

# FLEXIBLE SEMANTIC SERVICE EXECUTION

**Inauguraldissertation**

zur

Erlangung der Würde eines Doktors der Philosophie

vorgelegt der

Philosophisch-Naturwissenschaftlichen Fakultät

der Universität Basel

von

Thorsten Möller

aus Saalfeld/Saale, Deutschland

Basel, 2012

Originaldokument gespeichert auf dem Dokumentenserver: <http://edoc.unibas.ch>.



Dieses Werk ist unter dem Vertrag "Creative Commons Namensnennung – Keine kommerzielle Nutzung – Keine Bearbeitung 3.0 Schweiz" lizenziert. Die vollständige Lizenz kann unter <http://creativecommons.org/licences/by-nc-nd/3.0/ch> eingesehen werden.




## Attribution-NonCommercial-NoDerivs 3.0 Switzerland


---


*You are free:*

 to share — to copy, distribute and transmit the work.

*Under the following conditions:*

 **Attribution** — You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

 **Noncommercial** — You may not use this work for commercial purposes.

 **No Derivative Works** — You may not alter, transform, or build upon this work.

*With the understanding that:*

**Waiver** — Any of the above conditions can be **waived** if you get permission from the copyright holder.

**Public Domain** — Where the work or any of its elements is in the **public domain** under applicable law, that status is in no way affected by the license.

**Other Rights** — In no way are any of the following rights affected by the license:

- Your fair dealing or **fair use** rights, or other applicable copyright exceptions and limitations;
- The author's **moral** rights;
- Rights other persons may have either in the work itself or in how the work is used, such as **publicity** or privacy rights.

**Notice** — For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to the web page <http://creativecommons.org/licenses/by-nc-nd/3.0/ch>.

**Disclaimer** — The Commons Deed is not a license. It is simply a handy reference for understanding the Legal Code (the full license) — it is a human-readable expression of some of its key terms. Think of it as the user-friendly interface to the Legal Code beneath. This Deed itself has no legal value, and its contents do not appear in the actual license.

Creative Commons is not a law firm and does not provide legal services. Distributing of, displaying of, or linking to this Commons Deed does not create an attorney-client relationship.

Genehmigt von der Philosophisch-Naturwissenschaftlichen Fakultät

auf Antrag von

Prof. Dr. Heiko Schuldt, Universität Basel, Dissertationsleiter

Prof. Dr. Birgitta König-Ries, Friedrich-Schiller-Universität Jena, Korreferentin

Basel, den 27. März 2012

Prof. Dr. Martin Spiess, Dekan



*To my mother and father*



Stillstand ist der Tod  
Geh voran, bleibt alles anders

---

HERBERT GRÖNEMEYER  
first line also in *Triptychon*, MAX FRISCH





# Zusammenfassung

Die vorliegende Arbeit widmet sich einer wichtigen Aufgabenstellung, die im Umfeld verteilter und dienstbasierter Architekturen auftritt: Die korrekte, zuverlässige und effiziente Ausführung softwarebasierter Dienste. Im Zentrum dieser Arbeit stehen dabei zwei Ansätze die jeweils die Flexibilität bei der Umsetzung dieser Aufgabe steigern. Erstens eine neuartige Methode zur automatisierten Vorwärtsbehandlung von Dienstfehlern zur Ausführungszeit, genannt Control Flow Intervention (CFI). Zweitens eine in sich geschlossene Ausführungstechnik die die Migration laufender Ausführungsinstanzen zwischen verfügbaren Ausführungsmaschinen gestattet. Beide tragen den spezifischen Anforderungen neuartiger internetbasierter und mobiler Anwendungsgebiete Rechnung. Wesentliche Merkmale dieser Anwendungen sind (i) Inhärenz entfernter Aufrufe, (ii) ad-hoc-Dienste um dynamisch sich verändernden Umgebungen und Benutzerpräferenzen gerecht zu werden und (iii) eine hohe Fehleranfälligkeit bedingt durch drahtlose Verbindungen und Volatilität angebotener Dienste.

Die zugrunde liegende Theorie für alle in dieser Arbeit angestellten Untersuchungen sind Semantische Dienste, insbesondere basierend auf deduktiven und entscheidbaren Beschreibungslogiken. In einem ersten Schritt greifen wir bisherige Arbeiten aus diesem Bereich auf und entwickeln diese weiter hin zu einem kohärenten formalen Systemmodell welches wesentliche Dimensionen in der Semantik von Diensten vereint.

Basierend auf diesem Systemmodell liegt anschliessend das Hauptaugenmerk auf CFI. Ziel dieser Methode ist es, Dienstfehler durch geeignete Ersetzungsstrategien nach vorn zu korrigieren, so dass das geplante Gesamtziel eines Dienstes in äquivalenter oder zumindest vergleichbarer Form trotzdem erreicht werden kann. Dies wird durch dynamisches Ausweichen auf semantisch äquivalente oder ähnliche Alternativen ermöglicht. Hierbei wird davon ausgegangen, dass konkrete Alternativen nicht Bestandteil der Dienstspezifikation sind. Es wird lediglich angenommen, dass sie in der Anwendungsdomäne vorhanden sind. Da Alternativen somit nicht vordefiniert sind, wird im Fehlerfall dynamisch nach ihnen gesucht.

Da das vorgestellte Systemmodell zwei Arten von Nebenläufigkeit bei der Dienstausführung zulässt, und da die Repräsentation des Zustandes verschiedener Ausführungsinstanzen in einer gemeinsamen Wissensbasis erfolgt, wird ausserdem der korrekte und inferenzvermeidende simultane Zugriff auf solche Wissensbasen untersucht. Diese Arbeit stellt dazu ein neuartiges Zugriffsmodell zur Koordination von nebenläufigen Transaktionen auf einer Web Ontology Language Wissensbasis vor. In diesem Zusammenhang werden dessen Leistungs- und Isolationseigenschaften diskutiert.

Um die praktische Anwendbarkeit der entwickelten Methoden untersuchen zu können, wurden diese prototypisch in unserem verteilten und dezentralen Ausführungssystem OSIRIS NEXT implementiert. Wir beschreiben den grundlegenden Aufbau dieses Systems. In diesem Zusammenhang stellt diese Arbeit dann die verteilte Ausführungstechnik vor, die insbesondere für (semi-)automatisch zusammengefügte und nur wenige Male ausgeführte ad-hoc-Dienste optimiert ist.

Schliesslich wurden die vorgestellten Verfahren durch verschiedene Experimente hinsichtlich ihres Laufzeitverhaltens quantitativ evaluiert. Die dabei gemachten Erfahrungen und Resultate zeigen das Potential der Verfahren für deren Einsatz in der Praxis.

# Abstract

This thesis deals with an important task in the context of distributed and service-oriented architectures: the correct, reliable, and efficient execution of software-based services. In the center of this work are two approaches that increase the flexibility in this task. First, a novel method for automated forward recovery of service failures at execution time, called Control Flow Intervention (CFI). Second, a self-contained technique that allows for migration of running execution instances among available execution machines. Both address requirements specific to novel Internet-based and mobile applications. Characteristic for such applications are (i) inherent remote invocation, (ii) ad-hoc services to cope with dynamically changing environments and user preferences, and (iii) frequent errors due to wireless connections and volatility of offered services.

The underlying theory for all investigations made in this thesis are Semantic Services, based in particular on deductive and decidable Description Logics. In a first step, we take up prior work in this area and develop it further towards a coherent formal system model that combines essential dimensions of service semantics.

Based on this model, the focus is then on CFI. The goal of this method is to correct service failures by appropriate replacement strategies in a forward-oriented way, meaning that the overall goal of a service remains attainable, though in a semantically equivalent or at least comparable form. This is achieved by dynamically shifting to semantically equivalent or similar alternatives. Alternatives are however not pre-defined as part of the service specification. Rather, it is assumed that they exist in the application domain and that they are searched for on demand in the presence of a failure.

Since the system model allows for two types of concurrency in the service execution, and since the state of execution instances is represented in a shared knowledge base, we also investigate the problem of ensuring correct concurrent access to knowledge bases so that inferences are avoided. Specifically, we present a novel concurrency control model for transactions operating over a Web Ontology Language knowledge base. Efficiency and isolation properties of the presented approach are furthermore discussed.

In order to investigate the practical applicability of the presented methods, they were prototypically implemented in our distributed and decentralized execution system OSIRIS NEXT. We describe the architecture of this system. In this context, the distributed execution technique is presented that is particularly optimized for ad-hoc services that are usually (semi-)automatically composed and executed a few times only.

Finally, the presented methods were evaluated quantitatively by various experiments with respect to their runtime behavior. The results and the experiences gained show the potential of the methods for their application in practice.



# Acknowledgments

I am deeply indebted to my adviser Prof. Heiko Schuldt for his friendly supervision, for many excellent ideas he provided, and for displaying patience on my occasional tendency towards action rather than intellect. He gave me the unique opportunity to write this thesis in his group. I also enjoyed the relaxed working environment he provided. This thesis would not have reached its altitude without him.

I wish to thank my former and present colleagues of the DBIS group at University of Basel. There have been numerous conversations and stimulating discussions over the last years. Out of the DBIS members, I would especially like to thank my officemate Nadine Fröhlich for sharing her insights on scientific and other topics, and for sharing all the (unavoidable) ups and downs in doing a Ph.D.

I would also like to express how valuable it was to collaborate with many knowledgeable people during the research projects CASCOM and LOCA. Among the people that contributed to this work, I would like to thank Marcel Büchler who supported me in the implementation. For her professional and uncomplicated character, and for agreeing to be a member of the thesis committee, cordial thanks go to Prof. Birgitta König-Ries. A big thanks also goes to Dr. Christian Hollmann and Roman Langfeld for reading preliminary versions and for making invaluable comments.

This thesis would probably never have been started without Sungyon So. Her natural acumen persuaded me (in an irresistible way) that I should do a Ph.D. I also like to thank her for staying close over the years despite the physical distance. Going even further back in time, it is to the credit of my sister Iris Möller who persuaded me in the same yet different way to start the voyage to Computer Science.

A special thanks goes to my parents Christine and Siegfried Möller for everything they gave to me, especially their continuous encouragement, support, and trust. This thesis is dedicated to them. Finally, I owe so much to my beloved Julia. Her strength and lightheartedness comforted and protected me throughout the last year of finishing this work – no one knows better these engaged days.



# Contents

Zusammenfassung	ix
Abstract	xi
Acknowledgments	xiii
<b>1 Introduction</b>	<b>1</b>
1.1 Service-based Applications	2
1.2 Problem Description	3
1.3 Thesis Goals and Contributions	6
1.4 Thesis Outline	7
<b>2 Motivation</b>	<b>9</b>
2.1 E-Commerce Scenario	9
2.2 E-Health Scenario	10
2.3 Application Dynamics and Consequences	13
<b>3 Fundamentals</b>	<b>15</b>
3.1 Description Logics	15
3.1.1 Description Logic <i>SHOIN</i>	17
3.1.2 Description Logic <i>SROIQ</i>	23
3.1.3 Datatype Maps and Data Ranges	24
3.1.4 Reasoning and its Computational Complexity	27
3.1.5 Operations on Knowledge Bases	31
3.2 Resource Description Framework	34
3.3 Web Ontology Language	35
3.3.1 Import Mechanism	36
3.3.2 Representation Formats	36
3.3.3 Profiles	37
3.3.4 Mapping to RDF Graphs	38
<b>4 System Model</b>	<b>41</b>
4.1 Basic Elements, Relations, and Assumptions	42
4.1.1 Functional Unit	43
4.1.2 Operation	45
4.1.3 Implementation	46
4.1.4 Service	47
4.1.5 Profile	47
4.1.6 Process	49
4.1.7 Service Description	50
4.2 Service Model	51

4.2.1	Profile Parameter . . . . .	51
4.2.2	Preconditions and Effects . . . . .	56
4.2.3	Profile, Operation, and Service . . . . .	73
4.3	Process Model . . . . .	76
4.3.1	Control Flow . . . . .	76
4.3.2	Data Flow . . . . .	85
4.3.3	Well-formed Processes . . . . .	90
4.4	Summary . . . . .	91
<b>5</b>	<b>Forward Failure Handling using CFI</b>	<b>95</b>
5.1	The Basic Control Flow Intervention Cycle . . . . .	97
5.2	Range of Application . . . . .	98
5.2.1	System Environments . . . . .	98
5.2.2	Failure types . . . . .	100
5.3	Replacements and their Structure . . . . .	102
5.4	Semantically Equivalent Execution . . . . .	104
5.4.1	The Matchmaking Task . . . . .	106
5.4.2	The Planning Task . . . . .	111
5.4.3	Functional Profile Equivalence . . . . .	117
5.4.4	Functional Equivalent Execution . . . . .	121
5.4.5	Similar Execution and Non-functional Properties . . . . .	131
5.5	Integration with Transactional Processes . . . . .	135
5.5.1	Guaranteed Termination . . . . .	136
5.5.2	Integration Strategies . . . . .	138
5.6	Repeated Intervention . . . . .	139
5.6.1	Threshold . . . . .	140
5.6.2	Progress . . . . .	140
5.6.3	Possibility to make Progress . . . . .	141
5.7	Discussion . . . . .	142
5.7.1	Disambiguating Profile Parameters . . . . .	142
5.7.2	To Plug-in Match or not to Plug-in Match . . . . .	143
5.7.3	Structure-aware versus Structure-nescent Replacements . . . . .	143
5.7.4	Replacement Composition Planning via Translation into PDDL . . . . .	144
5.8	Summary . . . . .	145
<b>6</b>	<b>Concurrency Control for Shared Knowledge Bases</b>	<b>147</b>
6.1	Motivation . . . . .	148
6.2	CC Model for OWL Knowledge Bases . . . . .	150
6.2.1	OWL Data Items . . . . .	150
6.2.2	Basic Operations . . . . .	152
6.2.3	Transactions . . . . .	153
6.2.4	Correct Concurrent Access . . . . .	154
6.2.5	Access Protocol . . . . .	155
6.2.6	Higher Level Conflicts . . . . .	157
6.2.7	Extended Commit Protocol . . . . .	161
6.2.8	Correctness of the Protocol . . . . .	162



---

6.3	RDF Triple Store Integration . . . . .	164
6.4	Integration of Inferencing Engines . . . . .	166
6.4.1	Online Computation of Implicit Knowledge . . . . .	167
6.4.2	Materialization of Implicit Knowledge . . . . .	167
6.5	CC applied to Semantic Service Execution . . . . .	169
6.6	Discussion . . . . .	171
6.6.1	Correctness . . . . .	171
6.6.2	Performance . . . . .	172
6.7	Summary . . . . .	174
<b>7</b>	<b>Implementation</b> . . . . .	<b>177</b>
7.1	OSIRIS NEXT . . . . .	177
7.1.1	Architectural Overview . . . . .	178
7.1.2	Peer-to-Peer Execution . . . . .	182
7.1.3	Control Flow Intervention . . . . .	187
7.2	KB Access Optimization Techniques . . . . .	188
7.2.1	Prepared Queries . . . . .	190
7.2.2	Frame Caching . . . . .	192
7.3	Snapshot Isolation OWL Data Store . . . . .	195
7.3.1	Interfacing with the OWL API . . . . .	195
7.3.2	Data Structures and Snapshot Management . . . . .	196
7.3.3	Transactions and Conflict Checking . . . . .	196
<b>8</b>	<b>Experimental Results</b> . . . . .	<b>199</b>
8.1	Control Flow Intervention . . . . .	199
8.1.1	Experimental Setup . . . . .	200
8.1.2	Results . . . . .	200
8.2	Execution Engine . . . . .	202
8.2.1	Experimental Setup . . . . .	202
8.2.2	Results . . . . .	202
8.3	KB Access Optimization Techniques . . . . .	203
8.3.1	Experimental Setup . . . . .	203
8.3.2	Results . . . . .	204
8.4	Snapshot Isolation OWL Data Store . . . . .	207
8.4.1	Experimental Setup . . . . .	208
8.4.2	Results . . . . .	209
<b>9</b>	<b>Related Work</b> . . . . .	<b>215</b>
9.1	Adaptation and Exception Handling . . . . .	215
9.2	Distributed Execution . . . . .	217
9.3	Concurrent Access to Knowledge Bases . . . . .	219
<b>10</b>	<b>Conclusions and Future Work</b> . . . . .	<b>223</b>
10.1	Summary . . . . .	223
10.2	Future Work . . . . .	225

---

<b>Appendix</b>	<b>227</b>
A.1 Effect System Algorithms . . . . .	227
A.2 Conditional Choice for Control Flow Graphs . . . . .	230
A.3 Properties of Read and Update Operations . . . . .	231
<b>Bibliography</b>	<b>233</b>
<b>Index</b>	<b>261</b>

# Figures

1.1	Integral parts and important requirements of the service execution task. . . . .	4
2.1	Example e-commerce scenario: Book Seller. . . . .	10
2.2	Example e-health scenario: Emergency Assistance. . . . .	11
3.1	Distinction between high level knowledge base updates and direct updates at the level of the storage layer. . . . .	33
3.2	Graphical representation of overlaps and containment regarding language expressiveness for FOL, OWL, and Logic Programs . . . . .	37
4.1	Classification of service semantics combined in the system model. . . . .	41
4.2	Basic elements of the system and their static structure. . . . .	43
4.3	Schematic diagram of assignment functions for exemplary input/output in different formats. . . . .	54
4.4	Links between representatives of profile parameters and variables in pre-conditions and effects (dotted lines represent possible links). . . . .	67
4.5	Unfolded control flow graph of the <i>emergency assistance</i> service. . . . .	80
4.6	Data flow primitives. . . . .	86
4.7	Summary of the system model depicting its main layers. . . . .	92
5.1	CFI in relation to conventional failure handling approaches. . . . .	96
5.2	Integral activities forming the Control Flow Intervention cycle. . . . .	97
5.3	Examples for structural substitutions in control flow graphs. . . . .	105
6.1	Overlapping read/update access on the KB for concurrent service execution. . . . .	150
6.2	Commit Pipe for OWL Concurrency Control. . . . .	162
6.3	CC model levels for RDF triple store integration. Transactions consisting of operations over OWL syntactic instances result in operations over disjoint sets of RDF triples at lowest level. . . . .	164
6.4	Data items at OWL and RDL level illustrated using a fictitious OWL syntactic instance. . . . .	165
6.5	System Architecture Types for Integration of an OWL Data Store with Inferencing Engines. . . . .	167
6.6	Mapping of read and update queries (see <a href="#">Figure 6.1</a> ) for service execution to read/update transactions and example operations over OWL syntactic instances. . . . .	170
7.1	High level organization of OSIRIS NEXT. . . . .	179
7.2	Internal design and functional decomposition of an OSIRIS NEXT peer. . . . .	180
7.3	<i>Emergency Assistance</i> process depicted as nested OWL-S constructs. . . . .	184

---

7.4	Simple example illustrating the execution strategy implemented in OSIRIS NEXT. . . . .	185
7.5	Internal structure and main components of an execution peer. . . . .	187
7.6	Comparison of (pre-) condition evaluation procedure for conventional and optimized approach using prepared queries. . . . .	192
8.1	Search and substitution times for increasing number of available services.	202
8.2	Fife-number summary of total execution time for <i>Dictionary</i> service as a function of increasing number of concurrent execution requests per peer. .	203
8.3	<i>Repeat-Until</i> service executed with different configurations. . . . .	206
8.4	Total execution time as a function of KB size for conventional and optimized configuration. . . . .	207
8.5	Execution times in comparison for basic workloads. . . . .	211
8.6	Execution times in comparison for additional workloads. . . . .	212

# Tables

1.1	Characteristics of applications considered in this thesis. . . . .	2
3.1	Syntax and Semantics of <i>SHOIN</i> concept expressions and roles and corresponding OWL constructs . . . . .	22
3.2	Additional Constructs in <i>SRIOQ</i> and their Semantics . . . . .	25
3.3	Model-Theoretic Semantics of $\mathcal{DL}+\mathcal{D}$ data ranges, concepts, axioms, and assertions . . . . .	27
3.4	Examples for mapping of OWL syntactic constructs to RDF triples . . . . .	39
4.1	Combinations of TBox axioms and ABox assertions that cause KB inconsistency for $\mathcal{L}_{Tra}$ and in the absence of the UNA. . . . .	71
5.1	Different dimensions of planning domains. . . . .	115
6.1	Commutativity and set-preservation of read, add, and delete operations on OWL data items. . . . .	153
7.1	Classification of OWL-S control constructs with regard to migration. . . . .	183
7.2	SWRL atoms, their semantics, and mapping to SPARQL BGP. . . . .	191
8.1	Search and substitution times for service profiles of different size (varying number of inputs, outputs) . . . . .	201
8.2	Exemplary services used for evaluation purposes. . . . .	204
8.3	Execution speedup of exemplary services. . . . .	205
8.4	Workloads used for the performance analysis and their characteristics. . . . .	208
8.5	Average time to normalize $n$ -ary axioms/assertions ( $2 \leq n \leq 10$ ). . . . .	210
8.6	Comparison of execution time and transactions per second as a function of increasing concurrency and workloads. . . . .	213



# 1

## Introduction

**I**F WE WERE ASKED TODAY to name major advancements and breakthroughs regarding systems, hardware, software, and information management technologies within the last two decades, one might frequently get the following answers. At systems level, methods that enabled pervasive and decentralized infrastructures. At hardware level, mobile devices and wireless communication technologies. At software level, the new paradigm of Service-oriented Computing where applications are built by combining reusable building blocks rather than being monolithic entities. And finally, methods and data models that enabled data and information management to scale up to the global level.

One particular example where all this went together is certainly the Internet with its prevalent application the Web. While in the early days of the Internet the main applications were electronic mail, instant messaging, and file transfer, it has evolved into a multi-purpose application platform with applications of various kinds. Notably, it is used today as a platform to build service-oriented applications. Around the millennium, however, it was found that methods used at that time for information representation in the Web (and other application areas) generally lack the ability to make the meaning – the semantics – of information understandable to machines. The goal of semantic technologies – which are being researched much longer – is to facilitate automation based on formal frameworks allowing machines to interpret and reason about the concepts, objects, and their relations within a given domain. The vision of the Semantic Web [BLHL01, FWL02] is to (i) bring these semantic technologies to the global level of the Web and other applications built on top of the Internet, and (ii) to enable the interlinking of information from diverse heterogeneous sources. These technologies therefore play an important role to information integration.

This thesis cannot be viewed independent of all these fields, as it is the progress in these fields that spawned novel application forms that call for appropriate methods to realize them. In fact, we see this thesis situated in the intersection of the following areas. First, pervasive and often decentralized infrastructures. Second, applications that are built based on the paradigm of Service-oriented Computing. Applications in which information technology is more and more integrated into everyday activities, with mobile users and mobile as well as embedded devices. Finally, methods to make the se-

antics of information and resources – most notably services – available for automated interpretation and reasoning by machines.

Having illuminated the general context of this thesis, we will now introduce it in more detail. In the remainder of this chapter we describe the envisioned application forms, outline the problems addressed, summarize the contributions made, and overview how this thesis is structured.

## 1.1 Service-based Applications

In this work, we consider applications that are built based on the paradigm of Service-oriented Computing (SOC) [Pap03]. In this programming and computing model, single services are re-usable pieces of software, designed to achieve intents of some sort either by physical transformations in the real world or information processing on data. We expect services to be available at a large variety of stationary, mobile, and embedded devices. Also, we expect services to be accessed mainly by users via mobile devices. [Table 1.1](#) summarizes the characteristics of service-based applications considered in this thesis. These characteristics reflect features sought in today’s and future Internet-based applications [CS06, SGA07].

Table 1.1: Characteristics of applications considered in this thesis.

Dimension	Explanation
Methodology	Applications are <i>composed</i> out of a set of pre-existing, reusable, and loosely coupled components – the services – each contributing certain units of functionality required by the application.
Creation	Not necessarily pre-defined and manually created by software engineers. Rather, they may be synthesized ad hoc using (semi-)automated service composition methods [RS04] in order to take into account dynamics of various origins such as user preferences or environmental and contextual properties.
Machinery	Services are deployed to a large variety of hardware, not only on stationary, but particularly on mobile or embedded devices of diverse computing, storage, and communication capabilities.
Operation	Services process data or information and might as well create effects in the real world. Regarding the data processed, we focus, however, on discrete services in contrast to stream-based services.*
Interaction	Remote interactions between services themselves and services and client devices in an asynchronous manner. Applications form (complex) interaction patterns over their constituting services.

\* The difference between discrete and continuous operation mode is clarified in [Section 4.1.1](#).



Applications may be built solely by combining pre-existing services. Such a combination of a set of services that altogether make up a value-added “larger” one is referred to as a *composite service* (CS) [MBE03].<sup>1</sup> Notably, a CS may represent a *workflow* or *business process*. In fact, the SOC paradigm together with composite services is increasingly adopted by IT-supported Workflow Management (WfM)<sup>2</sup> and Business Process Management (BPM)<sup>3</sup>. The underlying assumption is that the tasks of workflows or activity units of business processes can be realized using software services and their operations.

## 1.2 Problem Description

An important task is the automated coordination of service execution. Analogous to automated workflow and process execution [AH02], *service execution* comprises all the activities that need to be carried out at runtime by a system in order to (i) invoke the services and operations of which it is composed in a coordinated manner as specified, (ii) to correctly manage (store and access) data that is processed unless execution finishes, and (iii) to detect, handle, and recover from runtime failures (see [Figure 1.1](#)). These activities also include initiation, control, and validation of service invocations, and invocation of services in parallel where possible. Typically, this task is carried out by a dedicated *execution engine*<sup>4</sup> acting on behalf of a user or software agent. Such an engine is responsible for correct service enactment and it should also include means to achieve a sufficient degree of reliability and efficiency. It may come stand-alone or can be an integral part of comprehensive service-based workflow management systems (WfMS)<sup>5</sup> or service-based business process management systems (BPMS)<sup>6</sup>.

When invoked, services create effects of some sort in the real world and/or process data of some kind. This is the *functional* dimension of services. On the other hand, they consume different resources at runtime (e.g., electric energy, disk space for data being processed, CPU cycles for computations). This is the *non-functional* dimension of services. As a matter of these two, both service users and service providers are interested that certain properties can be ensured in the course of execution. Regarding the functional dimension, the most important properties are correctness, reliability, and

---

<sup>1</sup>The technical concept can be compared to *mashups*, a term coined more recently to refer to content aggregation technologies [BDS08]. Mashups, however, aim at combining data and presentation in addition to functionality. Mashups therefore target a broader spectrum of *composite applications*.

<sup>2</sup>WfM is commonly viewed as including concepts, methods, and technologies to support the design, administration, configuration, enactment, and analysis of business processes [Wes07].

<sup>3</sup>According to [AHW03], BPM can be considered an extension of WfM that originates from office automation [JB96].

<sup>4</sup>Note that the singular form used here shall not imply any system related property. Such an engine may be a centralized (and autonomous) system. On the other hand, multiple engines may cooperate in a distributed (and decentralized) manner for accomplishing the execution task.

<sup>5</sup>In [Law97], a WfMS is defined as a system that defines, creates and manages the execution of workflows through the use of software, running on one or more workflow engines, which is able to interpret the process definition, interact with workflow participants and, where required, invoke the use of Information-Technology (IT) tools and applications.

<sup>6</sup>In [Wes07], a BPMS is defined as a generic software that is driven by explicit process representations to coordinate the enactment of business processes.

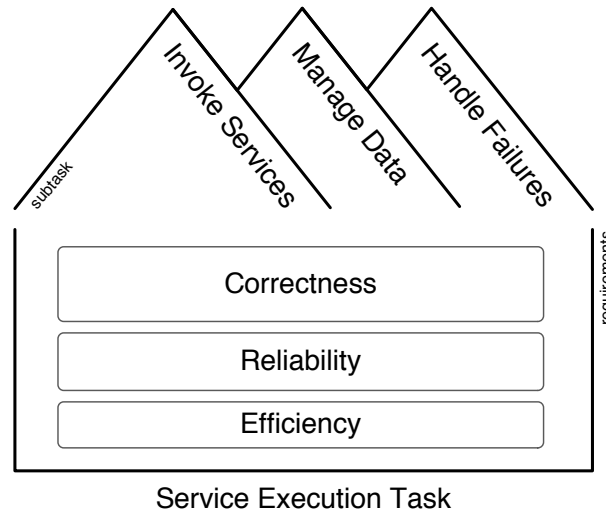


Figure 1.1: Integral parts and important requirements of the service execution task.

efficiency (see Figure 1.1). While the latter two should be clear, the former refers to preservation of a consistent state upon termination (assuming that one started from a consistent state) even in the presence of failures or exceptional situations. Apart from failures caused by errors made at design time of a system, failures can occur because systems are subject to various phenomena of a stochastic nature. Also, one of the eight golden rules of process management [Dvt05, Chapter 15] states that exceptions should be considered the rule because not everything can be determined beforehand.

The aspect of system-supported failure handling and recovery is important for the service execution task, especially when it comes to service execution in distributed environments such as the Internet where multiple systems (software, hardware) as well as humans can be involved. In general, one wants to ensure that once execution of a service has been started it will not arbitrarily halt somewhere before its end, caused by a failure or an exceptional situation. If this would happen, the final outcome of the service would be achieved only partially, up to a level where the outcome has not been achieved at all. This would displease service users as well as service providers. Moreover, resources that have been used may remain in an undefined state, possibly resulting in inconsistent data seen on subsequent use. Also, subsequent use of resources may be impeded in case they were not properly released. Consequently, failure handling for service execution aims at two things. First, ensuring consistency regarding data and resources. This is achieved, second, by methods that ensure that one can recover from an error either by *rolling back* to the previous correct state of the system as if nothing was done or by *rolling forward* to a new consistent state. These two approaches are commonly categorized as *backward* and *forward* recovery, respectively [LA90, ALRL04].<sup>7</sup> Among prominent methods in this regard are transactional ones (e.g., [GMS87]). Their basic principle is that a service execution or a part thereof is understood as a transaction. Backward recovery then either reverses effects of a partial execution (due to an error) by applying the in-

<sup>7</sup>It should be noted that the term forward recovery is understood in a very general way in the field of dependable computing, meaning that the system reaches *some* new state without an error (i.e., there are no further requirements on what particular state this is).

verse or compensates for these effects. From a semantic point of view, compensation does not necessarily directly undo the effects but may be done in a countervailing way (e.g., issuing a credit note and mark an order as canceled instead of completely deleting it). In contrast, forward-oriented recovery aims at achieving either the original outcome or a semantically equivalent final outcome. The latter – methods for forward-oriented failure handling and recovery by achieving semantically equivalent or similar outcomes – is the first and major dimension of flexibility for the service execution task subject to be systematically studied in this thesis.

Another aspect belonging to the non-functional dimension is efficiency. One is likely interested in keeping costs low, both resource usage costs and costs of the service execution task itself (e.g., in terms of time, space, money). For instance, having the option to choose from a set of execution engines that are all able to execute a certain service but varying in terms of costs to do so, one would likely want to choose the one that induces lowest costs. One may even want to migrate from one execution engine to another in the course of execution if it turns out that another one can do better; for instance, because it is faster or consumes less energy. This applies in a similar way to services. Having determined that a set of services are functionally equivalent or similar, one may want to choose the one that provides the best value regarding some non-functional property (e.g., shortest time required for a computation). In a heterogeneous and large scale setting such as the Internet it can often be assumed – in fact, it is usually the case – that different options exist to choose from semantically equivalent or similar services, from resources of different capacities, and from devices that may be better suited to execute a particular service than another.

These two features – (i) forward-oriented semantic service failure handling and (ii) the ability to migrate an ongoing execution to an engine at runtime that best fits a set of context and situation specific criteria – are what we consider as constituting flexibility for the composite service execution task. The main focus is put on a novel approach to forward-oriented semantic service failure handling which we call *Control Flow Intervention* (CFI). The general idea of CFI is to allow an execution engine to intervene in the default control flow of a service in the presence of an invocation failure and allow it to replace one or several failed services or operations by a semantically equivalent (or similar) one. Rather than being pre-defined as part of the service specification, a replacement is dynamically searched by the engine at failure time. To achieve this, CFI essentially proposes a combined Description Logic and Petri-net based approach to formalize and reason about the semantics of services.

In order to implement (i) and (ii) in practice so that computers handle them mostly in an automatic way, a couple of issues need to be solved:

- How to determine whether services or single operations of them are equivalent regarding their functional and/or non-functional properties. Essentially, this comes down to formalizing a decidable notion of *equivalence*. In addition, this formalization should be compatible to also allow representing a broader notion of *similarity*.
- How to represent functional and non-functional properties of services and how to store these representations such that one can (efficiently) find candidates for the purpose of forward failure handling.

- How to ensure that a replacement that has been selected also preserves executability of the service and that it complies with its data flow.
- How to ensure consistency of data especially for concurrent execution of multiple services and concurrent execution threads within a service. These two types of concurrency are an essential requirement as they exist in many practical application scenarios.
- How to determine whether there are other execution engines available that can take over an ongoing execution as they can do it in a more efficient way (i.e., whether it would be beneficial to migrate an ongoing execution to another engine).
- How to ensure consistent execution state *migration* at runtime from one engine to another so that it can be seamlessly resumed at the new engine from the state where it was paused.

The next section outlines each of the contributions made on how we address these questions.

### 1.3 Thesis Goals and Contributions

The overarching goal of this thesis is to further the research into Semantic Services and the service execution task in particular. Our work builds upon the state-of-the-art in Semantic Service research. From this perspective, our work should be seen as one step towards integration with other research areas, namely, Process and Workflow Management and Transactional Information Systems. While many of the visions of Semantic Services have been described in detail at a conceptual level, we also see this work as a valuable step towards implementing this vision in practical systems.

