, and the second reaction waits for the transient trigger from the external environment. The trigger (event) processing can be started as soon as the transient trigger is received. If the transient trigger is received before the task is ready, it will be lost. The following example shows a process to handle transient alarm signals. The handling is ready when a message *monitor(ready)* is received (Line 2). Fireplugs are opened (Line 7) as soon as an alarm signal is received (Line 5). However, it will be lost if the process is not ready to receive alarm signals. Note that the alarm signal detection has a different conversation identifier *XID1* because the alarm signals are usually from the external sensors.

Listing 7.9: Transient Trigger Implemented in Prova

```
1 handle_alarm(XID):-
2     rcvMsg(XID,Protocol,From,request, monitor(ready)),
3     println(["Ready for receiving alarm signals."]),
```

```
5     rcvMsg(XID1,Protocol,From,request,alarm(Sensor)),
6     println(["An alarm signal is received from Sensor"]),
7     open(fireplugs).
```

WCP-24 Persistent Trigger - The ability for a task to be triggered by a signal from another part of the process or from the external environment. These triggers are persistent in form and are retained by the process until they can be acted on by the receiving task [136].

The RAWLS supports *Persistent Trigger*. Different with *Transient Trigger*, the persistent triggers (events) are inherently durable in nature, ensuring that they are not lost in transit and are buffered until they can be handled. The RAWLS presents an event-driven workflow execution, i.e., a task is started when an agent receives a task request (event) from other agents or from the external environment (e.g., the RAWLS user client). Moreover, reaction rules used to specify control flows are associated with a conversation identifier to reflect the process execution, thereby enabling Prova agents to deal with different process instances accurately.

Summary

The results of control-flow pattern-based evaluation are summarized in Table 7.1, including the evaluation results regarding to Kepler, Taverna and Triana from [185]. "+" denotes that a pattern is directly supported. If a pattern is not supported, it is rated to "-".

The evaluation considers 35 control-flow patterns delivered by the Workflow Patterns Initiative, except the patterns that involve cancellation and force completion of a workflow activity (WCP-19, 20, 25, 26, 27, 29, 32, 35). The results show that the RAWLS supports 26 patterns, which are much more than Kepler which supports 18; Taverna which supports 10 patterns; Triana which supports 13 patterns. To be more specific, the rule-based workflow specification of the RAWLS also supports the basic workflow patterns (i.e., WCP-01–05 in the table) as Kepler, Taverna and Triana. With respect to the advanced patterns, it has superiority over other three systems, especially to the advanced branching and synchronization patterns (i.e., WCP-06–09, 28–33, 37–38, 41, 42), the state-based patterns (i.e., WCP-16–18, 39–40) and the trigger patterns (i.e., WCP-23 and 24). The advanced branching and synchronization patterns characterize more complex branching and merging concepts in workflows. For example, *Structured Discriminator* (WCP-09) requires a join connector to select one branch from two or more branches and ignore others. None of Kepler, Taverna and Triana supports it, because they are not able to reset the join construct when exactly one piece of data is received [185]. Prova well supports this pattern via the reaction group, *@or*, which requires *either* of the event channels to be successfully proved. Based on the semantics of the event-driven computation of complex event patterns (see Section 5.5), the RAWLS also supports the partial join, thereby supporting the patterns WCP-30, 34 and 41. The state-based patterns are the ones, in which decisions are made according to data associated with current execution, including the status of activities as well as process-relevant

working data. The RAWLS not only integrates human dynamic decisions, but also provides an expressive decision logic description, thereby supporting the patterns WCP-16 and 18 that are not supported by other three systems. Moreover, with the benefits of Prova reactive event messaging, the RAWLS supports two trigger patterns: *Transient Trigger* (WCP-23) and *Persisten Trigger*(WCP-24).

Table 7.1: Control-Flow Pattern-Based Comparison

Control Flow Patterns	Kepler	Taverna	Triana	RAWLS
WCP-01. Sequence	+	+	+	+
WCP-02. Parallel Split	+	+	+	+
WCP-03. Synchronization	+	+	+	+
WCP-04. Exclusive Choice	+	-	+	+
WCP-05. Simple Merge	+	+	+	+
WCP-06. Multi-Choice	+	+	+	+
WCP-07. Structured Synchronizing Merge	-	-	-	+
WCP-08. Multi-Merge	+	+	+	+
WCP-09. Structured Discriminator	-	-	-	+
WCP-10. Arbitrary Cycles	+	-	+	+
WCP-11. Implicit Termination	+	+	+	+
WCP-12. Multiple Instances without Synchronization	+	+	+	+
WCP-13. Multiple Instances with a Priori Design-Time Knowledge	-	-	-	+
WCP-14. Multiple Instances with a Priori Runtime Knowledge	-	-	-	-
WCP-15. Multiple Instances without a Priori Runtime Knowledge	-	-	-	-
WCP-16. Deferred Choice	-	-	-	+
WCP-17. Interleaved Parallel Routing	+	-	-	-
WCP-18. Milestone	-	-	-	+
WCP-21. Structured Loop	+	-	+	+
WCP-22. Recursion	-	-	-	+
WCP-23. Transient Trigger	-	-	-	+
WCP-24. Persistent Trigger	+	+	+	+
WCP-28. Blocking Discriminator	-	-	-	-
WCP-30. Structured Partial Join	-	-	-	+
WCP-31. Blocking Partial Join	-	-	-	-
WCP-33. Generalized AND-Join	+	+	+	+
WCP-34. Static Partial Join for Multiple Instances	-	-	-	+
WCP-36. Dynamic Partial Join for Multiple Instances	-	-	-	-
WCP-37. Local Synchronizing Merge	-	-	-	+
WCP-38. General Synchronizing Merge	-	-	-	+
WCP-39. Critical Section	-	-	-	-
WCP-40. Interleaved Routing	+	-	-	-
WCP-41. Thread Merge	+	-	-	+
WCP-42. Thread Split	+	-	-	+
WCP-43. Explicit Termination	+	-	-	-

7.1.2 Data Patterns

Workflow Data Patterns (WDPs) captures various ways in which data is represented and utilized in workflows [187]. Like the control-flow-based evaluation, the numbering and definition of data patterns in this section also follows the Workflow Patterns Initiative for the clarity.

7.1.2.1 Data Visibility

Data visibility patterns describe the ways of data elements are defined and utilized. In other words, they define a scope in which data elements are accessible.

WDP-01 Task Data - Data elements can be defined by tasks which are accessible only within the context of individual execution instances of that task [187].

The RAWLS supports the *Task Data* pattern, since it is a basic requirement for every workflow system, and variables defined in a primitive task can be accessed during the task execution.

WDP-04 Multiple Instance Data - Tasks which are able to execute multiple times within a single case can define data elements which are specific to an individual execution instance [187].

The RAWLS supports multiple concurrent instances of an activity or sub-process, and each instance can have different datasets or varying parameter settings. For example, two instances of task a are provided with different inputs (see Listing 7.5).

WDP-08 Environment Data - Data elements which exist in the external operating environment are able to be accessed by components of processes during execution [187].

Prova employed by the RAWLS can access external data via query languages. It not only provides access to relational databases via SQL, but also supports access to Semantic Web Data available on the Internet (see Section 4.4.2).

The RAWLS does not support the data visibility patterns WDP-02 *Block Data*, WDP-03 *Scope Data*, WDP-05 *Case Data*, WDP-06 *Folder Data* and WDP-07 *Workflow Data*, which define shared data in a subprocess, a subset of tasks, a process instance (a case), multiple process instances (multiple cases), all components of a process, respectively. This is because, in the RAWLS, the agents communicate each other by sending and receiving messages, data is local in a task or agent. Data shared by a set of tasks cannot be defined, unless the tasks are locally in one agent, and the shared data is globally defined. For more details of such patterns, see [187].

7.1.2.2 Internal Data Interaction

WDP-09 Task to Task - The ability to communicate data elements between one task instance and another within the same case [187].

In the RAWLS, the agents communicate each other by sending and receiving messages, i.e., passing data between from one task to another. Based on messaging reaction rules, both control flows and data flows can be specified and implemented.

WDP-10 Block Task to Sub-Workflow Decomposition - The ability to pass data elements from a block task instance to the corresponding subprocess that defines its implementation [187].

WDP-11 Sub-Workflow Decomposition to Block Task - The ability to pass data elements from the underlying subprocess back to the corresponding block task [187].

Messaging reaction rules can not only capture global ECA rules, but also maintain local conversation states to perform complex tasks in a sub-process. Like the *Task to Task* pattern (WDP-09), the pattern WDP-10 can be implemented by sending a message (data) to an agent which manages the sub-process implementing a block task. At completion, the agent can send the results back to the requester (WDP-11).

WDP-12 To Multiple Instance Task - The ability to pass data elements from a preceding task instance to a subsequent task which is able to support multiple execution instances [187].

The RAWLS creates a task instance as soon as required data is received. As shown in Listing 7.5, two distinct messages are sent to agent *agent1* to create two instances of task *a*.

WDP-13 From Multiple Instance Task - The ability to pass data elements from a task which supports multiple execution instances to a subsequent task [187].

Since the RAWLS supports the synchronization of multiple task instances at completion before any subsequent tasks can be triggered (WCP-13), it also supports the result synchronization of multiple task instances. As shown in Listing 7.5, the results of two instances of task *a* are collected in the variable *Events*.

WDP-14 Case to Case - The passing of data elements from one case of a process during its execution to another case that is executing concurrently [187].

The RAWLS does not support this pattern.

7.1.2.3 External Data Interaction

WDP-15 (WDP-19, WDP-23) Task (Case/Workflow) to Environment - Push Oriented - The ability of a task (a case or a process environment) to pass data elements to resources or services in the operational environment [187].

Prova combines declarative with imperative programming styles and allows calling external procedural attachments (e.g., Java methods) in declarative rules. Therefore, it is possible for a task (a case or a process environment) to pass data to the external environment.

WDP-16 (WDP-20, WDP-24) Environment to Task (Case/Workflow) - Pull Oriented - The ability of a task (a case or a process environment) to request data elements from resources or services in the operational environment [187].

Prova can implement such patterns by accessing external data with query languages, such as SQL, SPARQL.

WDP-17 (WDP-21, WDP-25) Environment to Task (Case/Workflow) - Push Oriented - The ability of a task (a case or a process environment) to receive

and utilize data elements passed to it from services and resources in the operating environment on an unscheduled basis [187].

The agents of the RAWLS are reactive, and the detection of external environments is implemented by reaction rules, which react to occurred events (external events or changed conditions) by executing certain actions (see Section 4.3.1). These patterns are not supported by Kepler, Taverna and Triana.

WDP-18 (WDP-22, WDP-26) Task (Case/Workflow) to Environment - Push Oriented - The ability of a task (a case or a process environment) to receive and respond to requests for data elements from services and resources in the operational environment [187].

Besides perceiving external environments, the agents of the RAWLS can react to the events from the external environment. Moreover, a group of agents can collaborate in cooperative ways to deal with external events. These patterns are not supported by Kepler, Taverna and Triana.

7.1.2.4 Data Transfer Patterns

WDP-27 (WDP-28) Data Transfer by Value - Incoming (Outgoing) - The ability of a process component to receive incoming data elements by value (pass data elements to subsequent components as values) avoiding the need to have shared names or common address space with the component(s) from which it receives them [187].

In the RAWLS, data passing between agents is implemented via sending and receiving event messages. The payload of a message contains data values, on which receivers (agents) can operate.

WDP-29 Data Transfer - Copy In/Copy Out - The ability of a process component to copy the values of a set of data elements from an external source (either within or outside the process environment) into its address space at the commencement of execution and to copy their final values back at completion [187].

Based on the data patterns WDP-08 and WDP-15 (WDP-19, WDP-23), the agents of the RAWLS can copy data from the external environment and copy back any changes to this data at the time of task completion.

WDP-30 Data Transfer by Reference - Unlocked - The ability to communicate data elements between process components by utilizing a reference to the location of the data element in some mutually accessible location. No concurrency restriction is applied to the shared data element [187].

In the RAWLS, the payload of the event messages between agents can be either simple data values or data references if the data is large.

WDP-31 Data Transfer by Reference - With Lock - Different with the previous pattern (WDP-30), concurrency restrictions are implied with the receiving component receiving the privilege of read-only or dedicated access to the data element. The required lock is declaratively specified as part of the data passing request [187].

This pattern is not supported since the lock cannot be specified.

WDP-32, WDP-33 Data Transformation - Input/Output - The ability to apply a transformation function to a data element prior to it being passed to a process component or passed out of a process component [187].

Prova does not directly support the data transformation. However, the RAWLS integrates distributed agents by Mule ESB, which supports converting data from one format to another [175]. For example, a workflow request is translated into a Prova list before the workflow engine (i.e., a Prova agent) processes it; at the workflow completion, the workflow output in a Prova list is translated back to a user friendly format.

7.1.2.5 Data-Based Routing

WDP-34 (WDP-36) Task Precondition (Postcondition) - Data Existence - Data-based preconditions (postconditions) can be specified for tasks based on the presence of data elements at the time of execution (at the time of task completion). The preconditions (postconditions) can utilize any data elements available to the task with which they are associated. A task can only proceed if the associated precondition (postconditions) evaluates positively [187].

Like other three scientific workflow systems, the RAWLS starts a task if required inputs are available. Moreover, it is also possible for the agents to impose postconditions at task completion to check if the outputs are generated.

WDP-35 (WDP-37) Task Precondition (Postcondition) - Data Value - Different with the previous two data patterns, the preconditions and postconditions of the current two patterns are specified based on the value of specific parameters at the time of execution [187].

Like the previous two data patterns based on data existence, the RAWLS also supports data value-based specification of preconditions and postconditions. Moreover, derivation rules provide high expressive power to describe complex domain-specific decision logic.

WDP-38 Event-Based Task Trigger - The ability for an external event to initiate a task and to pass data elements to it [187].

The RAWLS presents an event-driven workflow execution and supports the trigger patterns (WCP-23, WCP-24). Therefore, this pattern is also supported by the RAWLS.

WDP-39 Data-Based Task Trigger - Data-based task triggers provide the ability to trigger a specific task when an expression based on data elements in the process instance evaluates as true. Any data element accessible within a process instance can be used as part of a data-based trigger expression [187].

The RAWLS specifies the task dependencies of a workflow by reaction rules, which perform actions in terms of occurred events. A data-based task trigger can also be implemented by a reaction rule, which is associated with a data element-based expression to start a task. The task is triggered as soon as the expression is evaluated to be true.

WDP-40 Data-Based Routing - Data-based routing provides the ability to

alter the control-flow within a case based on the evaluation of data-based expressions [187].

The implementation of *Data-Based Routing* has been shown in the control-flow patterns *Exclusive Choice* (WCP-04) and *Multiple Choice* (WCP-06). As seen in Listings 7.2 and 7.3, the variables X and Y attached to the split construct determine the selection of outgoing branches.

Table 7.2: Data Pattern-Based Comparison

Data Patterns	Kepler	Taverna	Triana	RAWLS
WDP-01. Task Data	+	+	+	+
WDP-02. Block Data	-	-	-	-
WDP-03. Scope Data	-	-	-	-
WDP-04. Multiple Instance Data	+	+	+	+
WDP-05. Case Data	-	-	-	-
WDP-06. Folder Data	-	-	-	-
WDP-07. Workflow Data	-	-	-	-
WDP-08. Environment Data	+	+	+	+
WDP-09. Task to Task	+	+	+	+
WDP-10. Block Task to Sub-Workflow Decomposition	+	+	+	+
WDP-11. Sub-Workflow Decomposition to Block Task	+	+	+	+
WDP-12. To Multiple Instance Task	+	+	+	+
WDP-13. From Multiple Instance Task	-	-	-	+
WDP-14. Case to Case	-	-	-	-
WDP-15. Task to Environment - Push-Oriented	+	+	+	+
WDP-16. Environment to Task - Pull-Oriented	+	+	+	+
WDP-17. Environment to Task - Push-Oriented	-	-	-	+
WDP-18. Task to Environment - Pull-Oriented	-	-	-	+
WDP-19. Case to Environment - Push-Oriented	+	+	+	+
WDP-20. Environment to Case - Pull-Oriented	+	+	+	+
WDP-21. Environment to Case - Push-Oriented	-	-	-	+
WDP-22. Case to Environment - Pull-Oriented	-	-	-	+
WDP-23. Workflow to Environment - Push-Oriented	+	+	+	+
WDP-24. Environment to Workflow - Pull-Oriented	+	+	+	+
WDP-25. Environment to Workflow - Push-Oriented	-	-	-	+
WDP-26. Workflow to Environment - Pull-Oriented	-	-	-	+
WDP-27. Data Transfer by Value - Incoming	+	+	+	+
WDP-28. Data Transfer by Value - Outgoing	+	+	+	+
WDP-29. Data Transfer - Copy In/Copy Out	+	+	+	+
WDP-30. Data Transfer by Reference - Unlocked	+	+	+	+
WDP-31. Data Transfer by Reference - With Lock	-	-	-	-
WDP-32. Data Transformation - Input	-	-	-	+
WDP-33. Data Transformation - Output	-	-	-	+
WDP-34. Task Precondition - Data Existence	+	+	+	+
WDP-35. Task Precondition - Data Value	-	-	-	+
WDP-36. Task Postcondition - Data Existence	-	-	-	+
WDP-37. Task Postcondition - Data Value	-	-	-	+
WDP-38. Event-based Task Trigger	-	-	-	+
WDP-39. Data-based Task Trigger	-	-	-	+
WDP-40. Data-based Routing	+	-	+	+

7.1.2.6 Summary

The results of data pattern-based evaluation are summarized in Table 7.2, including the evaluation results regarding to Kepler, Taverna and Triana from [185]. "+" denotes that a pattern is directly supported. If a pattern is not supported, it is rated to "-".

The evaluation considers all 40 data patterns delivered by the Workflow Patterns Initiative. The results show that the RAWLS supports 33 patterns, which are much more than Kepler which supports 19; Taverna which supports 18 patterns; Triana which supports 19 patterns.

To be more specific, the RAWLS employs messaging reaction rules to describe agent interactions by sending and receiving messages, and data is local in a task or agent and data shared by a set of tasks cannot be defined. Therefore, the data visibility patterns which involve data sharing between tasks or cases are not supported (WDP-02, 03, 05, 06 and 07). Based on messaging reaction rules, the RAWLS supports all external data interaction patterns, especially the ones which receive and respond to requests for data elements from the external environment (WDP-17–25). However, in Kepler, Taverna and Triana, since only the tasks of a workflow can start a connection with external environment, they are not reactive to support such patterns. Moreover, with the agent-oriented execution framework and derivation rules, it is possible to provide complex domain-specific preconditions and postconditions to perform tasks, thereby supporting the patterns WDP-34–37. In addition, based on Mule ESB, two data transformation patterns (WDP-32 and 33) are also supported by the RAWLS.

7.1.3 Scientific Workflow Patterns

The workflow patterns delivered by the Workflow Patterns Initiative are originally for business workflows and this section considers four new scientific workflow patterns identified in [185].

SWP-01 Dynamic Input Size - The ability to consume n data elements from the same input channel, where n is determined at runtime on the basis of the value received from another input channel [185].

This pattern can be considered as the dynamic version of WCP-41 *Tread Merge*, and the number of tokens from the same input channel required to execute a task is determined at runtime [185]. However, the RAWLS does not support this pattern. Although the partial join is supported by the RAWLS, the logic of the partial join is predefined and the number of tokens required to be received can not be dynamically determined.

SWP-02 Dynamic Token Replication - The ability to generate n copies of a data element d received in input, where the number n is determined at runtime on the basis of the value received from another input channel [185].

In the RAWLS, whenever an agent receives a task request, a new task instance is created. This pattern can be implemented by the iteration patterns (see Section

7.1.1.5), which can dynamically generate multiple copies of a data element and process them one by one. Therefore, the RAWLS supports this pattern.

SWP-03 Dynamic Balancing of Input Tokens - The ability to replicate a data element received from an input channel in order to balance the number of tokens received from another input channel [185].

The key of implementing this pattern is to balance the data elements produced by two or more tasks with different production rates. To implement this pattern, all data elements are stored locally the one, which has the highest production rate. Such data elements are updated whenever their latest ones are received. When the data element that has the highest production rate is received, it is sent to execute another task together with those data elements stored locally.

SWP-04 Cartesian Product of Input Tokens - The ability to compute the cartesian product of the data values contained into two or more channels connected to the same task, so that this task can be executed on each possible combination of inputs [185].

This pattern is supported by the RAWLS. Prova provides a builtin *element/2 (io,i)*, which extracts one element each time from a list, and combining data produced by different tasks can be implemented by using two or more element builtins. For example, the combination of two lists: L = [1, 2, 3] and M = [9, 8, 7] can be simply implemented as follows:

Listing 7.10: Combination of Two Lists

```
1 listCombination():-
2     L = [1, 2, 3],
3     M = [9, 8, 7],
4     element(L1, L),
5     element(M1, M),
6     println(["(",L1,",", M1,")"]).

8 %This program returns:
9 %(1,9),(1,8),(1,7),(2,9),(2,8),(2,7),(3,9),(3,8),(3,7)
```

The above evaluation results are summarized in Table 7.3, including the evaluation results regarding to Kepler, Taverna and Triana from [185]. "+" denotes that a pattern is directly supported. If a pattern is not supported, it is rated to "-". "±" denotes that a pattern is partially supported.

Table 7.3: Scientific Workflow Pattern-Based Comparison

Data Patterns	Kepler	Taverna	Triana	RAWLS
1. Dynamic Input Size	+	-	-	-
2. Dynamic Token Duplication	+	-	-	+
3. Dynamic Input Balancing	-	±	±	+
4. Cartesian Product of Input Tokens	-	+	-	+

7.2 Evaluation of the Domain Knowledge Representation

The RbAF exploits the benefits of both DL and LP to express (domain-specific) decision logic in workflows, i.e., integrating existing Semantic Web data into declarative rules (see Section 4.4). Although the DL-based ontologies and LP rules are expressible in each other to some extent, a unified logic based on DL and LP is still on the way. This section evaluates the expressive power of the domain knowledge representation in the RbAF from both LP and DL perspectives in Sections 7.2.1 and 7.2.2, respectively.

7.2.1 LP-based Knowledge Representation Evaluation

Different scientific workflow applications usually involve different decision logic. They may deal with entirely different datasets and access various resources in their implementation. In order to describe scientific workflows, different forms of logic programs may be used either separately or in combination. However, the expressive power of a logic program is highly related to its complexity. To be useful in practical applications it is necessary to trade expressivity for complexity. This section discusses which kind of logic program can provide modest expressivity and complexity to describe the decision logic involved in the WsSWFs, and how Prova supports it.

As mentioned in Section 2.8, there are different forms of logic programs and their combinations, which vary in expressivity power and complexity, as shown in Figure 2.14. Plain Datalog programs can express useful queries of relational algebra, e.g., the queries represented by SQL select-from-where, but they are function-free. Definite logic programs support functions, but they exclude negative information and default statements. Normal logic programs allow negation (NaF to be more precisely) in their bodies. The head of a rule of a normal logic program must be an atom. There are also extended logic programs, which support both NaF and classical negation. Each definite, normal and extended logic program can be a propositional and Datalog program. In addition, there are extensions that have been made to such logic programs, e.g., to allow disjunctions as heads of rules.

Scientific knowledge representation usually involves non-monotonic reasoning, i.e., propositions derived from a knowledge base may be changed by adding or removing its clauses. Moreover, scientific knowledge representation usually needs to describe exceptions, which do not conform to general rules. Among aforementioned logic programs, the most suited for scientific knowledge representation is normal logic programs. A normal logic program inherits the expressiveness of propositional and finite logic programs. Also, normal logic programs support NaF and provide a simple and practical formalism for expressing defaults and exceptions, and other forms of non-monotonic reasoning. Disjunctive normal logic programs extend normal logic programs by adding disjunction in the rule heads. However, computing answer sets of disjunctive normal logic programs are hard (Full disjunctive logic programs under SMS is π_1^1-complete [188]) and there are also few solid and efficient

implementations. Extended logic programs support both NaF and classical negation, however, the classical negation might lead to logical conflicts between rules. Therefore, normal logic programs have modest expressiveness to describe scientific policies and can be regarded as general logic programs to specify (domain-specific) decision logic in the WsSWFs.

However, there are two problems that need to be considered when using normal logic programs: *recursion-through-negation* and *undecidability when using function symbols with no restrictions* (see Section 2.8). The former can be solved by checking if a logic program can be stratified or not, and Prova can be used to execute stratified programs directly. Moreover, NaF is *safe* only when the test goal is ground. As known in the example in Listing 2.3, *jack* is dead since it cannot fly. But for a goal that queries all dead animals, i.e., *:- eval(died(X)).*, Prova returns nothing. The reason is that the call to *not(fly(X))* does not return the animals that are dead. It fails because there is at least an animal (i.e., *tweety*) can fly. Therefore, domain experts need to use NaF with ground goals. This is can be done by Prova *bound* built-in, which is capable of testing if arguments supplied to a rule are bound. More specific examples can be found in Section 7.4.

But for using function symbols with no restrictions, the problem becomes complex. As mentioned in Section 2.8, using function symbols in logic programs makes reasoning tasks undecidable in general cases. To overcome this issue, there are solutions that have been proposed to impose restrictions on the program syntax to guarantee the decidability of reasoning tasks (see Section 2.8). A decidable fragment mentioned is *nonrecursive logic programs* [63]. However, the restriction (i.e., nonrecursive) is strong and causes a loss of expressive power to express recursion relations.

This thesis follows the spirits of γ-*restricted* and *FP2 programs* to guarantee the decidability of the decision logic in the WsSWFs.

***γ-restricted* programs** [66, 64]. A normal logic program is γ-*restricted* if for any rule r defining predicate p, each variable occurring in r is initiated by means of an occurrence of a predicate q in $B^+(r)$ such that $\gamma(q) < \gamma(p)$. In other words, the feasible ground instances of r are determined by predicates from lower levels than the one of p. A detailed definition of γ-*restricted* programs can be found in [66].

For example, the following program P_1 adapted from [64] is a γ-*restricted* program:

$r(1)$. $r(2)$. $q(X)$:- $r(X),\ p(X)$. $p(X)$:- $q(X),\ not\ p(X)$.

The program can be finitely instantiated: first, the rule $q(X)$:- $r(X), p(X)$. can be finitely instantiated by $r(X)$, which is defined by finite facts. Then the rule $p(X)$:- $q(X), not\ p(X)$ is finitely instantiated because $q(X)$ is in its body. A level mapping γ could be $\gamma(r) = 1$, $\gamma(q) = 2$, $\gamma(p) = 3$.

The restriction of the γ-*restricted* programs ensures the decidability of grounding reasoning (i.e., checking the presence of specific ground atoms among the consequences of a program) and computability of non-ground reasoning (i.e., computing all answers to non-ground queries) [64]. Hence, both ground and non-ground reasoning over the above program P_1 is decidable.

FP2 programs [69, 64]. The definition of an FP2 program is based on two key concepts: *recursion patterns* and *call-safeness*. A recursion pattern π is a function mapping each predicate p to a subset of its arguments, and such arguments either strictly decrease or almost never get larger at each recursion. A program is call-safe with respect to a recursion pattern π if variables appearing in negative subgoals or in an argument of a subgoal selected by π are initialized by previous resolved goals. Furthermore, a normal logic program belongs to FP2 if there is a recursion pattern π such that the program is call-safe with respect to π. A detailed definition of FP2 programs can be found in [69].

For example, the following program P_2 for appending lists belongs to FP2:

$append([], L, L)$. $\quad$ $append([H|T], L, [H|LT])$:- $append(T, L, LT)$.

A recursion pattern π can be obtained by mapping *append* on all of their arguments, i.e. $\pi_{append} = \{1,\ 2,\ 3\}$; in addition, two rules are call-safe because the variables H, T, L and LT occur in the selected arguments.

FP2 programs have decidable ground reasoning, but its non-ground reasoning is uncomputable [64].

Both γ-*restricted* and *FP2 programs* are designed for normal logic programs. In other words, if the decision logic in the WsSWFs is also a γ-*restricted* program or an FP2 program, its reasoning is decidable. In terms of the classification in [64] (see Section 2.8), γ-*restricted* and *FP2 programs* are in the *bottom-up computable* and *top-down computable* groups, respectively. Programs in the bottom-up commutable group allow for stable model computation and query answering by searching over finite ground programs. Programs in the top-down computable group are designed for query answering, and the programs usually have an infinite number of answer sets and a finite number of atoms must be identified to guarantee the decidability of answering a query. This also means that γ-*restricted* and *FP2 programs* can be used to test the decidability of decision logic if there exists a finite ground program or not, respectively.

It is worth noticing that, if a logic program is γ-*restricted* or FP2, its reasoning is decidable. However, not all decidable logic programs are either γ-*restricted* or FP2 logic programs. As mentioned in Section 2.8, there are other decidable logic programs, but they are out of the scope of this thesis because they are more expressive and complicated and are not originally designed for normal logic programs.

7.2.2 DL-based Knowledge Representation Evaluation

DL is mainly used for formal description of concepts, roles and their relations, and is also known as the basis for many widely used ontology languages. Since different scientific workflow applications involve different domain ontologies, this section presents the complexity of ontologies that have different expressive power, and discusses if the RAWLS provides an expressive query to them.

Domain ontologies involved in this work can be represented in RDFS and OWL. RDFS is built on top of RDF and provides well defined meanings for RDF vocabularies. RDFS defines classes, properties and relations by using subClassOf, Class,

Property, subPropertyOf, Resource, range, domain, etc. Reasoning RDFS data is an entailment process from one (source) RDF graph to another target RDF graph by means of a set of rules that defines the semantics of RDFS. According to the results from [189], the entailment for RDFS is decidable, *NP-complete*, and in *P* if the target graph does not contain blank nodes.

RDFS enriches the data model represented by RDF and provides support for describing simple ontologies. However, RDFS cannot express: range restrictions, disjointness of classes, Boolean combinations of classes, cardinality restrictions and special characteristics of properties. OWL is another ontology format, but it is more expressive than RDFS and provides a larger vocabulary to express the relationships between things. The main facilities that OWL can express over RDFS are object property relations between classes, constraints on properties, equivalences between classes, properties of properties and Boolean combinations of classes and constraints. However, it is impossible to compute all interesting logical conclusions from an OWL ontology since the OWL reasoning could be exponential or even undecidable. To address this issue, OWL 1.0 provides three increasingly expressive sub-languages: OWL Lite, OWL DL and OWL Full [190]:

- **OWL Lite** is the least expressive sub-language of OWL and supports those users primarily needing a classification hierarchy and simple constraints; it corresponds to the SHIF(D) description logic.
- **OWL DL** is more expressive than OWL Lite while retaining computational completeness and decidability (because it is based on the decidable Description Logic); OWL DL includes all OWL language constructs, but such constructs can be used only under certain restrictions; it corresponds to the SHION(D) description logic.
- **OWL Full** is meant for users who want maximum expressiveness and syntactic freedom with no computational guarantees.

The semantics of OWL Lite and OWL DL are based on DL, and thus both of them are decidable. With regard to the computational complexity, OWL Lite entailment is known to be complete for *EXPTIME*, while the entailment for OWL DL is known to be complete for *NEXPTIME*. OWL Full entailment is known to be undecidable.

OWL 2 [191] is a subsequent compatible revision to its previous versions (aka. OWL 1) and became a W3C Recommendation in October 2009. OWL 2 provides an increased expressive power, such as qualified cardinality restrictions, property chain inclusion axioms and reflexive, irreflexive and asymmetric object properties. The detailed differences between OWL 2 and OWL 1 can be found in [191]. Like its previous versions, OWL 2 also defines three tractable sub-languages (aka. profiles) that offer different advantages depending on specific applications [192]:

- **OWL 2 EL** is particularly useful in applications employing ontologies that contain large numbers of properties and/or classes; it captures the expressive power used by such ontologies.

- **OWL 2 QL** aims at applications that use large volumes of instance data, and enables easier access and query to data stored in databases; in OWL 2 QL, conjunctive query answering can be implemented using conventional relational database systems.
- **OWL 2 RL** aims at applications that require scalable reasoning without sacrificing too much expressive power; it is designed to accommodate OWL 2 applications that can trade the full expressivity of the language for efficiency, as well as RDF(S) applications that need some added expressivity.

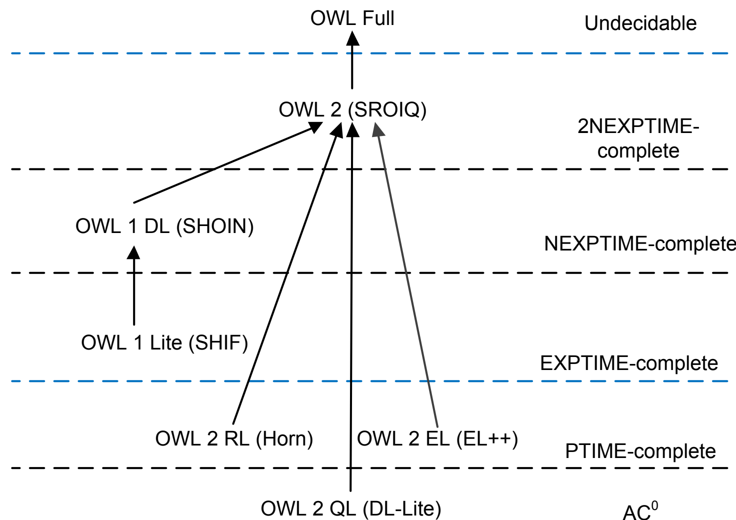


Figure 7.1: Expressiveness and Complexity of OWL Family

Each of the profiles trades off different aspects of OWL expressive power in return for different computational and/or implementational benefits. The basic reasoning problems of OWL 2 EL can be performed in polynomial time with respect to the size of the ontology. The OWL 2 QL enables a tight integration with relational database systems. It is designed so that sound and complete query answering is in LOGSPACE (more precisely, AC^0) with respect to the size of the data. In other words, it has the same complexity as Datalog. OWL 2 RL reasoning systems can be implemented using rule-based reasoning engines, as a mapping to logic programs. In the OWL 2 RL fragment, the ontology consistency, class expression satisfiability, class expression subsumption, instance checking and conjunctive query answering problems can be solved in polynomial time.