The main contributions that result from theoretical work are:

- A formal system model that provides precise semantics for the execution of Semantic Services. It combines, first, the description of the functional and non-functional properties of services and representation of their semantics based on Description Logics. Second, the behavior of services by viewing them as processes. This includes the flow of control and data, allows for concurrency within and among service instances (intra- and inter-service concurrency), and captures how a world state representation in a knowledge base evolves in the course of execution. The system model furthermore includes distributed environments.
- A novel method to semantic forward-oriented failure handling for the service execution task, called *Control Flow Intervention* [MS08, MS10b]. CFI proposes the integrated use of Semantic Service matchmaking and composition with process-based execution [MSGK06]. More specifically, we (i) analyze failure types that can be covered by CFI, (ii) define different types of replacements and requirements on how to find or create them, and (iii) describe requirements to preserve executability under a replacement.

- A general concurrency control model and protocol that provides transactional read and update access to shared Web Ontology Language [W3C09] (OWL) knowledge bases. This model jointly considers (i) data level consistency properties according to serializability theory in databases and (ii) consistency requirements at Description Logic level. This is achieved by transferring the notion of transactions from databases to read and update access over knowledge bases. Second, transactions consist of read and update operations that directly operate over OWL axioms, assertions, and annotations rather than at the lower physical data level. This allows to analyze and control conflicting access at both the semantic and data level.

Furthermore, the main contributions that result from practical work are:

- A distributed and peer-to-peer style execution system for efficient semantic service execution, called OSIRIS NEXT [MS07]. This system is the platform in which most of the experimental work has been carried out.
- An implementation of CFI in OSIRIS NEXT and an empirical evaluation of its runtime performance [MS10b]. The evaluation demonstrates the practical applicability of CFI. For the implementation and evaluation, we have also developed a simple service repository based on an RDF triple store that can be queried using SPARQL.
- An execution technique that allows for migrating ongoing service executions among execution peers in OSIRIS NEXT [MS07]. The migration process is self-contained and optimized for ad hoc services, as it does not require additional system services.
- Two optimization techniques that are used in our implementation of a service execution engine [MS10a]. The first one is used to speed up repeated precondition checking. The second one is a caching technique that provides rapid access to frequently reused parts of a service specification. We present speedup results for an empirical performance evaluation. Moreover, these techniques are applicable beyond the service execution task to efficiently read information from graph based RDF triple stores.
- An implementation of the concurrency control model and protocol together with a main memory OWL store. We present a detailed empirical performance evaluation for which a benchmark for OWL updates has been defined that mimics typical access patterns of practical applications.

## 1.4 Thesis Outline

This thesis is organized in ten chapters that can be grouped in five parts. The introductory part consists of this chapter and [Chapter 2](#). In this chapter we have started by guiding the reader to the place in the Service-oriented Computing research field where this

thesis is located. This included a description of the service execution task and its main activities. Based on characteristics of service-based applications, we have highlighted the problems that this thesis addresses and have summarized the main contributions. The problem description is further set forth in [Chapter 2](#) where we present two such service-based applications taken from the e-commerce and e-health domain. We show how the methods presented in this thesis contribute to these applications. Throughout this thesis, we will often refer back to these applications for the purpose of illustrative examples.

The second part is made up by [Chapter 3](#). In this chapter, we give a rather detailed introduction to the fundamentals on which this thesis builds. This includes the theory of Description Logics, the Web Ontology Language, and the Resource Description Framework. Additional foundational information that is relevant to our work, namely the theory of Petri nets and principles of service matchmaking and planning, is intentionally provided in situ throughout subsequent chapters.

The conceptual part of this thesis is divided into [Chapter 4](#), [5](#), and [6](#). [Chapter 4](#) takes up the current state-of-the-art on Semantic Service research. In this chapter, we present a formal system model for the Semantic Service execution task. It combines representation of the semantics of functional and non-functional properties with the behavior of services by viewing them as processes. The model also includes a formal notion of executability. All methods introduced in subsequent chapters will be applied to this model. In [Chapter 5](#) we present in detail CFI for optimistic and semantic forward failure handling, describe the types of failures that can be covered, and discuss its properties. [Chapter 6](#) is then entirely devoted to a model for concurrency control on shared knowledge bases so as to avoid different types of inferences. We apply this approach to OWL knowledge bases, show that it is compatible with representation of OWL axioms, assertions, and annotations as RDF triples, and provide two architecture blueprints for efficient integration of reasoning engines.

The fourth part is devoted to the practical work of this thesis. In [Chapter 7](#), we describe the implementation of CFI in our peer-to-peer style distributed service execution system called OSIRIS NEXT. We also describe techniques for efficient semantic service execution that have been implemented in OSIRIS NEXT. First, two techniques to speed up repeated access to the same information of a knowledge base. Second, a technique that allows for dynamic migration of ongoing executions among execution peers, which was especially designed for ad hoc services and mobile environments. Finally, [Chapter 7](#) also presents how the concurrency control model introduced in [Chapter 6](#) has been implemented. [Chapter 8](#) then describes how we have evaluated our methods and techniques. We present and discuss experimental results.

The final part is divided into [Chapter 9](#) and [10](#). The former reviews the most important related work and discusses qualitative differences. The latter summarizes the results of this thesis and discusses possible future work.

# 2

## Motivation

IN THIS CHAPTER, we describe two exemplary, albeit simplified, practical application scenarios to further illustrate the potentials of having flexible semantic service failure handling using CFI and the possibility of dynamically migrating ongoing executions. The scenarios are chosen from diverse domains. One from the e-commerce domain, the other from the e-health domain. We will refer back to them throughout this thesis for illustration purposes. Finally, advantages of the CFI approach are summarized and put into relation to rollback and compensation based approaches.

### 2.1 E-Commerce Scenario

This application scenario describes a simplified book ordering and shipment composite service. Typically, online book sellers would provide such a service in the Internet for (prospective) customers that want to order some book(s). Apart from interactions with the customers, this CS shall integrate an additional online shipper service to deliver ordered book(s) to the customer. [Figure 2.1](#) shows the structure of this CS, depicted as an control flow graph that specifies the local execution dependencies among the single services it consists of. The nodes represent the enclosed services *find book*, *order & pay*, and *shipment*. The connecting directed arrows specify a precedence – the order in which they need to be invoked. [Figure 2.1](#) also shows data items processed and their flow. This is depicted by enumerated item names inside the right-copped rectangles. The first service is an atomic service provided by a third-party online library. It is used as the first step to retrieve the unique ISBN number (7) for a book searched by the customer based on its title (1), author name (2), and publisher information (3). The second service *order & pay* would be provided by the book seller. It is used to place an order of one or more items (4) of the book identified before (7) as well as to handle payment by credit card or the like (5). This service may be a composite service itself; its decomposition is not illustrated here. Upon completion this service produces an order and payment acknowledge (8). The acknowledge shall include information about (i) how many items of the book were actually ordered and (ii) the actual value charged to the credit card account. Finally, the service *shipment*, provided by a third-party shipper,

is used as the last step to request delivering the ordered book(s) (4,7) to the customer's address (6). Since this may involve a fee charged by the shipper, the credit card number (5) is required again to debit the fee. Upon completion, this service also produces an acknowledge (9) informing about whether it actually accepted the request and if so, the expected delivery date and the value charged to the credit card account.

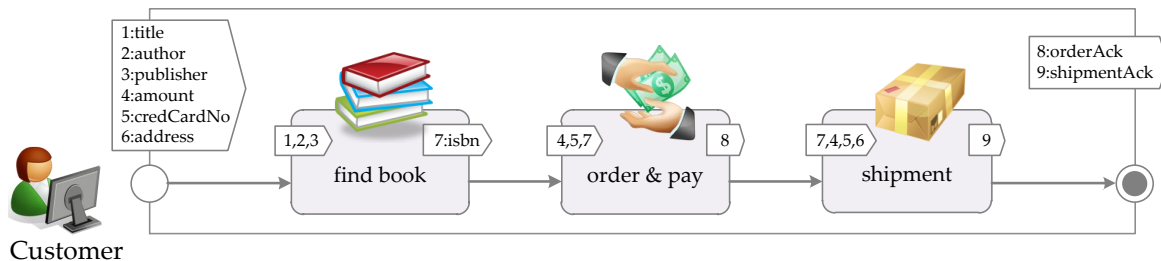


Figure 2.1: Example e-commerce scenario: Book Seller.

If this CS is executed in an automated way by a system on behalf of a customer, the customer certainly expects particular guarantees. Apart from correctness guarantees (e.g., the correct book is ordered, the credit card is not charged incorrectly), this also includes guarantees concerning the overall goal that the CS is supposed to achieve. From a transactional point of view, this basically refers to the atomicity property: either the expected result is achieved upon completion or a state is preserved as if it were never executed. This is commonly referred to as the all-or-nothing rule. For instance, once the book was ordered and paid it should be asserted that it will be delivered (e.g., by issuing a shipment acknowledge). Likewise, if the book would not be found or if placing the order failed (e.g., because it was not on stock or the service was temporarily off-line for maintenance), the CS should terminate as if it were never executed.

On the other hand, under some conditions, it would still be possible to achieve the overall result. For example, the customer might be fine with buying the book from another book seller. There might be other online book sellers available offering their own online book selling services. Also, the book seller might run multiple *order & buy* services for different sites. If the services provided by the other book sellers or the additional site services qualify as *semantically equivalent*, the system executing the CS can recover in a forward-oriented way instead. After having determined a semantically equivalent alternative service, the system would intervene by modifying the “default” execution flow. In the example, the original *order & pay* service would be replaced by an alternative. Finally, the system would resume execution from its current position with invocation of the replacement; thus, allowing to complete in a way that would still satisfy the user's needs. This is the basic idea underlying the CFI approach.

## 2.2 E-Health Scenario

In this scenario, we consider an emergency medical assistance application scenario. It was subject to the design, implementation, and evaluation in the EU-funded interna-



tional research project CASCOM [BLF<sup>+</sup>06]. The scenario starts from a person that faces a situation where she/he needs to request (immediate) medical assistance because of a sudden disease or emergency. Further actions triggered to handle the case include (i) the selection and activation of a local ambulance, (ii) the gathering of (recent) medical data from the persons medical record, and (iii) the submission of this data to a mobile device carried by the emergency physician who is in charge of giving primary care. The latter aims at providing the physician with relevant medical information about the person in order to gain prior insight in its past and current health state, current medication, allergies, or drug intolerances.

Considering a service based coverage of this scenario, specialized services would have to exist that provide assistance in identifying the person that issued the request, to discover, select and trigger a local ambulance, to query the person's medical record for relevant information and documents, and to transfer them to the physicians mobile device. For this purpose, a CS like the one shown in Figure 2.2 may have been created.

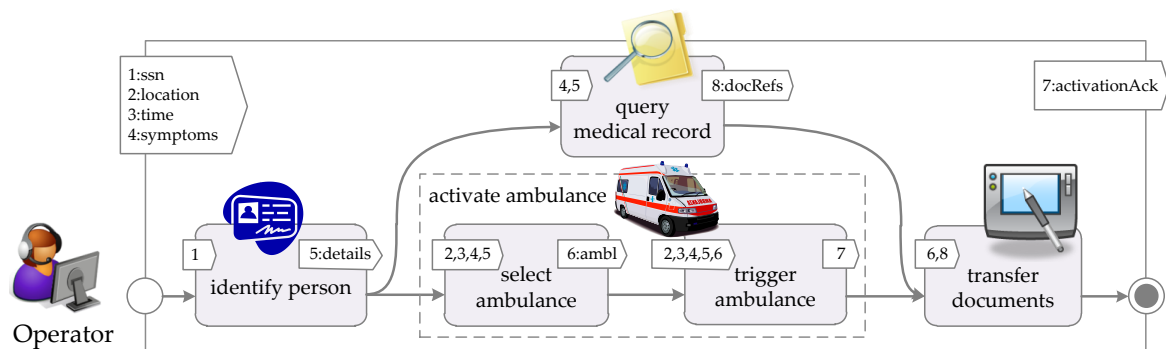


Figure 2.2: Example e-health scenario: Emergency Assistance.

This CS may be invoked by a telephone operator in a local emergency dispatch center. In a more visionary setting, it may also be an intelligent agent running on a mobile device carried by the person in need of emergency care, that automatically triggers the service by placing the request. No matter how it gets invoked, as a first step a service *identify person* is used to identify and to retrieve personal details. This may be done based on the social security number (1) or other personal data. Afterwards, the service splits into two paths that can be performed in parallel; that is, a precedence order exists only for subsequent services within a path but not between paths.<sup>1</sup> The lower path consists of another composite service *activate ambulance* whose decomposition is also shown. Typically, it would be provided by a local emergency center. This embedded CS is responsible for selecting a local ambulance and to activate it subsequently. The selection would typically be done based on criteria supporting the decision such as the location of the person (2), the time when the request was received (3), symptoms that were reported (4), and personal details (5); the latter two being optional. The service *trigger ambulance* actually triggers an alarm signal on a mobile device used by the crew of the selected ambulance (6) accompanied by submitting the mission information

<sup>1</sup>Note that this statement loses its generality as soon as synchronization primitives between parallel paths would be introduced.

(2-5). Its output is a positive or negative acknowledge (7) confirming whether the activation was successful or not. The second parallel path consists of the service *query medical record*. Imagine this service as being able to retrieve document references (8) to (relevant) documents from various sources of the medical record of some person (5), based on, e.g., a set of search keywords (4). Because this is likely to be a rather complex task, it is expected to be realized by yet another CS. However, its actual structure is not of particular interest here, which is the reason why it is not further decomposed. Finally, the two parallel paths join so that as a last step the service *transfer documents* is invoked upon completion of both paths. This service takes the document references found by the *query medical record* service and submits them to the mobile device of the ambulance crew based on the ambulance identifier that has been selected and activated before. The physician can then use this device to download and read the medical documents, assuming that it is authorized to access them. Ideally, this is done while still being on the way to the person's place.

Similar to the book seller scenario, computer aided execution of this CS would be hardly accepted if certain execution guarantees were not provided. For instance, if the service *trigger ambulance* fails. In this case, a crucial part of the overall result cannot be achieved. Such failures may happen, for instance, for technical reasons when the connection to the mobile device used by the ambulance crew could not be established. Even for non-technical reasons when a negative acknowledge is returned by this service, e.g., because the ambulance crew was already busy with handling another mission. A forward-oriented strategy to automatically recover from these service failures can be achieved by online replacement of the entire service *activate ambulance* by a semantically equivalent service. The fact that a qualifying alternative service would be available in practice can be assumed in this application scenario because regions are usually covered by more than one ambulance center, and ambulance centers often keep more ambulances ready than statistically required.

In principle, invocation of the *identify person* service may also fail for technical reasons; albeit this should only rarely be the case because availability of such a service is required to be very high, for obvious reasons. The forward-oriented approach of CFI would also be applicable in this case provided that qualifying semantic equivalences exist. In practice, this is likely the case, for similar reasons than with *trigger ambulance*. There is almost always a redundant coverage of a region by more than one emergency center. On the other hand, assuming that the service *query medical record* or *transfer documents* fails should not cause the overall CS to fail as their success is not crucial. The result of these services is basically optional. The emergency physician should be able to handle the medical case sufficiently even without additional information from the health record of the person.

It was mentioned above that execution of the *emergency assistance* CS might be automatically triggered by a software agent running on a mobile device carried by the person. As a matter of the fact that mobile (as well as embedded) devices do have limited resources compared to stationary devices, it might be inadvisable to execute the CS completely on the local execution engine running on the mobile device. For instance, battery capacity might be low, network bandwidth might be small, wireless network connectivity might not be as reliable as a wired network connection, and computational

and/or memory resources might also be insufficient. Instead of a single local execution system, a distributed and decentralized execution system consisting of multiple interacting engines deployed to physically separated devices (nodes) would be advantageous, as nodes can cooperate in sharing the execution task. This allows to overcome two problems. First, the problem of limited and/or less reliable resources. Having the possibility to migrate an ongoing execution from the engine running on the mobile device to an engine running on a better equipped and/or more reliable (stationary) device would likely be beneficial in this case. Just imagine the mobile device would run out of power while execution is still in progress. Second, the problem of centralized resources. A distributed and decentralized system not requiring supervision or global coordination of ongoing executions also facilitates a high degree of scalability as it does prevent the need for central resources which may become bottlenecks.

## 2.3 Application Dynamics and Consequences

Inherent to the e-commerce and e-health application scenario presented above is a high degree of application dynamics. Rather than being composed of the same services for all possible users, the composite services would consist of different services for different groups of users. In case of the book seller scenario, it can almost be taken for granted nowadays that enterprises target customers on a nationwide level, if not even on the global level. For instance, the same enterprise may run multiple instances for different countries. Also, it is the nature of any market that more or less many (competing) enterprises act on it; hence, there will be more than one book seller, in the same way as there will be more than one shipper, all providing their own services. These two dimensions span an area of semantically equivalent or similar services from which the book order CS can be composed. What is more, users usually have preferences for service selection (e.g., a book seller close to the customers place or a book seller known to have a high reputation). Albeit not the primary scope of this thesis, this calls for ad hoc composition of the CS using (semi) automatic service composition methods [RS04]. An important consequence is that there is not a single common book seller CS, but rather different ones. Each of them would be executed probably just once; at least fewer times than a common CS used by many users. This calls for a flexible, self-contained, and peer-to-peer like approach to distributed execution. We have found that the architectural design of a distributed execution system and its strategy used to coordinate execution among nodes is a result of application workload patterns, based on the general technical setting. Given the workload of (i) many different CSs that are (ii) executed few times only and where single services are (iii) rather short-running, an approach is required that does not come with initial overhead required to subsequently coordinate execution.

These kind of dynamics and its consequences apply in a similar way to the emergency medical assistance scenario. Here, however, the current local place of the person having urgent health problems determines which *identify person* and *activate ambulance* service would be a candidate to be used – they should be close together. Obviously, as a matter of the emergency-related setting, services are also required to be short-running.



# 3

## Fundamentals

THE PURPOSE OF THIS CHAPTER is to give an overview of (i) the theory of *Description Logics* (DLs), (ii) the *Web Ontology Language* (OWL) and how it relates to DLs, and (iii) to briefly introduce the *Resource Description Framework* (RDF). The former are a family of languages for representing knowledge in a way that enables computers to reason about it. DLs have become very popular in recent years. They are successfully used in various application domains such as medical informatics (e.g., [Ope, Int]) and digital libraries (e.g., [KM09]). Most notably, DLs are the underlying formalism of OWL, the de-facto knowledge representation framework in the Semantic Web. RDF is another framework that provides a (lower-level) general data model for conceptual description and modeling of information in the Semantic Web.

As stated in [Chapter 1](#), CFI proposes the use of the DL based approach to formalize and reason about the semantics of services. OWL and RDF, on the other hand, are used as a concrete representation means. This chapter is, therefore, included to provide a sound basis for subsequent chapters and to make this thesis self-contained. For reasons of brevity, the presented level of detail is representative but not exhaustive. For instance, the section on DLs entirely skips inferencing procedures. Readers already familiar with those theories may skim through the sections to get familiar with the exact notation that is being used.

### 3.1 Description Logics

Description Logics (DLs) are a family of knowledge representation formalisms based on deductive logic based reasoning. They evolved from early frame-based systems [Min74] and semantic networks [Qui67] developed in the 1970s. These early systems were, however, not fully satisfactory because of their lack of precise semantic characterization. As a result, reasoning results were strongly dependent on the implementation strategies (i.e., for the same input different tools may return different results). The question then arose as to (i) how to provide formal semantics to those knowledge representations so

that (ii) reasoning procedures can be built that are sound<sup>1</sup> and complete<sup>2</sup> with respect to the intended semantics. One important step towards DLs was the recognition that many of the features to express structures and relationships in frames and semantic networks could be given a semantics by relying on first-order logic (FOL); and that already a fragment of FOL is sufficient to express them [BL85]. Increasing levels of modeling expressivity have been introduced over time. Not surprisingly, however, it turned out that higher expressiveness results in harder computational complexity of reasoning up to constructs that are undecidable, in general. Yet, the computational complexity resulting from the various DL constructs is well understood (for a summary see [Zol]) and most recent DLs are mostly limited to be decidable. If not, then because deliberate decisions were made to offer very high expressivity to application domains where automated reasoning is not of utmost importance. On the other hand, DLs trading expressive power for performance of reasoning tasks have been defined (e.g., [LB87, BKM99, BBL08]) in which reasoning procedures are known to be tractable. Altogether, the family of DL languages is probably the most thoroughly understood set of formalisms in all of knowledge representation.

The basic notions in DLs are *individuals* (a.k.a. objects), *concepts* (a.k.a. classes), and *roles* (a.k.a. properties).<sup>3</sup> Altogether, they make up the *vocabulary* (or names) of the domain of interest. In short, an individual name identifies a physical or virtual object existing in the domain of interest such as *J.S. Bach* or *Basel*. Concept names identify the abstract notions of the domain and are essentially classes of individuals such as *Composer*, *City*, *Book*, or *Hospital*. Role names identify the relations among individuals such as *writtenBy* or *partOf*. Concepts and roles are used, first, for modeling a (hierarchical) structure (a.k.a. terminology) representing *intensional* knowledge about the domain or “world” by means of terminological *axioms*. The entire vocabulary is then used to describe *extensional* knowledge by making *assertions* about the individuals. Both axioms and assertions are statements that are true by definition in the world. Assertions, in particular, express concept memberships of individuals, relationships among individuals using roles, and individual name (in)equalities if the use of alias names is permitted. Informally, extensional knowledge describes the state of affairs in the domain. Extensional knowledge is thought to be contingent or dependent on a single set of circumstances and therefore subject to occasional or even constant change. Intensional knowledge, however, is thought to change seldom – to be “timeless”, in a way. Because terminological knowledge is clearly different from assertional knowledge, they get represented by dedicated containers, called TBox, RBox, and ABox.<sup>4</sup> They contain

---

<sup>1</sup>In mathematical logic, a deductive reasoning system is sound iff its rules of proof do not allow for a false inference from a true premise. If a reasoning system is sound and its axioms are true then its theorems are also guaranteed to be true.

<sup>2</sup>The converse of the soundness property is the completeness property. A deductive reasoning system is complete iff there are no true sentences that cannot – at least in principle – be proved by the reasoning system. In other words, every logical consequence can be deduced.

<sup>3</sup>Individuals, concepts, and roles correspond to constants, unary, and binary predicates, respectively, in FOL.

<sup>4</sup>In the literature, the RBox is often considered part of the TBox; hence, it is not distinguished from the TBox.

concept inclusion axioms, role inclusion axioms, and individual assertions, respectively. Together they make up a *knowledge base*.

Another aspect of knowledge representation based on DLs is the integration of *concrete domains* (i.e., pre-defined) such as numbers, strings, date times, and so on. Almost all “data” oriented applications require such a feature in order to express binary relationships whose *range* of allowed values – the codomain – maps to a concrete domain. In the application scenarios described in [Chapter 2](#) we find, for instance, the publication *date* of a book, the credit card *number* of a customer, the *weight* of a person, the blood *pressure* or respiration *rate* of a patient, the social security *number* of a person, the departure and (expected) arrival *time* of an ambulance, or the *costs* of using some service. All these examples express (finite) ranges of values over concrete domains such as integers, date times, real numbers, strings, that are often also associated with a measurement unit. In early versions of DLs, extensions to such concrete domains were designed in an ad hoc way unless a general method was established for integrating knowledge about concrete domains within a DL language [BH91]. Recent works then addressed the integration of datatypes to form *datatype maps*, defined the notion of *data ranges*, and analyzed the aspect of decidability and computational complexity of reasoning with common data types [HS01, MH08].

In the following two sections we summarize the syntax and semantics of the two most widely known DLs today, namely *SHOIN* and *SROIQ*. The former corresponds to DL “species” of version 1.0 of the Web Ontology Language (OWL DL) [MH04], while the latter underlies version 2.0 [W3C09]. At the time of writing, the latter is the latest OWL release. [Section 3.1.3](#) then introduces the extension to integrate concrete domains. [Section 3.1.4](#) briefly discusses the main reasoning tasks and lists important computational complexity results. [Section 3.1.5](#) discusses basic operations over knowledge bases, their assumptions, and implications. The definitions follow closely the corresponding literature [HS01, HPSH03, HKS06, BCM<sup>+</sup>07, MH08].

### 3.1.1 Description Logic *SHOIN*

*SHOIN* belongs to DLs of high expressive power. In short, it allows to define (i) transitive and non-transitive roles, (ii) their inverse, (iii) comprises various constructors to create complex concept expressions, and (iv) allows to describe inclusion hierarchies over roles and concepts. These are syntactically defined as follows.

#### Syntax of *SHOIN*

**Definition 3.1** (*SHOIN* roles). Let  $V_{OP}$  be a countable set of role names.<sup>5</sup> The set of *SHOIN* roles (or roles for short) is  $V_{OP} \cup \{R^- \mid R \in V_{OP}\}$ , where  $R^-$  is the inverse role of  $R$ . A role transitivity axiom is of the form  $\text{Tra}(R)$  where  $R \in V_{OP}$ .

A role inclusion axiom is of the form  $R \sqsubseteq S$ , for two roles  $R$  and  $S$ , called sub and super role, respectively. A role is simple if it is not transitive and none of its subroles is transitive.

<sup>5</sup>We use the subscript *OP* to indicate that roles are called object properties in OWL and for distinguishing them later on from concrete roles (a.k.a. data properties).

Since there is also the notion of concrete roles, which will be introduced later when we introducing data ranges, a role  $R \in V_{OP}$  is also called an *abstract* role.

**Definition 3.2** (*SHOIN* concepts). Let  $V_C, V_I$  be countable sets of concept names and individual names respectively, that may, under certain restrictions, have non-empty intersections.<sup>6</sup> The set of *SHOIN* concepts (or concepts for short) over the vocabulary  $V_C, V_I$ , and roles is inductively defined as follows.

- (1) Every concept name  $A \in V_C$  is a concept.
- (2)  $\top$  (top),  $\perp$  (bottom) are concepts.
- (3) If  $C, D$  are concepts,  $R$  is a role,  $S$  is a simple role,  $a \in V_I$  is an individual name, and  $n, m$  are non-negative integers, then the following are also concepts:
  - $\neg C$  (negation),  $C \sqcap D$  (conjunction),  $C \sqcup D$  (disjunction),
  - $\{a_1, \dots, a_m\}$  (nominal),
  - $\exists R.C$  (existential restriction),  $\forall R.C$  (universal restriction),
  - $\geq n S$  (min cardinality restriction),  $\leq n S$  (max cardinality restriction).<sup>7</sup>

A concept is atomic if it is a concept name. A concept is complex otherwise.

The operators available in a DL to formulate complex concepts (roles) are also called concept (role) *constructors*. If every constructor available in a DL  $\mathcal{L}_1$  is also in another DL  $\mathcal{L}_2$  then  $\mathcal{L}_1$  is said to be a *sublanguage* of  $\mathcal{L}_2$ . For instance, the basic DL  $\mathcal{ALC}$ <sup>8</sup> is a sublanguage of *SHOIN* because it contains top, bottom, negation, conjunction, disjunction, existential restriction, and universal restriction.

As will be seen when formal semantics are given to the symbols and constructors, the symbol  $\top$  is used to denote the universal concept (having the same semantics as  $C \sqcup \neg C$ ) and  $\perp$  denotes its complement the empty concept ( $C \sqcap \neg C$ ). In fact, in DLs that allow for negation there are dualities regarding constructors: Conjunction is dual to disjunction under negation since  $C \sqcap D \Leftrightarrow \neg(\neg C \sqcup \neg D)$  (De Morgan's law) where  $\Leftrightarrow$  means logically equivalent. Analogously, existential and universal restriction are dual under negation ( $\exists R.C \Leftrightarrow \neg \forall R. \neg C$  and  $\forall R.C \Leftrightarrow \neg \exists R. \neg C$ ). Consequently, for each complex concept created from these constructors there is always an equivalent dual concept; hence, such DLs provide syntactic sugar.

## Notational Conventions

Throughout this thesis, we adopt the following notational conventions in definitions and examples:

<sup>6</sup>With *punning*, a meta modeling technique allowing to reuse names in cases in which it is possible to disambiguate the exact use of a name,  $V_C, V_I$  and the set of role names  $V_{OP}$  need not, under certain restrictions, be mutually disjoint. For more details see <http://www.w3.org/2007/OWL/wiki/Punning>.

<sup>7</sup>Number restrictions are limited to simple roles in order to retain decidability [HST99].

<sup>8</sup> $\mathcal{ALC}$  stands for *Attributive Language with Complement*.



- Definitions and abstract examples are typeset in math mode. The upper-case letters  $A, B$  are used for atomic concepts ( $A, B \in V_C$ );  $C, D$  for concepts;  $R, S$  for roles where  $S$  sometimes denotes a simple role; the lower-case letters  $a, b$  for individual names ( $a, b \in V_I$ );  $m, n$  for natural numbers.
- Concrete examples are typeset in slanted font shape. Concept names start with an uppercase letter followed by lowercase letters (e.g., *Customer, Person, Book*), role names start with a lowercase letter (e.g., *hasOrdered, likes*), and individual names are composed of uppercase letters (e.g., *ALICE, BOB*).

### Terminological Knowledge

No matter which particular DL is considered, concepts and roles are used in *terminological axioms* to express how concepts or roles relate to each other in the domain of interest. In the most general case, these axioms have the form

$$C \sqsubseteq D \quad (R \sqsubseteq S) \quad \text{or} \quad C \equiv D \quad (R \equiv S) .$$

The former are called *inclusions* whereas the latter are called *equalities*. Informally, an inclusion states that the right-hand side concept (role) subsumes the left-hand side (i.e., a super concept/role that is more general than the sub concept/role). In logical terms, an inclusion  $C \sqsubseteq D$  says that in order to be a member of  $D$  it is sufficient to be a member of  $C$  and that it is necessary to be a member of  $D$  to be a member of  $C$ . In fact, an inclusion can be understood as an implication  $C \rightarrow D$ ; we will come back to this later when detailing OWL 2 RL in [Section 3.3.3](#). This is analogous the case for role inclusions  $R \sqsubseteq S$ . Equalities, on the other hand, express that two concepts (roles), even though they might differ in *intension*, have the same *extension*. This means that an equality  $C \equiv D$  expresses necessary and sufficient conditions for concept membership in either direction; analogous for role equalities  $R \equiv S$ . Therefore, an equality  $X_1 \equiv X_2$  is an abbreviation for two symmetric inclusion axioms  $X_1 \equiv X_2 \Leftrightarrow X_1 \sqsubseteq X_2 \wedge X_2 \sqsubseteq X_1$  where  $X_1, X_2$  are either concepts or roles.

An inclusion  $C \sqsubseteq D$  where the left-hand side might possibly be a complex concept is called a *general concept inclusion axiom* (GCI). An equality  $A \equiv C$  whose left-hand side is an atomic concept  $A \in V_C$  is called a *concept definition*. Equalities of this kind are most often used to introduce symbolic names for complex concepts.

We will now formally define the most general form of collections of these axioms.

**Definition 3.3** (RBox). *An RBox  $\mathcal{R}$  is a finite set of role inclusion axioms of the form  $R \sqsubseteq S$  and transitivity axioms  $\text{Tra}(R)$  where  $R, S$  are roles.*

**Definition 3.4** (TBox). *A TBox  $\mathcal{T}$  is a finite set of general concept inclusion axioms of the form  $C \sqsubseteq D$  where  $C, D$  are concepts.*

An atomic concept occurring on the left-hand side of a concept definition in  $\mathcal{T}$  is *defined* whereas an atomic concept that only occurs on the right-hand side in  $\mathcal{T}$  is *primitive*. Analogously, we can speak of primitive and defined roles. As a matter of fact, instances of primitive concepts can only be declared explicitly; how this can be done follows below.

Unfortunately, the presence of GCIs in the TBox, which is correspondingly called *general* then, causes worst-case computational complexity of terminological reasoning to become intractable. Therefore, a restricted form of the TBox is often considered where reasoning has tractable complexity for several DLs such as ALC [Lut99]. More precisely, an *acyclic* TBox  $\mathcal{T}$  is a TBox such that

- $\mathcal{T}$  contains only concept definitions,
- there is at most one definition for each concept name  $A \in V_C$  in  $\mathcal{T}$ , and
- there does not exist a concept definition  $A \equiv C$  in  $\mathcal{T}$  where  $A$  occurs either directly or indirectly in  $C$  (i.e., if the definition of  $A$  does not transitively use itself).

If the last item does not hold then  $\mathcal{T}$  is called *cyclic*. The characteristic of acyclic TBoxes is that they are unequivocal regarding each defined concept. Furthermore, the extension of each defined concept is uniquely determined by the extension of primitive concepts. As a result, it is possible to compile away an acyclic TBox by a technique called *unfolding*: iteratively replace defined concepts occurring on the right-hand side of a concept definition by its definition unless only primitive concepts occur on the right-hand side of each concept definition.

### Assertional Knowledge

The second part of knowledge representation means expresses the state of affairs in a domain: In the ABox one uses concepts and roles to make *assertions* about individuals. These assertions have the form

$$C(a), \quad R(a, b), \quad a = b, \quad a \neq b .$$

A *concept assertion*  $C(a)$  states that  $a$  is a member of  $C$ . A *role assertion*  $R(a, b)$  is used to state that  $b$  is a *filler* of the role  $R$  for  $a$ . More intuitively,  $a$  is related to  $b$  through  $R$ .