OWL 2 itself is based on the SROIQ description logic, and its reasoning complexity is *2NEXPTIME-complete*. To sum sup, the expressive power and complexity of the OWL family are summarized in Figure 7.1 [193].

The SPARQL-API query engine employed by the RAWLS is built on OWL API and fully aligned with the OWL 2. It uses SPARQL syntax and is more expressive than existing DL query languages by allowing mixed TBox, RBox and ABox

queries (see Section 6.4). Moreover, it acts as a SPARQL-DL interface to every reasoner supporting OWL API 3, such as Pellet [194], RacerPro [195], FaCT++ [196] and HermiT [155]. In other words, such reasoners can be easily configured as a main ontology reasoner in terms of specific requirements. Some prominent reasoners providing the implementation of OWL API are as follows:

- Pellet [194] is a Java-based capable OWL-DL reasoner that supports reasoning with the full expressivity of OWL-DL and has been extended to support the OWL 2 specification.
- The RacerPro system [195] is an optimized tableau reasoner for SHIQ(D). RacerPro is a commercial reasoner, but free trials and research licenses are available.
- FaCT++ [196] is a tableaux reasoner written in C++ which supports the full OWL 2 DL profile.
- JFact [197] is a Java implementation of the FaCT++ reasoner with extended datatype support.
- HermiT [155] is a Java-based OWL reasoner for the Description Logic SHOIQ+.
- Chainsaw [198] is a *metareasoner* which computes ontology modules first and then delegates the processing of the modules to an existing OWL 2 DL reasoner. Currently Chainsaw has FaCT++ as the delegate reasoner and supports the same expressivity as FaCT++.

All of these reasoners support OWL 2 and are available as open source software (except RacerPro). A summary of their features is shown in Table 7.4.

Table 7.4: Comparison of Reasoners Implementing OWL API

	Pellet	**RacerPro**	**FaCT++**	**HermiT**	**Chainsaw**	**JFact**
OWL-DL Support	Yes	Yes	Yes	Yes	Yes	Yes
OWL 2 Support	Yes	Yes	Yes	Yes	Yes	Yes
Supported Expressivity	SROIQ(D)	SROIQ(D)	SROIQ(D)	SROIQ+	SROIQ(D)	SROIQ(D)
Algorithm	Tableau	Tableau	Tableau	Hyper-tableau	AD/sub-reasoner	Tableau
License	Open/closed-source	Closed-source	Open-source	Open-source	Open-source	Open-source

The RAWLS employs HermiT, which is based on hypertableau calculus, to be a real reasoner behind the SPARQL-DL query engine to reason domain ontologies. According to the W3C OWL 2 compliance test suite for ontology reasoners [199], HermiT passes all test cases of OWL 2 DL and OWL 2 EL that are applicable under

Direct Semantics [200]. Also, according to the test results performed by the ORE 2013 reasoner competition [201], HermiT performs better than other reasoners implementing OWL API on three standard reasoning tasks: *classification*, *consistency* and *concept satisfiability*. Note that, as it aforementioned, different workflow applications may need different domain ontologies. Based on the flexible SPARQL-DL query engine, users can choose any ontology reasoner implementing OWL API in terms of specific requirements of their workflow applications.

7.3 Computational Model-Based Empirical Evaluation

This thesis presents a distributed rule-based multi-agent system, called RAWLS, for the WsSWFs. This section evaluates the RAWLS in terms of typical properties of computational models.

Cycles in execution graph: Cycles are common during the process modeling when individual activities or groups of activities need to be repeated. Based on the control-flow pattern evaluation in Section 7.1.1.1, the RAWLS supports three distinct types of the repetition: *Arbitrary Cycles* (WCP-10), *Structured Loop* (WCP-21), and *Recursion* (WCP-22). Among the tested workflow systems, the RAWLS is only one that supports *Recursion*, which is implemented by a pair of declarative rules: one describes a recursive task in terms of itself, and the other one specifies the termination condition. *Structure Loop* includes repetitions based on dedicated programmatic constructs *while...do* and *repeat...until* statements. Prova engine can implement them based on recursive rules. Moreover, based on messaging reactions rules, an agent can not only receive event messages from multiple sources, but also send event messages to multiple destinations, thereby implementing *Arbitrary Cycles*. The *Arbitrary Cycles* pattern brings the flexibility to the workflow description, but it also poses a risk to create an infinite loop and make a workflow get stuck. To overcome this problem, the RAWLS employs an infinite loop detector in Mule ESB to count messages passing between the agents, and an infinite loop happens if a message is repeatedly sent over a specified number of times (e.g., 100) in a workflow (see Section 6.6). However, the drawback of this solution is that, a normal loop may be detected as infinite if the number of normal loop is more than the number of duplicate messages used to detect an infinite loop. To address this issue, users need to configure the detector to set the number of duplicate messages of detecting an infinite loop.

Deterministic or non-deterministic: Workflows can be implemented as deterministic or nondeterministic models. A deterministic model is when one choice is allowed from each place or transition in the workflow, and a nondeterministic model is where there may be choices for each transition or place and conditions are placed on the edges to allow for the determination of which edge the token should choose [202]. In this work, the workflow execution is often nondeterministic. This is because the RAWLS focuses on the WsSWFs and provides an abstract distributed multi-agent model to represent scientific workflows. On one hand, the distributed

execution environment provides a number of resources. But on the other hand, it also brings problems if the required resources are unavailable. Therefore, the physical details of resources on which the tasks are performed in this thesis are only known at runtime. Alternative resources with the same effect can be dynamically chosen if one resource is unavailable. Moreover, for the workflows, whose execution is based on the latest available information (e.g., current weather), the execution of one instance of such processes might be different from another.

Consistency: In general, a language is consistent if it does not contain a contradiction. In other words, a language is consistent if there exists an interpretation that satisfies all formulas in the language. Static analysis techniques for consistency checking of workflows can analyze the control flow of individual tasks as well as the consistency of how data of the workflow is represented, collected and utilized [203]. Control flow inconsistencies are often caused by unsatisfied conditions or deadlocks that lead to complete failure of the workflow execution. Data inconsistencies are often caused by contradictory data between workflow (task) inputs and workflow (task) description. The workflow execution in this work supports access to external data via query languages and also allows for calling external procedural attachments (e.g., Java methods). Moreover, human users are allowed to conduct manual tasks or handle the unexpected exceptions. All these factors may make the workflow execution inconsistent. Since this thesis focuses on the workflow execution phase, in the RAWLS, some basic inconsistencies are often handled as dynamic exceptions at runtime, i.e., they can be solved by intelligent agents by finding alternative execution paths. Also, the RAWLS supports the infinite loop detection by counting duplicate messages passing between distributed Prova agents. If some inconsistencies still cannot be solved, human users are allowed to revise workflows or provide missing resources.

Parallel and concurrent execution: Concurrency is another significant property of computational models and increases the flexibility, performance and power of programming languages. Concurrency means that two or more computations happen within the same time frame, and they also may interact with each other. The workflow language of this work is based on CTR, which is a deductive database language that integrates concurrency, communication and database updates in a complete logic framework. Prova engine can implement the concurrent execution by concurrent rule processing. In other words, two or more processes executing concurrently can synchronize their execution by passing event messages between distributed agents. Section 7.4.3 will present the implementation of the ant identification and treatment, which involves interactions between distributed agents to complete the process.

Parallelism is related to concurrency but distinct from concurrency. Parallel execution means that two or more computations happen simultaneously. Such computations are performed on separate processors of a multi-processor machine or different machines in a network. The RAWLS of this work offers a rule-based, distributed agent system for the WsSWFs. Based on the evaluation in Section 7.1.1.1, the RAWLS supports the *Parallel Split* pattern (WCP-02), which splits a branch

into two or more parallel sub-branches. Such sub-branches can be performed on either different threads in one agent or multiple distributed agents. In addition, the RAWLS supports even more sophisticated parallel execution based on the conditions imposed on parallel sub-branches, such as *Exclusive Choice* (WCP-04) and *Multi-Choice* (WCP-06).

Distributed computation (data/knowledge): Distributed computation not only supports easy collaboration with organizations, but also offers advantages, such as high reliability and availability, high performance and local self-sufficiency. The RAWLS offers a rule-based platform for distributed agent programming. Prova rule engines are deployed as distributed inference services on Mule ESB. They have a local knowledge base and also dynamic access to external data sources and object representations.

Moreover, the interactions between distributed agents are implemented by messaging reaction rules. Such agents can be involved in completing a complex process by receiving task assignments and sending the results back. They are process-agnostic and know nothing about the process they are embedded in. Moreover, they can build choreography workflows via messaging reaction rules. Messaging reaction rules are associated with conversation identifiers, which make all tasks of a process to be performed in one conversation.

Synchronous or asynchronous communication: The RAWLS supports asynchronous communication between distributed agents. In this work, messaging reaction rules describe abstract message-driven conversations between distributed agents and represent their associated interactions via sending and receiving event messages. In other words, when an agent (requester) sends a request to another agent, say agent a, it also uses a receiving activity to wait for the answer. The receiving activity does not consume any resources but acts as a stub to allow the agent a to call back, thereby achieving asynchronous communication. The requester can choose to synchronize with the agent a immediately or go on doing its business and synchronizing later. This is crucial when the agents are engaged in long running conversations.

The RAWLS also can support synchronous communication by Mule ESB synchronous protocols, e.g., HTTP. The synchronous communication is often used between the Web-based user client and the workflow engine via HTTP. However, since scientific workflows often contain long running activates, the RAWLS employs the AJAX technology to implement an asynchronous interaction with its user client.

7.4 Use Case-Based Experimental Evaluation

Sections 7.2.1 and 7.2.2 evaluate the expressive power of the domain knowledge representation in the RbAF from a theoretical respective. Different logic programs and ontologies are examined to analyze their expressive power. To experimentally analyze the performance and demonstrate the expressive power of the domain knowledge representation in the RbAF, this section presents the implementation of three

WsSWF use cases mentioned in Section 2.5.2.

The implementation of such real-world use cases is based on the prototype system RAWLS. A detailed description of these use cases can be found in Section 2.5.2.

Note that the rule engine Prova employed in the RAWLS not only allows for calling external procedural attachments (e.g., Java methods), but also provides access to external data via query languages. Moreover, human users are allowed to involve in the workflow execution to perform manual tasks or handle unexpected exceptions. Such uncontrollable factors may bring uncertainties to the workflow execution and make workflows undecidable. For clarity, the evaluation of this section assumes that these factors are well-functioning and terminate in a finite amount of time.

7.4.1 Protein Prediction Result Analysis

The following Prova code shows the logic of an agent to analyze the protein prediction results. A main task in the logic is to analyze if any reliable GO term of a protein lies on a path in the gene ontology of a predicted GO term (see Section 2.5.2). Note that the overall process also contains tasks, such as obtaining protein annotation and selecting reliable GO terms implemented by accessing Web services provided by *Quick GO*, their implementation is omitted here for simplicity.

Listing 7.11: Protein Prediction Result Analysis

```
1 processMessage(XID,From,Primitive, proteinPredicitonAnalysis(
2             inArgs(PredictedGOTerm, ReliableGOTerms), outArgs(Result))):-
3      bound(PredictedGOTerm),
4      validatePredictedGOTerm(XID, From, PredictedGOTerm),
5      analysis(ReliableGOTerms, PredictedGOTerm, Result),
6      sendMsg(XID,esb,From, "answer", [inArgs(PredictedGOTerm, ReliableGOTerms),
7               outArgs(Result)]).

9 validatePredictedGOTerm(XID, From, PredictedGOTerm):-
10     semanticDataConnection("http://beta.sparql.uniprot.org/", Connection),
11     println(["==> Connected."]),
12     PredictedGOTerm1 = PredictedGOTerm.substring(3),
13     QueryString = '
14           PREFIX up:<http://purl.uniprot.org/core/>
15           PREFIX uniprot:<http://purl.uniprot.org/uniprot/>
16           ASK{
17               <http://purl.uniprot.org/go/$PredictedGOTerm> a up:Concept.
18           }',
19     sparql_ask(Connection, QueryString, QueryId1),
20     sparql_results(QueryId1),
21     !.

23 validatePredictedGOTerm(XID, From, PredictedGOTerm) :-
24    Result = "PredictedGOTerm is not correct.",
25    sendMsg(XID,esb,From, "answer", [inArgs(Protein), outArgs(Result)]),
26    fail().

28 analysis(ReliableGOTerms, PredictedGOTerm, Result):-
29   Onto = de.fub.csw.protein.prediction.DataProcessor.getOnto(PredictedGOTerm),
30   sparqldl_create(Engine, Onto),
31   element(GOTerm, ReliableGOTerms),
32   QueryString = '
```

```
33                  PREFIX : <http://www.geneontology.org/go#>
34                  ASK {
35                    SubClassOf(:$PredictedGOTerm, :$GOTerm)
36                  }
37          ',
38     askQuery(Engine, QueryString, Result),
39     Result = "yes",
40     !.

42  analysis(ReliableGOTerms, PredictedGOTerm, Result):-
43     Result = "no".

45  askQuery(Engine, QueryString, Result):-
46     sparqldl_ask(Engine, QueryString, QueryId),
47     sparqldl_results(QueryId),
48     Result = "yes",
49     !.

51  askQuery(Engine, QueryString, Result):-
52     Result = "no".
```

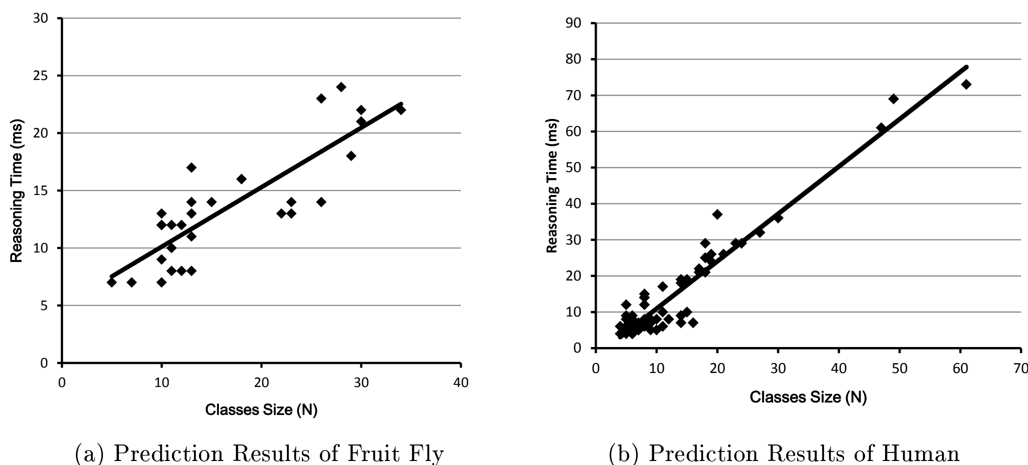
Processing the analysis request is implemented by a Prova rule *processMessage* (Line 1-7). The analysis request contains a predicted GO term of a protein and the reliable GO terms of the protein as its inputs. The predicate *validatePredictedGOTerm* checks if the predicted GO term is valid (Line 4). The *validatePredictedGOTerm* rule employs a SPARQL query (Line 13-18) to check if the given predicted GO term is an instance of the *Concept* defined in Uniprot core vocabulary [130] (i.e., *<http://purl.uniprot.org/go/$PredictedGOTerm> a up:Concept*, see Line 17). The query is performed by accessing the Uniprot SPARQL endpoint (`http://purl.uniprot.org/uniprot/`) via the Sesame API. If the given predicated GO term is invalid, an error message will be sent back to users (Line 23-26).

The prediction result analysis is conducted by analyzing if any reliable GO term of the protein is the parent class of the predicated GO term. Each GO term has a gene ontology that describes its relationships with other terms. The reasoning of the gene ontology of the predicted GO term is performed by the SPARQL-DL query engine on top of HermiT reasoner. The SPARQL-DL query asks whether class *PredictedGOTerm* is the subclass of *ReliableGOTerm* (Line 35), i.e., if class *ReliableGOTerm* is the parent class of *PredictedGOTerm*. It is a TBox query and returns *true* if *PredictedGOTerm* is a child of *ReliableGOTerm* or equivalent to *ReliableGOTerm* in the hierarchy tree. Note that the variables used in SPARQL and SPARQL-DL queries are attached with "$" at the beginning for clarity (Lines 17 and 35), since Prova can not directly concatenate strings with variables by "+" operator. In practice, Prova uses its builtin *concat/2(i, io)* to concatenate two or more strings [173].

From the LP perspective, the predicate *element(GOTerm, ReliableGOTerms)* in the first *analysis* rule iteratively gets a GO term from the list of reliable GO terms until one GO term is found as the parent class of the predicated GO term. If no GO term is found, the second *analysis* rule will be checked, and the variable *Result* is initiated to "no" (Line 42-43). The predicate is finitely recursive since the reliable GO terms of a protein are countable, thereby guaranteeing the decidability

of the analysis. Therefore, the rule *processMessage* is decidable. Additionally, the CUT(!) in the first *analysis* rule stops analyzing other GO terms if the required one is found and improves efficiency of the analysis. Note that the above logic also involves access to external data queries, which may cause failure in the execution of the process and make the workflow undecidable. Following the assumption at the beginning of this section, such uncontrollable factors are assumed to be well functioning and terminate in a finite amount of time.

As shown in Listing 7.11, the main part of the implementation is iteratively reasoning the gene ontology of a predicted GO term to check if one reliable GO term of the protein is found as the parent class of the predicted GO term. According to the description of the Gene ontology Consortium, a gene ontology describes a hierarchy of GO terms and provides an expressivity of AC^1. In other words, the time of reasoning a gene ontology depends on its size.



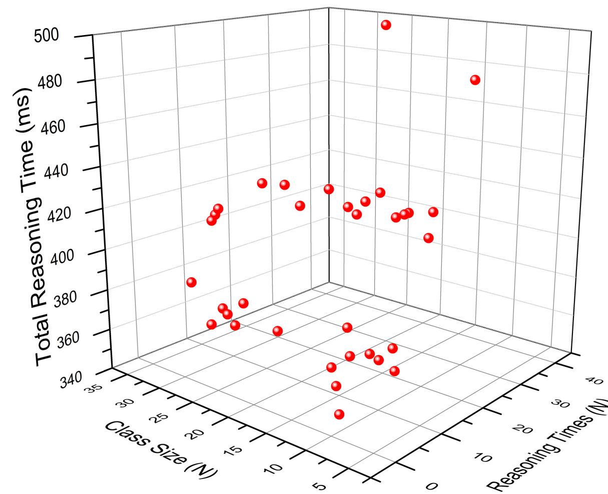
(a) Prediction Results of Fruit Fly (b) Prediction Results of Human

Figure 7.2: Complexity of Gene Ontology

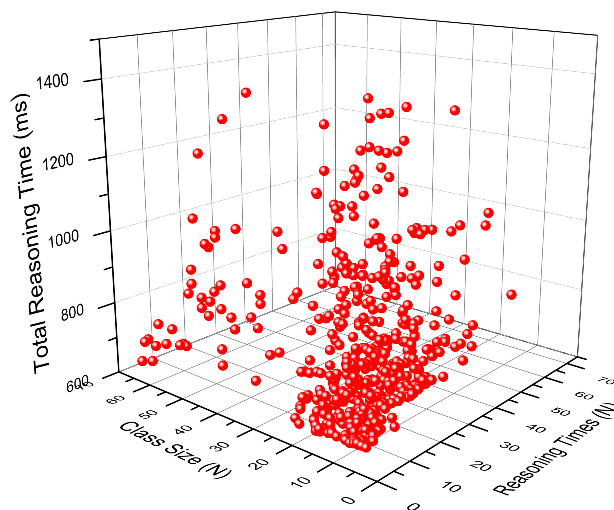
To test the complexity of reasoning a gene ontology, the protein prediction results (i.e., GO terms) of *fruit fry (drosophila melanogaster, to be more precisely)* and *human (homo sapiens, to be more precisely)* are selected. The test takes 42 and 551 prediction results of two species, respectively generated by NetCoffee [49] in an experiment. Moreover, for each gene ontology of a predicted GO term, GO term *GO:0008150* (representing a biological process) and the predicted GO term are the parent class and subclass of all other classes in the gene ontology, respectively, thus *SubClassOf(:$PredictedGOTerm, :GO:0008150)* is selected as a representative query to test a gene ontology. The test is performed on a Windows machine with a quad-core CPU and 2GB of RAM, and the test results are shown in Figure 7.2 (a) and (b), respectively.

In Figure 7.2 (a) and (b), the number of classes declared (i.e., the number of GO terms) in a gene ontology is plotted along the horizontal axis. The values of

[1] AL stands for Attribute Language and is a minimal DL language.



(a) Prediction Results of Fruit Fly



(b) Prediction Results of Human

Figure 7.3: Gene Ontology Reasoning Analysis

reasoning time are plotted on the vertical axis. Based on the scatter diagram of Figure 7.2, there is a definite relationship between the class size and the reasoning time, indicating that as the number of GO terms of a gene ontology increases, the time used to reason the gene ontology also increases. This is easy to understand, since there are more classes in a gene ontology, a reasoner needs more time to check if the predicted GO term is the subclass of such classes.

The gene ontology reasoning in the analysis to both prediction results of two species is summarized in Figure 7.3 (a) and (b), respectively. It shows how both the class size of a gene ontology (X) and the reasoning times (Y) affect the analysis of a prediction result, more precisely, the overall gene ontology reasoning time in the analysis (Z). As it shown in Figure 7.3, the overall gene ontology reasoning time in each analysis depends on both the size of the gene ontology of a predicted GO term and how many times the ontology is reasoned. The more classes a gene ontology has and more it is reasoned, the longer the analysis takes. Moreover, it can be noticed that all prediction analysis instances of two species finish in a finite amount of time, i.e., the above logic program is decidable.

7.4.2 Snow Depth Data Screening

One advantage of declarative rule-based logic programming is that it simplifies the development of applications where rule-based knowledge is used for decision making. The following Prova rules implement the criteria of screening the snow depth data in the experiment with a purpose of establishing a snow depth model for the pastoral area of northern Xinjiang (see Section 2.5.2).

Listing 7.12: Snow Depth Data Screening

```
1 processMessage(XID,From,Primitive, snowSampleIdentification(inArgs(File),
2               outArgs(Result))):-
3      screenSamples(File, Result),
4      sendMsg(XID,esb,From, "answer", [snowSampleIdentification,
5               inArgs(File), outArgs(Result)]).

7 screenSamples(File, Result):-
8      findall(R,screening(File, R), RS),
9      Result = de.fub.csw.snow.model.SampleProcessing.processResults(RS).

11 screening(File, Result):-
12      fopen(File,Reader),
13      read_enum(Reader,Line),
14      List = de.fub.csw.snow.model.SampleProcessing.getSampleItems(Line),
15      Station = List.get(0),
16      Year = List.get(1),
17      Month = List.get(2),
18      Day = List.get(3),
19      DayMaxTemp = List.get(4),
20      Tb36V = List.get(5),
21      Tb18V = List.get(6),
22      Depth = List.get(7),
23      MonthAvgTemp = List.get(8),
24      sampleValidation(sample(Station, Year, Month, Day,
25                  DayMaxTemp, Tb36V, Tb18V, Depth, MonthAvgTemp), R),
26      R = 'true',
27      concat([Line, " ", R], Result).

29 sampleValidation(sample(Station, Year, Month, Day,
30             DayMaxTemp, Tb36V, Tb18V, Depth, MonthAvgTemp), Result):-
31      bound(Depth),
32      checkDepth(Depth),
33      bound(Month), bound(DayMaxTemp),
34      not(thaw(Month, DayMaxTemp)),
35      bound(Tb36V), bound(Tb18V),
```

```
36       drySnow(Tb36V, Tb18V),
37       bound(Station),
38       bound(Year),
39       bound(MonthAvgTemp),
40       not(frostLayer(Station, Year, Month, Depth, MonthAvgTemp)),
41       checkElevation(Station),
42       Result = 'true',
43       !.

45 sampleValidation(sample(Station, Year, Month, Day,
46              DayMaxTemp, Tb36V, Tb18V, Depth, MonthAvgTemp), Result):-
47       Result = 'false'.

49 frostLayer(Station, Year, Month, Depth, MonthAvgTemp):-
50       MonthAvgTemp < 10,
51       Depth > 0.5,
52       Depth < 10.

54 drySnow(Tb36V, Tb18V):-
55       Tb36V > 195.0,
56       Tb36V < 225.0,
57       Tb18V < 255.5.

59 thaw(Month, DayMaxT):-
60       Month = 3,
61       DayMaxT > 6.

63 checkTemperature(Temp) :-
64     tempMetric(TempMetric),
65     Temp < TempMetric.

67 checkDepth(Depth) :-
68     depthMetric(DepthMetric),
69     Depth >= DepthMetric.

71 checkElevation(Station) :-
72     stationElevation(Station, Ele),
73     elevationMetric(EleMetric),
74     Ele < EleMetric.

76 depthMetric(3.0).
77 tempMetric(6).
78 elevationMetric(2000).

80 stationElevation('Fuhai', 500.9).
81 stationElevation('Aletai', 735.3).
82 stationElevation('Fuyun', 823.6).
83 stationElevation('Qinghe', 1218.2).
84 ...
```

Unlike the process of analyzing protein prediction results, this process does not involve external ontology reasoning but only implements the criteria for screening snow depth data by declarative rules. The process is also implemented by a Prova rule *processMessage* (Line 1-5) to screen snow depth data. It is started with reading a file that stores the samples of snow depth. This is done by the Prova predicate *fopen(File, Reader)* that opens the file and returns a BufferedReader (i.e., *Reader*) (Line 12), which can read a document line by line. The predicate *read_enum(Reader, Line)* enumerates the lines of the file, and it produces inde-

pendent solutions, one for each line. Each line of the file stores a sample of snow depth, and the Java method *getSampleItems(Line)* processes each line and stores the parameters of a sample into a Java list (Line 14). After that, the variables denoting the parameters of a sample are initialized based on the list (15-23). Like Prova predicate *element/3*, *read_ enum(Reader, Line)* is decidable since the samples in the file are always countable.

Two rules of *sampleValidation* implement the criteria of screening snow depth data (Line 29-47). The subgoal *not(thaw(Month, DayMaxTemp))* in the first rule denotes that a valid sample must not be thaw (Line 34). It is proved if *thaw(Month, DayMaxTemp)* fails. As mentioned in Section 7.2.1, NaF is safe only when the test goal is ground. The reasoning of *not(thaw(Month, DayMaxTemp))* is safe because the arguments *Month* and *DayMaxTemp* are initiated by reading the file storing the samples, and the Prova *bound* builtin guarantees them to be bound (Line 33). In other words, *bound(Month)* fails if *Month* is a free variable. Similarly for the subgoal *not(frostLayer(Station, Year, Month, Depth, MonthAvgTemp))*, which describes that the snow must not be covered by deep frost (Line 40). The rule *drysnow* involves three mathematical expressions, and the *bound* builtin ensures the variable involved in the rule to be bound, and furthermore guarantees the decidability of the rule reasoning. Similarly for other rules, such as *thaw*, *checkTemperature*, *checkDepth* and *checkElevation*.

The Prova predicate *findall(R,screening(File, R), RS)* accumulates all solutions of the goal *screening(File, Result)* (i.e., the screening results) in the variable *RS* (Line 8), which is further processed by a Java method *processResults(RS)* (Line 9) and then sent back to users (Line 4-5). Since the external procedural attachments are decidable in terms of the assumption, then the rule *screenSamples* (Line 7-9) is decidable and so is the rule *processMessage*.

7.4.3 Ant Identification and Treatment

The ant identification and treatment process shown in Figure 2.10 involves the collaboration between *fieldworker*, *taxonomist* and *curator*, and its main process is presented as follows. The predicate *executeTask* (Line 3, 9) is responsible for performing a task and involves task allocation and invocation (that are omitted for simplicity). The *rcvMsg* predicate right after the predicate *executeTask* is used to asynchronously receive the task results. As shown in the implementation, the tasks *antIdent*, *archive* and *treatment* are performed sequentially (Line 3, 9 and 11-12). Note that the *archive* task does not have any output and its subsequent task *treatment* does not need to wait for its completion.

Listing 7.13: Main Process of Ant Identification and Treatment

```
1 workflow(XID, From, "antIdentProcess", [inArgs|Paras], outArgs(Res, Treatment)) :-
2   bound(Paras),
3   executeTask(XID, 'antIdent', [inArgs|Paras], outArgs(Res)),
4   rcvMsg(XID, esb, Agent, "answer", ['antIdent',
5               [inArgs|Paras], outArgs(Res)]),
6   bound(Res),
```

```
7   processResult(XID, From, Paras, Res),

9   executeTask(XID, 'archive',[inArgs|Paras]),

11  sendMsg(XID, esb, From, "answer", ['treatment', Res, Treatment]),
12  rcvMsg(XID, esb, Agent, "answer", ['treatment', Treatment]),

14  sendMsg(XID, esb, From, "answer",
15            ["antIdentProcess", [inArgs|Paras], outArgs(Res, Treatment)]).

17 processResult(XID, From, Paras, Res), :-
18   not(isIdentFailed(Res)), !.

20 processResult(XID, From, Paras, Res), :-
21   isIdentFailed(Res),
22   sendMsg(XID, esb, humanAgentProxy, "request", [antIdent,
23                           antIdentProcess, [inArgs|Paras], outArgs(Res)]),
24   rcvMsg(XID, esb, humanAgentProxy, "answer", [antIdent, HumanReply]),

26   sendMsg(XID, esb, From, "answer", ["antIdentProcess",
27                        [inArgs|Paras], outArgs(HumanReply)]).

29 isIdentFailed(failed).
```

The workflow execution after the identification is determined by two *processResult* rules (Line 17-27). If the identification fails, the request will be escalated to a human task and ask domain experts for help (Line 20-27). Nothing happens if the identification is successful, and the workflow execution moves forward (Line 17-18). This exclusive choice is implemented by *isIdentFailed(Res)* and *not(isIdentFailed(Res))*. *not(isIdentFailed(Res))* is safe because the predicate *bound(Res)* (Line 6) guarantees the variable *Res* to be bound before reasoning *not(isIdentFailed(Res))*.

The execution of the above process is controlled by a centralized workflow engine, which allocates tasks to agents and manages data passing between them. Among the tasks, the *antIdent* task is performed by a master agent, which manages a group of agents to perform the identification. The logic of the master agent is presented as follows. It starts with assigning the *antIdent* task to an appropriate agent (worker) in terms of the location, where the ant is discovered. After that, the master agent forwards the identification request to the worker agent and then waits for the results from it (Line 9-10).

Listing 7.14: Ant Identification Management

```
1 processMessage(XID,From,Primitive, antIdent([inArgs|Paras], outArgs(Res))):-
2   antIdentAllocation(XID,Paras, Res),
3   sendMsg(XID,esb,From, "answer", [antIdent, [inArgs|Paras], outArgs(Res)]).

5 antIdentAllocation(XID, Paras, Res):-
6   last(Location,Paras),
7   allocate(Location, Agent),
8   !,
9   sendMsg(XID,esb, Agent, "request", antIdent(Paras, Res)),
10  rcvMsg(XID, esb, Agent, "answer", antIdent(Res)).

12 antIdentAllocation(XID, Paras, Res):-
13   Res = "failed".
```

```
15 allocate("Germany", antIdentAgentGermany).
16 allocate("UK", antIdentAgentUK).
17 allocate("Poland", antIdentAgentPoland).
18 ...
```

The predicate *last(Location,Paras)* (Line 6) obtains the last element of the given *Para* list to initialize the variable *Location*. The *Para* list is provided by a field-worker, who has to describe the discovered ant and specify the location of discovered ants as the last element of the list. The predicate *last* is implemented as follows:

Listing 7.15: Obtaining the Last Element of a Prova List

```
1 last(L,[L]).
2 last(L,[H|T]) :-
3         last(L,T).
```

It is an FP2 program. A recursion pattern π can be obtained by mapping *last* on their second argument, i.e., $\pi_{last} = \{2\}$. Both rules are call-safe, because in each of them all variables of the rule occur either in the selected argument or in the rule body with respect to π.(see Definition 5.4 of [69]). In other words, a goal *last(L, List)* is call-safe with regard to π iff *List* is bound. The Prova builtin *bound(Paras)* in Listing 7.13 (Line 2) guarantees the variable *Paras* to be bound, and thus *last(Location,Paras)* is decidable. The subgoal *allocate(Location, Agent)* (Line 7) denotes a query of finding a worker agent in terms of the given location. The query is performed by matching the *allocate* rules and goals with unification. It can be regarded as a simple Datalog program, and thus it is decidable.

The following Prova code presents the specific logic of ant identification implementation in a worker agent. The identification is successful if the body feature, nest structure and food preference of an ant can be matched to the facts in the knowledge base. Note that the facts in the knowledge base are summarized in terms of the description in [43]. The rule *antIdent(Res, Paras)* does not involve negation and function, and can be regarded as a decidable Datalog program. If the identification fails, the variable *Res* will be initialized to *failed* (Line 11-13).