In contrast to the relational data model [Cod70, Cod90], DLs usually do not adopt the *Unique Name Assumption* (UNA). In the absence of the UNA the same individual might have different names (i.e., there can be aliases). This means that given only two distinct individual names  $a, b$  without having further knowledge about them one can neither conclude that they identify different individuals nor the same individual. Hence, the absence of the UNA necessitates the latter two types of assertions. An *individual equality*  $a = b$  asserts that the individual names  $a$  and  $b$  represent the same individual (i.e.,  $a$  and  $b$  are different names identifying the same individual) and an *individual inequality*  $a \neq b$  asserts the opposite (i.e., that  $a$  and  $b$  represent distinct individuals in the domain). It should be clear that (in)equalities are dispensable under the UNA.

**Definition 3.5** (ABox). *An ABox  $\mathcal{A}$  is a finite set of assertions of the form  $C(a), R(a, b), a = b, a \neq b$  where  $a, b \in V_I, C$  a concept, and  $R$  a role.*

## Knowledge Base

Depending on the context, slightly different definitions of a knowledge base (KB) can be found in the literature. One possibility is to state that it consists of a TBox  $\mathcal{T}$ , an RBox  $\mathcal{R}$ , and an ABox  $\mathcal{A}$ , denoted with  $\mathcal{K} = (\mathcal{T}, \mathcal{R}, \mathcal{A})$ . More often, however, a KB is defined to be made up by a TBox and an ABox only. This is achieved by defining the TBox to also contain role inclusion axioms (of an RBox) in addition to concept inclusions. Finally, one can define a KB as the union of a TBox and an ABox. Formally,

$$\mathcal{K} := (\mathcal{T}, \mathcal{A}) \quad \text{or} \quad \mathcal{K} := \mathcal{T} \cup \mathcal{A} . \quad (3.1)$$

To simplify expositions, we will adopt the definition as a pair by default but sometimes prefer the latter for convenience.

Given a DL  $\mathcal{L}$ , a knowledge base  $\mathcal{K}$  is an  $\mathcal{L}$ -*knowledge base* if and only if each complex concept and complex role occurring in the axioms and assertions in  $\mathcal{K}$  is built using the constructors available in  $\mathcal{L}$ . Analogously, we can speak of  $\mathcal{L}$ -*concepts*,  $\mathcal{L}$ -*inclusions*,  $\mathcal{L}$ -*assertions*,  $\mathcal{L}$ -*TBoxes*, and  $\mathcal{L}$ -*ABoxes*.

Finally, we use the general term *syntactic construct* to refer to any of the different types of axioms and assertions defined by some DL. The term *syntactic instance* is used to refer to a concrete instance of any of the available syntactic constructs. Consequently, we can say that a knowledge base is a set of syntactic instances.

## Semantics of $\mathcal{SHOIN}$

Now that we have seen how the DL  $\mathcal{SHOIN}$  is syntactically defined, “meaning” is given to concepts, roles, assertions, and axioms in a formal way. This is achieved in terms of *model-theoretic semantics* [Tar56], enabling to interpret each of them in a non-empty domain of interest.

**Definition 3.6** (Semantics of  $\mathcal{SHOIN}$ ). *An interpretation for  $\mathcal{SHOIN}$  is a tuple  $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$  where  $\Delta^{\mathcal{I}}$  is the non-empty interpretation domain. The interpretation function  $\cdot^{\mathcal{I}}$  assigns each individual name  $a \in V_I$  to an individual  $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$ , each concept name  $A \in V_C$  to a subset  $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ , and each role name  $R \in V_{OPN}$  to a subset  $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ . The interpretation function is extended to transitive, inverse roles and complex concepts as shown in Table 3.1.*

*If  $C$  and  $R$  is a concept and role, respectively, then  $C^{\mathcal{I}}$  and  $R^{\mathcal{I}}$  is called the extension of  $C$  and  $R$ , respectively, in  $\mathcal{I}$ . An individual name  $a \in V_I$  represents an instance of a concept  $C$  if  $a^{\mathcal{I}} \in C^{\mathcal{I}}$  in  $\mathcal{I}$ .*

*An interpretation  $\mathcal{I}$  satisfies a GCI  $C \sqsubseteq D$  if  $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ ; analogous for role inclusion axioms, see Table 3.1. An interpretation  $\mathcal{I}$  satisfies a TBox  $\mathcal{T}$ , written  $\mathcal{I} \models \mathcal{T}$ , if it satisfies all axioms in  $\mathcal{T}$ . Such an interpretation is called a model of  $\mathcal{T}$ .*

*An interpretation  $\mathcal{I}$  satisfies ABox assertions  $C(a)$  if  $a^{\mathcal{I}} \in C^{\mathcal{I}}$ ,  $R(a, b)$  if  $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$ ,  $a = b$  if  $a^{\mathcal{I}} = b^{\mathcal{I}}$ , and  $a \neq b$  if  $a^{\mathcal{I}} \neq b^{\mathcal{I}}$ .  $\mathcal{I}$  is a model of an ABox  $\mathcal{A}$ , written  $\mathcal{I} \models \mathcal{A}$ , if it satisfies all assertions in  $\mathcal{A}$ .*

*An interpretation  $\mathcal{I}$  is a model of the knowledge base  $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ , written  $\mathcal{I} \models \mathcal{K}$ , if it is a model of both  $\mathcal{T}$  and  $\mathcal{A}$ . If there exists a model  $\mathcal{I}$  for  $\mathcal{K}$  then  $\mathcal{K}$  is said to be consistent (or satisfiable); synonymously, we say that  $\mathcal{A}$  is consistent w.r.t.  $\mathcal{T}$ .*

Table 3.1: Syntax and Semantics of  $\mathcal{SHOIN}$  concept expressions and roles and corresponding OWL constructs

Ex.	DL		OWL
	Syntax	Semantics	
$\mathcal{S}$	$A$	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$	Class
	$\top$	$\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$	Thing
	$\perp$	$\perp^{\mathcal{I}} = \emptyset$	Nothing
	$R$	$R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$	ObjectProperty
	$\text{Tra}(R)$	$R^{\mathcal{I}} = (R^{\mathcal{I}})^+$	TransitiveProperty
	$C \sqcap D$	$(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$	intersectionOf
	$C \sqcup D$	$(C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$	unionOf
	$\neg C$	$(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$	complementOf
	$\exists R.C$	$\{x \mid \exists y.(x, y) \in R^{\mathcal{I}} \text{ and } y \in C^{\mathcal{I}}\}$	someValuesFrom
	$\forall R.C$	$\{x \mid \forall y.(x, y) \in R^{\mathcal{I}} \text{ implies } y \in C^{\mathcal{I}}\}$	allValuesFrom
$\mathcal{H}$	$R_1 \sqsubseteq R_2$	$R_1^{\mathcal{I}} \subseteq R_2^{\mathcal{I}}$	subPropertyOf
$\mathcal{O}$	$\{a_1, \dots, a_n\}$	$\{a_1, \dots, a_n\}^{\mathcal{I}} = \{a_1^{\mathcal{I}}\} \cup \dots \cup \{a_n^{\mathcal{I}}\}$	oneOf*
	$\exists R.\{a\}$	$\{x \mid (x, a^{\mathcal{I}}) \in R^{\mathcal{I}}\}$	hasValue
$\mathcal{I}$	$R^-$	$\{(x, y) \mid (y, x) \in R^{\mathcal{I}}\}$	inverseOf
$\mathcal{N}$	$\geq n S$	$\{x \mid \#\{y \mid (x, y) \in S^{\mathcal{I}}\} \geq n\}$	minCardinality
	$\leq n S$	$\{x \mid \#\{y \mid (x, y) \in S^{\mathcal{I}}\} \leq n\}$	maxCardinality
$\mathcal{F}$	$\text{Fun}(S)$	$(x, y) \in S^{\mathcal{I}} \text{ and } (x, z) \in S^{\mathcal{I}} \text{ implies } y = z$	FunctionalProperty

$\#N$  denotes the cardinality of the set  $N$ .

$\text{Fun}(S)$  is a syntactic variant of  $\leq 1 S$  (listed for completeness sake).

Based on [Definition 3.6](#) we can formally express the semantics of the UNA.

**Definition 3.7** (Unique Name Assumption). *An interpretation  $\mathcal{I}$  respects the unique name assumption iff  $\cdot^{\mathcal{I}}$  is an injection regarding interpretation of individual names; that is,  $\forall a, b \in V_I: a^{\mathcal{I}} = b^{\mathcal{I}} \text{ implies } a = b$ .*

The cases where  $\cdot^{\mathcal{I}}$  is a surjection or a bijection regarding individual names are respectively referred to as the *parameter names assumption* (i.e., no unnamed individuals) and *standard names assumption* (e.g., the identity function  $a^{\mathcal{I}} = a$ ).

In contrast to database schema frameworks such as the ER model [Che76, SS77] that make the assumption of defining a single model, a KB does not define a single model. In fact, it can be seen as a set of constraints that may be satisfied by a possibly infinite set of models. This is easiest to understand when considering that no constraints at all – the empty KB – means that any model is possible. Adding more constraints usually means fewer models, up to the point where no model remains possible due to the existence of contradictory constraints. The latter is called an *inconsistent* KB (see [Definition 3.6](#)). Speaking of constraints here, it is important not to confuse this with the notion of *integrity constraints*. “Rather than being statements about the world, [integrity] constraints are statements about what the KB can be said to know” [Rei88]. In other words, integrity constraints are meant for enforcing the acceptable states (content) of the KB.

Yet one might want to interpret axioms in the KB both as integrity constraints and in the “standard” way [MHS09, TSBM10].

Moreover, different assumptions on the cardinality of the domain  $\Delta^{\mathcal{I}}$  can be made; hence, on the size of models. In open environments such as the Web one typically assumes an open (infinite) domain  $\Delta^{\mathcal{I}}$  since one cannot assume to have complete knowledge. On the other hand, one can strictly close the domain by stating  $\top \sqsubseteq \{a_1, \dots, a_n\}$  where  $a_1, \dots, a_n$  are the named individuals that shall exist in the domain. An open versus closed domain has consequences, for instance, on the existential constructor: whereas  $\exists hasFriend.\top$  can refer to new, otherwise unknown, individuals in an open domain, it does refer to an  $a_i$  in a closed domain (and without further knowledge it is not known which particular  $a_i$  this is). In contrast, a database model is considered to be a finite structure since a database is understood as a complete representation of an application domain: as noted above, the database *is* a particular model.

The aspect whether the domain is open versus closed is not to be confused with the *open world assumption* (OWA) versus the *closed world assumption* (CWA), which refer to different reasoning paradigms. The former means that given an assertion or axiom  $\psi$  and a KB  $\mathcal{K}$ , if  $\psi$  is satisfied only in some models of  $\mathcal{K}$  then neither  $\mathcal{K} \models \psi$  nor  $\mathcal{K} \not\models \psi$  can be concluded. In such a case we can merely draw the conclusion that the truth-value of  $\psi$  is *not known*;  $\mathcal{K}$  is underspecified w.r.t.  $\psi$ . In contrast, a failure to proof  $\psi$  to be true due to lack of sufficient information implies that it is false under the CWA. Formally,  $\mathcal{K} \not\models \psi$  implies  $\mathcal{K} \models \neg\psi$ . This reflects common-sense reasoning in which one conjectures  $\psi$  to be false if it is not true, which builds on the assumption that all relevant knowledge is available (as opposed to the assumption under the OWA that knowledge is incomplete). The CWA is common to databases. For instance, if the product database of a book seller does not contain a certain book then it is concluded that the book is not offered by the book store; under the OWA we can only conclude that it might be offered, but we cannot be certain based on the information stored in the database.

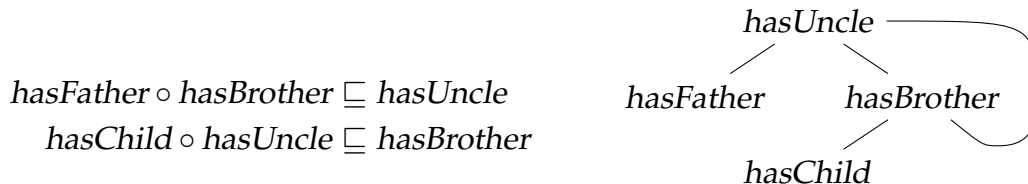
Finally, the syntactic constructs allowed in  $\mathcal{SHOIN}$  can interact in such a way that a KB does not admit finite models (i.e., it admits only models with an infinite domain  $\Delta^{\mathcal{I}}$ ). For example, DLs allowing for inverse roles, functional roles ( $\leq 1 R$ ), and cycles in the TBox [CGLN01]. In general, a DL where each concept or TBox admits a finite model is said to have the *finite model property* [EF95]. This becomes important for reasoning procedures and especially if one wants to compute and materialize the complete set of implicit axioms and assertions entailed by a KB. Obviously, this is impossible if the KB admits only infinite models.

### 3.1.2 Description Logic $\mathcal{SROIQ}$

Roughly speaking,  $\mathcal{SROIQ}$  [HKS06] extends  $\mathcal{SHOIN}$  by a number of new constructs that have proven useful in practical domains such as the medical domain and that do not affect decidability and practicability. This includes the following.

1. Complex role inclusion axioms of the form  $R_1 \circ \dots \circ R_n \sqsubseteq R$  ( $n \geq 2$ ), where  $\circ$  denotes composition of roles (binary relations). They are used to express propagation of one property along another one. For instance, one can state that the friend of an enemy is also an enemy  $hasEnemy \circ hasFriend \sqsubseteq hasEnemy$ , or that

a brother of the father is an uncle  $hasFather \circ hasBrother \sqsubseteq hasUncle$ . In order to preserve decidability, complex role inclusions in a TBox must adhere to the syntactic restriction of not forming a cycle; a negative example [MHRS06] being:



2. Qualified number restrictions of the form  $\geq nS.C$  and  $\leq nS.C$  to restrict role fillers to a certain concept, where  $S$  is a simple role (see Section 3.1.1). For instance, the axiom  $BookStoreCustomer \equiv (\geq 1 hasOrdered.Book)$  states that for being a book store customer it is sufficient and necessary to have ordered at least one book. Observe that qualified number restrictions can be seen as generalizations of existential and universal restriction since  $\exists R.C \Leftrightarrow \geq 1 R.C$  and  $\forall R.C \Leftrightarrow \leq 0 R.\neg C$ .
3. Reflexive, irreflexive, symmetric, asymmetric, and disjoint roles, denoted with  $Ref(R)$ ,  $Irr(S)$ ,  $Sym(R)$ ,  $Asy(S)$ , and  $Dis(S_1, S_2)$ , respectively, where  $S_{(i)}$  are simple roles. Note, however, that symmetric and transitive roles do not increase expressive power when inverse roles and role composition is available:  $Sym(R)$  is equivalent to  $R^- \sqsubseteq R$  and  $Tra(R)$  is equivalent to  $R \circ R \sqsubseteq R$ .
4. The universal role  $U$ . In order to preserve decidability, the universal role cannot, however, be used in complex RIAs.
5. The class restriction constructor  $\exists S.Self$ , where  $S$  is a simple role, to allow describing concepts such as a *narcissist* as  $\exists likes.Self$ .

Table 3.2 summarizes syntax and semantics of these new constructs, whereby role and concept symbols are used analogous to Definition 3.1 and Definition 3.2. Finally, negated role assertions of the form  $\neg S(a, b)$  have been added where  $S$  is a simple role. This is especially worthwhile under the OWA to make negative ABox assertions such as  $\neg likes(ALICE, THE\_LORD\_OF\_THE\_RINGS)$ . Formally, an interpretation  $\mathcal{I}$  satisfies a negated role assertion  $\neg S(a, b)$  if  $(a^{\mathcal{I}}, b^{\mathcal{I}}) \notin S^{\mathcal{I}}$ .

### 3.1.3 Datatype Maps and Data Ranges

A valuable extension to DLs are *concrete domains*, based on the framework proposed in [BH91].<sup>9</sup> In contrast to individuals from the general-purpose (or abstract) domain  $\Delta^{\mathcal{I}}$  we have considered thus far, *values* of concrete domains are “eternal” (mathematical) abstractions. Notable examples include numerical, temporal, and spatial concrete domains, the latter of which is not considered in the following. Characteristic to values of concrete domains is that their identity is determined by some procedure usually involving their structure. For instance, the strings 24.10.2010 and 2010/10/24 both identify

<sup>9</sup>Combinations of “standard” DLs with concrete domains are commonly denoted by appending (D) to the name of the DL (e.g.,  $SHOIN(\mathbf{D})$ ,  $SROIQ(\mathbf{D})$ ).

Table 3.2: Additional Constructs in  $\mathcal{SROIQ}$  and their Semantics

Ex.	DL		OWL
	Syntax	Semantics	
$\mathcal{S}$	$\exists S.\text{Self}$	$\{x \mid (x, x) \in S^{\mathcal{I}}\}$	SelfRestriction
	$U$	$U^{\mathcal{I}} = \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$	topObjectProperty
$\mathcal{R}$	$S_1 \circ \dots \circ S_n \sqsubseteq R$	$S_1^{\mathcal{I}} \circ \dots \circ S_n^{\mathcal{I}} \subseteq R^{\mathcal{I}}$	propertyChain
$\mathcal{Q}$	$\geq nS.C$	$\{x \mid \#\{y \mid (x, y) \in S^{\mathcal{I}} \text{ and } y \in C^{\mathcal{I}}\} \geq n\}$	minCardinalityQ with onClass
	$\leq nS.C$	$\{x \mid \#\{y \mid (x, y) \in S^{\mathcal{I}} \text{ and } y \in C^{\mathcal{I}}\} \leq n\}$	maxCardinalityQ with onClass
Role assertions	$\text{Sym}(R)$	$(x, y) \in R^{\mathcal{I}} \text{ implies } (y, x) \in R^{\mathcal{I}}$	SymmetricProperty
	$\text{Asy}(S)$	$(x, y) \in S^{\mathcal{I}} \text{ implies } (y, x) \notin S^{\mathcal{I}}$	AsymmetricProperty
	$\text{Ref}(R)$	$\text{Diag}^{\mathcal{I}} \subseteq R^{\mathcal{I}}$	ReflexiveProperty
	$\text{Irr}(S)$	$S^{\mathcal{I}} \cap \text{Diag}^{\mathcal{I}} = \emptyset$	IrreflexiveProperty
	$\text{Dis}(S_1, S_2)$	$S_1^{\mathcal{I}} \cap S_2^{\mathcal{I}} = \emptyset$	disjointObject- Properties

$\circ$  is overloaded to denote the standard composition of binary relations.

$\#N$  denotes the cardinality of the set  $N$ .  $\text{Diag}^{\mathcal{I}}$  denotes the set  $\{(x, x) \mid x \in \Delta^{\mathcal{I}}\}$ .

the same date – October 24<sup>th</sup> 2010 – after normalization from the (locale-specific) *lexical space* into the *value space*, which is done by a procedure based on the structure.

Since concrete domains, in their unrestricted form, have severe consequences on decidability and computational complexity of reasoning, more recent works then introduced practicable notions of *datatype maps* and *data ranges* [HS01, MH08]. In short, datatype maps are formalizations of (i) a set of available datatypes, (ii) their *value space* and *lexical space*, and (iii) *facets* and *facet expressions*. The latter are expressions over a datatype, that further restrict their range of values. Datatypes are usually thought to be unary. Nevertheless, the definitions introduced in the following can be extended to  $n$ -ary datatypes as in [PH03] (e.g., date can be considered a 3-ary datatype consisting of the components year, month, and day). Finally, data ranges are basically expressions over the elements in a datatype map.

**Definition 3.8** (Datatype Map). *A datatype map is a 4-tuple  $\mathcal{D} = (V_D, V_{LS}, V_F, \cdot^{\mathcal{D}})$ , where*

- $V_D$  is a set of datatypes  $d$ ,
- $V_{LS}$  is a function assigning a set of lexical forms  $V_{LS}(d)$  to each  $d \in V_D$  where  $V_{LS}(d)$  is called the lexical space of  $d$ ,
- $V_F$  is a function assigning a set of facets  $V_F(d)$  to each  $d \in V_D$ , and
- $\cdot^{\mathcal{D}}$  is a function assigning a datatype interpretation  $d^{\mathcal{D}}$  to each datatype  $d \in V_D$  called the value space, a facet interpretation  $f^{\mathcal{D}} \subseteq d^{\mathcal{D}}$  to each facet  $f \in V_F(d)$ , and a data value  $v^{\mathcal{D}} \in d^{\mathcal{D}}$  to each lexical form (constant)  $v \in V_{LS}(d)$ .

A facet expression for a datatype  $d \in V_D$  is a formula  $\varphi$  built using propositional connectives over the elements from  $V_F(d) \cup \{\top_d, \perp_d\}$ . The function  $\cdot^{\mathcal{D}}$  is extended to facet expressions by setting, for  $f_{(i)} \in V_F(d)$ ,  $(\top_d)^{\mathcal{D}} = d^{\mathcal{D}}$ ,  $(\perp_d)^{\mathcal{D}} = \emptyset$ ,  $(\neg f)^{\mathcal{D}} = d^{\mathcal{D}} \setminus f^{\mathcal{D}}$ ,  $(f_1 \wedge f_2)^{\mathcal{D}} = f_1^{\mathcal{D}} \cap f_2^{\mathcal{D}}$ , and  $(f_1 \vee f_2)^{\mathcal{D}} = f_1^{\mathcal{D}} \cup f_2^{\mathcal{D}}$ .

Observe that the symbols  $\top_d$  and  $\perp_d$  denote built-in universal and empty facets, respectively, specific to a datatype  $d \in V_D$ .

### Example 3.1

Imagine a datatype map  $\mathcal{D}$  with  $V_D = \{\text{string}, \text{real}\}$ , where  $\text{string}^{\mathcal{D}}$  shall be the set of all strings over some alphabet  $\Sigma$  (i.e.,  $\text{string}^{\mathcal{D}} = \Sigma^*$ ) and  $\text{real}^{\mathcal{D}}$  the set of real numbers. The set  $V_{LS}(\text{string})$  would then contain all lexical string forms and  $V_{LS}(\text{real})$  shall contain decimal representations of real numbers.<sup>10</sup> Finally, the set  $V_F(\text{real})$  might contain the facet  $\text{int}$ , interpreted as the set of integers, and facets of the form  $<_q$ ,  $>_q$ ,  $\leq_q$ , and  $\geq_q$  for decimal numbers  $q$ . The facet expression  $\text{int} \wedge >_{10} \wedge <_{20}$  would then represent the integers 11, ..., 19.

The syntax and semantics of the description logics  $\mathcal{SHOIN}(\mathbf{D})$  and  $\mathcal{SROIQ}(\mathbf{D})$  will be introduced next. In essence, they are obtained by extending the “core” description logic with a datatype system. Prior to that, the notion of data ranges is yet to be introduced.

**Definition 3.9** (Data Ranges). Let  $\mathcal{D} = (V_D, V_{LS}, V_F, \cdot^{\mathcal{D}})$  be a datatype map and let  $\mathbb{V}_{LS} = \bigcup_{d \in V_D} V_{LS}(d)$  be the union of all lexical forms over all datatypes in  $\mathcal{D}$ . The set of data ranges for  $\mathcal{D}$  is the smallest set that contains

- (1)  $\top_{\mathcal{D}}$  (universal data range),
- (2)  $d$  (datatype range),
- (3)  $d[\varphi]$  (facet data range over a datatype),
- (4)  $\{v_1, \dots, v_n\}$  (enumeration),
- (5)  $\overline{dr}$  (negated data range),

for  $d \in V_D$ ,  $\varphi$  a facet expression for  $d$ ,  $v_i \in \mathbb{V}_{LS}$ , and  $dr$  a data range.

**Definition 3.10** (Syntax of  $\mathcal{DL} + \mathcal{D}$ ). Let  $\mathcal{DL}$  be a description logic defined over the vocabulary  $(V_C, V_{OP}, V_I)$ . Let  $V_{DP}$  be a countable set of concrete role names<sup>11</sup> pair-wise disjoint from  $V_C, V_{OP}, V_I$ . Let  $\mathcal{D}$  be a datatype map. Let  $T_{(i)} \in V_{DP}$  be a concrete role,  $a \in V_I$  an individual,  $v \in \mathbb{V}_{LS}$  a lexical form,  $dr$  a data range for  $\mathcal{D}$ , and  $n$  a non-negative integer.

The logic  $\mathcal{DL} + \mathcal{D}$ , obtained by extending  $\mathcal{DL}$  with  $\mathcal{D}$ , extends the concepts of  $\mathcal{DL}$  with concepts of the form  $\exists T.dr$ ,  $\forall T.dr$ ,  $\geq nT.dr$ , and  $\leq nT.dr$ . The set of TBox axioms is extended by inclusion axioms of the form  $T_1 \sqsubseteq T_2$  and disjointness axioms of the form  $\text{Dis}(T_1, T_2)$ . The set of ABox assertions is extended by assertions of the form  $T(a, v)$ .

<sup>10</sup>Of course, one could also represent real numbers in binary or hexadecimal form.

<sup>11</sup>We use the subscript  $DP$  to indicate that they are called data properties in OWL.



Table 3.3: Model-Theoretic Semantics of  $\mathcal{DL} + \mathcal{D}$  data ranges, concepts, axioms, and assertions

Data Ranges	
$(\top_{\mathcal{D}})^{\mathcal{D}} = \Delta^{\mathcal{D}}$	$(d[\varphi])^{\mathcal{D}} = \varphi^{\mathcal{D}}$
$(\{v_1, \dots, v_n\})^{\mathcal{D}} = \{v_1^{\mathcal{D}}, \dots, v_n^{\mathcal{D}}\}$	$\overline{dr}^{\mathcal{D}} = \Delta^{\mathcal{D}} \setminus dr^{\mathcal{D}}$
Concepts	Axioms
$(\exists T.dr)^{\mathcal{I}} = \{x \mid \exists y.(x, y) \in T^{\mathcal{I}} \text{ and } y \in dr^{\mathcal{D}}\}$	$\text{Dis}(T_1, T_2) \Rightarrow T_1^{\mathcal{I}} \cap T_2^{\mathcal{I}} = \emptyset$
$(\forall T.dr)^{\mathcal{I}} = \{x \mid \forall y.(x, y) \in T^{\mathcal{I}} \text{ implies } y \in dr^{\mathcal{D}}\}$	$T_1 \sqsubseteq T_2 \Rightarrow T_1^{\mathcal{I}} \subseteq T_2^{\mathcal{I}}$
$(\geq nT.dr)^{\mathcal{I}} = \{x \mid \#\{y \mid (x, y) \in T^{\mathcal{I}} \text{ and } y \in dr^{\mathcal{D}}\} \geq n\}$	Assertions
$(\leq nT.dr)^{\mathcal{I}} = \{x \mid \#\{y \mid (x, y) \in T^{\mathcal{I}} \text{ and } y \in dr^{\mathcal{D}}\} \leq n\}$	$T(a, v) \Rightarrow (a^{\mathcal{I}}, v^{\mathcal{D}}) \in T^{\mathcal{I}}$

$\#N$  denotes the cardinality of a set  $N$ .

In order to define formal semantics for the constructs of data ranges, the datatype domain  $\Delta^{\mathcal{D}}$  needs to be introduced, which is done by [Definition 3.11](#). The function  $\cdot^{\mathcal{D}}$  is then extended for the constructs as shown in [Table 3.3](#).

**Definition 3.11** (Semantics of  $\mathcal{DL} + \mathcal{D}$ ). *An interpretation for  $\mathcal{DL} + \mathcal{D}$  is a triple  $\mathcal{I} = (\Delta^{\mathcal{I}}, \Delta^{\mathcal{D}}, \cdot^{\mathcal{I}})$ , where  $\Delta^{\mathcal{I}}$  and  $\Delta^{\mathcal{D}}$  are nonempty disjoint sets such that  $d^{\mathcal{D}} \subseteq \Delta^{\mathcal{D}}$  for each  $d \in V_D$ . The interpretation function  $\cdot^{\mathcal{I}}$  is derived from  $\mathcal{DL}$  and extended to assign to each concrete role  $T \in V_{DP}$  the interpretation  $T^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{D}}$ . Furthermore,  $\cdot^{\mathcal{I}}$  and  $\cdot^{\mathcal{D}}$  are extended to data ranges, complex concepts, axioms, and assertions as shown in [Table 3.3](#).*

An interpretation  $\mathcal{I}$  is a model of a  $\mathcal{DL} + \mathcal{D}$  knowledge base  $\mathcal{K}$ , written  $\mathcal{I} \models \mathcal{K}$ , if it satisfies all axioms of the TBox  $\mathcal{T}$  and all assertions of the ABox  $\mathcal{A}$ .

In [MH08] it is shown that, without losing generality, the assumption can be made that datatypes  $d_{(i)} \in V_D$  are pairwise disjoint; that is, if  $d_1, d_2 \in V_D$  and  $d_1 \neq d_2$  then  $d_1^{\mathcal{D}} \cap d_2^{\mathcal{D}} = \emptyset$ . This allows for a modular treatment of different datatypes for reasoning (i.e., handling a datatype  $d$  does not necessitate considering other supported datatypes  $V_D \setminus d$ ). Furthermore, it has been pointed out in [MH08] that  $\mathcal{K}$  can be interpreted by considering only those datatypes from the datatype map that are explicitly mentioned in  $\mathcal{K}$ . This is the reason why it is not necessary to define the consequence relation  $\models$  in [Definition 3.11](#) w.r.t. the entire datatype map (i.e., including those datatypes not mentioned in  $\mathcal{K}$ ) provided that  $\Delta^{\mathcal{D}}$  is the set that contains at least the interpretations for each  $d \in V_D$ , which is the case. Note, however, that the consequences of  $\mathcal{K}$  might change under the extension of a datatype map with a new datatype (i.e., if  $\Delta^{\mathcal{D}}$  is enlarged).

### 3.1.4 Reasoning and its Computational Complexity

A key feature of DLs is the possibility to reason about the axioms and assertions in a KB and to infer additional implicit knowledge, thereby making it explicit. In general, what can be inferred – the *consequences* – depends on (i) the rules of inference defined by a (description) logic and obviously on (ii) what is known already – the *premises*. The rules of inference can be defined so that the set of consequences changes either monotonically or non-monotonically with the set of premises. A (description) logic is

said to be *monotonic* only if its rules of inference do not allow for reduction of the set of consequences when new premises are added. Otherwise it is non-monotonic. Simply put, it means that learning a new piece of knowledge cannot reduce what is implicitly known by inference. Most DLs, in particular *SHOIN* and *SRIQ*, are monotonic if interpreted under OWA, while interpretation under CWA results in non-monotony since a negative consequence  $\neg\psi$  is no longer entailed if a positive assertion/axiom  $\psi$  is learned (added to a KB).

In the following, we will briefly introduce main reasoning tasks in DLs without going into detail on reasoning algorithms as their internals are not of importance throughout this thesis. Main results on their worst-case computational complexity for *SHOIN* and *SRIQ* are given at the end of this section.

### Standard Reasoning Tasks

Several reasoning tasks (problems) exist for TBoxes and ABoxes. Reasoning over an ABox can further be done in isolation or w.r.t. a TBox. We start with TBox reasoning tasks, which fall into the category of terminological reasoning.

First, when conceptualizing a domain, it is often needed to find out whether concepts are contradictory and whether they actually make sense. Intuitively, a concept  $C$  makes sense w.r.t. a set of interrelated concepts – a TBox – if there exists an interpretation  $\mathcal{I}$  that satisfies each concept (cf. [Definition 3.6](#)) and where  $C$  has at least one individual as a member in  $\mathcal{I}$ . Such a concept is said to be *satisfiable* w.r.t. the TBox and *unsatisfiable* otherwise. Other important reasoning tasks over TBoxes are whether one concept subsumes another one, whether two concepts are equivalent, and whether two concepts are disjoint. These tasks can be transferred analogously to roles. Formally, they are defined as follows.

**Definition 3.12** (TBox Reasoning Problems). *Let  $C, D$  be concepts,  $\mathcal{T}$  a TBox, and  $\mathcal{A}$  an ABox.*

- **Concept Satisfiability:**  $C$  is satisfiable w.r.t.  $\mathcal{T}$  iff there exists a model  $\mathcal{I}$  of  $\mathcal{T}$  such that  $C^{\mathcal{I}}$  is nonempty.
- **Concept Subsumption and Equivalence:**  $C$  is subsumed by (resp. equivalent to)  $D$  w.r.t.  $\mathcal{T}$  iff  $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$  ( $C^{\mathcal{I}} = D^{\mathcal{I}}$ ) for every model  $\mathcal{I}$  of  $\mathcal{T}$ , written  $\mathcal{T} \models C \sqsubseteq D$  ( $\mathcal{T} \models C \equiv D$ ).<sup>12</sup>
- **Concept Disjointness:**  $C$  and  $D$  are disjoint w.r.t.  $\mathcal{T}$  iff  $C^{\mathcal{I}} \cap D^{\mathcal{I}} = \emptyset$  for every model  $\mathcal{I}$  of  $\mathcal{T}$ .

In DLs where the concept intersection constructor ( $\sqcap$ ) exists and which contain the unsatisfiable concept ( $\perp$ ), all problems above can be reduced to concept subsumption. This means that it is sufficient to implement subsumption checking in order to implement the other. Similarly, these problems can be reduced to unsatisfiability for DLs having concept intersection ( $\sqcap$ ) and negation ( $\neg$ ).

<sup>12</sup>The relation symbol  $\models$  is either understood as “satisfies” or as “entails” depending on whether the left-hand side is an interpretation or a set of axioms and/or assertions.

The task of computing the entire subsumption hierarchy of parents and children of each named concept in a TBox is the so-called *classification* process. It is an important feature for verification and graphical visualization of a conceptualization.

Standard reasoning tasks in the ABox (and w.r.t. a TBox) are *instance checking*, *consistency checking*, the *retrieval problem*, and its dual the *realization problem*. Similar to reduction of TBox reasoning tasks to concept subsumption (or satisfiability), the latter three can all be accomplished by reduction to instance checking.