Listing 7.16: Ant Identification

```
1 processMessage(XID,From,Primitive, antIdent(Paras, Res)):-
2   antIdent(Res, Paras),
3   sendMsg(XID,esb,From, "answer", antIdent(Res, Paras)).

5 antIdent(Res, [Nest, Node, Color, Thorax, Seg, Club,
6            Des, HColor, S, Worker, Length, Food, Location]) :-
7   bodyFeature(X, Node, Color, Thorax, Seg, Club, Des, HColor, S, Worker, Length),
8   nest(Res,Nest),
9   food(Res, Food), !.

11 antIdent(Res, [Nest, Node, Color, Thorax, Seg, Club,
12            Des, HColor, S, Worker, Length, Food, Location]) :-
13   Res = "failed".

15 % facts
16 node(argentineAnt, one).
17 node(littleBlackAnt, two).
```

```
19 color(argentineAnt, lightbrown).
20 color(littleBlackAnt, black).

22 thorax(argentineAnt, uneven).
23 thorax(littleBlackAnt, uneven).

25 antennae(argentineAnt,twelve,zero).
26 antennae(littleBlackAnt,twelve,three).

28 hair(argentineAnt, sparse, _).
29 hair(littleBlackAnt, _, _).

31 stinger(argentineAnt, no).
32 stinger(littleBlackAnt, indeterminate).

34 worker(argentineAnt, monomorphic).
35 worker(littleBlackAnt, monomorphic).

37 length(argentineAnt, 0.125,0.125).
38 length(littleBlackAnt, 0.0625,0.0625).

40 nest(argentineAnt, moist).
41 nest(littleBlackAnt, fineSoil).

43 food(argentineAnt, sweets).
44 food(argentineAnt, proteins).
45 food(littleBlackAnt, grease).
46 food(littleBlackAnt, vegetable).
47 food(littleBlackAnt, insects).

49 % rules
50 bodyFeature(Res, Node, Color, Thorax, Seg, Club, Des, HColor, S, Worker, Length) :-
51    node(Res, Node),
52    color(Res, Color),
53    thorax(Res, Thorax),
54    antennae(Res, Seg, Club),
55    hair(Res, Des, HColor),
56    stinger(Res, S),
57    worker(Res, Worker),
58    length(Res, Min1, Max1),
59    Length >= Min1,
60    Length <= Max1.
```

The following code presents the logic of finding ant treatments. However, the treatments will be sent to the fieldworkers, who often know nothing about the knowledge encoded by declarative rules. To overcome this issue, a webpage providing the ant description and the ant treatments will be sent back to the fieldworkers. Finding the webpage is implemented by proving the subgoal *treatment(Ant, Treatment)* with unification. Like *allocate(Location, Agent)*, it is decidable.

Listing 7.17: Finding Ant Treatments

```
1 processMessage(XID,From,Primitive, treatment(Ant, Treatment)):-
2   treatment(Ant, Treatment),
3   !,
4   sendMsg(XID,esb,From, "answer", treatment(Treatment)).

6 processMessage(XID,From,Primitive, treatment(Ant, Treatment)):-
7   sendMsg(XID,esb,From, "answer", treatment("unknown")).
```

```
9 treatment(argentineAnt, "http://en.wikipedia.org/wiki/Argentine_ant").
10 treatment(littleBlackAnt, "http://en.wikipedia.org/wiki/Little_black_ant").
11 ...
```

To sum up, different workflows have different forms of logic, and it is impossible to evaluate the presented rule-based workflow language by analyzing every real-world workflow. The expressiveness and decidability of other workflows can be analyzed similarly with the same way presented in this section. Prova itself is undecidable because it has unrestricted functions and external procedural attachments. But users can use the Prova features, such as the *bound* builtin, to make decidable logic programs (e.g., FP2 programs), thereby guaranteeing the decidability of their decision logic (see Section 7.4.2).

7.5 System Performance Evaluation

The implementation of the RAWLS is based on Mule ESB, which integrates distributed rule-based agents to create sophisticated workflows. The communication between such distributed Prova agents is based on JMS transport protocol. This section evaluates the message passing overhead during the communication and the system concurrency.

7.5.1 Message Passing Overhead

In the RAWLS, Mule ESB integrates distributed Prova agents and uses JMS transport protocol for their communication. The JMS messages are managed by Apache ActiveMQ, which is an open source message broker.

To evaluate the communication overhead of the RAWLS, a set of workflows consisting of different number of tasks is selected to simulate different communication complexity. In the RAWLS, the workflow engine can allocate a task to either one agent or a group of agents. For simplicity, the experiment of this section assumes that a task is performed by one agent, which communicates twice with the workflow engine, namely, the workflow engine sends a task request (message) to the agent, which sends the task results (message) back to the workflow engine afterwards. Moreover, since the event messages passing between distributed Prova agents of the RAWLS carry either primitive data types or the logical pointers of large data (see Section 4.3.2), the experiment assumes that the size of each message passing between the agents is 2Kb, and that each agent takes a constant time (0.01 seconds) to perform each task in the experiment. The communication overhead of a workflow is then calculated as follows:

$$T_{comm} = T_{wf} - N * 0.01$$

Here, N denotes the number of tasks in the workflow; T_{wf} denotes the workflow execution time; and T_{comm} denotes the message passing overhead of the workflow execution. Note that, the execution time of each task (i.e., 0.01 seconds) is deducted

from the workflow execution time and has nothing to do with the calculation of the message passing overhead.

Moreover, since the execution of a workflow may involve the agents deployed across a WAN, the simulation workflows are performed separately on a WAN and a LAN, respectively in the experiment. The workflow execution is managed by a Windows machine (one quad-core CPU and 2GB of RAM) located at Free University of Berlin. To simulate the resources on a LAN and a WAN, a Linux server (two single-core CPUs and 2 GB of RAM) located at Free University of Berlin and a server (two single-core CPUs and 2 GB of RAM) located at Lanzhou University (China) are employed, respectively to deploy the agents used to perform the workflow tasks. The experiment simulation data and results are summarized in Table 7.5.

Table 7.5: Data Sets of Communication Overhead Evaluation

Number of tasks (N)	**Messaging times (N)**	**Workflow execution time (LAN) (s)**	**Communication overhead (LAN) (s)**	**Workflow execution Time (WAN) (s)**	**Communication overhead (WAN) (s)**
25	50	0.723	0.473	27.31	27.205
50	100	1.3	0.8	55.552	55.322
100	200	2.421	1.421	112.123	111.643
250	500	5.701	3.201	286.482	285.252
500	1000	10.551	5.551	571.232	568.752
1000	2000	39.8	29.8	1000.638	995.658

As shown in Table 7.5, the communication overhead increases as the number of messages passing between agents increases. Moreover, the communication overhead during the workflow execution is much low if a workflow is executed on the LAN. A workflow consisting of 1000 tasks takes only 29.8 seconds on the message passing, and the same workflow executed on the WAN, which takes 995.658 seconds. In other words, the communication overhead for each task in the LAN and the WAN is about 0.0298 and 0.996 seconds, respectively, which are low compared to the real-world workflow execution itself.

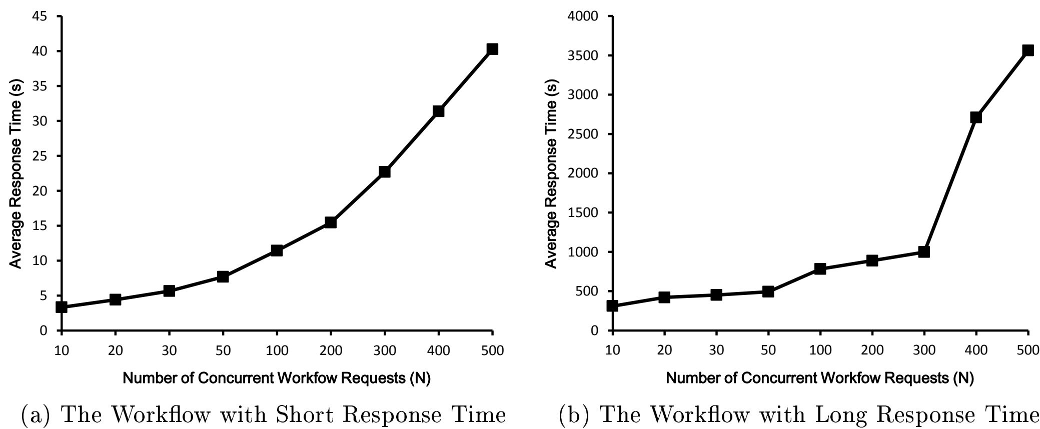
Note that the communication overhead in a real-world environment is more complicated than in the simulation experiment of this section. In a real-world environment the communication overhead is often affected by other factors, such as network bandwidth.

7.5.2 System Concurrency

Mule ESB has an SEDA architecture, which decomposes a complex, event-driven application into a set of stages connected by event queues. SEDA not only decouples event and thread scheduling from application logic, but also avoids the high overhead associated with thread-based concurrency models [179]. This section evaluates the

concurrency of the RAWLS, i.e., how the RAWLS scales with varying number of workflow requests.

In the evaluation, the average response time of all workflows is used as a metric to evaluate the concurrency of the RAWLS. Here, the response time refers to the amount of time taken by the RAWLS to process a workflow request, i.e., the workflow execution time. Two workflows which take 3.3 and 301.8 seconds, respectively are selected to represent workflows with short and long response time. During the experiment, they are continuously called by different number of workflow requests, and corresponding response time (i.e., the workflow execution time) is recorded. The experiment results are shown in Figure 7.4 (a) and (b), respectively.



(a) The Workflow with Short Response Time (b) The Workflow with Long Response Time

Figure 7.4: Concurrency with Increasing Number of Workflow Requests

As shown in Figure 7.4 (a) and (b), with the number of concurrent workflow requests increases, the average response time increases slightly. In other words, the RbAF is capable of not only processing multiple concurrent workflow requests, but also having good elasticity and scalability. For the workflow that has a long response time (301.8 seconds), after the number of concurrent requests is more than 300, the average response time grows faster. This is because at this point the performance of the RAWLS begins to saturate, in turn, leads to a decline in the throughput.

Because of the limit of the test instrument (a Linux machine with two single-core CPUs and 2 GB of RAM), concurrent workflow requests more than 500 are not considered in the experiment. But the RAWLS can handle more workflow requests by adding more processing power in practice.

7.6 Summary

This chapter presented a detailed evaluation of the RAWLS from different perspectives. First, based on the workflow pattern-based evaluation, the rule-based workflow language of the RAWLS showed a higher expressivity than other three prominent scientific workflow systems. Besides the basic control-flow patterns, in

particular, the RAWLS has superiority over other three systems to support the advanced branching and synchronization patterns, the state-based patterns and the trigger patterns. With respect to the data patterns, the RAWLS supports all external data interaction patterns thanks to messaging reaction rules. Moreover, with the combination of messaging reaction rules and derivation rules, the RAWLS can easily specify preconditions and postconditions associated with workflow tasks.

In the RAWLS, the (domain-specific) decision logic of the WsSWFs is mainly expressed by both derivation rules and Semantic Web ontologies. Based on the evaluation from the LP perspective, the results showed that the general (domain-specific) decision logic of the WsSWFs can be represented by normal logic programs, which support NaF and are more expressive than propositional and definite logic programs. From the DL perspective, the SPARQL-DL query engine integrated in the RAWLS provides an expressive DL query language and acts as an interface to every ontology reasoner that supports OWL API. The RAWLS employs HermiT, which is a Java-based OWL reasoner for the DL SHOIQ+, to be a real reasoner behind the SPARQL-DL query engine to reason domain ontologies.

This chapter also evaluated the RAWLS in terms of typical properties of computational models. The RAWLS supports most of the properties, including different forms of execution cycles, non-deterministic execution branches, parallel and concurrent execution, distributed computation and asynchronous (synchronous) communication. The only limitation is that the RAWLS does not provide mechanisms to check the workflow consistency at design time but handles some basic inconsistencies as dynamic exceptions at runtime. An experimental evaluation based on three real-world WsSWF use cases was also given to analyze the performance and demonstrate the expressive power of the domain knowledge representation in the RbAF. Moreover, the system performance evaluation at the end of this chapter showed that the Mule-based RAWLS has low message passing overhead and supports effective concurrency.

Part IV

Conclusion

Chapter 8

Conclusion and Outlook

Contents

8.1 Summary 175
8.2 Outlook 176

Scientific workflows accelerate the pace of scientific experiments in different disciplines and have attracted a great deal of interest to liberate scientists from tedious and time-consuming operations in their experiments. Driven by the explicit benefits of automating large-scale and complex scientific processes, many research efforts have been put into orchestrated and structured scientific workflows, which have fixed logic and are executed frequently with different datasets or varying parameters. Instead of focusing on such efficiency-critical structured processes, this thesis explicitly considered the WsSWFs, which require high expressiveness with respect to both processes and decision activities depending on domain-specific knowledge. They are more error-prone and have manual tasks that need to be conducted by human users.

For the purpose of supporting the WsSWFs, this thesis presented a rule-based, agent-oriented framework RbAF, which combines the declarative programming using rules with the agent technology to support them.

8.1 Summary

The main contribution of this thesis is a rule-based, agent-oriented framework that addresses the requirements of the WsSWFs. The corresponding solutions are summarized as follows:

(i) **Declarative rule-based scientific workflow language combining messaging reaction rules and derivation rules**: messaging reaction rules specify workflow processes in terms of message-driven conversations between parties and describe their associated interactions via sending and receiving event messages asynchronously. In particular, with the combination of messaging reaction rules and derivation rules, it is possible to reason over events, actions and their effects. In addition, a CTR-based formal semantics which precisely defines the rule-based workflow language is presented. Based on the workflow pattern-based evaluation, the RbAF shows higher expressive power than other three considered scientific workflow systems.

(ii) **Expressive (domain-specific) decision logic description combining LP and DL**: on one hand, the RbAF represents domain-specific knowledge by derivation rules, which are more expressive than typical simple gateways. Generally, different workflows have different logic. The evaluation results from the logic programming perspective show that, general (domain-specific) decision logic can be represented by *normal logic programs* which support NaF and are more expressive than propositional and finite logic programs. On the other hand, the RbAF provides different flexible ways to access domain data encoded by Semantic Web technologies: outsourcing RDF data storage and querying to an extensible and configurable framework; initializing the workflow variables with concepts defined in external ontologies; reasoning domain ontologies with an expressive DL query language using SPARQL syntax.

(iii) **Distributed inference agents as an adaptive workflow execution environment**: the RbAF employs distributed inference agents as the workflow execution environment. Distributed agents are deployed as inference services on Mule ESB, and each agent has a local knowledge base and also provides dynamic access to external data sources and object representations. Moreover, the RbAF combines two ways of the workflow composition: *orchestration* and *choreography*, which support both centralized workflow execution and peer-to-peer conversation-based interactions. As to the workflow exception handling, first, declarative rules have inherent advantages in specifying alternative execution paths. The RbAF can replace exceptional resources by reasoning the workflow ontology which structures all resources that are used in the workflow execution. Moreover, the asynchronous human interaction enables human users to deal with unexpected exceptions at runtime.

(iv) **Asynchronous human interaction**: the RbAF implements the interactions between distributed agents via asynchronously sending and receiving event messages. The asynchronous communication between agents benefits long running conversations and nested sub-conversations, where the requesters do not have to waste resources for waiting for a reply. Moreover, the conversation identifiers carried by event messages keep all tasks of a process instance running in one conversation. To support asynchronous human interaction, a human agent managing the life cycle of human tasks is employed. On one side, a human task requester freezes the current execution context and asynchronously waits for the results from the human agent. On the other side, scientists operate on human tasks when they are available and call back the requester to resume its execution.

8.2 Outlook

The rule-based workflow specification of this thesis has the advantages of flexibility and expressiveness, but it also brings difficulties to the workflow modeling. Scientific workflows are normally composed by scientists themselves—experts in their specific domains, who are responsible of both workflows modeling and domain decision ex-

pression. A common solution to this problem is providing graphical user interfaces to facilitate domain experts to compose scientific workflows through dragging and dropping workflow components. However, the flexibility and expressive power of declarative rules are weakened in this way. A better solution would be reducing the complexity of rule-based workflow specifications by providing a powerful workflow editor with advanced capabilities to ease the workflow definition, such as syntax highlighting, context sensitive content assist, syntax error indicator, code completion and template navigation. As a programming language, Prova employed in the RAWLS has an editor that can be provided as an Eclipse plugin and supports programming Prova rules within Eclipse IDE. However, the current version of Prova editor only supports simple syntax highlighting and error indicator, and much more needs to be done to improve user experience.

Provenance is also another important requirement of scientific workflows. Although the RbAF provides an expressive workflow description and supports a flexible workflow execution, the workflow provenance is not supported. Provenance provides human users with an explanation of the workflow execution and ensures workflows can be reproduced and extended. However, provenance is a broad standalone topic in itself, and this thesis does not consider it as a main research question. In the RAWLS, only the workflow exceptions are recorded at runtime. Although most of them can be handled by the rule-based agents automatically, they are useful for users to improve the workflow system during the downtime. Open Provenance Model (OPM) is a generic provenance model and provides an interchangeable format between heterogeneous provenance systems. Since OPM was devised, there are existing SWFMSs that have enhanced their provenance sub-systems to support OPM. The RbAF describes task dependencies by messaging reaction rules, and it is possible to employ OPM to record provenance information at a task level, although some efforts are required. However, the WsSWFs involves complex domain-specific decision logic and searching solutions of a goal is often performed by unification and backtracking. It is therefore necessary to record the derivation history of such decision logic and visually present decision procedures to domain experts.

Part V

Appendix

Appendix A

Zusammenfassung

Bestehende Lösungen für Geschäftsabläufe sowie wissenschaftliche Workflows konzentrieren sich hauptsächlich auf die orchestrierte und vorstrukturierte Ausführung rechenintensiver und datenorientierter Aufgaben. Im Gegensatz hierzu werden in der vorliegenden Arbeit ausdrücklich schwach strukturierte wissenschaftliche Workflows (WsSWFs) betrachtet, diese benötigen nicht nur eine aussagekräftige Prozess und (domänenspezifische) Spezifikation der zugrundeliegenden Entscheidungslogik. Vielmehr erfordern sie auch flexible Ausführungspfade und menschliche Interaktion.

Das Hauptforschungsproblem in dieser Arbeit ist die Kombination von regelbasierter Wissensrepräsentation mit Agenten-Technologie zum Zweck der Unterstützung der Ausführung von WsSWFs aus technischer Sicht, und ein regelbasiertes Agenten-orientiertes Framework (RbAF) wird vorgeschlagen.

Die erste Herausforderung besteht darin, Arbeitsabläufe durch deklarative Regeln zu beschreiben. Diese Arbeit verwendet Messaging Reaction Rules, die über globale Ereignis-Bedingung-Aktion (ECA) Regeln hinausgehen und die lokale Durchführung komplexer Aktionen in bestimmten Kontexten unterstützen. Die zweite Herausforderung besteht in der Beschreibung (domänenspezifischer) Entscheidungslogik in Workflows. Diese Arbeit behandelt das Problem durch die Kombination von Logik-Programmierung (LP) und Description Logic (DL). Die dritte Herausforderung ist es, die von den WsSWFs erforderliche Flexibilität zu unterstützen. Das RbAF setzt verteilte regelbasierte Agenten als Workflow-Ausführungsumgebung ein und unterstützt asynchrone Interaktion zwischen verteilten Agenten. Darüber hinaus kombiniert das RbAF zwei Möglichkeiten der Workflow-Komposition: Orchestrierung und Choreographie. Ein weiterer Mechanismus ist die flexible Ausnahmebehandlung zur Laufzeit auf Basis einer Workflow-Ontologie zur Strukturierung der Workflow-Ressourcen. Eine weitere Herausforderung, die in dieser Arbeit aufgegriffen wurde, ist die Integration von menschlichen Benutzern in die Workflow-Ausführung. Das in dieser Arbeit entwickelte Framework definiert neben autonomen Agenten einen menschlichen Agenten zur Verwaltung des Lebenszyklus der menschlichen Aufgaben in Form einer Web-Schnittstelle für die Interaktion von Wissenschaftlern mit dem System.

In dieser Arbeit wurde das RbAF aus verschiedenen Perspektiven evaluiert. Es konnte gezeigt werden, dass die regelbasierte Spezifikation von Workflows im Vergleich zu drei bekannten Workflowsystemen die Definition ausdrucksstärkerer Workflow-Muster ermöglicht. In Bezug auf die Repräsentation von Domänenwissen zeigen die Ergebnisse der Analyse, dass allgemeine (domänenspezifische) Entscheidungslogik in den WsSWFs durch normale Logikprogramme repräsentiert werden kann. Eine ausdrucksstarke Abfragesprache für DL wurde eingesetzt, und verschiedene Reasoner können leicht im RbAF konfiguriert werden. Im Sinne einer empirischen Evaluation unterstützt das RbAF die meisten der typischen Eigenschaften von Rechenmodellen. Eine experimentelle Auswertung basierend auf drei realen Anwendungsfällen für WsSWFs wurde ebenfalls durchgeführt, um die Performanz zu analysieren und die Ausdruckskraft der Repräsentation von Domänenwissen im RbAF zu demonstrieren. Zusammenfassend lässt sich sagen, dass das RbAF sowohl die ausdruckstarke Beschreibung sowie eine flexible Ausführung wissenschaftlicher Workflows unterstützt, und somit die Anforderungen an WsSWFs (außer Herkunft) erfüllt.

Appendix B

About the Author

Zhili Zhao received his master degree in Computer Science from Lanzhou University in 2009. Supported by China Scholarship Council (2010-2014), he joined in Corporate Semantic Web group of Freie Universität Berlin led by Prof. Dr. Adrian Paschke in 2010. His research interests include scientific workflow management, business process management, knowledge representation, logic programming and Semantic Web.

Bibliography

[1] I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields, *Workflows for e-Science: Scientific Workflows for Grids*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006. 1

[2] "Taverna Workflow Management System." http://www.taverna.org.uk/. Accessed: 2010-12-15. 1, 13, 47

[3] M. Sonntag, D. Karastoyanova, and E. Deelman, "Bridging the Gap between Business and Scientific Workflows," in *Proceedings of the IEEE 6th International Conference on e-Science, Brisbane, Australia, December 7-10, 2010*, pp. 206–213, IEEE Computer Society, December 2010. 1, 3, 48, 80

[4] A. Barker and R. G. Mann, "Flexible Service Composition," in *Cooperative Information Agents X, 10th International Workshop, CIA 2006, Edinburgh, UK, September 11-13, 2006, Proceedings* (M. Klusch, M. Rovatsos, and T. R. Payne, eds.), vol. 4149 of *LNCS*, pp. 446–460, Springer, 2006. 2, 41

[5] M. Frincu and C. Craciun, *Dynamic and Adaptive Rule-Based Workflow Engine for Scientific Problems in Distributed Environments*, ch. 10, pp. 227–251. CRC Press, 2010. 2, 44, 45, 50

[6] G. Papavassiliou, G. Mentzas, and A. Abecker, "Integrating Knowledge Modelling In Business Process Management," in *ECIS*, pp. 851–861, 2002. 2, 48

[7] K. Görlach, M. Sonntag, D. Karastoyanova, F. Leymann, and M. Reiter, *Conventional Workflow Technology for Scientific Simulation*, pp. 323–352. Guide to e-Science, Springer-Verlag, März 2011. 3, 78

[8] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li, "Taverna: A Tool for the Composition and Enactment of Bioinformatics Workflows," *Bioinformatics*, vol. 20, pp. 3045–3054, Nov. 2004. 3, 14

[9] "Design Science Research in Information Systems and Technology ." http://www.desrist.org/desrist/. Accessed: 2012-02-08. 3

[10] V. K. Vaishnavi and W. Kuechler, Jr., *Design Science Research Methods and Patterns: Innovating Information and Communication Technology*, ch. 2. Auerbach Publications, 2007. 3

[11] U. Yildiz, A. Guabtni, and A. H. H. Ngu, "Business versus Scientific Workflows: A Comparative Study," in *Proceedings of the 2009 Congress on Services - I*, SERVICES '09, (Washington, DC, USA), pp. 340–343, IEEE Computer Society, 2009. 11

[12] T. Pothoven, "Workflow Usage in the Healthcare Environment," April 2010. 11

[13] D. Roure and C. Goble, "Supporting e-Science Using Semantic Web Technologies – The Semantic Grid," in *Semantic e-Science* (H. Chen, Y. Wang, and K.-H. Cheung, eds.), vol. 11 of *Annals of Information Systems*, pp. 1–28, Springer US, 2010. 12, 75

[14] I. Foster, Y. Zhao, I. Raicu, and S. Lu, "Cloud Computing and Grid Computing 360-Degree Compared," *2008 Grid Computing Environments Workshop*, pp. 1–10, Nov. 2008. 12

[15] J. Qin and T. Fahringer, *Scientific Workflows–Programming, Optimization, and Synthesis with ASKALON and AWDL*, ch. 1, pp. 3–13. Springer Berlin Heidelberg, 2012. 13

[16] M. K. Anand, *Managing Scientific Workflow Provenance*. PhD thesis, UNIVERSITY OF CALIFORNIA, 2010. 13

[17] J. Kästnera and E. Arnold, *When can a Computer Simulation act as Substitute for an Experiment? A Case-Study from Chemisty*. University of Stuttgart, 2011. 13

[18] R. Tolosana-Calasanz, J. A. Bañares, O. F. Rana, P. Álvarez, J. Ezpeleta, and A. Hoheisel, "Adaptive Exception Handling for Scientific Workflows," *Concurrency and Computation: Practice & Experience*, vol. 22, pp. 617–642, Apr. 2010. 14, 16, 20

[19] B. Ludäscher, M. Weske, T. McPhillips, and S. Bowers, "Scientific Workflows: Business as Usual?," in *Proceedings of the 7th International Conference on Business Process Management (BPM)* (U. Dayal, J. Eder, J. Koehler, and H. Reijers, eds.), LNCS 5701, (Ulm, Germany), 2009. 14, 16, 17, 18, 87

[20] A. Haller, E. Oren, and S. Petkov, "Survey of Workflow Management Systems," 2005. 14

[21] C. Lin, S. Lu, X. Fei, A. Chebotko, D. Pai, Z. Lai, F. Fotouhi, and J. Hua, "A Reference Architecture for Scientific Workflow Management Systems and the VIEW SOA Solution," *IEEE Transactions on Services Computing*, vol. 2, no. 1, pp. 79–92, 2009. 14, 15

[22] I. Altintas, B. Ludaescher, S. Klasky, and M. A. Vouk, "Introduction to Scientific Workflow Management and the Kepler System," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, (New York, NY, USA), ACM, 2006. 14, 46, 67

[23] S. Majithia, M. Shields, I. Taylor, and I. Wang, "Triana: A Graphical Web Service Composition and Execution Toolkit," in *Proceedings of the IEEE International Conference on Web Services*, ICWS '04, (Washington, DC, USA), pp. 514–521, IEEE Computer Society, 2004. 14, 46, 47

[24] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, and H. T. Vo, "VisTrails: Visualization Meets Data Management," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006* (S. Chaudhuri, V. Hristidis, and N. Polyzotis, eds.), pp. 745–747, ACM, 2006. 15

[25] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz, "Pegasus: A Framework for Mapping Complex Scientific Workflows onto Distributed Systems," *Scientific Programming*, vol. 13, pp. 219–237, July 2005. 15, 46

[26] Y. Zhao, M. Hategan, B. Clifford, I. T. Foster, G. von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, and M. Wilde, "Swift: Fast, Reliable, Loosely Coupled Parallel Computation," in *2007 IEEE International Conference on Services Computing - Workshops (SCW 2007), 9-13 July 2007, Salt Lake City, Utah, USA*, pp. 199–206, IEEE Computer Society, 2007. 15, 47

[27] A. Barker and J. Hemert, "Scientific Workflow: A Survey and Research Directions," in *Parallel Processing and Applied Mathematics* (R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, eds.), vol. 4967 of *Lecture Notes in Computer Science*, pp. 746–753, Springer Berlin Heidelberg, 2008. 15

[28] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana, "Business Process Execution Language for Web Services Version 1.1," tech. rep., BEA, IBM, Microsoft, SAP, Siebel, 2003. 16, 28, 40

[29] R. Barga and D. Gannon, "Scientific versus Business Workflows," in *Workflows for e-Science* (I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields, eds.), pp. 9–16, Springer London, 2007. 16, 18

[30] M. Sonntag, D. Karastoyanova, and F. Leymann, "The Missing Features of Workflow Systems for Scientific Computations," in *Proceedings of the 3rd Grid Workflow Workshop (GWW), Software Engineering Conference, GI-Edition Lecture Notes in Informatics (LNI), P-160*, pp. 209–216, Gesellschaft für Informatik e.V. (GI), February 2010. 17

[31] S. Perera and D. Gannon, "Enabling Web Service Extensions for Scientific Workflows," *2006 Workshop on Workflows in Support of LargeScale Science*, pp. 1–10, 2006. 17

[32] A. Malinova and S. Gocheva-Iliev, "Using the Business Process Execution Language for Managing Scientific Processes," *Information Technologies and Knowledge*, vol. 2, pp. 257–261, 2008. 17

[33] D. Karastoyanova, "On Scientific Experiments and Flexible Service Compositions," in *From Active Data Management to Event-Based Systems and More* (K. Sachs, I. Petrov, and P. Guerrero, eds.), vol. 6462 of *Lecture Notes in Computer Science*, pp. 175–194, Springer Berlin/Heidelberg, 2010. 17

[34] J. Rollinger, H. Stuppner, and T. Langer, "Virtual Screening for the Discovery of Bioactive Natural Products," in *Natural Compounds as Drugs Volume I* (F. Petersen and R. Amstutz, eds.), vol. 65 of *Progress in Drug Research*, pp. 211–249, Birkhäuser Basel, 2008. 18

[35] V. K. Kasam, *In Silico Drug Discovery on Computational Grid for Finding Novel Drugs Against Neglected Diseases*. PhD thesis, University of Bonn, 2009. 18

[36] E. Santos, D. Koop, H. T. Vo, E. W. Anderson, J. Freire, and C. Silva, "Using Workflow Medleys to Streamline Exploratory Tasks," in *Proceedings of the 21st International Conference on Scientific and Statistical Database Management*, SSDBM 2009, (Berlin, Heidelberg), pp. 292–301, Springer-Verlag, 2009. 20

[37] M. Caeiro-Rodriguez, T. Priol, and Z. Németh, "Dynamicity in Scientific Workflows," Tech. Rep. TR-0162, Institute on Grid Information, Resource and Workflow Monitoring Services, CoreGRID - Network of Excellence, August 2008. 20, 26

[38] D. Collins, J. Montagnat, A. Zijdenbos, A. Evans, and D. Arnold, "Automated Estimation of Brain Volume in Multiple Sclerosis with BICCR," in *Information Processing in Medical Imaging* (M. Insana and R. Leahy, eds.), vol. 2082 of *Lecture Notes in Computer Science*, pp. 141–147, Springer Berlin Heidelberg, 2001. 21

[39] S. Lu and J. Zhang, "Collaborative Scientific Workflows," *IEEE International Conference on Web Services*, vol. 0, pp. 527–534, 2009. 21, 48

[40] A. Paschke, "A Semantic Rule and Event Driven Approach for Agile Decision-Centric Business Process Management," in *Towards a Service-Based Internet* (W. Abramowicz, I. Llorente, M. Surridge, A. Zisman, and J. Vayssiére, eds.), vol. 6994 of *Lecture Notes in Computer Science*, pp. 254–267, Springer Berlin Heidelberg, 2011. 22, 44, 45, 64

[41] H. Yu, Q. Feng, X. Zhang, X. Zhang, and T. Liang, "An Approach for Monitoring Snow Depth Based on AMSR-E Data in the Pastoral Area of Northern Xinjiang," *Acta Prataculturae Sinica*, 2009. 22

[42] H. Yu, X. Zhang, W. Wang, Q. Feng, and T. Liang, "Monitoring Model and Accuracy Evaluation of Snow Depth in Qinghai Province Based on AMSR-E Data," *Arid Zone Research*, vol. 2, pp. 255–261, 2011. 23, 75

[43] B. E. Science, *Ant Identification Guide*. Bayer Environmental Science, 2010. 23, 167

[44] R. Nair and B. Rost, "Protein Subcellular Localization Prediction Using Artificial Intelligence Technology," in *Functional Proteomics* (J. Thompson, M. Ueffing, and C. Schaeffer-Reiss, eds.), vol. 484 of *Methods in Molecular Biology*, pp. 435–463, Humana Press, 2008. 23

[45] B. Rost, J. Liu, R. Nair, K. Wrzeszczynski, and Y. Ofran, "Automatic Prediction of Protein Function," *Cellular and Molecular Life Sciences CMLS*, vol. 60, no. 12, pp. 2637–2650, 2003. 23

[46] "The Gene Ontology." `http://www.geneontology.org/`. Accessed: 2013-11-28. 23

[47] R. Sharan, S. Suthram, R. M. Kelley, T. Kuhn, S. McCuine, P. Uetz, T. Sittler, R. M. Karp, and T. Ideker, "Conserved Patterns of Protein Interaction in Multiple Species," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 102, pp. 1974–1979, 2005. 24, 115

[48] "A Fast Browser for Gene Ontology Terms and Annotations." `http://www.ebi.ac.uk/QuickGO/`. Accessed: 2013-12-11. 24

[49] J. Hu, B. Kehr, and K. Reinert, "NetCoffee: A Fast and Accurate Global Alignment Approach to Identify Functionally Conserved Proteins in Multiple Networks," *Bioinformatics*, pp. 540–548, Dec. 2013. 24, 161

[50] C. Lee, B. Michel, E. Deelman, and J. Blythe, "From Event-Driven Workflows Towards a Posteriori Computing," in *Future Generation Grids* (V. Getov, D. Laforenza, and A. Reinefeld, eds.), pp. 3–28, Springer US, 2006. 25

[51] B. Cantalupo, L. Giammarino, N. Matskanis, M. Surridge, and F. Silvestri, "Semantic Workflow Representation and Samples," tech. rep., University of Southampton IT Innovation Centre, 2005. 27

[52] "Workflow Patterns." http://www.workflowpatterns.com/. Accessed: 2012-03-23. 28, 68, 86, 129, 130