**Definition 3.13** (ABox Reasoning Problems). *Let  $C$  be a concept,  $\alpha$  an ABox assertion,  $\mathcal{T}$  a TBox, and  $\mathcal{A}$  an ABox.*

- **Instance Checking:**  $\mathcal{A}$  entails  $\alpha$  w.r.t.  $\mathcal{T}$  (or  $\alpha$  is a consequence of  $\mathcal{A}$  w.r.t.  $\mathcal{T}$ ) if every interpretation that satisfies  $\mathcal{T}$  and  $\mathcal{A}$  also satisfies  $\alpha$ , written  $\mathcal{T}, \mathcal{A} \models \alpha$ .
- **Retrieval Problem:** Find all individuals  $a$  such that  $\mathcal{T}, \mathcal{A} \models C(a)$ ; analogous for roles.
- **Realization Problem:** Given an individual  $a$  and a set of concepts  $\mathbf{C}$ , find the most specific concepts  $C_i \in \mathbf{C}$  (i.e., there is no  $C' \in \mathbf{C}$  such that  $C' \neq C_i$  and  $C' \sqsubseteq C_i$ ) such that  $\mathcal{T}, \mathcal{A} \models C_i(a)$ ; analogous for roles.

If the TBox is empty (e.g., when it is acyclic and has been compiled away) then we can drop  $\mathcal{T}$  in **Definition 3.13**. Finally, it is worth mentioning that ABox instance checking for negated assertions can be polynomially reduced to ABox consistency without negated assertions, and vice versa [Mil08, Section 2.2]. Formally, if  $\varphi$  is either  $C(a)$  or  $R(a, b)$  then  $\mathcal{T}, \mathcal{A} \models \neg\varphi$  iff  $\mathcal{A} \cup \{\varphi\}$  is inconsistent w.r.t.  $\mathcal{T}$ .

### Queries as Advanced Instance Retrieval

The retrieval problem can be seen as a very limited querying facility to knowledge bases. Data-intensive applications, however, usually have demanding requirements to querying facilities. More powerful instance retrieval can be done using so-called *conjunctive ABox queries* [CGL98, HT00] of the form

$$q := \alpha_1 \wedge \cdots \wedge \alpha_n$$

where  $\alpha_i$  is an *atom* of the form  $C(x)$  or  $R(x, y)$ ,  $C$  is a concept, and  $R$  is either a simple but possibly inverse abstract role or a concrete role (concrete roles do not have inverses). Let  $V_V$  be a finite set of variable names disjoint from  $V_I$  and  $V_{LS}$ . Then  $x$  is either an individual  $x \in V_I$  or a variable  $x \in V_V$  and  $y$  is either a variable  $y \in V_V$ , an individual  $y \in V_I$  if  $R$  is an abstract role, or a lexical form  $y \in V_{LS}$  if  $R$  is a concrete role.<sup>13</sup> Note that in this form we allow for (i) direct use of individuals in atoms analogous to [KRH07] and (ii) the use of data values and concrete roles analogous to [HM05]. These are unproblematic extensions compared to the original form considered in [CGL98, HT00]. As usual, a

<sup>13</sup>The general concept of a *conjunctive query* refers to the class of FOL formulas that can be built from atomic formulas, the conjunction connector, and the existential quantifier; that is, a conjunctive query is of the form  $x_1, \dots, x_k, \exists x_{k+1}, \dots, \exists x_l (\alpha_1 \wedge \cdots \wedge \alpha_m)$  where  $x_1, \dots, x_k$  are free variables (distinguished),  $x_{k+1}, \dots, x_l$  are bound variables (undistinguished), and  $\alpha_1, \dots, \alpha_m$  are atomic formulas (i.e., n-ary predicates over constants and the variables  $x_1, \dots, x_l$ ).

variable is represented by a *symbol* and can be substituted by a *value*. Variables are partitioned into *distinguished variables* also called answer or solution set variables (i.e., where the substituted value is part of a solution) and existentially quantified *undistinguished variables* that are not part of a solution. A query without distinguished variables is called a Boolean query because it can only be used to test whether “something” is entailed by an ABox (true) or not (false). Let  $\text{Var}(q)$  be the set of variables occurring in a conjunctive ABox query  $q$ . Let  $\mathcal{A}$  be an ABox,  $\mathcal{I}$  a model of  $\mathcal{A}$ ,  $\text{Var}_F \subseteq \text{Var}(q)$  be the set of variables that occur at the filler position of a concrete role  $R$  (i.e., if there is an  $\alpha = R(x, y)$  where  $R$  is a concrete role and  $y \in \text{Var}(q)$  then  $y \in \text{Var}_F$ ), and  $\pi : \text{Var}(q) \cup V_I \cup \mathbb{V}_{LS} \rightarrow \Delta^{\mathcal{I}} \cup \Delta^{\mathcal{D}}$  a total function such that

$$\pi(x) = \begin{cases} a^{\mathcal{I}} \text{ with } a \in V_I & \text{if } x \in \text{Var}(q), x \notin \text{Var}_F, \text{ and } x \text{ distinguished} \\ l^{\mathcal{D}} \text{ with } l \in \mathbb{V}_{LS} & \text{if } x \in \text{Var}(q), x \in \text{Var}_F, \text{ and } x \text{ distinguished} \\ \varepsilon \in \Delta^{\mathcal{I}} & \text{if } x \in \text{Var}(q), x \notin \text{Var}_F, \text{ and } x \text{ undistinguished} \\ \varepsilon \in \Delta^{\mathcal{D}} & \text{if } x \in \text{Var}(q), x \in \text{Var}_F, \text{ and } x \text{ undistinguished} \\ x^{\mathcal{I}} & \text{if } x \text{ is an individual name } x \in V_I \\ x^{\mathcal{D}} & \text{if } x \text{ is a lexical form } x \in \mathbb{V}_{LS} . \end{cases}$$

$\pi$  is called a *match* (solution) for  $\mathcal{I}$  and  $q$  if  $\mathcal{I} \models_{\pi} \alpha$  for every  $\alpha \in q$ . For  $\alpha$  a concept membership assertion  $C(x)$  or a role membership assertion  $R(x, y)$  then

$$\mathcal{I} \models_{\pi} C(x) \text{ if } \pi(x) \in C^{\mathcal{I}}$$

and

$$\mathcal{I} \models_{\pi} R(x, y) \text{ if } (\pi(x), \pi(y)) \in R^{\mathcal{I}} .$$

If there is a match  $\pi$  for  $\mathcal{I}$  and  $q$  then it is also said that  $\mathcal{I}$  *satisfies*  $q$  w.r.t.  $\pi$ , written  $\mathcal{I} \models_{\pi} q$ . The *query entailment problem* (QEP) is deciding whether all models  $\mathcal{I}$  of a knowledge base  $\mathcal{K}$  also satisfy  $q$  for some match  $\pi$ , written  $\mathcal{K} \models q$ . In fact, query entailment is the decision procedure used for Boolean queries. The solutions of a query with distinguished variables are tuples of individual names or lexical forms where each tuple is obtained by substituting the distinguished variables according to each match entailed by the  $\mathcal{K}$ . Finding all solutions corresponds to the *query answering problem* (QAP) [GLHS08]. QEP and QAP can be mutually reduced [CGL98, HT00].

Finally, the *query containment problem* (or *query subsumption*) is the reasoning task of deciding whether a conjunctive query generally has at least the matches that another query has. Formally, given a DL  $\mathcal{L}$ , a query  $q$  is subsumed by a query  $q'$  w.r.t. an  $\mathcal{L}$ -KB  $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ , denoted with  $\mathcal{K} \models q \sqsubseteq q'$ , iff for every  $\mathcal{L}$ -ABox  $\mathcal{A}'$  and the KB  $\mathcal{K}' = (\mathcal{T}, \mathcal{A}')$  it holds that the solution set of  $q$  is a subset of the solution set of  $q'$ . Observe that this assumes that  $q, q'$  share the same set of distinguished variables.

## Computational Complexity of Reasoning Tasks

The worst-case computational complexity of subsumption reasoning for *SHOIN* is known to be intractable since it is NExpTime-complete [Tob01]. In the general case *SRQIQ* is even harder, namely N2ExpTime-complete but NExpTime under the syntactic restriction of bounded role hierarchies [Kaz08]. The former is due to the complex

role inclusion axioms  $R_1 \circ \dots \circ R_n \sqsubseteq R$ . For a summary of complexity result for various sublanguages of  $\mathcal{SHOIN}$ ,  $\mathcal{SROIQ}$  and other DLs see [Zol].

Complexity of conjunctive query answering is analyzed either as a function of the size of the query only, the size of the ABox only, or both together. They are respectively called *query complexity*, *data complexity*, and *combined complexity*. At the time of writing, decidability of query answering in  $\mathcal{SHOIN}$  and  $\mathcal{SROIQ}$  is still open, though signs that this is the case take shape [GR10]. Recently, it has been shown that combined complexity in the Horn fragment of  $\mathcal{SHOIQ}$  and  $\mathcal{SROIQ}$  is ExpTime-complete and 2ExpTime-complete, respectively [ORS11], which means that it is not harder than subsumption in the full versions of these DLs. Complexity of the fairly expressive sublanguage  $\mathcal{SHIQ}$  is known to be 2ExpTime-complete in the presence of inverse roles and ExpTime-complete otherwise [Lut08].

As usual, asymptotic worst-case complexity results say little about average complexity in practice. Several studies have given empirical evidence that optimized reasoning procedures yield reasonable response times in practical settings [Hor98, HM01a, HM08]. It has also been noted that exponentially hard cases can be exponentially rare in practice [Har06]. As a response to intractable complexity results, however, less expressive DLs have been devised of which reasoning is known to be tractable and query answering implementable on top of conventional relational database technology (see [Section 3.3](#)). They are the result of profoundly understanding which particular interactions of modeling constructs lead to intractability. In other words, these DLs reflect the desire to get to the highest expressivity for which worst-case tractability is retained.

### 3.1.5 Operations on Knowledge Bases

From a purely syntactical point of view, a knowledge base is essentially a set of axioms and assertions. Therefore, one can apply standard set operations such as creating the union, intersection, or difference of two knowledge bases  $\mathcal{K}_1, \mathcal{K}_2$ . This allows then to determine – at a syntactical level – whether they are disjoint, have parts in common, or are actually the same. On the semantical level, however, the result and the practicability of these operations depends on the expressivity and the interpretations (models) of the single knowledge bases  $\mathcal{K}_i$ . Obviously, two knowledge bases  $\mathcal{K}_1$  and  $\mathcal{K}_2$  that are satisfiable when viewed in isolation need not be satisfiable when building their union  $\mathcal{K}_1 \cup \mathcal{K}_2$  because they may model contradicting knowledge. On the other hand, difference  $\mathcal{K}_1 \setminus \mathcal{K}_2$  and intersection  $\mathcal{K}_1 \cap \mathcal{K}_2$  are unproblematic regarding satisfiability for monotonic DLs because the result cannot be “larger” than the arguments (i.e., the result is a subset of  $\mathcal{K}_1, \mathcal{K}_2$  anyway). [Example 3.2](#) and the next paragraph illustrates this.

#### Example 3.2

Consider the following KBs where the semicolon “;” delimits the TBox from the ABox and the comma “,” delimits axioms and assertions inside the TBox/ABox.

$$\begin{aligned} \mathcal{K}_1 &= \{ \text{Person, Female, Woman} \equiv \text{Person} \sqcap \text{Female}; \text{Woman}(\text{CURIE}) \}, \\ \mathcal{K}_2 &= \{ \text{Person, Female, Woman} \equiv \text{Person} \sqcap \text{Female}, \text{Man} \equiv \text{Person} \sqcap \neg \text{Woman}; \\ &\quad \text{Man}(\text{EINSTEIN}) \}. \end{aligned}$$

The union, intersection, and difference are then

$$\begin{aligned}\mathcal{K}_1 \cup \mathcal{K}_2 &= \{Person, Female, Woman \equiv Person \sqcap Female, \\ &\quad Man \equiv Person \sqcap \neg Woman; Woman(CURIE), Man(EINSTEIN)\}, \\ \mathcal{K}_1 \cap \mathcal{K}_2 &= \{Person, Female, Woman \equiv Person \sqcap Female\}, \\ \mathcal{K}_1 \setminus \mathcal{K}_2 &= \{Woman(CURIE)\}.\end{aligned}$$

**Example 3.2** is an innocuous one. None of the operations yields an inconsistent knowledge base, i.e., all concepts, axioms, and ABox assertions are still satisfiable in every resulting knowledge base; observe that they are also satisfiable in isolation. This is due to the fact that  $\mathcal{K}_1$  and  $\mathcal{K}_2$  do not model contradicting knowledge. Adding, for instance,  $Woman(CALLIOPE)$  to  $\mathcal{K}_1$  and  $Man(CALLIOPE)$  to  $\mathcal{K}_2$  would, however, result in an inconsistency in the union  $\mathcal{K}_1 \cup \mathcal{K}_2$  because the concepts  $Man$  and  $Woman$  were indirectly described as disjoint. That is, there can be no model in which the individual referred to by the name  $CALLIOPE$ <sup>14</sup> is both a man and a woman. Observe that  $\mathcal{K}_1$  and  $\mathcal{K}_2$  remain consistent when viewed in isolation. It is beyond the application of the union operator to knowledge bases to ensure that the resulting knowledge base is still satisfiable. Consequently, a consistency check should be done before committing an operation if an application requires a consistent knowledge base at any time.

Apart from dealing with knowledge bases by means of set operators, almost all applications not only require the ability to read their content but to modify or edit them in order to accommodate new, revise existing, or retract obsolete knowledge [FMK<sup>+</sup>08]. In the AI research field this is well known as the *belief revision* and *belief update* problem [Pep08]. In short, belief revision refers to the process of how to modify a knowledge base “in the light of new information that was previously inaccessible” and where the initial knowledge base needs to be modified because it is incomplete or parts have become obsolete or incorrect in light of the new information. In contrast, the belief update problem refers to the process of how to modify a knowledge base that needs to be brought up-to-date to changes in a dynamic domain (in the world) because it is out-of-date after changes have occurred in the domain. The update problem gets into focus in **Chapter 4** when we discuss how to represent the effects of service and service operation invocations. In the context of **Chapter 6**, however, updating a KB is viewed from a data management point of view. More specifically, we will define the lower level storage layer of a knowledge base management system (KBMS) to provide update operations with *direct* semantics, meaning that operations directly add or delete axioms and assertions. To make this data-centric point of view more explicit, we classify belief revision and belief update as *indirect update semantics* and the latter as *direct update semantics*, see **Figure 3.1**. In [HPSK06] the notion of *edit semantics* has been defined for ABoxes, which is equivalent to our notion of direct updates. We follow this definition but generalize it for the knowledge base as follows.

**Definition 3.14** (Direct KB Update). *Let  $\mathcal{L}$  be a Description Logic and let  $\mathcal{K} = \mathcal{T} \cup \mathcal{A}$  be an  $\mathcal{L}$ -knowledge base. Then, updating  $\mathcal{K}$  by adding (deleting) a new (existing)  $\mathcal{L}$ -syntactic*

<sup>14</sup>Calliope is the intersexual protagonist in the novel *Middlesex* by Jeffrey Eugenides.

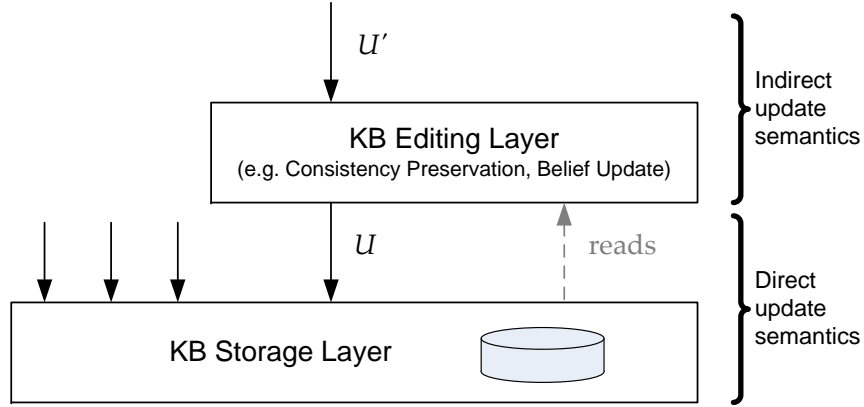


Figure 3.1: Distinction between high level knowledge base updates and direct updates at the level of the storage layer.

instance  $\psi$ , written  $\mathcal{K} + \psi$  ( $\mathcal{K} - \psi$ ), results in an updated KB  $\mathcal{K}'$  such that  $\mathcal{K}' = \mathcal{K} \cup \{\psi\}$  ( $\mathcal{K}' = \mathcal{K} \setminus \{\psi\}$ ).

A direct KB update (or direct update for short) is a finite and non-empty set of these additions and deletes. Given  $\mathcal{K}$  and a direct update  $U$ ,  $\mathcal{K}'$  is the result of updating  $\mathcal{K}$  with  $U$ , written  $\mathcal{K} \Longrightarrow_U \mathcal{K}'$ , obtained by applying all adds and deletes in  $U$  to  $\mathcal{K}$ .

A direct update  $U$  is usually thought to be applied in an atomic way: either none or all additions and deletes are applied. Obviously, allowing direct updates on  $\mathcal{K}$  may affect its entailments (i.e., the implicit knowledge). Moreover, applying  $U$  may result in an inconsistent  $\mathcal{K}'$  if  $U$  adds axioms or assertions that either contradict with existing knowledge or among each other (e.g., if an add of  $\psi$  and  $\neg\psi$  is in  $U$ ). Practical KBMSs might, therefore, be equipped with a knowledge base editing<sup>15</sup> layer on top of the storage layer as depicted in Figure 3.1. Its purpose is to transform a belief revision or belief update represented by  $U'$  at the higher level (i.e., an indirect update, which may contradict with  $\mathcal{K}$ ), into a direct update  $U$  such that (i) consistency at DL level is preserved and (ii) the desired update semantics as defined by the actual Belief Revision or Belief Update approach is achieved. In order to achieve this, the editing layer may need to interrogate the knowledge base (i.e., read-access it), which is indicated by the dashed arrow in Figure 3.1. Finally, we make the following observation.

**Observation 3.1.** *Given a monotonic DL  $\mathcal{L}$  such as  $\mathcal{SHOIN}(\mathbf{D})$  or  $\mathcal{SROIQ}(\mathbf{D})$ , an  $\mathcal{L}$ -KB  $\mathcal{K}$ , and a delete-only direct update  $U$  with  $\mathcal{K} \Longrightarrow_U \mathcal{K}'$ , if  $\mathcal{K}$  is consistent, so is  $\mathcal{K}'$ .*

The reason is that every delete can only reduce explicit and implicit knowledge. In other words, only addition of new axioms or assertions can lead to inconsistency. This is not the case in general for non-monotonic logics; thus, deletes would need to be considered as well by the consistency preservation mechanism.

<sup>15</sup>The term refers to the field of modifying a knowledge base either to resolve inconsistencies or in response to a change request [FMK<sup>+</sup>08].

## 3.2 Resource Description Framework

As the name indicates, RDF is a framework for representing information about (Web) resources. Originally designed as a meta data model, RDF has come to be used as a general framework for conceptual information representation. This is also due to the design goal of facilitating Web-scale processing, exchange, and semantic interpretation of information by machines rather than being only displayed to humans. Its basic idea is to allow anybody to make statements about any *resource* (in the Web), represented in the form of so-called subject-predicate-object *triples*. The *subject* denotes the resource about which a statement is made (i.e., the answer to the question about whom or what a statement is made). The *predicate* (a.k.a. property) denotes traits that are true of the subject and establishes a relationship between the subject and the *object*. The latter is somebody or something involved in the subject's predicate.

RDF's data model is a labeled, directed multi-graph whose nodes are partitioned into *named*, *blank*, and *literal* nodes. As a matter of the abstract graph based nature, it is independent of any concrete (serialization) format for representing or storing a graph. A named node represents a resource that has an *Internationalized Resource Identifier* (IRI) [DS05] as its name.<sup>16</sup> A blank node also represents a resource, but one that has no separate form of identification – it cannot be identified outside a graph. A blank node is to be seen as a local and existentially quantified variable representing an anonymous resource. For instance, the triples  $\langle \text{ALICE}, \text{ownsBook}, \_1 \rangle, \langle \_1, \text{genre}, \text{Fiction} \rangle$  state that Alice owns an otherwise unknown fiction book. Literals are data values such as strings, numbers, dates, and so on. Literals may be *plain* or *typed*. The former are strings and may optionally have a language tag. They are intended to be used for plain text in a natural language. Plain literals are formally interpreted to denote themselves (lexical space = value space). In contrast, a typed literal is a string (lexical form) associated with a datatype, the latter identified by an IRI. Analogous to data values of concrete domains (see Section 3.1.3), the value of a typed literal is found by applying the lexical-to-value mapping associated with the datatype to the lexical form. Finally, the edges of the graph are named *predicates*. Predicates are generally identified by an IRI analogous to named nodes. RDF graphs are formally defined as follows.

**Definition 3.15** (RDF Graph). *Let  $V_R$  be a set of IRI resources and  $V_B$  a set of blank nodes disjoint from  $V_R$ . Let  $V_L$  be a set of literals disjoint from  $V_R$  and  $V_B$ . An RDF graph  $\mathcal{G}$  is a set of triples  $\langle s, p, o \rangle$  with*

$$\mathcal{G} \subseteq (V_R \cup V_B) \times V_R \times (V_R \cup V_B \cup V_L)$$

*where  $s, p, o$  is called the subject, predicate, and object, respectively. The set of nodes of  $\mathcal{G}$  is the set of subjects and objects of triples in  $\mathcal{G}$ . The set of edges of  $\mathcal{G}$  is the set of predicates of triples in  $\mathcal{G}$ . An RDF graph is ground if it has no blank nodes ( $V_B = \emptyset$ ).*

It follows from Definition 3.15 that literals cannot be subjects and that an IRI resource may occur both as a subject and predicate (because  $V_R$  is not further partitioned). The

<sup>16</sup>The RDF specification actually defines named nodes as URI resources. The generalization to IRIs is unproblematic since they are generalized URIs over the Unicode character set (as opposed to URIs whose alphabet is a subset of the ASCII character set).



use of literals as subjects is a long debated restriction and it is still open whether it will be integrated in future RDF versions (for a summary of the discussion see [W3C]).

An *RDF dataset* is a collection of RDF graphs. More precisely, it is a set of one *default graph* and zero or more *named graphs*, written

$$DS = \{\mathcal{G}, (u_1, \mathcal{G}_1), \dots, (u_n, \mathcal{G}_n)\} \quad n \geq 0$$

where  $\mathcal{G}, \mathcal{G}_1, \dots, \mathcal{G}_n$  are RDF graphs with  $\mathcal{G}$  being the default graph,  $u_1, \dots, u_n$  are distinct IRIs, and the pairs  $(u_i, \mathcal{G}_i)$  denote the named graphs (i.e., the default graph does not have a name and each named graph is identified by an IRI). Usually, the default graph is the *RDF merge* of named graphs in the dataset. The RDF merge of a set of RDF graphs  $\mathcal{G}_1 \dots \mathcal{G}_n$  is an RDF graph

$$\hat{\mathcal{G}} = \bigcup \mathcal{G}'_i$$

where  $\mathcal{G}'_i$  has been obtained from  $\mathcal{G}_i$  by standardizing apart blank nodes such that  $\hat{\mathcal{G}}$  does not share blank nodes with  $\mathcal{G}_1 \dots \mathcal{G}_n$ . However, the definition of RDF datasets does not impose a relationship between the default and named graphs. Another possible arrangement is to have meta information about the named graphs in the default graph and have triples of named graphs not visible (merged) in the default graph. For instance, to have triples in the default graph representing provenance information about the named graphs.

Analogous to DLs, RDF has a formal model-theoretic semantics in order to support reasoning about the meaning of triples and graphs. In particular, it defines a notion of entailment between graphs. In short, a graph  $\mathcal{G}'$  (constructed, for instance, by some procedure from another graph  $\mathcal{G}$ ) is entailed by  $\mathcal{G}$  if every interpretation which satisfies all triples in  $\mathcal{G}$  also satisfies all triples in  $\mathcal{G}'$ . For details we refer to [Hay04].

### 3.3 Web Ontology Language

OWL's features are influenced by a couple of (difficult to combine) requirements and design goals. On one hand it was clearly intended to be an ontology language for representing knowledge in terms of classes of objects and how they are interrelated. Taking existing knowledge representation languages such as SHOE [HHL99] and OIL [FHH<sup>+</sup>01] to the next level was one goal. On the other hand it was intended to extend the Semantic Web technology stack on top of RDFS, RDF, and XML such that Web scale exchange and combination of OWL ontologies is supported. In particular, OWL has a mechanism allowing to include (import) ontologies specified somewhere else into another ontology. This constitutes the hyperlinking effect for knowledge representation just like information hyperlinking in the classical Web, which made it so powerful.

An *ontology* is the basic information “container” in OWL. In contrast to information science where an ontology is understood as a shared conceptualization of a domain [Gru93], the notion of an OWL ontology goes beyond this. Specifically, it can additionally contain:

- assertions representing the (current) state of affairs in some domain (i.e., an ABox),

- *entity declarations* that define the vocabulary of some domain, and
- *annotations* that associate additional (meta) information with axioms, assertions, or entities.

Annotations are, however, outside the underlying DL framework: they do not contribute to the implicitly entailed knowledge and are therefore also called *non-logical axioms*.<sup>17</sup> An annotation is made by means of a so-called *annotation property*, which is a binary relation analogous to an object or data property. We will also call an annotation a syntactic instance. An entity corresponds to a name in either of the vocabulary sets  $V_L, V_C, V_D, V_{OP}, V_{DP}, V_{AP}$ , where  $V_{AP}$  is a set of annotation property names. Syntactically, an entity is always identified by an IRI.

An OWL ontology can actually be seen as an extended knowledge base. We define the notion of an *OWL knowledge base* that, for technical reasons, keeps the “core” knowledge base, entity declarations, and annotations apart. The terms OWL ontology and OWL knowledge base are thus understood as interchangeable.

**Definition 3.16** (OWL Knowledge Base). *An OWL knowledge base is a 3-tuple  $\mathcal{W} = (\mathcal{K}, \mathbf{D}, \mathbf{A})$  where  $\mathcal{K}$  is a knowledge base (see Equation (3.1)),  $\mathbf{D}$  is a set of entity declarations, and  $\mathbf{A}$  is a set of annotations.*

We assume that  $\mathbf{D}$  and  $\mathbf{A}$  can be updated under the same direct update semantics than  $\mathcal{K}$ , as stated by Definition 3.14. Finally, it should be clear that entity declarations are implicit and annotations are disregarded when dealing with the “core” KB  $\mathcal{K}$ .

### 3.3.1 Import Mechanism

The import mechanism of OWL works as follows. The *import closure*  $IC(\mathcal{O})$  of an ontology  $\mathcal{O}$  is a set containing  $\mathcal{O}$  and all ontologies that  $\mathcal{O}$  imports.<sup>18</sup> The *syntactic instance closure*  $SIC(\mathcal{O})$  is the union of syntactic instances of each ontology in the import closure

$$SIC(\mathcal{O}) = \bigcup_{o \in IC(\mathcal{O})} o,$$

where anonymous individuals have been standardized apart; that is, they are treated as being different. Anonymous individuals are analogous to blank nodes in RDF and cannot be identified outside an ontology.

### 3.3.2 Representation Formats

The *functional-style* syntax [MPSP09] (a.k.a. abstract syntax) defines the syntactic constructs of OWL and their structure. The primary storage and exchange format of OWL ontologies is RDF/XML based on the OWL-to-RDF mapping (see Section 3.3.4). Other formats are the Manchester Syntax [HPS09], Turtle [BBL11], and OWL/XML [MPPS09], sorted in descending order of readability for humans.

<sup>17</sup>There are, however, some simple inferences for annotations under RDF-based formal semantics.

<sup>18</sup>There are two exceptions when an imported ontology must not belong to the import closure: in case ontologies are different versions from the same ontology series and when ontologies are declared to be incompatible.

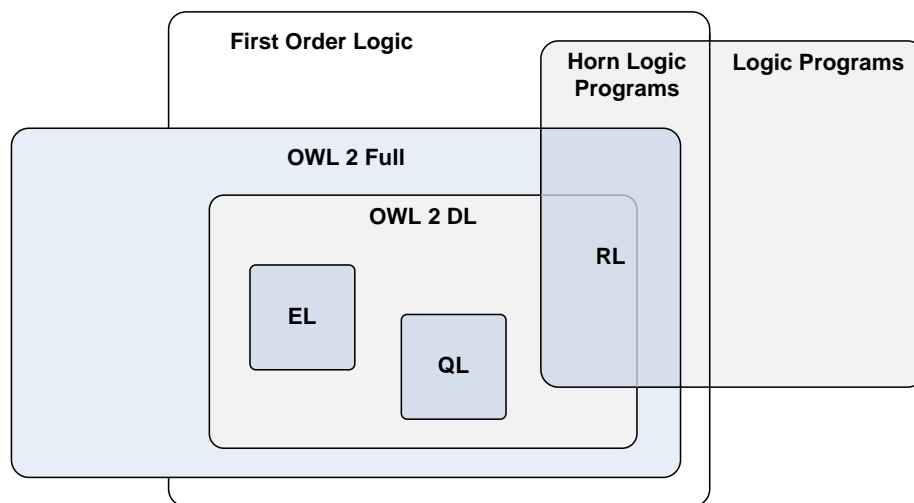


Figure 3.2: Graphical representation of overlaps and containment regarding language expressiveness for FOL, OWL, and Logic Programs

### 3.3.3 Profiles

*OWL 2 DL* can be seen as the default modeling level that provides the highest expressivity while retaining decidability of standard reasoning tasks. The acronym reflects its origin in the description logic  $\mathcal{SROIQ}(\mathbf{D})$  (and  $\mathcal{SHOIN}(\mathbf{D})$  for *OWL 1 DL*); in fact, they can be considered syntactic variants: The formal model-theoretic semantics of *OWL 2 DL* [MPG09] precisely matches the formal semantics introduced in [Section 3.1](#).

*OWL* has also been given formal model-theoretic semantics based on and compatible with *RDF*'s formal semantics [Sch09]. In fact, this yields an even more expressive modeling level (language) because *RDF* features can be used along with *OWL* features. However, standard reasoning tasks are no longer decidable in general under these *RDF*-based semantics. The modeling level is correspondingly called *OWL Full*.

In addition, *OWL 2* comes with three profiles [MGH<sup>+</sup>09], namely *OWL 2 EL*, *OWL 2 QL*, and *OWL 2 RL*. They are mainly motivated by intractability of reasoning in *OWL DL*. Each of them is essentially a subset of *OWL 2* sufficient for different types of applications. They generally scale better due to their tractable complexity of reasoning. [Figure 3.2](#) illustrates the relationship between the different language levels of *OWL 2* and also puts them into relation with *FOL* and *Logic Programs*.

#### OWL 2 EL

The *EL* acronym reflects the profile's origin the *EL DL* family [BBL08]. These *DLs* are generally known to have polynomial worst-case complexity of subsumption reasoning w.r.t. a *TBox*. Consequently, they are especially useful in domains where the *TBox* can become very large. For instance, *SNOMED CT* [Int] has about 380000 concepts built using only conjunction ( $\sqcap$ ), existential restriction ( $\exists R.C$ ), and the top concept ( $\top$ ). Amongst other restrictions, *OWL 2 EL* disallows a number of concept constructors, namely negation ( $\neg$ ), disjunction ( $\sqcup$ ), universal restriction ( $\forall R.C$ ), and also disallows inverse roles, which can (in combination) lead to exponential complexity.

## OWL 2 QL

The QL acronym reflects the profile’s aim to be realized on top of existing relational DBMS. In fact, it targets application domains where the ABox can become very large with billions of individuals and efficient ABox query answering is the most important reasoning task. The latter can be implemented by rewriting queries into, for instance, standard SQL queries. Complexity of query answering w.r.t. a TBox in OWL 2 QL is in PTime, for some cases even in LogSpace.

## OWL 2 RL

The RL acronym reflects the profile’s purpose of resembling an OWL-based rule language [GHVD03]. The basic idea is that a concept inclusion axiom  $C \sqsubseteq D$  can be understood as a rule-like implication

$$\underbrace{D}_{\text{head}} \leftarrow \underbrace{C}_{\text{body}}$$

where the sub concept is the rule body (antecedent) and the super concept is the rule head (consequent).<sup>19</sup> OWL 2 RL is best used for application domains requiring scalable reasoning without sacrificing too much expressive power. In order to ensure decidability and to avoid the need for nondeterministic reasoning, OWL 2 RL imposes restrictions on the rules that can be expressed. Amongst other things, it disallows rules (concept inclusion axioms) where the existence of an individual enforces the existence of another individual. Also rules are restricted to be asymmetric, i.e., there are concepts that can be used as the body (subconcept) but cannot be used as the head (superclass).

### 3.3.4 Mapping to RDF Graphs

The OWL mapping to RDF graphs [PSM09] defines a bidirectional transformation based on unique rules from the OWL abstract syntax to RDF graphs while preserving formal OWL semantics for a roundtrip (OWL  $\rightarrow$  RDF  $\rightarrow$  OWL). Formally, let  $\psi$  be a possibly nested syntactic instance; note that several syntactic constructs can be nested (e.g., a complex concept expression). Then,  $T(\psi)$  denotes the RDF graph (set of triples) obtained by recursively applying the OWL-to-RDF mapping rules to  $\psi$ . We overload  $T$  for ontologies. The RDF graph for  $\mathcal{O}$  is obtained by applying  $T$  to all syntactic instances  $\psi \in \mathcal{O}$ ; that is,

$$T(\mathcal{O}) = \bigcup_{\psi \in \mathcal{O}} T(\psi).$$

Observe that this does not include imported ontologies (if any).  $T$  can be analogously overloaded for OWL knowledge bases; that is,  $T(\mathcal{W}) = \bigcup_{\psi \in \mathcal{W}} T(\psi)$ .