[53] M. Weske, G. Vossen, and F. Puhlmann, "Workflow and Service Composition Languages," in *Handbook on Architectures of Information Systems* (P. Bernus, K. Mertins, and G. Schmidt, eds.), International Handbooks on Information Systems, pp. 369–390, Springer Berlin Heidelberg, 2006. 28

[54] W. van der Aalst and A. ter Hofstede, "YAWL: Yet Another Workflow Language," *Information Systems*, pp. 245 – 275, 2005. 28, 40

[55] T. Oinn, M. Greenwood, M. Addis, M. N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P. Lord, M. R. Pocock, M. Senger, R. Stevens, A. Wipat, and C. Wroe, "Taverna: Lessons in Creating a Workflow Environment for the Life Sciences: Research Articles," *Concurrency and Computation: Practice & Experience*, vol. 18, pp. 1067–1100, Aug. 2006. 28, 46, 47

[56] T. Allweyer, *BPMN 2.0: Introduction to the Standard for Business Process Modeling*. Books on Demand GmbH, 2010. 28

[57] M. Dumas and A. Hofstede, "UML Activity Diagrams as a Workflow Specification Language," in *UML 2001 – The Unified Modeling Language. Modeling Languages, Concepts, and Tools* (M. Gogolla and C. Kobryn, eds.), vol. 2185 of *Lecture Notes in Computer Science*, pp. 76–90, Springer Berlin Heidelberg, 2001. 28

[58] J. Sroka, J. Hidders, P. Missier, and C. Goble, "A Formal Semantics for the Taverna 2 Workflow Model," *Journal of Computer and System Sciences*, vol. 76, no. 6, pp. 490 – 508, 2010. 29

[59] "Prova Rule Language." https://prova.ws/. Accessed: 2010-10-26. 30, 105

[60] A. Paschke, "Rules and Logic Programming for the Web," in *Reasoning Web. Semantic Technologies for the Web of Data* (A. Polleres, C. d'Amato, M. Arenas, S. Handschuh, P. Kroner, S. Ossowski, and P. Patel-Schneider, eds.), vol. 6848 of *Lecture Notes in Computer Science*, pp. 326–381, Springer Berlin Heidelberg, 2011. 31, 35, 45, 72, 73

[61] T. Eiter, G. Gottlob, and H. Mannila, "Disjunctive Datalog," *ACM Transactions on Database Systems*, vol. 22, pp. 364–418, Sept. 1997. 32

[62] A. Calì, G. Gottlob, and T. Lukasiewicz, "A General Datalog-Based Framework for Tractable Query Answering over Ontologies," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 14, no. 0, pp. 57 – 83, 2012. Special Issue on Dealing with the Messiness of the Web of Data. 32

[63] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov, "Complexity and Expressive Power of Logic Programming," *ACM Computing Surveys*, vol. 33, pp. 374–425, Sept. 2001. 33, 35, 151

[64] M. Alviano, F. Calimeri, Faber Wolfgang, G. Ianni, and N. Leone, "Function Symbols in ASP: Overview and Perspectives," in *Nonmonotonic reasoning. Essays celebrating its 30th anniversary. Papers from the conference (NonMon30), Lexington, KY, USA, October 22–25, 2010.*, pp. 1–24, London: College Publications, 2011. 33, 151, 152

[65] T. Syrjänen, "Omega-Restricted Logic Programs," in *Logic Programming and Nonmotonic Reasoning* (T. Eiter, W. Faber, and M. Truszczyński, eds.), vol. 2173 of *Lecture Notes in Computer Science*, pp. 267–280, Springer Berlin Heidelberg, 2001. 33

[66] M. Gebser, T. Schaub, and S. Thiele, "GrinGo: A New Grounder for Answer Set Programming," in *Logic Programming and Nonmonotonic Reasoning* (C. Baral, G. Brewka, and J. Schlipf, eds.), vol. 4483 of *Lecture Notes in Computer Science*, pp. 266–271, Springer Berlin Heidelberg, 2007. 33, 151

[67] F. Calimeri, S. Cozza, G. Ianni, and N. Leone, "Computable Functions in ASP: Theory and Implementation," in *Logic Programming* (M. Garcia de la Banda and E. Pontelli, eds.), vol. 5366 of *Lecture Notes in Computer Science*, pp. 407–424, Springer Berlin Heidelberg, 2008. 33

[68] Y. Lierler and V. Lifschitz, "One More Decidable Class of Finitely Ground Programs," in *Logic Programming* (P. Hill and D. Warren, eds.), vol. 5649 of *Lecture Notes in Computer Science*, pp. 489–493, Springer Berlin Heidelberg, 2009. 33

[69] S. Baselice and P. a. Bonatti, "A Decidable Subclass of Finitary Programs," *Theory and Practice of Logic Programming*, vol. 10, pp. 481–496, July 2010. 33, 152, 167

[70] F. Calimeri, S. Cozza, G. Ianni, and N. Leone, "Magic Sets for the Bottom-Up Evaluation of Finitely Recursive Programs," in *Logic Programming and Nonmonotonic Reasoning* (E. Erdem, F. Lin, and T. Schaub, eds.), vol. 5753 of *Lecture Notes in Computer Science*, pp. 71–86, Springer Berlin Heidelberg, 2009. 33

[71] M. Alviano, W. Faber, and N. Leone, "Disjunctive ASP with Functions: Decidable Queries and Effective Computation," *Theory and Practice of Logic Programming*, vol. 10, pp. 497–512, July 2010. 33

[72] P. A. Bonatti, "Reasoning with Infinite Stable Models," *Artificial Intelligence*, vol. 156, no. 1, pp. 75 – 111, 2004. 33

[73] M. imkus and T. Eiter, "FDNC: Decidable Non-monotonic Disjunctive Logic Programs with Function Symbols," in *Logic for Programming, Artificial Intelligence, and Reasoning* (N. Dershowitz and A. Voronkov, eds.), vol. 4790 of *Lecture Notes in Computer Science*, pp. 514–530, Springer Berlin Heidelberg, 2007. 33

[74] T. Eiter and M. Šimkus, "Bidirectional Answer Set Programs with Function Symbols," in *Proceedings of the 21st International Jont Conference on Artifical Intelligence*, IJCAI'09, (San Francisco, CA, USA), pp. 765–771, Morgan Kaufmann Publishers Inc., 2009. 33

[75] V. P. Luong, "Between Well-Founded Semantics and Stable Model Semantics.," in *IDEAS*, pp. 270–278, 1999. 34

[76] K. R. Apt, H. A. Blair, and A. Walker, *Foundations of Deductive Databases and Logic Programming*, ch. 2, pp. 89–148. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1988. 35

[77] W. Lu and U. Furbach, "Disjunctive Logic Program = Horn Program + Control Program," in *Logics in Artificial Intelligence* (J. Dix, L. n. Cerro, and U. Furbach, eds.), vol. 1489 of *Lecture Notes in Computer Science*, pp. 33–46, Springer Berlin Heidelberg, 1998. 35

[78] "Deductive, Inductive and Abductive Reasoning." `http://butte.edu/departments/cas/tipsheets/thinking/reasoning.html`. Accessed: 2014-06-15. 35, 36

[79] W. Tan and M. Zhou, *Business and Scientific Workflows: A Web Service-Oriented Approach.* IEEE Press Series on Systems Science and Engineering, Wiley, 2013. 40

[80] A. Charfi and M. Mezini, "AO4BPEL: An Aspect-Oriented Extension to BPEL," *World Wide Web*, vol. 10, pp. 309–344, 2007. 40

[81] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in *ECOOP'97 – Object-Oriented Programming* (M. Akşit and S. Matsuoka, eds.), vol. 1241 of *Lecture Notes in Computer Science*, pp. 220–242, Springer Berlin Heidelberg, 1997. 40

[82] R. Davide and T. Elisa, "What Your Next Workflow Language Should Look Like," in *Proceedings of the 2nd International Workshop on Coordination and Organization*, 2006. 40

[83] P. A. Buhler and J. M. Vidal, "Adaptive Workflow = Web Services + Agents," in *Proceedings of the International Conference on Web Services*, pp. 131–137, CSREA Press, 2003. 41

[84] A. Barker and R. G. Mann, "Agent-Based Scientific Workflow Composition," in *Astronomical Data Analysis Software and Systems XV ASP Conference Series* (C. Gabriel, C. Arviset, D. Ponz, and E. Solano, eds.), vol. 351, pp. 485–488, 2006. 41

[85] T. Wagner, "Agentworkflows for Flexible Workflow Execution," in *Proceedings of the International Workshop on Petri Nets and Software Engineering* (L. Cabac, M. Duvigneau, and D. Moldt, eds.), vol. 851 of *CEUR Workshop Proceedings*, pp. 199–214, CEUR-WS.org, 2012. 41

[86] J. Lam, F. Guerin, W. Vasconcelos, and T. J. Norman, "Building Multi-agent Systems for Workflow Enactment and Exception Handling," in *Proceedings of the 5th international conference on Coordination, organizations, institutions, and norms in agent systems*, COIN'09, (Berlin, Heidelberg), pp. 53–69, Springer-Verlag, 2010. 42, 78

[87] I. Horrocks and P. F. Patel-Schneider, "Reducing OWL Entailment to Description Logic Satisfiability," in *Journal of Web Semantics*, pp. 17–29, Springer, 2003. 42

[88] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean, "SWRL: A Semantic Web Rule Language Combining OWL and RuleML," tech. rep., World Wide Web Consortium, May 2004. 42

[89] A. Fleischmann and C. Stary, "Whom to Talk to? A Stakeholder Perspective on Business Process Development," *Universal Access in the Information Society*, vol. 11, no. 2, pp. 125–150, 2012. 42, 43

[90] A. Fleischmann, W. Schmidt, C. Stary, S. Obermeier, and E. Börger, *Subject-Oriented Business Process Management*. Springer, 2012. 43

[91] H. Weigand, W.-j. V. D. Heuvel, and M. Hiel, "Rule-Based Service Composition and Service-Oriented Business Rule Management," *Business*, pp. 1–12, 2008. 44, 45

[92] A. Paschke and K. Teymourian, "Rule-Based Business Process Execution with BPEL+," in *Proceedings of I-KNOW and I-SEMANTICS 2009* (A. Paschke, H. Weigand, W. Behrendt, K. Tochtermann, and T. Pellegrini, eds.), (Graz, Austria), pp. 588–601, Verlag der Technischen Universität Graz, September 2009. 44, 69

[93] J. Bae, H. Bae, S.-H. Kang, and Y. Kim, "Automatic Control of Workflow Processes Using ECA Rules," *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 8, pp. 1010–1023, 2004. 44

[94] D. Lin, H. Sheng, and T. Ishida, "Interorganizational Workflow Execution Based on Process Agents and ECA Rules," *IEICE - Transactions on Information and Systems*, vol. E90-D, pp. 1335–1342, Sept. 2007. 44

[95] L. Chen, M. Li, and J. Cao, "A Rule-Based Workflow Approach for Service Composition," in *Proceedings of the Third international conference on Parallel and Distributed Processing and Applications*, ISPA'05, (Berlin, Heidelberg), pp. 1036–1046, Springer-Verlag, 2005. 45

[96] A. Paschke and A. Kozlenkov, "A Rule-Based Middleware for Business Process Execution," in *Proceedings of Multi-Conference Information Systems*, 2008. 45, 46

[97] H. Boley and A. Paschke, "Rule Responder Agents Framework and Instantiations," in *Semantic Agent Systems* (A. Elçi, M. Koné, and M. Orgun, eds.), vol. 344 of *Studies in Computational Intelligence*, pp. 3–23, Springer Berlin/Heidelberg, 2011. 45

[98] A. Paschke, "Rule Responder HCLS eScience Infrastructure," in *Proceedings of the 3rd International Conference on the Pragmatic Web: Innovating the Interactive Society*, ICPW '08, (New York, NY, USA), pp. 59–67, ACM, 2008. 45

[99] A. Paschke and H. Boley, "Rule Responder: Rule-Based Agents for the Semantic-Pragmatic Web," *International Journal on Artificial Intelligence Tools*, vol. 20, no. 6, pp. 1043–1081, 2011. 45

[100] A. Paschke, A. Kozlenkov, and H. Boley, "A Homogenous Reaction Rules Language for Complex Event Processing," in *International Workshop on Event Drive Architecture for Complex Event Process*, 2007. 45

[101] Y. Gil, E. Deelman, M. H. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. A. Goble, M. Livny, L. Moreau, and J. Myers, "Examining the Challenges of Scientific Workflows," *IEEE Computer*, vol. 40, no. 12, pp. 24–32, 2007. 46

[102] I. Taylor, M. Shields, I. Wang, and A. Harrison, "The Triana Workflow Environment: Architecture and Applications," in *Workflows for e-Science* (I. Taylor, E. Deelman, D. Gannon, and M. Shields, eds.), pp. 320–339, Springer London, 2007. 47

[103] T. Fahringer, R. Prodan, R. Duan, J. Hofer, F. Nadeem, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-L. Truong, A. Villazon, and M. Wieczorek, "ASKALON: A Development and Grid Computing Environment for Scientific Workflows," in *Workflows for e-Science* (I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields, eds.), pp. 450–471, Springer, 2007. 47

[104] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, "Swift: A Language for Distributed Parallel Scripting," *Parallel Computing*, vol. 37, pp. 633–652, Sept. 2011. 47

[105] L. van Elst, F.-R. Aschoff, A. Bernardi, and S. Schwarz, "Weakly-Structured Workflows for Knowledge-Intensive Tasks: an Experimental Evaluation," in *Proceedings of the Twelfth International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pp. 340–345, June 2003. 48

[106] M. Pesic, H. Schonenberg, and W. van der Aalst, "DECLARE: Full Support for Loosely-Structured Processes," in *Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference*, pp. 287–298, Washington, DC, USA: IEEE Computer Society, 2007. 49, 102

[107] "OWL-S: Semantic Markup for Web Services." `http://www.w3.org/Submission/OWL-S/`. Accessed: 2012-03-18. 50, 53, 54

[108] I. Niles and A. Pease, "Origins of the IEEE Standard Upper Ontology," in *Working Notes of the IJCAI-2001 Workshop on the IEEE Standard Upper Ontology*, pp. 4–10, 2001. 51

[109] T. Vitvar, J. Kopecky, J. Viskova, and D. Fensel, "WSMO-Lite Annotations for Web Services," in *Proceedings of the 5th European Semantic Web Conference* (M. Hauswirth, M. Koubarakis, and S. Bechhofer, eds.), LNCS, (Berlin, Heidelberg), Springer Verlag, June 2008. 51

[110] "Web Service Semantics - WSDL-S." `http://www.w3.org/Submission/WSDL-S/`. Accessed: 2012-03-18. 51

[111] "Semantic Annotations for WSDL." `http://www.w3.org/2002/ws/sawsdl/`. Accessed: 2012-03-18. 51

[112] "Web Service Modeling Ontology." `http://www.w3.org/Submission/WSMO/`. Accessed: 2012-03-18. 51

[113] D. Fensel and C. Bussler, "The Web Service Modeling Framework WSMF," *Electronic Commerce Research and Applications*, vol. 1, no. 2, pp. 113 – 137, 2002. 51

[114] "Semantic Web Services Framework (SWSF) Overview." `http://www.w3.org/Submission/SWSF/`. Accessed: 2012-03-18. 51

[115] "Semantic Web Services Language." `http://www.w3.org/Submission/SWSF-SWSL/`. Accessed: 2012-03-18. 51

[116] "Semantic Web Services Ontology." `http://www.w3.org/Submission/SWSF-SWSO/`. Accessed: 2012-03-18. 51

[117] "WSMO-Lite: Lightweight Semantic Descriptions for Services on the Web." `http://www.w3.org/Submission/WSMO-Lite/`. Accessed: 2012-03-18. 51

[118] A. Paschke, *Rule Based Service Level Agreements: RBSLA; Knowledge Representation for Automated E-contract, SLA and Policy Management*. PhD thesis, Technical University Munich, 2007. 52

[119] A. Paschke and M. Bichler, "Knowledge Representation Concepts for Automated SLA Management," *Decision Support Systems*, vol. 46, pp. 187–205, Dec. 2008. 52

[120] M. Lee, S. S. Park, and J.-W. Lee, "Ontology-Based Service Layering for Facilitating Alternative Service Discovery," in *Proceedings of the 2nd international conference on Ubiquitous information management and communication*, ICUIMC '08, (New York, NY, USA), pp. 465–470, ACM, 2008. 52

[121] J. Qin and T. Fahringer, *Scientific Workflows–Programming, Optimization, and Synthesis with ASKALON and AWDL*, ch. 7, pp. 115–134. Springer Berlin Heidelberg, 2012. 52

[122] P. Pinheiro da Silva, L. Salayandia, and A. Q. Gates, "WDO-It! A Tool for Building Scientific Workflows from Ontologies," tech. rep., University of Texas at El Paso, El Paso, TX, September 2007. 53

[123] N. Matskanis, M. Surridg, J. Ferris, and B. Cantalupo, "Adaptive Workflow Technology," tech. rep., University of Southampton IT Innovation Centre, 2006. 54