<sup>19</sup>Whereas  $C \sqsubseteq D$  and  $C \rightarrow D$  are logically equivalent, they are not to be confused with a *trigger* (a.k.a. *production*) rule  $C \Rightarrow D$  [BCM<sup>+</sup>07, Section 2.2.5]. The semantics of the trigger rules is given in a declarative way by equivalence to an epistemic inclusion  $\mathbf{K}C \sqsubseteq D$  that basically states that an individual  $a$  is a member of  $D$  if it is *known* either explicitly or implicitly by inference that  $a$  is a member of  $C$ . In other words,  $C \Rightarrow D$  is not equivalent to its contrapositive  $\neg D \Rightarrow \neg C$ . Trigger rules are therefore constructive; their semantics can also be defined in an operational way as a forward-chaining process.

Table 3.4 illustrates the mapping using some examples. Observe that the number of triples to which a OWL syntactic instance is mapped is not only determined by the type of syntactic construct. Some of them (e.g., *ObjectOneOf*, *DisjointUnion*) map to a number of triples that is proportional to the arity of actual syntactic instance.

Table 3.4: Examples for mapping of OWL syntactic constructs to RDF triples

Syntactic Construct $\Psi$	$T(\Psi)$	$ T(\Psi) $
<i>Declaration</i> ( <i>Class</i> ( $x$ ))	$x \text{ rdf:type owl:Class.}$	1
<i>Declaration</i> ( <i>ObjectProperty</i> ( $x$ ))	$x \text{ rdf:type owl:ObjectProperty.}$	1
<i>ObjectHasValue</i> ( $p \ y$ )	$x \text{ rdf:type owl:Restriction.}$ $x \text{ owl:onProperty } p. \ x \text{ owl:hasValue } y.$	3
<i>ObjectOneOf</i> ( $y_1 \dots y_n$ )	$x \text{ rdf:type owl:Class.}$ $x \text{ owl:oneOf } ( T(\text{SEQ}(y_1 \dots y_n)) ) .$	$\geq 6$ for $n \geq 2$
<i>ObjectMaxCardinality</i> ( $p \ m \ y$ )	$x \text{ rdf:type owl:Restriction.}$ $x \text{ owl:onProperty } p. \ x \text{ owl:onClass } y.$ $x \text{ owl:maxQualifiedCardinality}$ $\text{"m"^^xsd:nonNegativeInteger.}$	4
<i>DisjointObjectProperties</i> ( $p_1 \ p_2$ )	$p_1 \text{ owl:propertyDisjointWith } p_2.$	1
<i>DifferentIndividuals</i> ( $x_1 \ x_2$ )	$x_1 \text{ owl:differentFrom } x_2.$	1
<i>ClassAssertion</i> ( $x \ y$ )	$x \text{ rdf:type } y.$	1
<i>ObjectPropertyAssertion</i> ( $x_1 \ p \ x_2$ )	$x_1 \ p \ x_2.$	1
Syntactic Instance $\psi$	$T(\psi)$	$ T(\psi) $
<i>Declaration</i> ( <i>Class</i> ( <i>:Father</i> ))	<i>:Father</i> $\text{rdf:type owl:Class.}$	1
<i>Declaration</i> ( <i>Class</i> ( <i>:Mother</i> ))	<i>:Mother</i> $\text{rdf:type owl:Class.}$	1
<i>Declaration</i> ( <i>Class</i> ( <i>:Parent</i> ))	<i>:Parent</i> $\text{rdf:type owl:Class.}$	1
<i>Declaration</i> ( <i>ObjectProperty</i> ( <i>:hasSon</i> ))	<i>:hasSon</i> $\text{rdf:type owl:ObjectProperty.}$	1
<i>DisjointUnion</i> ( <i>:Parent</i> <i>:Mother</i> <i>:Father</i> )	<i>:Parent</i> $\text{owl:disjointUnionOf } \_ :1.$ $\_ :1 \text{ rdf:first } \textit{:Mother}. \ \_ :1 \text{ rdf:rest } \_ :2.$ $\_ :2 \text{ rdf:first } \textit{:Father}. \ \_ :2 \text{ rdf:rest rdf:nil.}$	5
<i>ClassAssertion</i> ( <i>egy:Isis</i> <i>:Mother</i> )	<i>egy:Isis</i> $\text{rdf:type } \textit{:Mother}.$	1
<i>ObjectPropertyAssertion</i> ( <i>egy:Isis</i> <i>:hasSon</i> <i>egy:Horus</i> )	<i>egy:Isis</i> <i>:hasSon</i> <i>egy:Horus.</i>	1

$\text{SEQ}(x_1 \dots x_n)$  is the linked RDF list, mapped to one triple for  $n = 0$  and  $n * 2$  triples for  $n \geq 1$ .  $\_ :i$  denotes a blank node;  $x, y$  denote an IRI or a blank node;  $p$  denotes an IRI;  $m, n$  non-negative integers.



# 4

## System Model

SEMANTIC SERVICE EXECUTION requires a system that manages this task. The purpose of this chapter is to describe an abstract system model that seeks to jointly represent the integral parts providing the basis for the types of flexibility introduced and motivated in [Chapter 1](#) and [2](#) in a general and formal way. First, the functional and non-functional *properties* of services reflecting the data, change, and non-functional semantics in the domain of their use. Second, the *behavior* that occurs when they are performed, comprising the interactions, communications, and synchronizations among acting parts.

This chapter is correspondingly divided into two main sections, see [Figure 4.1](#). First, [Section 4.2](#) in which we present a service model that captures these properties and their semantics. Second, [Section 4.3](#) in which a process model is presented that captures the behavior along with execution semantics. Together with a preceding section where we introduce the basic elements, they make up the system model that is taken as the basis for subsequent chapters.

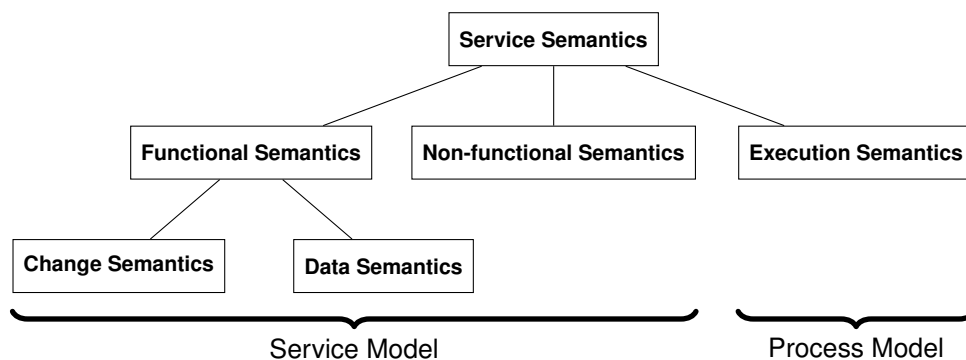


Figure 4.1: Classification of service semantics combined in the system model.

The service model builds largely upon previous research and initiatives on (formal) Semantic Service frameworks, namely OWL-S [MBH<sup>+</sup>04], SAWSDL [FL07], and WSMO [LPR05] (in alphabetical order) and adopts the DL based approach to represent the semantics of services and their operations. General introductions to principles of

Semantic Services have been given, for instance, in [CS06, SGA07, SHS08, FFST11]. The process model builds upon previous research in the area of modeling workflows and concurrent systems, in particular formal process models. Development of such models is strongly connected with their intended use and objectives. A categorization into five different uses has been presented in [CKO92] of which we quote the category that is relevant for this work:

*Automate execution support* requires a computational basis for controlling behavior within an automated environment.

Well-studied directions in the area of process theories are activity-oriented approaches and process algebras. A prominent representative for the latter is the  $\pi$ -calculus [Mil99], whereas PETRI nets [Mur89] belong to the former category. Many process modeling languages build on these formalisms [LS07]. More recently, the set of process theories has been further extended by applying the declarative and constraint-based approach to process modeling [PA06, PSSA07]. In this thesis, we will adopt the PETRI net formalism to provide a model of the behavior of services. It provides a well-understood theory general enough to model distributed and concurrent systems, which are prerequisites to capture the nature of the application forms presented in the introduction (see [Section 1.1](#)). Equally important to this work, they can be used to provide firm executable semantics.

Finally, our goal is to devise a generic and principled approach. Analogous to the Reference Model for Service-oriented Architectures [MLM<sup>+</sup>06], we aim at being independent of specific standards, technologies, implementations, or other concrete details. On the other hand, there are additional (technical) aspects related to service execution, notably: service engineering, programming, and provision; quality-of-service negotiation and enforcement between service providers and users; security and privacy concerns such as policies that should control the use of services in open environments. But these are all non-goals to this thesis and therefore not addressed by the model.

## 4.1 Basic Elements, Relations, and Assumptions

The purpose of this section is to provide a big picture on how the system is structured. More specifically, we will introduce and define the basic elements in the system informally, identify how they relate to each other, provide important background information, and point out basic assumptions that are made in this thesis. Whenever possible, elements are introduced in bottom-up succession. [Figure 4.2](#) depicts the static structure of the system model. For the most part, this represents a common denominator over basic notions and their relations in prominent Semantic Service and Web Service description frameworks. Though frameworks do not fully agree on a common terminology and differ in their formal representation, these notions and the relations can be identified more or less directly in DSD [KKRM05], OWL-S, SAWSDL, WSMO, WSDL [CMRW07],



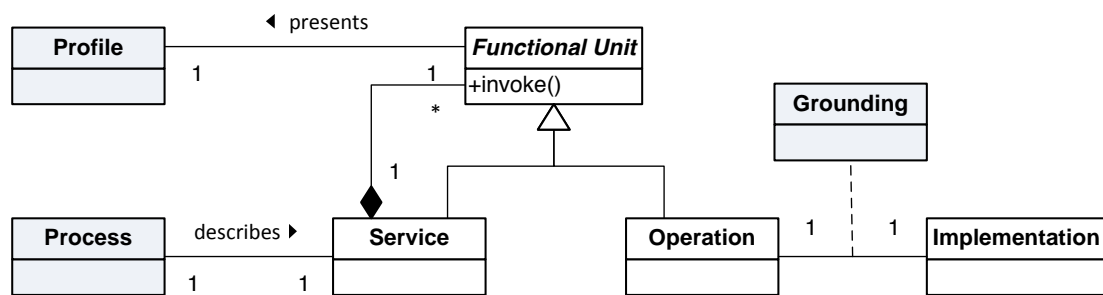


Figure 4.2: Basic elements of the system and their static structure in UML notation. Colored elements are part of a service description.

and the more recent hRESTS [KGV08] and SA-REST [GRS10] for microformat-style embedding.<sup>1</sup> The terminology that we will use follows mostly OWL-S and WSDL.

Normative assumptions that are made about the basic elements in the following subsections are set out-of-line and are highlighted using enumeration labels of the form **(An)**. Assumptions that have an informative character or assumptions that are consequences of the structure and the relations depicted in Figure 4.2 are not highlighted and will be described in-lined.

Also, we do not yet detail how semantics nor behavior is formalized in the following subsections. Finally, we abstract from a number of conceptual, infrastructural, and technical aspects that are all relevant to concrete implementations of the system model. This includes (i) how elements are created and how they are actually represented, (ii) technical means that are used, for instance, by clients to interact with services, (iii) technical means on how the elements are provided, (iv) technical details concerning the different actors in the system such as service clients, service providers, execution engines, or other infrastructure related entities, and (v) the environment in which the system is deployed (e.g., centralized, distributed, peer-to-peer).

### 4.1.1 Functional Unit

As the name suggests, a functional unit is an abstract notion meant to encapsulate some well-defined functionality. There are two kinds of functional units in this model: services and operations. No matter which, they are designed to achieve either application or system related intents of some sort. Generalizing the definition given in [Pre04] to a functional unit, it has “the capacity to perform something of value, in the context of some domain of application”. This includes essentially information processing over data and/or physical transformations in the real world. In this thesis, we consider functional units of the following kind.

- (A1)** A functional unit is *deterministic* regarding information processing and physical transformations.

<sup>1</sup>There is also WSMO-Lite [VKVF08] and MicroWSMO [MKP09]. The former slightly extends SAWSDL, whereas the latter relies on hRESTS. Both can be seen as light-weight successors of WSMO.

(A2) Data processed by a functional unit is *discrete*.<sup>2</sup>

**Assumption 1** means that services and operations are devoid of internal randomness regarding information processing and physical transformations. More specifically, they are devoid of what is often referred to as *internal* as opposed to *external* nondeterminism [Hoa85].<sup>3</sup> Therefore, they can be seen as partial functions of the input data to be processed and the current state of affairs in the application domain; hence, they may not have different results with different probabilities for each result when invoked with the same input data and starting from the same state. Note that this understanding of determinism does very well include services and operations that are implemented based on nondeterministic algorithms.

**Assumption 2** means that a functional unit may consume and produce single data items (of a well-defined format) rather than *continuous* streams of individual data items – known as data streams [BBD<sup>+</sup>02] – whose end may not be known in advance and where data items may arrive in a time-varying manner. This is what we call discrete versus continuous *operation mode* (cf. Table 1.1) and this classification shall relate only to data being processed. The reason for this restriction is that data stream processing yields some fundamental new research problems that are out of scope to this thesis. We submit that discrete operation mode resembles today’s standard SOAP and REST based Web service technologies, as they consider processing bounded messages (of a well-defined structure). Support for continuous stream services has been addressed, however, for service-oriented platforms in various works (e.g., [ABM04, SPL<sup>+</sup>04, BFL<sup>+</sup>07, GRL<sup>+</sup>08, BS11]).

A functional unit is assumed to provide the (technical) means to *invoke* it, meaning that this is the interaction primitive for starting its execution. However, a service does actually not provide a direct technical instrument to invoke it. Invocation here is understood as its instantiation (by an execution engine). In an ideal world, invocation as well as the invoked functional unit itself never fails. In a real world with concurrency, conflicting access to shared resources, and undesired phenomena of a stochastic nature they may occasionally fail. For example, there might be sudden changes that do not allow an operation to finish. This topic together with recovery will get into main focus in Chapter 5.

Finally, Figure 4.2 shows that every functional unit *presents* exactly one profile. This might appear a restriction considering the fact that there are reasons that suggest a model in which especially services can present multiple profiles, which is the case, for instance, in the OWL-S framework. In fact, this is a simplification rather than a restriction, justified by the fact that we are concerned with the service execution task. A service might actually present multiple profiles but this aspect is not relevant when it comes to execution and is therefore not represented in the model. This will be further explained in Section 4.1.5 when discussing the profile.

---

<sup>2</sup>The nature of data that can be processed is not always made clear in the literature on Web Services and Semantic Services thereby leaving space for unintended interpretation.

<sup>3</sup>Internal nondeterminism refers to the case where a choice is made by the system as the result of an internal nondeterministic decision. In contrast, external nondeterminism reflects the case where a choice is made by the environment outside the control of the system.

### 4.1.2 Operation

An operation is one of possibly many functional units of which a service is composed of. Operations can be compared to the notion of *actions* in action and planning theory [GNT04, HLP08] or *tasks* (activities) in workflow theory [JB96]. That said, we understand an operation as having no externally visible substructure: from the outside it is observed as an indivisible, atomic unit. In contrast to a service, an operation is therefore not associated with a process. This means that it is understood as a black box of which internals and its internal behavior are not known (not of further interest).

Usually, an operation realizes some functionality that is either too “small” to make up a service on its own or that is an idiosyncratic part of a service. Conversely, determining the operations of a service depends, in part, on how it can be functionally decomposed. Note that an operation is not assumed to be private or hidden. Analogous to a service, an operation is publicly visible in the system; it is just an integral part of a service.

Invocation of operations is inherently remote for services provided in a network (e.g., the Internet) and is done by means of a request message for *request-reply* and *one-way* interaction protocol operations. We understand a one-way interaction as asynchronous, meaning that the invoker can proceed immediately. A request-reply interaction is understood as synchronous: control proceeds only after the reply has been received (which does not necessarily imply busy-waiting of the invoker for the reply).

One-way and request-reply interaction protocols imply that operations are *stateless*. This is not a limitation since a *stateful* operation, as it is understood here, can equally well be viewed as a composite service. Without loss of generality we can therefore make the following assumption.

**(A3)** Operations are *stateless*.

Statelessness refers to the property of not maintaining *conversational* state beyond the duration of an invocation. When an invocation completes, the state is no longer retained. In other words, interaction with a stateless operation is subject to a total amnesia of conversational memory once invocation completes. Statelessness is important insofar as it is one means of facilitating loose coupling, which is perhaps the main principle in service-oriented computing. However, the importance here lies in the constriction of operations to simple one-way or request-reply interaction protocols. All interaction protocols that go beyond the simple request-reply message exchange inevitably require maintaining a conversational state (a.k.a. session); hence, are stateful. We submit that a stateful operation can always be converted into a stateless one either by putting conversational state to the invoker side (frontend) and including relevant parts in each invocation as additional input data or by putting it to secondary and possibly persistent memory (backend) used by the operation on each invocation [Jos07, Chapter 15].

In the system model, an operation has exactly one *implementation* that performs the functionality when invoked. In practice, however, one might want to extend this and allow an operation to be implemented in multiple while functionally equivalent ways. There are two main reasons that motivate this. First, to allow the provider to make an operation accessible in multiple technical ways (e.g., using various protocol and mes-

sage/data formats). Second, to provide different implementations with varying quality characteristics (e.g., performance). Imagine, for instance, a text retrieval operation implemented using different algorithms of which one performs best for long query input texts whereas the other for rather short query inputs.

The simplification of the model to exactly one implementation per operation is made for the same reason than the simplification that functional units present exactly one profile (further explained in [Section 4.1.5](#)): invocation of an operation at execution time actually invokes one and only one of its implementations. Deciding which one to chose if there are multiple implementations available is an optimization problem that is relevant but not in the immediate focus of this thesis. Clearly, the decision can involve (i) the different quality characteristics, (ii) the preferred access technology, or (iii) the combination of both. The time when such a decision is made is a typical instance of early binding (where the decision is made at design time, hence, it is static at runtime) versus late binding (where the decision is made dynamically at runtime whenever necessary).

It is important to understand that an operation – analogous to a service – is thought of as an abstract representation of functionality. Only implementations deal with concrete levels of realization and usage. Therefore, an operation that has no implementation is said to be *abstract*. Obviously, an abstract operation is only useful for static analysis of its properties but invocation is actually not possible. Since we concentrate on service execution – which does not make sense for abstract operations (and abstract services) anyway – we restrict the model to require at least one implementation per operation; this restriction can otherwise be dropped.

For the sake of completeness, we mention other (technical) aspects that are, however, not important for the purpose of this chapter. First, remote invocation of an operation is assumed to take place in a best effort network where messaging is reliable to a level as provided by the data transmission protocol used such as TCP. Second, an operation may, optionally, provide means to get canceled prematurely. Otherwise an operation is assumed to be indivisible, which is the more common case. If cancellation is supported then it is understood analogous to a transactional rollback: Once an invocation has been canceled all effects that have been created up to this point will be reversed. Finally, completion of one-way interaction style operations may be accompanied by an acknowledge/notification message.

### 4.1.3 Implementation

An implementation is a concrete realization of the functionality specified by an operation. It is, therefore, associated to one and only one operation. Moreover, an implementation is assumed to be accompanied by a declarative description of all technical details on how to access and use it. This includes message and data formats (e.g., JSON, XML, YAML), transport protocols (e.g., HTTP), and address information (i.e., an endpoint reference such as an URL). The technical concept is known in OWL-S as a *grounding* and in WSDL as a *binding*; from a conceptional point of view, they are so closely related that they can be considered synonyms. These technical details are represented in the system model by the Grounding association class.

Assuming that different implementations of the same operation realize the intended functionality correctly and only this functionality, they can only differ in non-functional properties, the conditions required to be operable (see [Section 4.1.5](#)), and their access technology (e.g., SOAP, REST, RPC).

#### 4.1.4 Service

A service is a functional unit that is composed of a finite set of operations and that can also include other services as sub services. This effectively yields a tree-structured containment relation that bottoms out in operations and allows to recursively decompose a service. However, the containment relation is of minor relevance compared to the behavioral perspective that views the routing of control and data between operations and sub services (see [Section 4.1.6](#)).

In order to clarify the notion of services, [Pre04] put forward a model-theoretic definition making a distinction between *abstract* and *concrete* services. An abstract service is understood as the set of all its concrete services. A concrete service is therefore an actual occurrence (or realization) of an abstract service (e.g., a concrete emergency case handled as defined by the *emergency assistance* service, see [Section 2.2](#)). The differentiation between an abstract and a concrete service is synonym to using the terms *service type* and *service instance*. Analogously, the terms *parametric* and *ground* service used in [BML<sup>+</sup>05] refer to the same concept (parametric = type, ground = instance). To be consistent with the process terminology set forth in [Section 4.1.6](#), we will use the terms *service instance* and *service*, the latter being the short form for an abstract service.

A service is assumed to be globally available in the system. Every actor (e.g., a client) in the system can possibly invoke a service provided that the actor has the technical means and is authorized to do so; note that the security aspect is out of scope to this thesis. This also means that it can be very well invoked by a client or the provider of the service. In the former case the service can be seen as passive (i.e., the provider awaits requests) while for the latter it can be seen as active (i.e., started on the initiative of the provider). Invocation of a service eventually results in invocation of operations as specified by its process. Finally, a service can be cross or just intra-organizational. A service is cross-organizational if it includes sub services provided by different organizations and intra-organizational otherwise.

#### 4.1.5 Profile

A profile<sup>4</sup> describes the functional and non-functional properties of a service or an operation, thereby making them self-describing. In other words, having access to the profile is sufficient and is the only means to get to know these properties in advance without actually using a service or an operation. The profile does not, however, tell anything about the behavior of a service.

Regarding the functional dimension, this includes (i) *input* data consumed, (ii) *output* data produced, (iii) *preconditions* required to be satisfied in order to operate effectively,

---

<sup>4</sup>In addition to the term profile, the terms *capability description* and *signature* are synonymously used in the literature (e.g., [PKPS02, SWKL02, GMP06]).

and (iv) *effects* that are asserted to hold upon successful termination. Preconditions and effects, therefore, express changes that are going to be made in the course of execution.<sup>5</sup> Capturing the functional aspect of a service or an operation by these four categories – hereafter abbreviated *IOPE* – originates from action theories in Artificial Intelligence and is known as the *change semantics*.<sup>6</sup> A key aspect in all Semantic Service frameworks is that *IOPEs* are described relative to a shared application domain conceptualization – an ontology. The ontology captures the terminological domain knowledge by defining the concepts of the domain. In our case this is a Description Logic theory – the TBox if loaded into a KB.

Regarding the non-functional dimension, a profile includes properties that describe how a functional unit is supposed to be and which can be used to judge its quality characteristics. Exemplary categories are performance, cost, security, trust, reliability, or transactional properties. Analogous to *IOPE*, properties in the non-functional category – hereafter abbreviated *N* – are semantically described relative to an ontology, which models typical QoS concepts in this case.

A profile might also include properties that neither fit well into the functional nor the non-functional dimension but rather onto a meta-level. Examples are a list of categories for the purpose to classify them according to a (business) taxonomy or the creation date and version number of the profile. Meta properties are rather relevant for service selection, but barely for the service execution task since they do not play an ultimate part in the course of execution.

As already indicated in [Section 4.1.1](#), there are reasons for allowing especially services to present multiple profiles. The main motivation is to allow service providers to publish different profiles for different use cases. For instance, the provider of the shipment service in the book seller scenario (see [Section 2.1](#)) might want to publish two profiles: one for express delivery and another one for regular delivery. The former would usually differ from the latter in the non-functional property representing its costs, as extra costs are charged. However, the instantiation of a service for the purpose of its execution ultimately involves a commitment for exactly one profile. Both the service provider and the service consumer certainly do not want to leave the properties of the service that is to be enacted open in the sense that the provider might deliver properties of one or another profile of a complex service. Rather, execution starts from an agreement upon exactly one profile. This also holds for the sub services (if any) of a service. Therefore, the system model starts from the point where the commitment for the single profile that is supposed to be enacted has been made.

Related to the aspect of single versus multiple profiles is the fact that profiles may be updated (frequently) as a result of functional or non-functional changes made by the service provider. In practice such updates are likely to be made more often to non-

---

<sup>5</sup>We note that the WSMO framework further distinguishes between preconditions versus assumptions and postconditions versus effects. Preconditions and postconditions are statements that relate to the inputs or outputs (referred to as the information space). Assumptions and effects are statements that neither relate to an input nor an output but the state of the world before and after an invocation. Since this distinction is not relevant from our perspective – both types can be checked at execution time (but not necessarily all of them can be checked at design time) – we consider preconditions/assumptions and postconditions/effects as synonyms, respectively, and will only use the terms precondition and effect.

<sup>6</sup>See [HLP08] for a comprehensive introduction to the most prominent action theories.

functional properties (e.g., changes in usage costs). We attribute a functional unit as *volatile* if its profile can be subject to changes. Clearly, the profile does not change for a service instance while it is executed for the same reason why there is one profile.

Finally, the profile maintainer and the service provider need not necessarily be the same. The profile of an e-commerce service, for example, may have been created by the service provider and augmented by a consumer organization or other parties. The reason is that non-functional properties can be classified in two categories: those *within* and those *beyond* the influence of the service provider [HKRK09]. Take as an example for the latter trustworthiness or reputation. It is hard to imagine that the service provider would honestly grade them. In fact, one would rather trust an independent third party that makes such assessments in an objective way. We therefore abstract from whether a profile is a single entity or not. A profile might be split into several parts that are even published in different places. On the other hand, profiles are obviously assumed to be made available for retrieval, which is usually done by means of system-wide directories – the service market, so to speak.

#### 4.1.6 Process

In essence, a process specifies and reflects the execution of a service. For the sake of clarity, we point out our understanding of the terms *process*, *process instance*, and *process model* first, as we intentionally decided to slightly deviate from the closely related terminology commonly used in Business Process Management [Wes07]. In our model, the term process is basically a short form (for the sake of brevity) loaded with two different meanings. First, it refers to a process specification (or schema) that describes the behavior of a service that occurs in the course of its execution. Clearly, a process instance is such a concrete occurrence of a process specification (occurring as a result of invocation of the service). This is the second meaning. Consequently, the term process is used interchangeably either to refer to the specification of a certain type of process or an instance of it (cf. service and service instance in Section 4.1.4). To avoid ambiguous cases when the meaning would not be clear from the context we will explicitly add specification, or instance then. Finally, the default meaning of the term process in our nomenclature corresponds to the notion of a *business process model* in [Wes07]; the latter should, therefore, not be confused with the process model described in Section 4.3 that describes the underlying formal model of processes.

It follows from Figure 4.2 that every service is described by one and only one process. Even services that are composed out of just one operation are described by a process. In this case the process is trivial and represents either a one-way or a request-reply style interaction.

Following activity-oriented process models, a process is essentially understood as a *control flow* together with a *data flow*. The former can be depicted by a control flow graph that describes the local execution dependencies between operations and embedded services. The data flow can be seen as unidirectional links connecting, for instance, an output of an operation to an input of a subsequent operation. Only by respecting the operational semantics when working off the control flow graph and handling the flow of data at execution time the intended functionality of the service is achieved.

There are two related notions in the areas of Web Services and BPM, namely *orchestration* and *choreography*. To clarify the relationship, we shall briefly review them. In [Pel03] an orchestration is characterized as “an executable business process that can interact with both internal and external Web Services”. Furthermore, it “represents control from one party’s perspective”; that is, from the perspective of the organization in which it is executed. An orchestration can, therefore, be understood as a cross-organizational composite service that includes external services. In contrast, a choreography rather takes a birds-eye perspective. It has been given similar but still different meanings in the literature. We list three of them here. A choreography

- “allows each involved party to describe its part in the interaction” and “tracks the message sequences among multiple parties and sources [...] rather than a specific business process that a single party executes” [Pel03];
- “describes peer-to-peer collaborations of participants by defining, from a global viewpoint, their common and complementary observable behavior; where ordered message exchanges result in accomplishing a common business goal” [KBR<sup>+</sup>05];
- “designates a business task performed by multiple roles, but does not give an implementation composed of a set of participants. The specification of the individual participants is at the level of BPEL-like languages” [QZCY07].

In essence, a choreography defines a collaboration protocol between the peers (parties) that cooperate in a business process (i.e., a contract between the peers to which they need to adhere when executing their part). Such a protocol may allow for many different realizations, meaning that there can be many different (cross-organizational) composite services each having different processes that adhere to the choreography. This shows that a choreography is not executable “as is”. It is therefore not directly relevant to this thesis.

#### 4.1.7 Service Description

The Web Services Architecture [BHM<sup>+</sup>04] defines a *service description* as machine-processable including – amongst other aspects – a service’s semantics. The latter is viewed as a “contract between the requestor entity and the provider entity concerning the effects and requirements pertaining to the use of a service”.

Following [Pre04], a service description has, as its model, an abstract service (service type). It is the goal of a service description framework to achieve *completeness* regarding the abstract service described, meaning that the model of a service description includes all instances of the abstract service. Completeness is basically achieved in all DL and FOL-based frameworks both in the functional as well as the non-functional dimension because of their grounding in TARSKI-style model-theoretic semantics [Tar56]. The basic idea is that the extension of the profile spans the model of an abstract service.

It has also been pointed out in [Pre04] that in practice service descriptions are usually not *correct* in the sense that there might be concrete services of an abstract service that



cannot be delivered at least temporarily by the service provider (e.g., the *order & pay* service of the book seller scenario that fails for some book because it is out of stock).

Analogous to the OWL-S framework, the profile, the process, and profiles of contained operations are considered the basic elements that constitute a declarative service description. However, without grounding information of operations, a service's description does not yet allow for its execution as the necessary technical details are missing on how to access and invoke the operations. Consequently, the service execution task inevitably requires a service description format that includes primitives for describing these grounding information, which we formulate by the following assumption.

(A4) Every service is annotated by an executable service description.

Clearly, a service description should have the same visibility in the system than the service about which it is made has. A service that is global, meaning that any actor in the system can possibly access it (provided that the actor is authorized to do so), should have a globally accessible service description.

Finally, the aspect of concrete service description formats, how they are stored, how they can be accessed, and appropriate means to achieve efficient and scalable discovery, aggregation, and search is a research topic of its own and largely beyond the scope of this thesis. This applies equally to security concerns. As there is no single solution, the system model is intentionally open in this regard. In distributed environments, however, dedicated service directories are usually considered for this, being either centralized systems or organized in a decentralized way.

## 4.2 Service Model

The service model described in the following essentially builds upon well-established works in two fields. First, the declarative while DL-based approach to conceptualize and reason about the semantics of functional and non-functional properties of services and their operations. Second, the action-based approach inspired by action and planning theories such as the Situation Calculus [Rei01] and the Fluent Calculus [Thi05] for representing the functional aspect in the behavior of services and operations.

The central notion here is the profile where a service or an operation is perceived as a black box. We start by defining the required notions of a *profile parameter*, a *precondition system* and an *effect system* upon which we eventually provide formal definitions for a profile, an operation, and a service.

### 4.2.1 Profile Parameter

A profile parameter (parameter for short) is the notion used to capture an input, an output, or a non-functional property; as noted in [Section 4.1.5](#), categories are abbreviated *I*, *O*, *N*, respectively. It consists of five elements. First, a *name* that is usually human-assigned. In the service model, the name of a parameter is merely used for identification purposes. Although practical names often carry superimposed semantics, we shall not

assume that this is generally the case for parameter names. Second, a *type* used to relate a parameter to a shared conceptualization (i.e., an ontology). The type is, therefore, the central means that captures the semantics of a profile parameter. Third, a *data value* that can be any kind of data item, but not a data stream as discussed in [Section 4.1.1](#). For an *IO* parameter, the data value is the actual input or output in the format as consumed or produced by an operation or a service. The data value of an *N* parameter is optional as it is not part of the data processed by a service; if used then it is the quantity or nominal assigned (e.g., `MAXRESPONSETIME = 30 sec`; `CERTIFICATION = ISO9001`). Fourth, a profile parameter includes a finite set of *representatives*. Representatives can be referred to by variables in preconditions and effects in order to instantiate (ground) them. Each representative is either an individual name or a lexical form (see [Definition 3.2](#) and [3.9](#)). The existence of representatives in addition to the data value is owing to the observation that practical operations and services are often heterogeneous in the sense that different data formats are in use. More specifically, a data value may be in a format that cannot be directly used within preconditions and effects.

Finally, a profile parameter includes an *assignment function*, denoted with  $\sigma$ , that captures how the representatives are determined. There are basically two types. For an *IO* parameter,  $\sigma$  establishes a relationship between the data value and the representatives (i.e., how the latter are derived from the former). For an *N* parameter,  $\sigma$  can also describe a query operating, for instance, over context information, a KB, or other sources of information. As will be seen,  $\sigma$  is simple (e.g., a syntactical transformation) down to trivial in some cases. A profile parameter is formally defined as follows.

**Definition 4.1** (Profile Parameter). *Let  $V_I, \mathbb{V}_{LS}$  be a set of individual names and lexical forms, respectively. A profile parameter (parameter for short) is a 5-tuple  $Pa = (id, type, val, Re, \sigma)$  where*

- *id is the name (or identifier) of the parameter,*
- *type is either a concept or a data range, called the type of the parameter,*
- *val is a data item, called the data value optionally assigned to the parameter,*
- *$Re = \{r_1, \dots, r_n\}$  is a finite set of representatives  $r_i \in V_I \cup \mathbb{V}_{LS}$  indexed by consecutive integers  $\{1, \dots, n\}$ , and*
- *$\sigma$  is an assignment function (or procedure) for  $Re$ .*

*The lowest numbered representative  $r_1$  is called the primary representative and the rest are secondary representatives.*

*A profile parameter is concrete if its type is a data range and general otherwise.<sup>7</sup> A profile parameter is instantiated (or ground) if  $Re$  is non-empty.*

We will use the function-like notation  $id(Pa)$ ,  $type(Pa)$ ,  $val(Pa)$ , and  $Re[i](Pa)$  to denote the name, type, data value, and the  $i$ -th representative of a parameter  $Pa$ , respectively. In concrete examples, we will write `PAR:Type` to denote a parameter named `PAR`, whose type is a concept or data range named `Type` (i.e.,  $type(PAR) = Type$ ).

<sup>7</sup>The terminology is chosen in analogy to concrete domains versus the general-purpose domain (see [Section 3.1.3](#)).

Two parameter  $Pa_1, Pa_2$  are said to have the *same name*, written  $id(Pa_1) = id(Pa_2)$ , iff the strings are the same sequence of characters. Two parameter are said to be *equivalent*, written  $Pa_1 \equiv Pa_2$ , iff their types are equivalent; that is, given a KB  $\mathcal{K}$ ,

$$\mathcal{K} \models type(Pa_1) \equiv type(Pa_2) .$$

Equivalent parameter therefore have the same extension in every interpretation  $\mathcal{I}$  that is a model of  $\mathcal{K}$ . Consequently, profile parameter are ascribed with Tarski-style model-theoretic semantics [Tar56] as membership over sets. Two parameter are said to be the *same*, written  $Pa_1 = Pa_2$ , iff they have both the same name and are equivalent.

The name and the type of a parameter are assumed to be static, meaning that they are assigned at design time. In contrast, we shall abstract from whether the data value  $val$  (and hence the set of representatives  $Re$ ) is constant or variable between different service instances since this is a relative matter.<sup>8</sup> In the same vein, we can abstract from whether there is a *default value* for  $val$ , which would be specified at design time and used by default whenever a value is not assigned at runtime. However, it is important that  $val$  and hence  $Re$  are service instance bound for  $IO$  parameter while we can also abstract from that for  $N$  parameter. An  $IO$  profile parameter can therefore also be understood as a statically typed instance variable while an  $N$  parameter need not be instance specific.

Details on data formats for  $val$  are intentionally left unspecified in [Definition 4.1](#). The reason is to allow the use of various kinds of (possibly “complex”) data depending on the actual data formats consumed and produced by an operation or service. Subject to the actual grounding, the data value might therefore further involve a second-party type. Note that *type* and the second-party type need not necessarily be the same, though both are certainly in a more or less close intensional relationship.

The primary representative  $Re[1]$  is special insofar as – analogous to a typed variable in a programming language – the type of a profile parameter defines the range of representatives that can be assigned for  $Re[1]$ . An assignment outside such a range is considered illegal. For a concrete parameter  $Pa$ ,  $Re[1]$  is generally a lexical form representing an element from the value space of the data range. Formally, given a  $\mathcal{DL} + \mathcal{D}$  interpretation  $\mathcal{I}$  with the datatype and data range interpretation function  $\cdot^{\mathcal{D}}$ ,

$$(Re[1](Pa))^{\mathcal{D}} \in (type(Pa))^{\mathcal{D}} . \quad (4.1)$$

Analogous, for a general parameter  $Pa$ ,  $Re[1]$  is required to be an instance of the concept; that is, given an interpretation  $\mathcal{I}$ ,

$$(Re[1](Pa))^{\mathcal{I}} \in (type(Pa))^{\mathcal{I}} . \quad (4.2)$$

We note that the requirement expressed by [Equation \(4.1\)](#) and [\(4.2\)](#) can be equally expressed by a precondition. Therefore, we leave it to the discretion of an implementation how they are actually enforced. Secondary representatives (if any) are exempted from this restriction and can even be a mixture of individuals and lexical forms. However, we require all representatives to be *compatible* to the preconditions and effects by whom they are referred. This will be detailed in [Section 4.2.2](#).

<sup>8</sup>One man’s constant is another man’s variable. [Per82]

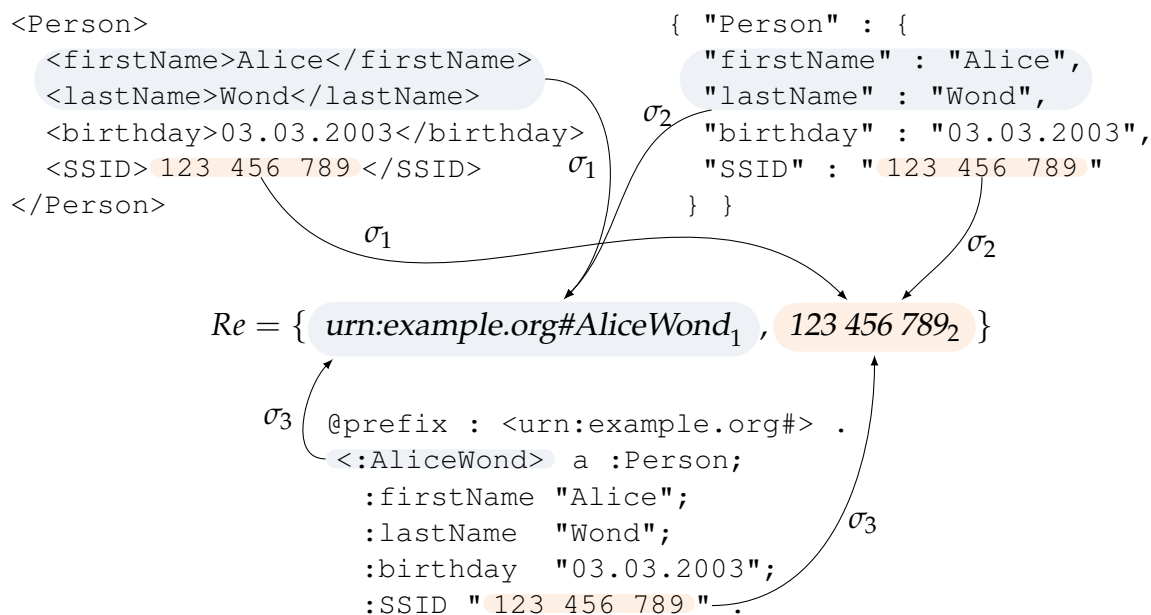


Figure 4.3: Exemplary input/output values in different data formats: XML (top left), JSON (top right), and RDF Turtle syntax (bottom). Value-specific assignment functions  $\sigma_1$ ,  $\sigma_2$ , and  $\sigma_3$  specify how representatives are determined. In this case the primary representative shall be a person's concatenated name plus a namespace prefix and the secondary representative its social security Id.

If a parameter is referred to by a precondition or an effect, or if it is taken into account for the purpose of dynamic failure recovery (the former of which detailed in [Section 4.2.2](#) and the latter in [Section 5.4.4](#)) then  $Re$  is non-empty. Elements of  $Re$  are then determined by the assignment function  $\sigma$ . In general,  $\sigma$  may be an  $n$ -ary function such that

$$Re := \sigma(\alpha_1, \dots, \alpha_n) \quad n \geq 0 \quad (4.3)$$

and where  $\alpha_1, \dots, \alpha_n$  are the arguments. As mentioned above, the representatives of  $IO$  parameters are solely derived from the data value; that is,  $Re := \sigma(val)$ . [Figure 4.3](#) illustrates the relationship between  $val$ ,  $\sigma$ , and  $Re$  using a simple practical example where also three different data formats are considered. Further details on  $\sigma$  are as follows.

### Concrete Input and Output Profile Parameter

Valid data values for concrete parameters are all lexical forms in the extension of the data range; that is,

$$(val(Pa))^{\mathcal{D}} \in (type(Pa))^{\mathcal{D}} . \quad (4.4)$$

This means that an operation or service directly takes or produces, for instance, an integer value. Therefore,  $\sigma$  is either a trivial identity mapping such that  $Re[1](Pa) = val(Pa)$  or a simple syntactical conversion from one lexical space into another (e.g., a date from the US locale specific date format into DE specific date format). Since there can be at

most syntactic differences between  $val$  and  $Re[1]$ , one of them – preferably the representative – is actually redundant and can be factored out in an implementation. Secondary representatives are usually not needed for unary datatypes such as numbers. For composite datatypes such as dates or other n-ary datatypes they might be needed in order to reference a component of a data value (e.g., the day of a date).

### General Input and Output Profile Parameter

There are basically two possibilities for general *IO* parameters. In the simple case, the data value is an individual name. Analogous to concrete *IO* parameters, this means again that  $Re[1](Pa) = val(Pa)$  and  $\sigma$  is the trivial identity mapping. Hence, in order to satisfy Equation (4.2),

$$(val(Pa))^{\mathcal{I}} \in (type(Pa))^{\mathcal{I}} . \quad (4.5)$$

An example for this simple case would be an operation that consumes or produces a data item that is an IRI and that refers to an OWL individual. Observe, that the data item can be an anonymous instance of the concept (i.e.,  $val(Pa) \notin V_I$ ). In this case, there would be an artificial name (that is not meaningful to the application) created by a Skolemization-style approach (e.g., an anonymous OWL individual corresponds to a blank node in RDF).

On the other hand, the data value might be a more complex structured data item. For instance, SOAP-based Web services use XML Schema elements to define the input/output parts of messages exchanged. Therefore,  $val$  can also be a data item that is an instance of a second-party type. Such a type is understood to allow for a possibly infinite set of data items where each data item is well-formed in the sense that it complies to the type's specification. Usually, the specification imposes structure and syntax on data (and might also include a set of operations that can be applied on instances of the type). Examples of such second-party types are XML Schema elements or types of (object-oriented) programming languages such as Java classes. The assignment function  $\sigma$  then represents a conversion that is supposed to select and/or extract the representatives (cf. Figure 4.3). Rather than being a generic function as in the simple case mentioned before,  $\sigma$  is then specific to the second-party type and needs to be defined as part of the design process. The Extensible Stylesheet Language (XSL) is one example for specifying and performing such conversions.

### Non-Functional Profile Parameter

Since a non-functional parameter – no matter whether concrete or general – differs from “real” *IO* parameters in the sense that it does neither represent a data item consumed nor produced,  $\sigma$  is a function that rather captures where the representatives come from. For instance, when the representatives are dynamically determined by a query  $q$  over a knowledge base  $\mathcal{K}$  (or any other information source), thus,  $Re := \sigma(\mathcal{K}, q)$ . Another example would be the case where  $\sigma$  is a trivial null-ary function returning a constant primary representative.

### Remark on Profile Parameter Types

The type of a parameter is taken from a domain ontology  $\mathcal{O}$  (that was loaded into a KB). In general,  $\mathcal{O}$  can be local or global in the domain. If it is local then there are possibly other local ontologies in the domain. They are designed independently, for instance, by different service providers, and all of them might model the domain in more or less different and overlapping ways. If  $\mathcal{O}$  is global then it is either shared by all participants in the system (which means that there is a high degree of standardization), or one can assume that it represents the combination of all the local ontologies where concepts and data ranges have been merged or aligned. The assumption of such a possibly dynamically constructed global ontology is (tacitly) made by numerous DL-based service discovery/selection approaches (e.g., [SWKL02, CWT08, SMM10, TRBD11]). The problem of (automated) merging or aligning multiple local ontologies [ES07] is not a goal of this thesis.

### 4.2.2 Preconditions and Effects

From the conceptual point of view taken in the service model our goal is it to capture the essentials of preconditions and effects in a general way. This is motivated by the abundance of different precondition and effect semantics in the AI literature that each have their own advantages and disadvantages. In fact, it has been argued in [Win90] that there is no single approach available that can be applied generally in any setting. For this reason, we will provide a generalization first by defining the notion of a *precondition system* and the notion of an *effect system* that capture the basics concerning change semantics of services/operations. After these notions have been introduced, we discuss concrete examples for each of them, some of which we have used in our implementation. We first clarify why one can separate the two systems.

**Observation 4.1.** *Preconditions and effects can be defined independently and dealt with in isolation.*

The question whether a precondition is satisfied is independent of the problem of how to update the current state of affairs according to an effect. Second, it is not necessary to take effects into account when reasoning about preconditions; vice versa when reasoning about effects. This shall not, however, preclude a combined treatment of both. For instance, reasoning whether all of a sequence of operations are executable one after the other and what their overall effect is.

Precondition as well as effect systems are made up of two elements, respectively:

1. A formal language to express concrete preconditions and effects.
2. An evaluation function.

The latter has either of the following purpose: For a precondition system it embodies the mechanism used to *check* whether a precondition holds true w.r.t. a representation of the current state of affairs in the domain of application, which is commonly referred to as the *world state*. For an effect system, the function embodies how to *update* the world state

representation according to an effect. The underlying basic idea taken from prominent action theories such as the Situation Calculus [Rei01] is that the world state and general *domain constraints*<sup>9</sup> of the application domain are axiomatized as a logical theory. Following the DL-based approach initially proposed in [LS02] and further extended, for instance, in [BML<sup>+</sup>05, LLMW06, BLL10], we consider the use of a DL knowledge base for representing the world state and general domain constraints. More precisely, the ABox represents the world state and the TBox domain constraints. In addition, an effect causes changes to the ABox but does not extend to the TBox:

(A5) The TBox is a *protected part*<sup>10</sup>, meaning that it is invariant under an effect.

For instance, an inclusion axiom  $Ambulance \sqsubseteq Vehicle \sqcap \leq 2 carries.Patient$  (which states that ambulances are vehicles that carry at most two patients) cannot be modified as the result of a service execution. A concrete assertion  $carries(A1, BOB)$  (which states that *BOB* is carried by ambulance *A1*) can very well be an effect of the execution of a service or one of its operations.

Changes to the TBox that might become necessary from time to time in concrete application domains are done outside the precondition and effect system. It should be evident that such changes need to be correctly coordinated with precondition checking and effect changes at runtime so that anomalies resulting from interfering accesses to the knowledge base can be avoided. Not only for this reason, we consider these three types of accesses to a KB to be made under a transactional regime, see [Chapter 6](#).

Though not explicitly included in the definition of a precondition system nor an effect system, *variables* in concrete precondition and effect systems are a means to establish links between (i) a precondition and an input or non-functional parameter, (ii) between a precondition and an effect, and (iii) between an input, output, or non-functional parameter and an effect. In other words, the use of variables in preconditions allows to “check” not only the current world state but, in addition, to check relevant parts of actual input and non-functional property values by referring to representatives of an input or non-functional property. Analogously, the use of variables in effects allows to express changes of the world state in two ways. First, to incorporate actual input, output, or non-functional parameter values by referring to a representative of them. As an example, consider the *trigger ambulance* service from [Section 2.2](#) that takes as an input the concrete ambulance that has been selected to get triggered. Then one would certainly want to express an effect that the selected ambulance is busy upon successful execution, which can only be done in this case based on the information provided by the input. Second, one can incorporate the results of preconditions in effects by referring to so-called solution set variables of preconditions, which essentially allows to change assertions about existing individuals that match a precondition. As an example, let us take again the *trigger ambulance* service and suppose that another precondition requires that there is a crew (e.g., consisting of a driver and an emergency physician) for the selected ambulance that is on standby. Again, one certainly wants to express

<sup>9</sup>Domain constraints are also referred to as *state constraints* or *integrity constraints* in the literature on actions and change (e.g., [LR94, HR99]) because they describe the possible states of the world.

<sup>10</sup>The notion of a protected part as a subset of a logical theory/knowledge base is used analogously, for instance, in [Win88b, GS88, CKNZ10].

that the crew is busy upon successful execution. Clearly, the crew is not provided as an input to the service but we shall assume that it can be retrieved in the process of precondition checking from the information in the KB; that is, it is returned as a solution if the precondition is satisfied. Hence, one would use a “crew” variable in the effect linked to a “crew” solution set variable in the precondition. Summing up, variables get instantiated at execution time thereby grounding preconditions and effects.

As the expression language might be “too powerful”, the evaluation function might be defined only for a subset of it (i.e., there can be classes of expressions for which it is not possible to define meaningful semantics of the check and/or the update function). Finally, since we aim at supporting automated precondition testing, effect application, and reasoning about them, we require the evaluation function to be decidable in its domain of definition.

## Precondition System

Having introduced at an informal level the basics of a precondition system, we formally define it as follows.

**Definition 4.2** (Precondition System). *Let  $\mathcal{L}$  be a Description Logic and let  $\mathbb{K}$  be the class of  $\mathcal{L}$ -knowledge bases. A precondition system is a pair  $PS = (\mathcal{L}^{PS}, f_{\text{chk}})$  where  $\mathcal{L}^{PS}$  is a formal language used to build expressions  $\varphi \in \mathcal{L}^{PS}$  and  $f_{\text{chk}}: \mathbb{K} \times \mathcal{L}^{PS} \rightarrow \{\text{true}, \text{false}\}$  is a partial function such that  $f_{\text{chk}}$  is decidable in its domain of definition. Let  $\mathbb{K} \times \mathcal{L}_{\text{chk}}^{PS}$  be the domain of definition where  $\mathcal{L}_{\text{chk}}^{PS} \subseteq \mathcal{L}^{PS}$ . An expression  $\varphi \in \mathcal{L}_{\text{chk}}^{PS}$  is called a precondition.*

The semantics of  $f_{\text{chk}}(\mathcal{K}, \varphi)$  is that it determines the truth-value of  $\varphi$  based on the KB  $\mathcal{K}$  at hand, which contains all the knowledge that, for instance, an execution engine actually has about the domain. Moreover, the exact semantics of  $f_{\text{chk}}$  depends on whether the OWA or the CWA is used (i.e., whether the KB is understood as an incomplete or complete representation of the domain). We do not prescribe the adoption of either paradigm as this is an application specific decision. However, in our implementation described in [Chapter 7](#) we will adopt the OWA.

Under the OWA,  $f_{\text{chk}}(\mathcal{K}, \varphi)$  returns true or false only if the truth-value of  $\varphi$  is either explicitly known or can be inferred (according to the rules of inference defined by the reasoning regime used) from what is explicitly known. A failure to prove  $\varphi$  to be false due to lack of knowledge has to be considered as *unknown*; analogous for true. One might be tempted to extend  $f_{\text{chk}}$  under the OWA such that it maps into a ternary target set  $\{\text{true}, \text{false}, \text{unknown}\}$ . However, under the principle of a conservative or cautious system, a result of unknown should also be taken as false: without further assumptions regarding error behavior of services/operations anything from success to harmful failure can happen if one would execute a service or an operation that has a precondition whose truth-value cannot be determined based on the information available. Hence, execution carries an element of risk in this case. To make this more explicit, we formulate the following general principle.

**Prudence principle.** *If one cannot determine whether a required condition to proceed is satisfied based on the information available about an environment that is otherwise not known then one should not proceed, as the consequences might not all be foreseeable.*



The only reasonable solution would be to expand efforts first in getting sufficient knowledge (provided that one can afford to do so). Therefore, we propose a conservative behavior under the OWA such that  $f_{\text{chk}}$  in [Definition 4.2](#) considers an unknown truth-value as false. In contrast, under the CWA, a failure to prove  $\varphi$  to be true implies that it is false. In other words, lack of knowledge of  $\varphi$  being true allows to infer everything that follows from  $\varphi$  being false.

Observe that  $f_{\text{chk}}(\mathcal{K}, \varphi) = \text{false}$  has different consequences on reasoning under the CWA versus the OWA. Whereas the CWA allows for additional inferences from the implication of  $\varphi$  being false, the OWA does not so if  $f_{\text{chk}}(\mathcal{K}, \varphi) = \text{false}$  actually results from an unknown truth-value of  $\varphi$  (i.e., one cannot infer anything from that in this case).

The same conservative behavior should be adopted for an  $\mathcal{L}$ -knowledge base where  $\mathcal{L}$  is a modal logic in which the truth-value of an axiom or assertion in the KB can be further qualified using the epistemic modal *possibly* operator  $\diamond$ . If a precondition  $\varphi$  is possibly true for all what is known – but not necessarily – then the opposite might as well be the case; hence,  $f_{\text{chk}}$  should return false in this case.

## Effect System

The notion of an effect system is defined in a similar way to the precondition system.

**Definition 4.3** (Effect System). *Let  $\mathcal{L}$  be a Description Logic,  $\mathbb{K}$  the class of  $\mathcal{L}$ -knowledge bases,  $\mathbb{U}$  the class of direct updates over  $\mathcal{L}$ -ABoxes. An effect system is a pair  $ES = (\mathcal{L}^{ES}, f_{\text{up}})$  where  $\mathcal{L}^{ES}$  is a formal language used to build expressions  $\varphi \in \mathcal{L}^{ES}$  and  $f_{\text{up}}: \mathbb{K} \times 2^{(\mathcal{L}^{ES})} \rightarrow \mathbb{U}$  is a partial function that maps a set of  $\mathcal{L}^{ES}$ -expressions  $E \in 2^{(\mathcal{L}^{ES})}$  into a direct update  $U \in \mathbb{U}$  over the ABox  $\mathcal{A}$  of  $\mathcal{K} \in \mathbb{K}$  such that  $f_{\text{up}}$  is decidable in its domain of definition. Let  $\mathbb{K} \times 2^{(\mathcal{L}_{\text{up}}^{ES})}$  be the domain of definition where  $\mathcal{L}_{\text{up}}^{ES} \subseteq \mathcal{L}^{ES}$ . An expression  $\varphi \in \mathcal{L}_{\text{up}}^{ES}$  is called an effect.*

The function  $f_{\text{up}}(\mathcal{K}, E)$  is understood as a realization of the *belief update* problem for knowledge bases [FUV83, Win90], which can be formulated as follows: given  $\mathcal{K}$  being a representation of some domain, how should  $\mathcal{K}$  be modified in the presence of changes within the domain represented by  $E$  so that the updated KB  $\mathcal{K}'$  still represents the domain after the change? The goal is to capture these changes syntactically by the direct update  $U$  that when applied to  $\mathcal{K}$  yields  $\mathcal{K}'$  ( $\mathcal{K} \xrightarrow{U} \mathcal{K}'$ ). Since the TBox is a protected part (see [Assumption 5](#)),  $U$  is obliged to affect the ABox only.

The reason for taking  $\mathcal{K}$  as one argument (as opposed to just an ABox  $\mathcal{A}$ ) is that this makes it possible to also take the TBox into account for determining  $U$ , which is an ultimate requirement in order to be able to address the frame and ramification problem. The reason for defining  $f_{\text{up}}$  over a set of effect expressions  $E$  is that interdependencies among a set of single effects might exist. While every sensible set of effects  $E$  should be pairwise consistent and consistent with the TBox, in the ABox update problem, it is naturally assumed that the effects in  $E$  may *conflict* (be inconsistent) with the existing ABox. If consistency of the KB needs to be preserved then the latter implies the need for a *conflict resolution* strategy embodied in the effect system.

Another reason for defining the notion of an effect system based on a function  $f_{\text{up}}$  is that this implies that it correctly represents deterministic operations/services (see

**Assumption 1**);  $f_{\text{up}}$  would not be a function under nondeterminism as the update  $U$  given  $\mathcal{K}$  and  $E$  might not be uniquely defined.

Apart from the restriction that effects do not extend to the TBox, **Definition 4.3** provides several degrees of freedom on how concrete effect systems can be defined. Notably, this includes the following: (i) effects might conflict with the existing ABox thereby necessitating a conflict resolution strategy; (ii) it is not stated whether  $\mathcal{K}$  must be consistent for an effect to be applicable; (iii) the precise semantics of how  $E$  is represented in the updated KB is not defined, which includes, for instance, *conditional effects* of the form  $\psi/\varphi$  where an effect  $\varphi$  is associated with a condition  $\psi$  that determines whether  $\varphi$  is enabled or not; and (iv) nothing is said about the DL  $\mathcal{L}$  nor  $\mathcal{L}^{ES}$ . Concrete effect systems, however, might not exploit the full spectrum spanned by these options. The main reason is that, depending on the expressivity of  $\mathcal{L}$ ,  $\mathcal{L}^{ES}$ , and the current state of  $\mathcal{K}$  (e.g., whether it is consistent or not), a decidable and practicable solution might not exist in general. This can be reflected in part by syntactic restrictions; that is, prohibited expressions in  $\mathcal{L}^{ES} \setminus \mathcal{L}_{\text{up}}^{ES}$  for which  $f_{\text{up}}$  is not defined. Major aspects along which concrete effect systems can differ are therefore (i) the expressivity of  $\mathcal{L}$ ,  $\mathcal{L}^{ES}$ , (ii) restrictions on the TBox (e.g., acyclic), and (iii) strategies to resolve conflicts. The former two are relevant for retaining decidability and practicability of the belief update problem.

## Classification of Effect Systems

A variety of proposals for semantics of updates to Propositional Logic and Description Logic knowledge bases have appeared in the AI and database literature. It has been argued in [Win90] that there is no update semantics that can be applied generally in any setting. As pointed out in [Win88a], the different semantics can be classified as either *model-based* or *formula-based*; with the *possible models approach* [Win88b, Her96] and the *possible worlds approach* [GS88] being prominent representatives, respectively.<sup>11</sup> No matter which, the basic principle that can be found in all update semantics<sup>12</sup> is that the knowledge base ought to change in a minimal way so that only those things that are forced to change by an effect are changed. This is inspired by Newton's law of inertia. As an example, consider the *trigger ambulance* service from **Chapter 2**. Suppose the occupancy of available ambulances is represented in an ABox using assertions  $\text{state}(A1, \text{IDLE})$  and  $\text{state}(A1, \text{BUSY})$  where  $A1$  shall identify a concrete ambulance. Further, suppose the only effect of *trigger ambulance* is that the selected ambulance changes its state from idle to busy. Consequently, nothing but the ambulance that has been selected (by the *select ambulance* service) should have changed state from idle to busy as a result of execution of *trigger ambulance*.

The *Katsuno and Mendelzon postulates* (KM postulates for short) [KM91] have captured what is the essence of the update process in a formal while logic-based way and also include a definition of the minimum principle. The authors argue that every sensible update semantics should satisfy these eight postulates. Yet it has been shown

<sup>11</sup>In the classification of updates that takes a data management point of view (see **Section 3.1.5**) all these semantics fall into the category of indirect update semantics (cf. **Figure 3.1**).

<sup>12</sup>See [HR99] for a comparison of various model-based semantics and [FMK<sup>+</sup>08] for a broad review of the research topic in relation to closely connected or overlapping research fields.

in [HR99] that prominent update semantics violate some of the postulates. As a result, they extracted uncontroversial postulates that are not violated by any of them.

As the minimum principle is sometimes too restrictive, the use of *occlusions* has been proposed for action theories, which have also been applied in the context of services [BML<sup>+</sup>05]. In short, occlusions are used as a device to exempt particular parts of the world state from the minimum principle under an update. In [BML<sup>+</sup>05], occlusions are expressions in the form of ABox assertions meant to be listed as part of a service description and are understood as occluding parts of the world state from the change semantics (i.e., these parts may change arbitrarily). Since occlusions open the gates for nondeterminism in the effects of services and their operations they are not considered in this thesis (see [Assumption 1](#)).

Conversely to occlusions, one might want to represent *rigid relations* [GNT04] in the knowledge base. In short, a rigid relation must not vary under an effect update nor under any other update per definition (e.g.,  $\text{birthplace}(J.S.BACH, EISENACH)$  with  $\text{birthplace}$  being a rigid role since the birthplace of a person cannot change over time). Therefore, they can be understood as protected parts. However, rigid relations are ABox assertions; hence, they are not covered by [Assumption 5](#). Yet there are basically two possibilities to represent them. First, in DLs having nominals one can use a technique to internalize (relevant parts of) an ABox into a TBox [RPZ10]. Using this technique, ABox assertions are rewritten as GCIs in the TBox in the following way:

$$\begin{aligned} a = b & \rightsquigarrow \{a\} \equiv \{b\}, \\ a \neq b & \rightsquigarrow \{a\} \sqcap \{b\} \sqsubseteq \perp, \\ C(a) & \rightsquigarrow \{a\} \sqsubseteq C, \\ R(a, b) & \rightsquigarrow \{a\} \sqsubseteq \exists R.\{b\}, \{b\} \sqsubseteq \exists R^-. \{a\} . \end{aligned}$$

This allows to move rigid relations into the TBox, thus, protecting them from changes over time. The second possibility is to flag an ABox assertion  $\alpha$  or even a role  $R$  indicating whether  $\alpha$  respectively each assertion  $R(a, b)$  is to be interpreted as rigid or not. This allows then to take into account the flag by an update procedure.

**Model-based (MB)** Under this paradigm, in which the OWA is usually adopted, an effect  $\varphi \in E$  is applied to the individual models  $\mathcal{I}$  of  $\mathcal{K}$  such that each new (updated) model  $\mathcal{I}'$  satisfies  $\varphi$ , written  $\mathcal{I}' \models \varphi$  (i.e.,  $\varphi$  is true in each new model). Consequently,  $\varphi$  expresses changes in an interpretation  $\mathcal{I}$  rather than in a knowledge base  $\mathcal{K}$ .<sup>13</sup> We write  $\mathcal{I} \xRightarrow{\varphi} \mathcal{I}'$  to denote the transition to the changed interpretation  $\mathcal{I}'$  that incorporates  $\varphi$  in some yet to be defined while minimal way; analogous for  $E$ . The classical meaning of minimal change as proposed for Propositional Logic in [Win88b] is defined in terms of a closeness relation between interpretations using symmetric set difference. One way of transferring this principle to DL interpretations (see [CKNZ10]) is to understand an interpretation  $\mathcal{I}$  as a set of atoms  $A(a), R(a, b)$ ; that is,

$$A(a) \in \mathcal{I} \Leftrightarrow a^{\mathcal{I}} \in A^{\mathcal{I}} \quad \text{and} \quad R(a, b) \in \mathcal{I} \Leftrightarrow (a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}} .$$

<sup>13</sup>Recap that a knowledge base represents the knowledge that, for instance, an execution engine has about the world, which is usually incomplete. In contrast, an interpretation is understood as a complete description of the world in this paradigm.

By using standard symmetric difference  $\mathcal{I} \Delta \mathcal{I}' = (\mathcal{I} \setminus \mathcal{I}') \cup (\mathcal{I}' \setminus \mathcal{I})$ , an interpretation  $\mathcal{I}'$  is strictly closer to  $\mathcal{I}$  than  $\mathcal{I}''$  if

$$\mathcal{I} \Delta \mathcal{I}' \subset \mathcal{I} \Delta \mathcal{I}'' .$$

This can be easily brought into a distance function by taking the cardinality of the difference

$$\text{dist}(\mathcal{I}, \mathcal{I}') = |\mathcal{I} \Delta \mathcal{I}'| .$$

In general, applying the effects  $\varphi \in E$  to  $\mathcal{I}$  may result in a set of new interpretations, which we will denote with  $E(\mathcal{I})$  (i.e.,  $\mathcal{I}' \in E(\mathcal{I})$ ). A set of effects  $E$  is *deterministic in  $\mathcal{I}$*  if  $|E(\mathcal{I})| = 1$  and *nondeterministic in  $\mathcal{I}$*  if  $|E(\mathcal{I})| > 1$  (e.g., for disjunctive effects  $\varphi_1 \vee \varphi_2$ ). Conversely, there may be no successor model  $\mathcal{I}'$  if an effect (or a set of effects) is *inconsistent with  $\mathcal{I}$*  (e.g., two effects  $\varphi_1 = \neg\varphi_2$ ). While one cannot generally know in advance whether an effect is consistent with  $\mathcal{I}$ , there is no value in a set of effects  $E$  in which effects are not mutually consistent. We shall therefore assume that there is an interpretation  $\mathcal{J}$  that satisfies each  $\varphi \in E$ . It is clear that if  $\mathcal{I}$  is a model of  $\mathcal{K}$  then  $\mathcal{I}'$  should be a model of the updated knowledge base  $\mathcal{K}'$ . Formally, given that  $U = f_{\text{up}}(\mathcal{K}, E)$ ,  $\mathcal{I} \Longrightarrow_E \mathcal{I}'$ , and  $\mathcal{K} \Longrightarrow_U \mathcal{K}'$  (see [Definition 3.14](#)) then

$$\mathcal{I}' \models \mathcal{K}' \text{ whenever } \mathcal{I} \models \mathcal{K}, \text{ thus, } \mathcal{K}' \text{ is consistent whenever } \mathcal{K} \text{ is.}$$

Moreover, if  $\mathcal{I} \models \varphi$  for some  $\varphi \in E$  then  $\mathcal{I} = \mathcal{I}'$ . In words, the interpretation should not change at all for each effect that is already satisfied, which relates to the principle of minimal change.

Let  $\mathcal{M}(\mathcal{K})$  be the set of models of a knowledge base  $\mathcal{K}$ . Under a so-called exact *logical update* [Liu10], the set of models of  $\mathcal{K}'$  is required to be exactly that set resulting from applying the effects in  $E$  to each model of  $\mathcal{K}$ ; that is,  $f_{\text{up}}$  realizes a logical update if

$$\mathcal{M}(\mathcal{K}') = \bigcup_{\mathcal{I} \in \mathcal{M}(\mathcal{K})} E(\mathcal{I}) . \quad (4.6)$$

This poses the following questions:

1. Given a DL  $\mathcal{L}$ , does a decidable procedure exist under which [Equation \(4.6\)](#) is generally ensured?<sup>14</sup>
2. Are there languages that are closed under such updates (i.e., is there a language  $\mathcal{L}$  in which the effects in  $E$  and the elements in  $U$  can always be represented)?<sup>15</sup>
3. If such a language exists, what is the size of  $U$  as a function of the overall input  $E$  and the initial  $\mathcal{K}$  (i.e., how does the size of the knowledge base change)?