[124] S. Beco, B. Cantalupo, L. Giammarino, N. Matskanis, and M. Surridge, "OWL-WS: A Workflow Ontology for Dynamic Grid Service Composition," in *Proceedings of the First International Conference on e-Science and Grid Computing*, vol. 46, pp. 148–155, July 2005. 54

[125] "NextGRID." `http://www.it-innovation.soton.ac.uk/projects/nextgrid`. Accessed: 2014-7-24. 54

[126] W. Ren, G. Chen, Z. Yang, J. Zhou, J.-B. Zhang, C. P. Low, D. Chen, and C. Sun, "Semantic Enhanced Rule Driven Workflow Execution in Collaborative Virtual Enterprise," in *10th International Conference on Control, Automation, Robotics and Vision, ICARCV 2008, Hanoi, Vietnam, 17-20 December 2008, Proceedings*, pp. 910–915, IEEE, 2008. 55

[127] M. Grüninger and C. Menzel, "The Process Specification Language (PSL) Theory and Applications," *AI Magazine*, vol. 24, pp. 63–74, Sept. 2003. 55

[128] Wooldridge, Michael, *An Introduction to MultiAgent Systems.* Wiley Publishing, 2nd ed., 2009. 59, 60

[129] "Semantic Web." `http://www.w3.org/standards/semanticweb/`. Accessed: 2011-08-11. 62, 75

[130] "UniProt Core Vocabulary." `http://www.uniprot.org/core/`. Accessed: 2014-01-08. 64, 113, 160

[131] A. Paschke and A. Kozlenkov, "Rule-Based Event Processing and Reaction Rules," in *Rule Interchange and Applications* (G. Governatori, J. Hall, and A. Paschke, eds.), vol. 5858 of *Lecture Notes in Computer Science*, pp. 53–66, Springer Berlin Heidelberg, 2009. 64

[132] A. Paschke and H. Boley, "Rules Capturing Events and Reactivity," in *Handbook of Research on Emerging Rule-Based Languages and Technologies* (A. Giurca, D. Gasevic, and K. Taveter, eds.), pp. 215–252, IGI Publishing, May 2009. 64, 65

[133] "Agent Communication Language Specifications." `http://www.fipa.org/repository/aclspecs.html`. Accessed: 2012-06-17. 66, 109, 121

[134] L. Vargas, J. Bacon, and K. Moody, "Event-Driven Database Information Sharing," in *Proceedings of the 25th British national conference on Databases: Sharing Data, Information and Knowledge*, BNCOD '08, (Berlin, Heidelberg), pp. 113–125, Springer-Verlag, 2008. 67

[135] J. Mendling, "Event-Driven Process Chains (EPC)," in *Metrics for Process Models*, vol. 6 of *Lecture Notes in Business Information Processing*, pp. 17–57, Springer Berlin Heidelberg, 2009. 68

[136] N. Russell, Arthur, W. M. P. van der Aalst, and N. Mulyar, "Workflow Control-Flow Patterns: A Revised View," tech. rep., BPMcenter.org, 2006. 69, 70, 71, 72, 86, 87, 93, 94, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141

[137] A. Paschke, "Reaction RuleML 1.0 for Rules, Events and Actions in Semantic Complex Event Processing," in *Proceedings of the 8th International Web Rule Symposium*, LNCS, Springer, 2014. 69, 120, 121

[138] D. Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems.* Addison-Wesley Longman, Amsterdam, 2002. 69

[139] C. Bădică, L. Braubach, and A. Paschke, "Rule-Based Distributed and Agent Systems," in *Rule-Based Reasoning, Programming, and Applications* (N. Bassiliades, G. Governatori, and A. Paschke, eds.), vol. 6826 of *Lecture Notes in Computer Science*, pp. 3–28, Springer Berlin Heidelberg, 2011. 69

[140] D. Luckham, W. R. Schulte, J. Adkins, P. Bizarro, H.-A. Jacobsen, A. Mavashev, B. M. Michelson, P. Niblett, and D. Tucker, "Event Processing Glossary - Version 2.0," tech. rep., Event Processing Technical Society, 2011. 69

[141] E. Kindler, "On the Semantics of EPCs: Resolving the Vicious Circle," *Data & Knowledge Engineering*, vol. 56, no. 1, pp. 23 – 40, 2006. 70, 86

[142] D. Hay, "Defining Business Rules – What Are They Really." Final Report, 2000. 73

[143] A. Carusi, T. Clark, and M. S. Marshall, "Web Semantics in Action: Web 3.0 in e-Science," in *IEEE International Conference on E-Science Workshops*, 2009. 75

[144] "International Workshop on Semantic Web Applications and Tools for the Life Sciences." `http://www.swat4ls.org/`. Accessed: 2013-11-03. 75

[145] "Identifiers.org." `http://identifiers.org/`. Accessed: 2013-11-18. 75

[146] "The European Bioinformatics Institute." `http://www.ebi.ac.uk`. Accessed: 2013-11-18. 75

[147] "EBI RDF Platform." `http://www.ebi.ac.uk/rdf/`. Accessed: 2013-11-18. 75

[148] "COEUS: Streamlined Back-end Framework for Rapid Semantic Web Application Development." `http://bioinformatics.ua.pt/coeus/`. Accessed: 2013-11-18. 75

[149] T. R. Gruber, "Toward Principles for the Design of Ontologies Used for Knowledge Sharing," *International Journal of Human-Computer Studies*, vol. 43, pp. 907–928, Dec. 1995. 75

[150] B. N. Grosof, I. Horrocks, R. Volz, and S. Decker, "Description Logic Programs: Combining Logic Programs with Description Logic," in *Proceedings of the 12th International Conference on World Wide Web*, WWW '03, (New York, NY, USA), pp. 48–57, ACM, 2003. 76

[151] M. Krötzsch, *Description Logic Rules*. PhD thesis, Karlsruher Institut für Technologie, 2010. 76

[152] D. Carral Martínez and P. Hitzler, "Extending Description Logic Rules," in *The Semantic Web: Research and Applications* (E. Simperl, P. Cimiano, A. Polleres, O. Corcho, and V. Presutti, eds.), vol. 7295 of *Lecture Notes in Computer Science*, pp. 345–359, Springer Berlin Heidelberg, 2012. 76

[153] M. Arenas and J. Pérez, "Querying Semantic Web Data with SPARQL," in *Proceedings of the Thirtieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '11, (New York, NY, USA), pp. 305–316, ACM, 2011. 77

[154] "UniProt Core Ontology." `http://www.uniprot.org/core/`. Accessed: 2013-11-18. 77, 107

[155] R. Shearer, B. Motik, and I. Horrocks, "HermiT: A Highly-Efficient OWL Reasoner," in *Proceedings of the 5th International Workshop on OWL: Experiences and Directions (OWLED 2008 EU)* (A. Ruttenberg, U. Sattler, and C. Dolbear, eds.), (Karlsruhe, Germany), October 26–27 2008. 78, 115, 155

[156] "SPARQL-DL API." `http://www.derivo.de/ressourcen/sparql-dl-api.html`. Accessed: 2014-02-06. 78, 115

[157] A. Karande, M. Karande, and B. B. Meshram, "Choreography and Orchestration Using Business Process Execution Language for SOA with Web Services," *International Journal of Computer Science Issues*, vol. 8, no. 2, pp. 224–232, 2011. 78, 79

[158] W. Jaradat, A. Dearle, and A. Barker, "A Dataflow Language for Decentralised Orchestration of Web Service Workflows," in *SERVICES*, pp. 13–20, 2013. 79

[159] T. Fleuren, J. Gotze, and P. Muller, "Workflow Skeletons: Increasing Scalability of Scientific Workflows by Combining Orchestration and Choreography," *European Conference on Web Services*, vol. 0, pp. 99–106, 2011. 79

[160] D. Ings, L. Clément, D. König, V. Mehta, R. Mueller, R. Rangaswamy, M. Rowley, and I. Trickovic, "WS-BPEL Extension for People (BPEL4People) Specification Version 1.1," tech. rep., OASIS, August 2010. 80

[161] "Web Services Human Task (WS-HumanTask) Specification Version 1.1." `http://docs.oasis-open.org/bpel4people/ws-humantask-1.1.html`. Accessed: 2011-08-13. 80, 81

[162] P. Mouallem, D. Crawl, I. Altintas, M. Vouk, and U. Yildiz, "A Fault-Tolerance Architecture for Kepler-Based Distributed Scientific Workflows," in *Proceedings of the 22nd international conference on Scientific and statistical database management*, SSDBM'10, (Berlin, Heidelberg), pp. 452–460, Springer-Verlag, 2010. 82

[163] J. H. Abawajy, "Fault-Tolerant Scheduling Policy for Grid Computing Systems," *Parallel and Distributed Processing Symposium, International*, vol. 14, p. 238b, 2004. 82

[164] S. Jablonski and C. Bussler, *Workflow Management Systems: Modelling Concepts, Architecture and Implementation*. ITCP Computer Science Series, International Thomson Publishing Services, 1996. 86

[165] A. J. Bonner and M. Kifer, "An Overview of Transaction Logic," *Theoretical Computer Science*, vol. 133, no. 2, pp. 205–265, 1994. 87, 88, 93

[166] A. J. Bonner and M. Kifer, "Transaction Logic Programming. (or, A Logic of Procedural and Declarative Knowledge)," tech. rep., Computer Systems Research Institute, University of Toronto, 1995. 88, 89

[167] A. J. Bonner and M. Kifer, "Concurrency and Communication in Transaction Logic.," in *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pp. 142–156, 1996. 88, 89, 90, 96

[168] A. F. Sleghel, "An Optimizing Interpreter for Concurrent Transaction Logic," Master's thesis, University of Toronto, 2000. 94

[169] A. J. Bonner, M. Kifer, and M. Consens, "Database Programming in Transaction Logic," in *Proceedings of the Fourth International Workshop on Database Programming Languages*, pp. 309–337, 1993. 94, 95

[170] D. Anicic, P. Fodor, R. Stuhmer, and N. Stojanovic, "Event-Driven Approach for Logic-Based Complex Event Processing," in *Proceedings of the 2009 International Conference on Computational Science and Engineering - Volume 01*, CSE '09, (Washington, DC, USA), pp. 56–63, IEEE Computer Society, 2009. 97, 100

[171] D. Roman, M. Kifer, and D. Fensel, "WSMO Choreography: From Abstract State Machines to Concurrent Transaction Logic," in *Proceedings of the 5th European Semantic Web Conference* (M. Hauswirth, M. Koubarakis, and S. Bechhofer, eds.), LNCS, (Berlin, Heidelberg), Springer Verlag, June 2008. 102

[172] "The Mandarax Project." `http://mandarax.sourceforge.net/overview.html`. Accessed: 2011-10-15. 105

[173] A. Kozlenkov, "Prova Rule Language Version 3.0 User's Guide," tech. rep., 2010. 105, 109, 110, 116, 119, 160

[174] "A Free, Open-Source Ontology Editor and Framework for Building Intelligent Systems." `http://protege.stanford.edu`. Accessed: 2012-05-14. 106

[175] "Mule ESB." `http://www.mulesoft.com/`. Accessed: 2012-09-30. 112, 116, 117, 146

[176] "OpenRDF Sesame." `http://www.openrdf.org/`. Accessed: 2014-01-06. 113, 127

[177] "SPARQL 1.1 Update." `http://www.w3.org/TR/sparql11-update/`. Accessed: 2014-01-13. 115

[178] E. Sirin and B. Parsia, "SPARQL-DL: SPARQL Query for OWL-DL," in *Proceedings of the 3rd OWL Experiences and Directions Workshop*, 2007. 115

[179] M. Welsh, D. E. Culler, and E. A. Brewer, "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services," in *Proceedings of the Eighteenth Symposium on Operating Systems Principles*, pp. 230–243, 2001. 116, 170

[180] "Apache ActiveMQ." `http://activemq.apache.org/`. Accessed: 2011-03-11. 119

[181] "Reaction RuleML." `http://ruleml.org/reaction/`. Accessed: 2011-11-13. 120

[182] A. Paschke, H. Boley, Z. Zhao, K. Teymourian, and T. Athan, "Reaction RuleML 1.0: Standardized Semantic Reaction Rules," in *Rules on the Web: Research and Applications*, vol. 7438 of *Lecture Notes in Computer Science*, pp. 100–119, Springer Berlin Heidelberg, 2012. 120, 121

[183] T. Osmun, D. Smith, H. Boley, A. Paschke, and Z. Zhao, *Rule Responder Guide*. RuleML Inc., 2011. 122

[184] A. Paschke, H. Boley, A. Kozlenkov, and B. L. Craig, "Rule responder: RuleML-Based Agents for Distributed Collaboration on the Pragmatic Web," in *Proceedings of the 2nd International Conference on the Pragmatic Web*, pp. 17–28, 2007. 124

[185] S. Migliorini, M. Gambini, M. La Rosa, and A. H. ter Hofstede, "Pattern-Based Evaluation of Scientific Workflow Management Systems," tech. rep., University of Verona, 2011. 130, 138, 141, 148, 149

[186] W. M. P. Van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros, "Workflow Patterns," *Distributed and Parallel Databases*, vol. 14, pp. 5–51, July 2003. 130

[187] N. Russell, A. H. M. ter Hofstede, D. Edmond, and W. M. P. van der Aalst, "Workflow Data Patterns: Identification, Representation and Tool Support," in *Proceedings of the 24th International Conference on Conceptual Modeling*, ER'05, (Berlin, Heidelberg), pp. 353–368, Springer-Verlag, 2005. 143, 144, 145, 146, 147

[188] T. Eiter and G. Gottlob, "On the Computational Cost of Disjunctive Logic Programming: Propositional Case," *Annals of Mathematics and Artificial Intelligence*, vol. 15, no. 3-4, pp. 289–323, 1995. 150

[189] H. J. ter Horst, "Completeness, Decidability and Complexity of Entailment for RDF Schema and a Semantic Extension Involving the OWL Vocabulary," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 3, pp. 79–115, Oct. 2005. 153

[190] "OWL Web Ontology Language Overview." `http://www.w3.org/TR/owl-features/`. Accessed: 2012-04-21. 153

[191] "OWL 2 Web Ontology Language Document Overview (Second Edition)." `http://www.w3.org/TR/owl2-overview/`. Accessed: 2013-12-04. 153

[192] "OWL 2 Web Ontology Language Profiles (Second Edition)." `http://www.w3.org/TR/owl2-profiles/`. Accessed: 2013-12-04. 153

[193] U. Straccia, "Fuzzy Logic, Annotation Domains and Semantic Web Languages," in *Scalable Uncertainty Management* (S. Benferhat and J. Grant, eds.), vol. 6929 of *Lecture Notes in Computer Science*, pp. 2–21, Springer Berlin Heidelberg, 2011. 154

[194] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz, "Pellet: A Practical OWL-DL Reasoner," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 5, no. 2, pp. 51 – 53, 2007. Software Engineering and the Semantic Web. 155

[195] V. Haarslev, R. Möller, and M. Wessel, "Querying the Semantic Web with Racer + nRQL," in *Proceedings of the KI-2004 International Workshop on Applications of Description Logics*, 2004. 155

[196] D. Tsarkov and I. Horrocks, "FaCT++ Description Logic Reasoner: System Description," in *Proceedings of the Third International Joint Conference on Automated Reasoning*, IJCAR'06, (Berlin, Heidelberg), pp. 292–297, Springer-Verlag, 2006. 155

[197] "JFact DL Reasoner." `http://jfact.sourceforge.net/`. Accessed: 2014-02-21. 155

[198] D. Tsarkov and I. Palmisano, "Chainsaw: a Metareasoner for Large Ontologies.," in *ORE* (I. Horrocks, M. Yatskevich, and E. Jiménez-Ruiz, eds.), vol. 858 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2012. 155

[199] "W3C OWL Test Case." `http://www.geneontology.org/`. Accessed: 2013-12-03. 155

[200] "OWL 2 Web Ontology Language Direct Semantics (Second Edition)." `http://www.w3.org/TR/owl2-direct-semantics/`. Accessed: 2014-02-10. 156

[201] R. S. Gonçalves, S. Bail, E. Jiménez-Ruiz, N. Matentzoglu, B. Parsia, B. Glimm, and Y. Kazakov, "OWL Reasoner Evaluation (ORE) Workshop 2013 Results: Short Report.," in *ORE* (S. Bail, B. Glimm, R. S. Gonçalves, E. Jiménez-Ruiz, Y. Kazakov, N. Matentzoglu, and B. Parsia, eds.), vol. 1015 of *CEUR Workshop Proceedings*, pp. 1–18, CEUR-WS.org, 2013. 156

[202] M. A. Bryant, "Representing Meaningful Provenance in Scientific Workflow Systems," Master's thesis, University of Wyoming, 2007. 156

[203] G. Weiler, A. Poetzsch-Heffter, and S. Kiefer, "Consistency Checking for Workflows with an Ontology-Based Data Perspective," in *DEXA* (S. S. Bhowmick, J. Küng, and R. Wagner, eds.), vol. 5690 of *Lecture Notes in Computer Science*, pp. 98–113, Springer, 2009. 157