The problem of model-based updates restricted to the ABox (which we are concerned with) has been studied in [Liu10]. In this work it was shown that even the basic DLs  $\mathcal{ALC}$ ,  $\mathcal{ALCO}$ , and  $\mathcal{ALCQIO}$  are not closed under logical updates to the ABox (i.e., it

<sup>14</sup>Observe that such a procedure would realize  $f_{\text{up}}$ .

<sup>15</sup>Observe that this assumes that  $\mathcal{L}$  is used for both effects and the knowledge base.

is not possible in general to express the elements in  $U$  in these DLs) or it is unknown whether this is possible (e.g.,  $\mathcal{ALCQI}$ ).<sup>16</sup> Expressing logical updates becomes possible only if these DLs are extended by the @ constructor known from Hybrid Logic [AdR00] or by moving to closely related Boolean ABoxes [ABHM03], which allow for assertions of the form  $\alpha_1 * \dots * \alpha_n$  where  $\alpha_i$  is either a concept or role assertion and  $*$  is either the Boolean connector  $\wedge$  or  $\vee$ . Therefore, weaker forms of updates have been studied in [Liu10], namely *approximate*, *projective*, and *approximate projective* update. Yet only some of the DLs mentioned are closed under these weaker updates while other still require to resort to Boolean ABoxes or the @ constructor; for details we refer to [Liu10]. It was also shown in this work that in the worst-case the size of the updated ABox in  $\mathcal{K}'$  is exponential in the size of the ABox of  $\mathcal{K}$  plus the size of  $E$  for logical updates and polynomial for projective updates. Similarly, it was shown in [GLPR09] that also  $DL\text{-}Lite_{\mathcal{F}}$  is not closed under logical ABox updates and therefore similar approximate update semantics have been proposed with worst-case polynomial complexity. In fact, in [CKNZ10] it was then shown that all the DLs in the  $DL\text{-}Lite$  family are not closed under the model-based update (and even revision) paradigm due to implicit disjunction intrinsically introduced by the principle of minimal change.

*Formula-based (FB)* Under this paradigm, the axioms and assertions in  $\mathcal{K}$  itself are the units of change (rather than interpretations as in MB). In short, given  $\mathcal{K}$  and a set of effects represented by  $E$ , the resulting direct update  $U$  contains a number of additions that represent the effects  $\varphi \in E$ . We call this the *primal* update and denote it with  $U_p$ . Analogous to the MB paradigm, the following properties should be ensured for the updated KB  $\mathcal{K}'$  (which are a subset of the KM postulates):

- $\forall \varphi \in E: \mathcal{K}' \models \varphi$ ;
- if  $\mathcal{K} \models \varphi$  then  $\mathcal{K}' = \mathcal{K}$ .

Under the premise that consistency is to be preserved for  $\mathcal{K}'$ , if  $E$  is inconsistent with  $\mathcal{K}$  (i.e., as soon as there is an effect in  $E$  that contradicts with  $\mathcal{K}$ ) then  $U$  contains, in addition, a number of deletes. We call this the *concomitant* update, denoted with  $U_c$ , such that, finally,  $U = U_p \cup U_c$ . In this case, the idea is to choose a *maximal* subset  $\mathcal{K}_{\max} \subseteq \mathcal{K}$  that is consistent with  $E$  (i.e.,  $U_c$  deletes the difference  $\mathcal{K} \setminus \mathcal{K}_{\max}$ ).<sup>17</sup> The problem here is that, in general, there might exist more than one such maximal  $\mathcal{K}_{\max}$ . As shown in [Example 4.1](#), this can also be the case for the ABox update problem we are concerned with. Consequently, this calls for a strategy on how to come up with one updated KB. Rather than a single obvious choice, there are basically two classes of possibilities to this. First, a maximal  $\mathcal{K}_{\max}$  is chosen based on some preference relation. This was proposed in [CKNZ10], called *bold semantics*; though the authors do not address the problem of appropriate preference relations. Second, all maximal  $\mathcal{K}_{\max}$  are combined into one. Several strategies have been advocated in the literature of which the *cross-product* [FUV83] (CP) and the *when in doubt throw it out* approach [Gin86] (WIDTIO) are prominent ones. Given  $\mathcal{K}$  and a set of effects  $E$ , let  $\max(\mathcal{K}, E)$  denote the set of all

<sup>16</sup>  $\mathcal{ALC}$ ,  $\mathcal{ALCO}$ ,  $\mathcal{ALCQI}$ ,  $\mathcal{ALCQIO}$  are all sublanguages of  $\mathcal{SROIQ}$ .

<sup>17</sup>Recap that for monotonic DLs deletion of existing axioms or assertions from  $\mathcal{K}$  is sufficient to resolve an inconsistency.

maximal  $\mathcal{K}_{\max}$  and  $\widehat{\mathcal{K}}$  the combined KB. CP defines the combination as the disjunction, whereby each  $\mathcal{K}_{\max}$  is understood as a conjunction<sup>18</sup> of its axioms and assertions

$$\widehat{\mathcal{K}}_{\text{CP}} = \bigvee_{\mathcal{K}_{\max} \in \max(\mathcal{K}, E)} \mathcal{K}_{\max} .$$

This ensures that no information gets lost. The downside is that  $\widehat{\mathcal{K}}_{\text{CP}}$  can be exponentially larger. Furthermore, since disjunction is needed to express  $\widehat{\mathcal{K}}_{\text{CP}}$ , it might no longer be expressible in the DL used. WIDTIO defines the combination as the intersection

$$\widehat{\mathcal{K}}_{\text{WIDTIO}} = \bigcap_{\mathcal{K}_{\max} \in \max(\mathcal{K}, E)} \mathcal{K}_{\max} .$$

The advantage of WIDTIO is that it is easy to compute. As pointed out in [Win88b], it also has a problem: progressively eliminating inconsistency (e.g., in the course of execution of services) can lead to less and less knowledge since axioms and assertions not in the intersection will be removed without further considerations.

#### Example 4.1

Consider the following knowledge base  $\mathcal{K}$  and an update  $E$  expressed in the form of an ABox assertion. In addition, suppose the UNA is not adopted.

$$\mathcal{K} = \{\text{Fun}(R); R(a, b), b \neq c\} \quad E = \{R(a, c)\}$$

Simply adding  $R(a, c)$  to  $\mathcal{K}$  clearly makes it inconsistent because  $R$  is functional and  $b, c$  are declared to refer to different individuals. To resolve the inconsistency one can either drop  $R(a, b)$  or the inequality assertion  $b \neq c$ . Consequently, there are two maximal  $\mathcal{K}_{\max}$ ; hence, at least two candidates for  $\mathcal{K}'$ :

$$\mathcal{K}'_1 = \underbrace{\{\text{Fun}(R); b \neq c\}}_{\mathcal{K}_{\max 1}} \cup \{R(a, c)\} \quad \mathcal{K}'_2 = \underbrace{\{\text{Fun}(R); R(a, b)\}}_{\mathcal{K}_{\max 2}} \cup \{R(a, c)\}$$

Whereas  $\mathcal{K}'_1$  still entails  $b \neq c$  as in  $\mathcal{K}$ ,  $\mathcal{K}'_2$  now entails  $b = c$ . Consequently, the first case prefers preservation of the interpretation of individual names over individual relations. The second case changes interpretation of names but preserves asserted relations. Under WIDTIO combination of  $\mathcal{K}'_1$  and  $\mathcal{K}'_2$ , the updated KB is  $\mathcal{K}'_{\text{WIDTIO}} = \{\text{Fun}(R)\} \cup \{R(a, c)\}$ ; thus, one loses the information regarding the (in)equality of  $b$  and  $c$ . Observe that the indeterminism illustrated by this example disappears under the UNA since synonyms cannot exist (i.e., (in)equality assertions are no longer needed).

The FB paradigm has also been criticised for the possibility of counterintuitive results (e.g., [BH93]). In particular, it has been shown in [Win88b] that in the presence of incomplete information the frame and ramification problem are not correctly solved

<sup>18</sup>Viewing a KB  $\mathcal{K}$  as a conjunction of its axioms and assertions is consistent with the definition that an interpretation  $\mathcal{I}$  satisfies  $\mathcal{K}$  if it satisfies *each* axiom and assertion.

since earlier FB update semantics considered only explicitly asserted knowledge but disregard implicit knowledge that can be derived (deduced). Therefore, more recent approaches to formula-based update semantics such as [CKNZ10] consider the deductive closure of a KB (i.e., all axioms and assertions that are entailed by a KB either explicitly or implicitly by inference). On the other hand, it has also been argued in [CKNZ10] that the KM-postulates are too strict in environments like the Web.

In summary, we conclude that there is still no single obvious approach available that would be ideally satisfactory and universal in terms of expressivity, computational properties, and avoiding unintuitive results. Moreover, we believe that it is necessary to examine whether the MB and FB paradigms are actually different. The interesting question is whether they can be mutually reduced or whether one is a proper subset of the other. The former would mean that they are equivalent. We conjecture that this is the case under the premise that one considers either the deductive knowledge closure or the set of models. In other words, if this conjecture is true then one can define any update semantic for DL knowledge bases equivalently in either way. Finally, an attempt to overcome the co-existence of different calculi by a unified action calculus has been published recently in [Thi11]. Yet it remains to be investigated whether it can be applied for the representation of effects in the context of services and whether it provides a more general approach.

### Concrete Precondition Systems

We give DL-based implementations of  $f_{\text{chk}}$  considering two different but related languages, thereby constituting two precondition systems:

**(PS1)** Conjunctive ABox queries (see [Section 3.1.4](#)) and the

**(PS2)** SPARQL query language [HS10] under the OWL 2 Direct Semantics Entailment Regime [GO10, Section 6] to ensure correct OWL DL semantics. We note that this approach is similar to the one presented in [SMM10].

Consequently, a precondition is expressed in terms of a query. For both **(PS1)** and **(PS2)**, the DL  $\mathcal{L}$  used can be  $\mathit{SROIQ}(\mathbf{D})$  or any sublanguage (i.e., any expressivity level up to OWL 2 DL). Regarding expressive power of queries, the main difference between them is that **(PS1)** allows for preconditions over the ABox of a knowledge base only while **(PS2)** allows formulating preconditions (i) over both the TBox and the ABox (i.e., over general domain constraints and the world state) and (ii) a particular form of higher order queries where variables can occur in the position of concept names and role names. More precisely, a SPARQL query can contain basic graph patterns that, when written as ABox query atoms, have the form

$$C?(x) \quad \text{and} \quad R?(x, y) \tag{4.7}$$

where  $C?$  and  $R?$  is a variable that can be substituted by a concept name or role name, respectively,  $x$  is either a variable or an individual name, and  $y$  is either a variable, an individual if  $R?$  is bound to an abstract role, or a data value if  $R?$  is bound to a concrete

role.<sup>19</sup> Moreover, the SPARQL approach allows queries over annotations of an OWL KB so that even the non-logical information part can be included.

No matter which of the two precondition systems is used, the basic idea is that if  $q$  is a query in either of them and  $\mathcal{K}$  is the information source queried then

$$f_{\text{chk}}(\mathcal{K}, q) = \begin{cases} \text{true} & \text{if } \mathcal{K} \models q \\ \text{false} & \text{otherwise .} \end{cases} \quad (4.8)$$

Consequently, precondition testing is reduced to query entailment checking if  $q$  does not contain solution set variables (i.e., for Boolean queries) and query answering otherwise. This also means that precondition testing inherits decidability and computational complexity properties resulting from the expressivity of the actual DL  $\mathcal{L}$  used (e.g., if  $\mathcal{L}$  is OWL 2 EL then combined query answering complexity for (PS1) is PSpace-complete [MGH<sup>+</sup>09]; see also Section 3.1.4). Finally, observe that Equation (4.8) adheres to the prudence principle no matter whether CWA or OWA is adopted.

The variables occurring in a query  $q$  are linked to profile parameters as follows; these interrelations are depicted in Figure 4.4. Let  $\text{Var}(q)$  be the possibly empty set of variables of a query  $q$  in either of the languages. If  $\text{Var}(q)$  is non-empty then it is one- up to three-partitioned into any combination of solution set variables  $\text{Var}_S$ , initially bound variables  $\text{Var}_I$ , and undistinguished variables  $\text{Var}_U$  (e.g.,  $\text{Var}(q)$  is two-partitioned if there are solution set and initially bound variables in  $q$  but no undistinguished variables). Note that undistinguished variables are understood as blank nodes in SPARQL. Each solution set variable  $v \in \text{Var}_S$  is referred to by an effect atom, thereby grounding the effect atom (see next subsection). Each variable  $v \in \text{Var}_I$  refers to a representative  $Re[i]$  of an input or non-functional profile parameter  $Pa$ . This means that such a variable is substituted by  $Re[i]$  already before query execution; one can also say that  $v$  is instantiated by  $Re[i]$ . Each undistinguished variable  $v \in \text{Var}_U$  cannot be linked at all (i.e., they are not used to relate parts of a precondition to profile parameters nor to effects). Finally, links are required to be *compatible*. A link between a representative  $Re[i]$  of a profile parameter  $Pa$  and a variable  $v$  is compatible if

$$\begin{aligned} Re[i](Pa) \text{ is an individual} &\leftrightarrow v \text{ occurs at the position of an individual,} \\ Re[i](Pa) \text{ is a lexical form} &\leftrightarrow v \text{ occurs at the position of a lexical form.} \end{aligned} \quad (4.9)$$

If  $q$  has solution set variables then it might be the case that its execution yields multiple solutions (results) in  $\mathcal{K}$ . As an example, consider the following precondition expressed as an ABox query

$$Person(x) \wedge CreditCard(y) \wedge hasCreditCard(x, y) \wedge validity(y, VALID) .$$

Suppose this precondition is specified for the *shipment* service of the book seller scenario from Chapter 2. It states that a person is required to have a valid credit card. Further, suppose that  $x$  is an initially bound variable (i.e., it refers to an input; hence, the actual person individual is determined by this input) and  $y$  is a result set variable (i.e.,

<sup>19</sup>In terms of FOL this means that variables can be used in the position of predicates (but not in the position of quantifiers and connectors.)



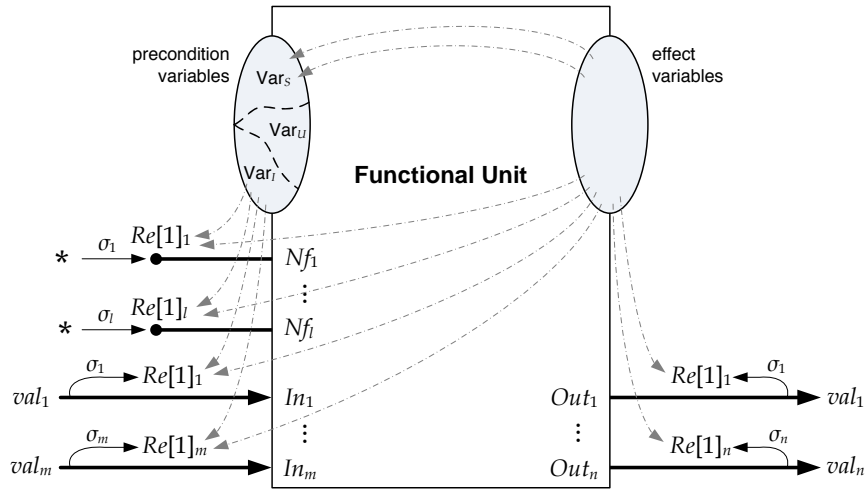


Figure 4.4: Links between representatives of profile parameters and variables in preconditions and effects (dotted lines represent possible links).

it is referred to by an effect). If this precondition is executed against a KB  $\mathcal{K}$  in which the person is known to have  $n > 1$  valid credit cards then there are  $n$  solutions for  $y$ . This poses the question which is the “best” solution to choose? In fact, this involves another question that has to be answered prior to that: Is such a situation allowed at all? It turns out that this depends on the actual use case; hence, it has to be answered individually. If any of the returned solutions can be chosen then such a situation is obviously allowed (i.e., it does not matter which solution is chosen). Otherwise there would be either a preference for a particular solution or a requirement that there must be at most one solution. While (PS2) allows for expressing such preferences over solutions (i.e., a ranking) as part of  $q$  by means of solution modifiers for ordering, this is not possible with (PS1). On the other hand, a requirement of at most one solution is not expressible in either of them. In fact, it is not difficult to see that such a requirement is a general domain constraint (e.g., each person might have at most one valid credit card, though this particular case clearly would not match well with reality). Consequently, it would have to be declared and enforced outside of preconditions.

As a matter of SPARQL’s grounding in RDF graph matching, (PS2) requires the KB to be represented in the form of RDF triples (see Section 3.3.4), which makes the use of an OWL knowledge base  $\mathcal{W}$  (see Definition 3.16) a natural consequence. Note that the use of an OWL knowledge base shall not imply that the full OWL expressivity spectrum is used in a certain domain of application; it might be a less expressive subset such as an OWL profile (see Section 3.3.3) or a sub DL. Also, we note that conjunctive ABox queries can be translated into SPARQL queries given that an appropriate entailment regime is used (i.e., the expressivity of SPARQL includes the former). In fact, this translation is straightforward and will be detailed in Section 7.2.1 when we describe our prototype implementation, which supports (PS2) and a slightly more expressive variant of (PS1).

Finally, we note that the use of the query language SPARQL-DL [SP07] constitutes another example for a precondition system where  $f_{chk}$  is again Equation (4.8). The expressive power of SPARQL-DL is stronger than ABox queries because the TBox can

be queried and the same form of higher order queries can be formulated (see [Equation \(4.7\)](#)). However, it is weaker than SPARQL under the OWL 2 direct semantics entailment regime because filter constraints (e.g., regular expressions over data values) and solution modifiers (e.g., ordering, limiting the number of solutions) do not exist. On the other hand, SPARQL-DL provides a more OWL friendly syntax because its syntax is closely related to the functional-style syntax for OWL. Also, a knowledge base need not be represented by RDF triples.

### Concrete Effect System

We give a DL-based implementation of  $f_{\text{up}}$  considering *positive* and *negative* ABox assertions as effect expressions. In the most general case, an effect  $\varphi$  is defined as follows:

$$\varphi := A(x) \mid R(x, y) \mid \neg A(x) \mid \neg R(x, y) \quad (4.10)$$

where  $A$  is a concept name ( $A \in V_C$ ),  $R$  is either an abstract or concrete role name ( $R \in V_{OP} \cup V_{DP}$ ), and the same rules apply to  $x, y$  as for conjunctive ABox query atoms (i.e., they can be variables, individuals, or lexical forms depending on the position where they occur and the role type, see [Section 3.1.4](#)). The first two forms are the positive while the latter two are the negative effects.

Following the formula-based approach for ABox updates proposed in [CKNZ10], we consider  $DL\text{-}Lite_{\mathcal{FR}}$  [CGL<sup>+</sup>07] (which is the basis of OWL 2 QL) and adoption of the UNA.<sup>20</sup> Together this ensures that the direct update  $U$  resulting from applying a set of effects  $E$  in the form given by [Equation \(4.10\)](#) to  $\mathcal{K}$  is (i) uniquely defined and that (ii) the elements in  $U$  are expressible in the same DL. As a consequence of (i) the problem of how to make a choice between multiple minimal knowledge bases (cf. [Example 4.1](#)) does not exist. Due to availability of existential restriction in  $DL\text{-}Lite_{\mathcal{FR}}$ , effects are, however, not necessarily deterministic in general, which will be detailed at the end of this subsection. Likewise, not imposing additional syntactic restrictions on TBoxes then effects may imply ramifications (i.e., implicit effects).

We show that  $U$  is still uniquely defined when  $DL\text{-}Lite_{\mathcal{FR}}$  is extended with (i) full existential restriction  $\exists R.C$ , (ii) concrete roles, (iii) all additional role types introduced with  $\mathcal{SROIQ}$  except transitive roles, (iv) role disjointness axioms, and (v) negated role assertions  $\neg R(a, b)$ . We call the extended version  $DL\text{-}Lite_{\mathcal{FR}}^{++}$ . Similarly, the extension to *conditional effects* of the form  $\psi / \varphi$  where  $\psi$  is a condition determining whether the effect  $\varphi$  is enabled is straightforward and does not pose any problems, but is not considered further here.

In short,  $DL\text{-}Lite_{\mathcal{FR}}^{++}$  is syntactically defined as follows. If  $A \in V_C$  is a concept,  $S \in V_{OP}$  and  $T \in V_{DP}$  are abstract and concrete roles, respectively, then the following are also concepts, roles:

$$B := A \mid \exists R.C \mid \exists T.dr \quad C := B \mid \neg B \quad R := S \mid S^-$$

where  $dr$  is a data range. TBox axioms are of the form:

$$\begin{array}{ccccc} \text{Fun}(R) & \text{Asy}(R) & \text{Sym}(R) & \text{Ref}(R) & \text{Irr}(R) \\ \text{Dis}(R_1, R_2) & \text{Dis}(T_1, T_2) & B \sqsubseteq C & T_1 \sqsubseteq T_2 & R_1 \sqsubseteq R_2 \end{array}$$

<sup>20</sup>The assumption of unique names is common to all standard action theories [Liu10, Thi11].

such that neither  $\text{Fun}(R_2)$  nor  $\text{Fun}(R_2^-)$  is in the TBox. ABox assertions are of the form:

$$A(a) \quad \neg A(a) \quad R(a, b) \quad \neg R(a, b) \quad T(a, v) \quad \neg T(a, v) .$$

The semantics of these constructs is defined in the standard way as introduced in [Section 3.1](#). Also, note that  $B_1 \sqcup B_2 \sqsubseteq C$  can be encoded by two inclusions  $B_1 \sqsubseteq C, B_2 \sqsubseteq C$ ;  $B \sqsubseteq C_1 \sqcap C_2$  can be encoded by  $B \sqsubseteq C_1, B \sqsubseteq C_2$ ; disjointness of two concepts  $B_1, B_2$  can be encoded by  $B_1 \sqsubseteq \neg B_2$ ; and  $\text{Sym}(R)$  is syntactic sugar since it is equivalent to  $R^- \sqsubseteq R$ .

Let  $\mathcal{L}^{ES}$  be the set of possible effects in the form given by [Equation \(4.10\)](#) such that all symbols are taken from a given vocabulary  $(V_C, V_{OP}, V_{DP}, V_I, V_{LS})$  and a set of variable names  $V_V$ . Then the effect system is:

**(ES1)**  $\mathcal{L}^{ES}$  with  $DL\text{-Lite}_{\mathcal{FR}}^{++}$ -knowledge bases, adoption of the UNA, and formula-based update semantics as detailed below.

By recalling what has been discussed before when we introduced the notion of an effect system, we make the following assumption specific to this effect system:

**(A6)** Given a consistent knowledge base  $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ , a set of effects  $E$  is consistent with  $\mathcal{T}$  but may be inconsistent with  $\mathcal{A}$ .

The semantics of the four types of effects in [Equation \(4.10\)](#) is defined as follows. Let  $\varphi$  be an effect in either form and  $\mathcal{I}, \mathcal{I}'$  be interpretations that (i) share the same domain  $\Delta^{\mathcal{I}} = \Delta^{\mathcal{I}'}$ , (ii) include the same datatype map  $\mathcal{D}$ , and that (iii) agree on interpretation of individual names, that is,  $a^{\mathcal{I}} = a^{\mathcal{I}'}$  for each individual name  $a \in V_I$ . Note that we do not need to state (iii) analogously for lexical forms since this is implicitly ensured by saying that  $\mathcal{I}, \mathcal{I}'$  include the same datatype map  $\mathcal{D}$ . Finally, let  $y^{\mathcal{I}\mathcal{D}}$  be a short form for  $y^{\mathcal{I}}$  if  $y$  is an individual name and  $y^{\mathcal{D}}$  if  $y$  is a lexical form. Then  $\mathcal{K}'$  accomplishes the application of  $E$  to  $\mathcal{K}$  if  $\mathcal{I}, \mathcal{I}'$  are models of  $\mathcal{K}, \mathcal{K}'$ , respectively, and the following holds:

$$\forall \varphi \in E: \text{if } \left\{ \begin{array}{ll} \varphi = A(x) & \text{then } x^{\mathcal{I}} \in A^{\mathcal{I}'} \\ \varphi = R(x, y) & \text{then } (x^{\mathcal{I}}, y^{\mathcal{I}\mathcal{D}}) \in R^{\mathcal{I}'} \\ \varphi = \neg A(x) & \text{then } x^{\mathcal{I}} \notin A^{\mathcal{I}'} \\ \varphi = \neg R(x, y) & \text{then } (x^{\mathcal{I}}, y^{\mathcal{I}\mathcal{D}}) \notin R^{\mathcal{I}'} \end{array} \right\} \text{ for each model } \mathcal{I}' \text{ of } \mathcal{K}'. \quad (4.11)$$

Observe that the meaning of a negative effect is that  $\mathcal{I}' \models \neg \varphi$  rather than  $\mathcal{I}' \not\models \varphi$ . Consequently, such an effect is not a knowledge retraction. Furthermore, [Equation \(4.11\)](#) implies that  $\mathcal{K}'$  is consistent but it does not yet show how to deal with the situation when an effect is inconsistent with the ABox of  $\mathcal{K}$ , which will be addressed below.

In order for an effect  $\varphi$  to be applicable to  $\mathcal{K}$ ,  $\varphi$  must be ground; that is, variables in  $\varphi$  must have been instantiated either by an individual name or a lexical form. Therefore, it is to be ensured at design time of a service description that a variable in an effect is linked either to a solution set variable of a precondition or a representative of a profile parameter such that the link is compatible as stated by [Condition \(4.9\)](#). This way all variables get instantiated at runtime (i.e., when a service instance is executed).

From [Equation \(4.11\)](#) we can directly derive the primal update  $U_p$ . The only significant detail is to distinguish between OWA and CWA. A positive effect  $\varphi$  as well as a

negative effect  $\neg\varphi$  is directly added to the ABox under the OWA due to fact that negative information needs to be maintained equally to positive information.<sup>21</sup> Under the CWA, however, a negative effect  $\neg\varphi$  need not be added. Recap, under the CWA it holds that  $\mathcal{K} \not\models \varphi$  implies  $\mathcal{K} \models \neg\varphi$ . Observe that it would not harm to add  $\neg\varphi$ ; it is just not necessary. In addition, and irrespective of OWA versus CWA, we need to ensure that in case the complement of an effect to be added is entailed by  $\mathcal{K}$ , no matter whether explicitly or implicitly by inference, it must no longer be entailed by  $\mathcal{K}'$ . Hence, the complement needs to be deleted if it is explicitly asserted. If the complement is implicitly entailed by  $\mathcal{K}$  then one needs to find and delete those assertions in  $\mathcal{A}$  that endorse the entailment. By slightly adapting [Algorithm 2](#) shown in [Appendix A.1](#) one gets an algorithm that finds them. In summary, given  $\mathcal{K} = (\mathcal{T}, \mathcal{A})$  and  $E$  then either

$$U_p = \{\mathcal{K} + \varphi \mid \varphi \in E\} \cup \underbrace{\{\mathcal{K} - \neg\varphi \mid \varphi \in E, \neg\varphi \in \mathcal{A}\}}_{\text{delete complement } \neg\varphi \text{ of effect } \varphi \text{ that is explicit in } \mathcal{A}} \cup \underbrace{\{\mathcal{K} - \psi \mid \psi \models_{\mathcal{K}} \neg\varphi \text{ and } \varphi \in E\}}_{\text{delete assertions } \psi \text{ that implicitly entail complement } \neg\varphi \text{ of effect } \varphi} \quad (4.12)$$

under OWA, or rather

$$U_p = \{\mathcal{K} + \varphi \mid \varphi \in E \text{ and } \varphi \text{ one of } A(a), R(a, b)\} \cup \{\mathcal{K} - \neg\varphi \mid \varphi \in E, \neg\varphi \in \mathcal{A}\} \cup \{\mathcal{K} - \psi \mid \psi \models_{\mathcal{K}} \neg\varphi \text{ and } \varphi \in E\} \quad (4.13)$$

under CWA.

As stated before, if  $\mathcal{K}'$  would become inconsistent by simply applying  $U_p$  to  $\mathcal{K}$  then one deletes a minimal number of assertions from  $\mathcal{A}$  that cause the inconsistency. Formally,

$$U_c = \{\mathcal{K} - \varphi \mid \varphi \in \mathcal{A}, \varphi \notin E, \text{ and } \varphi \text{ causes inconsistency}\} \quad (4.14)$$

and finally

$$f_{\text{up}}(\mathcal{K}, E) = U_p \cup U_c .$$

Clearly, we need to detail how  $U_c$  is determined. We first show that  $U_c$  is unique in general regardless of  $\mathcal{K}$  and  $E$ ; hence, there is exactly one  $\mathcal{K}'$ . In order to proof this we need to find all *minimal* combinations of axioms/assertions that lead to inconsistency. By showing that for each combination there is a single way of resolving the inconsistency we are done; if there were multiple ways then this would imply the need to make a nondeterministic choice.

The following chain of argumentation is based on [CKNZ10, Section 5]. We include transitive roles ( $\text{Tra}(R)$ ) in order to show that they lead to two ways of resolving an inconsistency, which justifies why  $DL\text{-Lite}_{\mathcal{FR}}^{++}$  is restricted in this regard. Let  $\mathcal{L}_{\text{Tra}}$  be  $DL\text{-Lite}_{\mathcal{FR}}^{++}$  extended by transitive role axioms.

<sup>21</sup>If a KB entails  $\varphi$  and there is an effect  $\neg\varphi$  then we can alternatively establish knowledge retraction semantics for negative effects by ensuring that the updated KB no longer entails  $\varphi$ .

Table 4.1: Combinations of TBox axioms and ABox assertions that cause KB inconsistency for  $\mathcal{L}_{\text{Tra}}$  and in the absence of the UNA. For the sake of brevity, we slightly abuse our notational conventions (see Section 3.1.1) since we use  $a, b, c$  at the filler position also for lexical forms if  $R$  is a concrete role.

TBox	ABox	TBox	ABox
	$C(a), \neg C(a)$	Fun( $R$ )	$R(a, b), R(a, c), b \neq c$
	$R(a, b), \neg R(a, b)$	Fun( $R^-$ )	$R(a, b), R(c, b), a \neq c$
	$R(a, b), \neg R^-(b, a)$ or $R^-(a, b), \neg R(b, a)$	Asy( $R$ )	$R(a, b), R(b, a)$
Dis( $C, D$ )	$C(a), D(a)$	Sym( $R$ )	$R(a, b), \neg R(b, a)$
Dis( $R, S$ )	$R(a, b), S(a, b)$	Ref( $R$ )	$\neg R(a, a)$
Tra( $R$ )	$R(a, b), R(b, c), \neg R(a, c)$	Irr( $R$ )	$R(a, a)$

Analyzing when an  $\mathcal{L}_{\text{Tra}}$ -KB becomes inconsistent under addition of new  $\mathcal{L}_{\text{Tra}}$ -ABox assertions and in the absence of the UNA then one finds exactly the following cases. First, if a single assertion  $C(a)$  is added for an unsatisfiable concept  $C \sqsubseteq \perp$ . Second, in any of the cases shown in Table 4.1. Out of these cases only those are relevant in which three (or more) assertions lead to an inconsistency. This is the case for transitive, functional, and inverse-functional roles. Each of them leads to indeterminism regarding which assertion to delete. It is easily seen that it is sufficient to delete either of the three assertions to resolve the inconsistency; hence, there is a choice. By disallowing transitive roles, this case ceases to exist. Furthermore, adopting the UNA renders (in)equality assertions unnecessary (cf. Example 4.1); hence, the two remaining cases of inverse-functional and functional roles “shrink” to two assertions. Consequently, there can only be cases where one or two assertions lead to inconsistency, which leads us to the following lemma.

**Lemma 4.4** (Adapted version of [CKNZ10, Lemma 12]). *Let  $\mathcal{T} \cup \mathcal{A}$  be an DL-Lite $_{\mathcal{FR}}^{++}$ -KB. If  $\mathcal{T} \cup \mathcal{A}$  is inconsistent then there is a subset  $\mathcal{A}_0 \subseteq \mathcal{A}$  with at most two assertions such that  $\mathcal{T} \cup \mathcal{A}_0$  is inconsistent.*

What is left to be shown is that there is a single choice if an inconsistency arises from two conflicting assertions. Observe that the case where already a single assertion  $\varphi$  causes inconsistency is not relevant because  $\varphi$  is inconsistent with the TBox and therefore precluded by Assumption 6.

Let  $\mathcal{A}_+, \mathcal{A}_-$  be the set of assertions that are added respectively deleted by a primal update  $U_p$  ( $U_p$  is defined according to Equation (4.12) resp. Equation (4.13) depending on whether OWA, CWA is used). Lemma 4.4 implies that if  $\mathcal{T} \cup (\mathcal{A} \setminus \mathcal{A}_-) \cup \mathcal{A}_+$  is inconsistent then there are two assertions  $\varphi, \psi$  such that  $\mathcal{T} \cup \{\varphi, \psi\}$  is inconsistent. The assertions  $\varphi, \psi$  can neither be both in  $(\mathcal{A} \setminus \mathcal{A}_-)$  nor both in  $\mathcal{A}_+$  since either  $(\mathcal{A} \setminus \mathcal{A}_-)$  or  $\mathcal{A}_+$  would be inconsistent then, which contradicts Assumption 6; observe that if  $(\mathcal{A} \setminus \mathcal{A}_-)$  is inconsistent so is  $\mathcal{A}$  due to monotonicity of satisfiability. Suppose  $\varphi \in \mathcal{A}_+$  and  $\psi \in (\mathcal{A} \setminus \mathcal{A}_-)$ . Since  $\varphi$  has to be added in order to accomplish the update, deletion of  $\psi$  is implied, which leads us to the summarizing theorem.

**Theorem 4.5** (Adapted version of [CKNZ10, Theorem 13]). *The concomitant update  $U_c$  for a set of (ES1)-effects  $E$  to be applied to a DL-Lite $_{\mathcal{FR}}^{++}$ -KB  $\mathcal{K}$  is uniquely defined.*

Using the algorithms listed in [Appendix A.1](#) we can compute  $U_c$ . They are extended versions of the corresponding algorithms in [CKNZ10] that take into account the additional features of  $DL-Lite_{\mathcal{FR}}^{++}$ .

There are two remarks concerning determinism and ramifications in the presence of a TBox in order. First, restricting effects such that only primitive concepts and roles are used precludes ramifications. This is easily seen by recalling that ramifications are introduced by concept/role inclusions. For instance, an effect  $A(x)$  and an inclusion  $A \sqsubseteq B$  in the TBox makes  $B(x)$  an implicit effect of  $A(x)$ ; analogous for roles. Since the criterion for being primitive is that a concept or role never occurs on the left-hand side of inclusions in a TBox, it becomes evident why primitive concepts/roles cannot cause ramifications. Second, under the further restriction that for every inclusion  $A \sqsubseteq C$  in the TBox such that  $A$  is used by an effect then (ES1) is guaranteed to be deterministic only if  $C$  is not defined using  $\exists R$ . As pointed out independently in [Mil08] and [CKNZ10], existential restriction can lead to unintuitive ramifications, attributed as a “higher degree’ of nondeterminism” in [Mil08].<sup>22</sup> [Example 4.2](#) illustrates this.

#### Example 4.2

Consider the following TBox, ABox, and effect:

$$\begin{aligned} \mathcal{T} &= \{Patient \sqsubseteq \exists treatedBy.Physician, RetiredPhysician \sqsubseteq \neg(\exists treadedBy^-.Patient)\} \\ \mathcal{A} &= \{Patient(ALICE), Physician(BOB), treatedBy(ALICE, BOB)\} \\ \varphi &= RetiredPhysician(BOB) \end{aligned}$$

In words, being a patient implies being treated by a physician, retired physicians do not have patients, Alice is a patient treated by the physician Bob, but Bob retires, which is represented by  $\varphi$ . One can see that besides switching Bob from a physician to a retired physician, the assertion that Alice is treated by Bob needs to be deleted in order to retain consistency; hence, the updated ABox is

$$\mathcal{A}' = \{Patient(ALICE), RetiredPhysician(BOB)\} .$$

$\mathcal{A}'$  entails that Alice is treated by some physician (which we do not know) since she is still a patient. Hence, the only option regarding Alice is that somehow she has immediately found another physician that treats her now. The option that she is (temporarily) no patient is not considered.

The problem with [Example 4.2](#) is that the update only makes explicit the new state of Bob but “forgot” to be explicit about Alice’s new state, or to be more general, the new state of all patients treated by Bob. In a way, the update is underspecified. One could resolve the indeterminism by enriching the update with the missing information, perhaps in an interactive way. This also gives raise to the idea of having an analogue to integrity constraint checking in databases. Having such a device, an update in which

<sup>22</sup>Nondeterminism is similarly introduced if  $C$  is defined using disjunction ( $\sqcup$ ) or universal restriction ( $\forall R$ ), which are however not present in  $DL-Lite_{\mathcal{FR}}^{++}$ .

Bob retires would be rejected by the system if this is not accompanied by stating what happens with all its patients – are they no longer patients versus treated by another physician.

### Related DL-based Approaches

Concluding this section, we would like to mention that formula-based effect semantics have also been considered in [Sir06, SMM10], which would constitute similar effect systems. While [SMM10] lacks details on how to choose a maximal  $\mathcal{K}_{\max}$  in case of a conflict between an effect and the current state in the ABox, [Sir06] considers the use of the *axiom pinpointing* reasoning service [SC03, Kal06] to find the ABox assertions that contradict with an effect.<sup>23</sup> However, it is unclear whether an update resulting from the application of a set of effects is uniquely defined in general. The approach first presented in [BML<sup>+</sup>05] and gradually advanced in [LLMW06, BLL10] essentially constitutes an effect system with model-based semantics. Especially the latest extension to general TBoxes combined with the representation of causal relationships appears to be an interesting alternative to investigate further.

### 4.2.3 Profile, Operation, and Service

We are now at the point where we can introduce formal definitions for profiles, operations, and services themselves, which is done in this order as they build upon each other. We tacitly assume that each of these definitions serve as the basis for declarative descriptions thereof.

#### Profile

Resuming [Section 4.1.5](#), a profile consists of the five types of properties and is formally defined as follows.

**Definition 4.6 (Profile).** *A profile is a 5-tuple  $Pr = (I, O, P, E, N)$  where*

- *$I$  is a finite set of input parameters such that  $\forall i_1, i_2 \in I: id(i_1) \neq id(i_2)$ ;*
- *$O$  is a finite set of output parameters such that  $\forall o_1, o_2 \in O: id(o_1) \neq id(o_2)$ ;*
- *$P$  is a finite set of preconditions;*
- *$E$  is a finite set of effects;*
- *$N$  is a finite set of non-functional parameters such that  $\forall n_1, n_2 \in N: id(n_1) \neq id(n_2)$ .*

We will write  $Pr.I$ ,  $Pr.O$ ,  $Pr.P$ ,  $Pr.E$ , and  $Pr.N$  to denote respective sets of a profile  $Pr$ . Note that each set  $I, O, P, E, N$  may possibly be empty; the profile where all sets are empty is correspondingly called the *empty profile*. The additional requirement on names

<sup>23</sup>In [Kal06] a black-box approach is described that relies only on a sound and complete reasoner for a particular DL, thus, is independent of the DL used. In addition, a glass-box approach tailored to a particular reasoner and to  $SHOIN(\mathbf{D})$  has also been described and implemented.

(identifiers) of input, output, and non-functional profile parameters ensures that they can be uniquely identified even if they are equivalent.

### Example 4.3

Recall the *find book* service and the *order & pay* service from Section 2.1. Suppose their profile is denoted with  $Pr_1$  and  $Pr_2$ , respectively. Furthermore, suppose there is an ontology (that has been loaded into a KB) consisting of the concepts *CreditCardNo*, *ISBN*, *BookInfo*, *Customer*, *Receipt*, and that there is a data range *int*. The input and output sets ( $I, O$ ) would then be specified as follows:

$$\begin{aligned} Pr_1.I &= \{\text{SEARCHPARAM:BookInfo}\}, \\ Pr_1.O &= \{\text{ISBN:ISBN}\}, \\ Pr_2.I &= \{\text{AMOUNT:int, CCNO:CreditCardNo, CUST:Customer, ISBN:ISBN}\}, \\ Pr_2.O &= \{\text{ACK:Receipt}\} . \end{aligned}$$

Consequently, all parameters are general except for AMOUNT, which is concrete. Suppose the parameters CCNO, ISBN, CUST, and AMOUNT have one representative where the former two shall be strings, the third an individual, and the latter an integer. If  $Pr_2$  is supposed to specify a precondition requiring the credit card to be valid and the book available from stock then it might be expressed by a conjunctive ABox query

$$Pr_2.P = \{\text{validity}(x, \text{Valid}) \wedge \text{inStock}(y, z)\}$$

where  $x$  links to  $Re[1](\text{CCNO})$ ,  $y$  to  $Re[1](\text{ISBN})$ ,  $z$  to  $Re[1](\text{AMOUNT})$ , and *validity*, *inStock* are roles in the domain ontology. Finally,  $Pr_2$  might specify an effect stating that the customer owns the book

$$Pr_2.E = \{\text{ownsBook}(u, y)\}$$

where  $u$  links to  $Re[1](\text{CUST})$  and *ownsBook* is yet another role in the domain ontology.

## Operation

Resuming Section 4.1.2, an operation is supplemented by a name and formally defined as follows.

**Definition 4.7** (Operation). *An operation is a 3-tuple  $Op = (id, Pr, Gr)$  where  $id$  is the name (or identifier) of the operation,  $Pr$  is a profile, and  $Gr$  is a grounding.*

Analogous to a profile parameter, the name of an operation is merely intended for identification purposes (cf. Section 4.2.1). The grounding  $Gr$  cannot be defined in more detail at the level of the system model since it is implementation-specific; hence, we abstract from technical details. Also, notice that the definition abstracts from whether an operation is one-way or request-reply; although it should be clear that a non-empty



set of outputs in the profile of an operation implies that it is request-reply. Protocol semantics of operations are addressed in the process model.

If we allow `nil` at the position of the grounding in [Definition 4.7](#) (i.e., where no grounding is provided) then we can also represent *abstract* operations. Operations are *implemented* otherwise.

[Definition 4.7](#) can be extended easily such that one can represent operations with multiple implementations (the motivation of which has been discussed in [Section 4.1.2](#)): instead of the single profile  $Pr$  and the grounding  $Gr$  one defines an operation as a pair  $Op = (id, PG)$  where  $PG$  is a non-empty set of pairs of the form  $(Pr, Gr)$  such that for each pair  $(Pr_1, Gr_1), (Pr_2, Gr_2) \in PG$ , the profiles  $Pr_1$  and  $Pr_2$  differ at most in their preconditions and non-functional parameters. The additional restriction on profiles is a consequence of the obvious requirement that all implementations of an operation are functionally equivalent (see [Section 4.1.3](#)). Since differences are possible for preconditions and non-functional properties, implementations may vary regarding required conditions for being operable and may come with differing quality characteristics.

We define the trivial while special-purpose *no-op* operation, denoted with `NOP`, that presents the empty profile and has no grounding:

$$\text{NOP} = (\text{"no-op"}, (\{\}, \{\}, \{\}, \{\}, \{\}), \text{nil})$$

According to what we have stated, `NOP` is abstract. This shall not worry us since an implementation would not do anything anyway. Although `NOP` rather lacks practical relevance, it is mentioned here because it is used later on as an ancillary tool.

## Service

Apart from the elements that we have already formally introduced, a service includes two more elements: a *control flow graph* and a *data flow graph*. Both will be introduced afterwards in [Section 4.3](#) since they represent the process model of a service, which we shall feature in its own section.

**Definition 4.8 (Service).** A service is a 5-tuple  $Sc = (id, Pr, \mathcal{U}, G_{cf}, G_{df})$  where  $id$  is the name (or identifier) of the service,  $Pr$  is a profile,  $\mathcal{U}$  is a finite, non-empty set of functional units that the service is composed of,  $G_{cf}$  is the control flow graph over elements in  $\mathcal{U}$ , and  $G_{df}$  is the data flow graph over elements in  $\mathcal{U}$ . Given a service  $Sc$ , an instance of  $Sc$  is denoted with  $\dot{Sc}$ .

A service is atomic if  $|\mathcal{U}| = 1$  and the single functional unit  $u \in \mathcal{U}$  is an operation. Otherwise, if  $|\mathcal{U}| > 1$  then it is composite. Finally, given two services  $Sc_{sub}$  and  $Sc$ ,  $Sc_{sub}$  is called a sub service of  $Sc$  if  $Sc_{sub} \in Sc.\mathcal{U}$ .

Because an atomic service consists of a single operation, their profiles are the same. Extending [Definition 4.8](#) such that a service can present multiple profiles (the motivation of which has been discussed in [Section 4.1.1](#) and [4.1.5](#)) is easily achieved by replacing  $Pr$  with a set of profiles, say  $\mathcal{P}$ , defined to be finite and non-empty. Doing so spawns an orthogonal dimension to the atomic versus composite dimension. If  $\mathcal{P}$  contains only one profile then we shall call the service *simple* and *complex* otherwise.

Analogous to the no-op operation but by a slight abuse of [Definition 4.8](#) we can also model the no-op service; that is, a service that presents the empty profile only, is composed of nothing, and has an empty control and data flow.

## 4.3 Process Model

The process model introduced in the following builds on the well-studied and established PETRI net formalism (PN) [Mur89], which is also well-known for its application to Workflow and Business Process modelling (e.g., [Aal98, SW01, AHW03, AS11]). It formalizes two key aspects. First, the dynamic *behavior* occurring in the course of an execution. Second, the *structure* especially of composite services. Applying the definitions given in [ALRL04] to services, the behavior is what is done when a service is executed to implement its functionality. The structure is what enables it to generate the behavior. Since the process model provides the means to make this structure and behavior explicit, one can say that a service is viewed as a white box. This completes the image of services that was started with the black box view in the service model.

In the process model, the structure is made up of two parts. First, a *control flow* that represents the routing along the invocations of operations and sub services a service is composed of (i.e., a precedence order, branching, and synchronisations). Second, a *data flow* that accompanies the control flow by representing the routing of data (e.g., an output that is consumed as an input by a subsequent operation). This section is correspondingly divided into two main subsections.

There are two important points to understand about the process model. First, the execution of a concrete service instance – its process instance – corresponds to an instance of a PN. Up to this point, different service instances are independent from each other, no matter whether of the same type of service or of different ones. In other words, up to the PN formalism used by the process model, the structure and behavior of single service instances is represented in an isolated manner and inter-process concurrency is not further considered. However, our process model combines execution semantics of the control flow with the preconditions and especially the effects that are created in the course of service execution in a KB and shows when updates are applied to the KB. Therefore, whether different service instances are isolated from each other depends on the assumptions made on the scope of the KB. If the KB is shared so that it (i) represents the state of affairs in an application domain, (ii) is used to check preconditions, and (iii) if effects from different and possibly concurrent executions are applied to it then interdependencies that might exist among service instances in the application domain are considered. This relationship is represented by distinguishing between a *local* and *global* execution state.

### 4.3.1 Control Flow

The invocation of services and operations in the form of request-reply or one-way interactions essentially constitutes a discrete process with discrete state changes over time. This makes the PN formalism an almost natural choice to describe such processes because their state-transition semantics fits well to that. Moreover, the PN formalism is general enough to (i) represent concurrency and to (ii) map control constructs existing in prominent process modeling frameworks onto PNs (e.g., BPEL [SS04, OVA<sup>+</sup>07, Loh08, LVOS09], OWL-S [NM02, BCI09]), thereby ascribing precise operational execution semantics to the control constructs existing in these languages.

Before we introduce the formal model of the control flow, we will introduce the basics of the PETRI net formalism subsequently required, which follows closely [Mur89, EKR95, Aal98]. Readers familiar with PNs and the notion of WorkFlow nets might skim through the following sub section.

### Petri Nets and WorkFlow Nets

In its basic form, a PETRI net (PN) is a bipartite directed graph  $G = (\mathbf{P}, \mathbf{T}, \mathbf{F})$ . Nodes are divided into the finite set of *place nodes*  $\mathbf{P}$  and the finite set of *transition nodes*  $\mathbf{T}$

$$\mathbf{P} = \{p_1, p_2, \dots, p_m\} \quad \text{and} \quad \mathbf{T} = \{t_1, t_2, \dots, t_n\} .$$

The set of edges  $\mathbf{F}$  for such graphs is called the *flow relation* and either connects a place node to a transition node or vice versa (but never two transitions or two places). Formally,

$$\mathbf{F} \subseteq (\mathbf{P} \times \mathbf{T}) \cup (\mathbf{T} \times \mathbf{P}) .$$

Observe that  $\mathbf{F}$  restricts every pair of nodes to be connected by at most one edge.  $\mathbf{F}$  is *acyclic* iff for each pair  $(x, y) \in \mathbf{F}$  then  $(y, x) \notin \mathbf{F}^*$  where  $\mathbf{F}^*$  is the transitive closure of  $\mathbf{F}$ .

A *path*  $W$  from a node  $x_1$  to a node  $x_k$  is a sequence  $W = \langle x_1, x_2, \dots, x_k \rangle$  such that  $(x_i, x_{i+1}) \in \mathbf{F}$  for  $1 \leq i < k$  (i.e.,  $(x_1, x_k) \in \mathbf{F}^*$ ). A node  $y$  is said to be on a path between node  $x$  and node  $z$  iff  $(x, y) \in \mathbf{F}^*$  and  $(y, z) \in \mathbf{F}^*$ . A path  $W$  is *elementary* [Aal98, Definition 6] iff all nodes on  $W$  are unique; that is, iff for any two nodes  $x_i, x_j$  on  $W$  and  $i \neq j$  implies  $x_i \neq x_j$ . Observe that an elementary path is acyclic. A PN is *strongly connected* iff there exists a path from every node to every other node in  $\mathbf{P} \cup \mathbf{T}$ .

If there is a directed edge from one node to another, the former is called the *input node* for the latter, while the latter is called the *output node* of the former. More generally, the *pre-set* and *post-set* of a place  $p \in \mathbf{P}$  is denoted with  $\bullet p$  and  $p \bullet$ , respectively. Analogously, the pre-set and post-set of a transition  $t \in \mathbf{T}$  is denoted with  $\bullet t$  and  $t \bullet$ , respectively. Formally, these sets are defined as follows:

$$\bullet p = \{t \mid t \in \mathbf{T} \text{ and } (t, p) \in \mathbf{F}\}, \quad p \bullet = \{t \mid t \in \mathbf{T} \text{ and } (p, t) \in \mathbf{F}\},$$

and

$$\bullet t = \{p \mid p \in \mathbf{P} \text{ and } (p, t) \in \mathbf{F}\}, \quad t \bullet = \{p \mid p \in \mathbf{P} \text{ and } (t, p) \in \mathbf{F}\} .$$

The set of *initial* and *final* places (a.k.a. source and sink places) is denoted with  $\mathbf{P}_i$  and  $\mathbf{P}_f$ , respectively, and defined as follows

$$\mathbf{P}_i = \{p \mid p \in \mathbf{P} \text{ and } \bullet p = \emptyset\} \quad \text{and} \quad \mathbf{P}_f = \{p \mid p \in \mathbf{P} \text{ and } p \bullet = \emptyset\} .$$

Inspired by [EKR95], we call the union  $\mathbf{P}_i \cup \mathbf{P}_f$  the *interface* of  $G$ .<sup>24</sup>

A *free-choice* PN is a PN where every arc is either the only incoming arc to a transition or is the only outgoing arc from a place, that is,

$$\forall p_1, p_2 \in \mathbf{P}: p_1 \bullet \cap p_2 \bullet \neq \emptyset \quad \text{implies} \quad |p_1 \bullet| = |p_2 \bullet| = 1 .$$

<sup>24</sup>This slightly differs from [EKR95, Definition 5.1] where the authors essentially consider PNs with exactly one initial and final place each (similar to WorkFlow nets) and define the interface as the union of these two places.

A *marking*  $M$  of a PN is an assignment of a number of *tokens* to a place node (i.e., the distribution of tokens over place nodes). Formally, it is described as the function

$$M: \mathbf{P} \rightarrow \mathbb{N}_0$$

and where  $M(p)$  denotes the number of tokens at place  $p \in \mathbf{P}$  and marking  $M$ . A marking can equally be understood as a multiset over the place nodes or as an element of the Cartesian product  $(\mathbb{N}_0)^{|\mathbf{P}|}$ . A *marked PN* is denoted with  $G = (\mathbf{P}, \mathbf{T}, \mathbf{F}, M_0)$  where  $M_0$  is the *initial marking*.

The algebraic structure of a PN  $G = (\mathbf{P}, \mathbf{T}, \mathbf{F})$  can be graphically represented. The common convention is to represent a place  $p \in \mathbf{P}$  by a circle, a transition  $t \in \mathbf{T}$  by a rectangle, a pair of nodes  $(x, y) \in \mathbf{F}$  by a directed arc pointing from  $x$  to  $y$ , and the number of tokens at a place  $p$  by an equal number of points drawn inside the circle representing  $p$  (see [Figure 4.5](#) for an example).

There are various extensions to the basic formalism such as token capacities for places, associating duration or delay with places/transitions/tokens, or *Coloured Petri nets* [Jen87] where tokens are distinguishable by associating them with a value. None of these extensions is required in the context of this thesis. Instead of extending the basic formalism, there are also well known restrictions such as the ones already seen (acyclic, free-choice) or *state machines* where transitions are restricted to have at most one incoming and outgoing edge, which essentially rules out concurrency. In the process model, we particularly consider a simplified form of so-called *WorkFlow nets* [vdA97, Aal98].<sup>25</sup> For this reason, we quote its definition here (notation and terminology slightly adapted).

**Definition 4.9** (WorkFlow Net [Aal98, Definition 6]). *A Petri net  $G = (\mathbf{P}, \mathbf{T}, \mathbf{F})$  is a WorkFlow net iff:*

- (1)  $G$  has two special places:  $p_i$  and  $p_f$ . Place  $p_i$  is the initial place:  $\bullet p_i = \emptyset$ . Place  $p_f$  is the final place:  $p_f \bullet = \emptyset$ .
- (2) If we add a transition  $t^*$  to  $G$  which connects place  $p_f$  with  $p_i$  (i.e.,  $\bullet t^* = \{p_f\}$  and  $t^* \bullet = \{p_i\}$ ), then the resulting PN is strongly connected.

The transition  $t^*$  can be seen as an ancillary tool used to short-circuit a PN.

We adopt another reasonable structural restriction from [Aal98] that ensures that a place with multiple output transitions (i.e., a split into multiple paths) is complemented by a place (rather than a transition) at which the previously spawned paths join eventually; analogous for transitions. In other words, this restriction precludes that two different paths spawned at a place join at a transition and vice versa. Let  $\circ$  be the unary operator that short-circuits a WorkFlow net  $G$  as defined by [Item \(2\)](#) in [Definition 4.9](#). Let  $\Sigma(W)$  be the alphabet of a path  $W$ ; that is, the set of unique nodes that occur in  $W$ .

**Definition 4.10** (Well-handled, well-structured [Aal98, Definition 9]). *A PN is well-handled iff for any pair of nodes  $x$  and  $y$  such that one of the nodes is a place and the other a transition and for any pair of elementary paths  $W_1$  and  $W_2$  leading from  $x$  to  $y$ ,  $\Sigma(W_1) \cap \Sigma(W_2) = \{x, y\}$  implies  $W_1 = W_2$ . A WorkFlow net  $G$  is well-structured iff  $\circ G$  is well-handled.*

<sup>25</sup>WorkFlow nets also have the concept of *triggers* (which are basically external conditions) and *work-flow attributes* (which are modeled using Coloured Petri nets), both of which we do not use.

Well-structuredness is necessary for proper realization of conditional routing and synchronization for parallel routing, which will become clear later when the execution semantics is introduced. Free-choiceness, in turn, inhibits improper mixing of parallel and conditional routing since they cannot occur both at the same time. What is more, these two properties are orthogonal since a PN may have neither, either, or both. However, the property of being free-choice and cyclic implies well-structuredness (i.e., if a PN is not well-structured while cyclic then it is not free-choice).

### Structure

The syntax of the control flow captures the static dimension of a process – its structure – and is represented by a *control flow graph*. The dynamic dimension of execution semantics over the control flow graph is introduced afterwards. In short, the control flow graph is a PN that satisfies the constraints of a free-choice and well-structured Workflow net. In addition, the control flow graph includes a mapping that assigns each transition to a single service or a single operation of which the service whose process is being described is composed of. Given a service  $Sc$ , its control flow graph is defined as follows.

**Definition 4.11** (Control Flow Graph). *A control flow graph (or control flow for short) for a service  $Sc$  is an enhanced, marked PN  $G_{cf} = (\mathbf{P}, \mathbf{T}, \mathbf{F}, M_0, fu)$  where*

- $\mathbf{P}, \mathbf{T}, \mathbf{F}$  are defined as for a Workflow net and  $\mathbf{F}$  additionally satisfies the properties of a free-choice and well-structured PN,
- $M_0$  is the initial marking such that  $M_0(p_i) = 1$  and  $M_0(p) = 0$  for every other place  $p \in \mathbf{P} \setminus \{p_i\}$ , and
- $fu: \mathbf{T} \rightarrow Sc.\mathcal{U} \cup \{\text{NOP}\}$  is a surjective mapping that assigns each transition  $t \in \mathbf{T}$  either to the no-op operation or an operation/sub service of which  $Sc$  is composed of.

A node  $x \in \mathbf{P} \cup \mathbf{T}$  is an ordinary node (or ordinary place, ordinary transition) iff  $|\bullet x| = |x \bullet| = 1$ ; it is a split node denoted with  $x_{\text{split}}$  iff  $|\bullet x_{\text{split}}| = 1$  and  $|x_{\text{split}} \bullet| > 1$ ; it is a join node denoted with  $x_{\text{join}}$  iff  $|\bullet x_{\text{join}}| > 1$  and  $|x_{\text{join}} \bullet| = 1$ .

Finally,  $fu(t) \in Sc.\mathcal{U}$  for  $t$  an ordinary transition and  $fu(t_{\text{split}}) = fu(t_{\text{join}}) = \text{NOP}$  for split and join transitions.

The initial and final place  $p_i, p_f$  correspond to instantiation and completing termination of execution, thereby framing the lifecycle of a service instance. Moreover, all places and transitions are on a path between  $p_i$  and  $p_f$ , which is ensured by the constraint that all transitions and places except  $p_i, p_f$  have non-empty pre-sets and post-sets. In other words, there are no dangling transitions nor places, which would actually not contribute to the behavior of the process.

It is easy to see that for  $G_{cf}$  a control flow graph, if  $G_{cf}$  contains ordinary transitions that map to a service then one can always *unfold*  $G_{cf}$  in linear time into a control flow graph  $G'_{cf}$  in which all ordinary transitions map to an operation and that preserves the overall structure. More precisely, let  $t$  be an ordinary transition in  $G_{cf}$  that is mapped to a

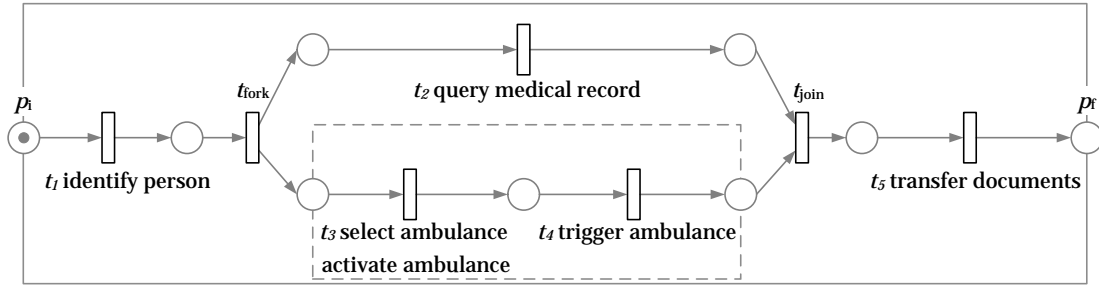


Figure 4.5: Unfolded control flow graph of the *emergency assistance service* (cf. [Figure 2.2](#)) with initial marking. The dashed rectangle frames the *activate ambulance* sub-service.

service  $Sc^t$ , let  $i$  be the input place of  $t$ ,  $o$  the output place of  $t$ ,  $G_{cf}^t$  the control flow graph of  $Sc^t$  that has the initial and final place  $p_i^t, p_f^t$ , respectively. In addition, we assume that the set of places (transitions) in  $Sc$  is mutually disjoint from the set of places (transitions) in  $Sc^t$ . Then, unfolding of  $t$  results in a control flow graph  $G'_{cf} = (\mathbf{P}', \mathbf{T}', \mathbf{F}', M'_0, fu')$  where

$$\begin{aligned}
 \mathbf{P}' &= (\mathbf{P} \cup \mathbf{P}^t) \setminus \{p_i^t, p_f^t\} \\
 \mathbf{T}' &= (\mathbf{T} \cup \mathbf{T}^t) \setminus \{t\} \\
 \mathbf{F}' &= (\mathbf{F} \cup \mathbf{F}^t \cup \{(i, x), (y, o) \mid x \in p_i^t \bullet, y \in \bullet p_f^t\} \\
 &\quad \setminus \{(i, t), (t, o), (p_i^t, x), (y, p_f^t) \mid x \in p_i^t \bullet, y \in \bullet p_f^t\}) \\
 M'_0 &= M_0 \\
 fu' &: \mathbf{T}' \rightarrow Sc.\mathcal{U} \cup Sc^t.\mathcal{U} \cup \{\text{NOP}\}
 \end{aligned} \tag{4.15}$$

such that  $fu'$  preserves the mappings of all ordinary transitions other than  $t$ ; that is,

$$fu'(u) = \begin{cases} fu(u) & \text{if } u \in \mathbf{T} \\ fu^t(u) & \text{if } u \in \mathbf{T}^t. \end{cases} \tag{4.16}$$

Unfolding is to be repeated iteratively until there are no more ordinary transitions that can be unfolded. The completely unfolded control flow graph is taken as the basis for execution because all ordinary transitions map to an operation invocation. [Figure 4.5](#) provides an example of an unfolded control flow graph depicting the process of the *emergency assistance service* from [Section 2.2](#).

It is also easy to see that the smallest valid control flow graph contains one transition  $\mathbf{T} = \{t\}$ , has the initial and final place only, and where  $\mathbf{F} = \{(p_i, t), (t, p_f)\}$ , which resembles the process of an atomic service.

Finally, we define the notion of a *sub control flow* or *subflow* for short as follows.

**Definition 4.12** (Sub Control Flow Graph). Let  $G_{cf} = (\mathbf{P}, \mathbf{T}, \mathbf{F}, M_0, fu)$  and  $G'_{cf} = (\mathbf{P}', \mathbf{T}', \mathbf{F}', M'_0, fu')$  be control flow graphs. We define the binary relation  $\trianglelefteq$  on control flow graphs by setting  $G'_{cf} \trianglelefteq G_{cf}$  iff  $\mathbf{P}' \subseteq \mathbf{P}$ ,  $\mathbf{T}' \subseteq \mathbf{T}$ ,  $\mathbf{F}' \subseteq \mathbf{F}$ ,  $fu' \subseteq fu$ , and the following holds  $\forall t \in \mathbf{T}: t \in \mathbf{T}'$  implies  $\bullet t \subseteq \mathbf{P}'$  and  $t \bullet \subseteq \mathbf{P}'$ .

If  $G'_{cf} \trianglelefteq G_{cf}$  then  $G'_{cf}$  is called a *sub control flow graph* (or *subflow for short*) of  $G_{cf}$ .

As an example, suppose  $G_{cf}$  is the control flow depicted in [Figure 4.5](#). Then the dashed rectangle frames a subflow  $G'_{cf} \trianglelefteq G_{cf}$  consisting of  $t_3, t_4$ , the three adjacent places, and the connecting arcs. Conversely, the path that begins with the input place of  $t_3$  and that ends in  $p_f$  is not a subflow because not all places of  $t_{join}$  are included.

### Execution Semantics

The execution semantics of the control flow graph builds on the standard PN state-transition semantics.<sup>26</sup> Subsequently we always consider unfolded control flow graphs. Intuitively, a transition corresponds to the invocation of the operation to which it is mapped. A transition is said to be *enabled* if and only if

1. there is a token in all its incoming places and
2. all preconditions of the associated operation are satisfied.

The first item should be clear: a marking  $M$  of a control flow is one part that determines whether a transition is enabled or not. The second item is actually relevant only for ordinary transitions because split and join transitions map to the no-op operation, which does not have preconditions per se. Since precondition checking is made against some KB  $\mathcal{K}$  and since  $\mathcal{K}$  is updated in the course of execution by effects of completed operation invocations, we combine  $M$  and  $\mathcal{K}$  into what we call the *execution state*, which is denoted with  $s$  and defined as the pair

$$s = (M, \mathcal{K}) . \quad (4.17)$$

The marking  $M$  represents the local *control state* and therefore scopes an execution state to a single service instance. As mentioned already,  $\mathcal{K}$  might have a broader scope beyond a single service instance if the *world state* that it represents spans multiple independent and concurrent service executions (and possibly other actors that query and update it). We define the global state  $\hat{s}$  over all executions in the system as the pair

$$\hat{s} = (\hat{M}, \mathcal{K}) \quad (4.18)$$

where  $\hat{M}$  is a finite set of markings. Notice that  $\mathcal{K}$  is understood as the same in  $s$  and  $\hat{s}$ . Details regarding advancement of execution states in the course of execution follows later after the execution semantics has been defined and explained.

An enabled transition can *fire*. The *firing rule* defines when it does fire. In our setting, firing depends on whether the invocation/execution of the operation to which a transition is mapped by  $fu$  succeeds or fails. For now it is not important what exactly the discriminating criterion for success versus failure is. This will be detailed in [Section 5.2.2](#). We denote the success and failure case with

$$exec(fu(t)) = succ \quad \text{and} \quad exec(fu(t)) = fail$$

for a transition  $t$ . Again, split transitions are special insofar as they always and instantly fire if they are enabled because  $exec(NOP) = succ$  per definition (i.e., the no-op operation never fails). The failure of an operation invocation/execution is the event that triggers a recovery procedure. Discussion of this topic is postponed to [Chapter 5](#).

<sup>26</sup>PN state-transition semantics is sometimes referred to as a *token game*, which is a more vivid analog for the flow of tokens through the net by moving tokens from places to other places.

Firing of a transition removes a token from all incoming places and adds a token in all outgoing places (i.e., the token flow), which is called the *transition rule*. Formally, execution semantics of a control flow graph is defined as follows.

**Definition 4.13** (Control Flow Graph Execution Semantics). *Let  $Sc$  be a service and  $G_{cf} = (\mathbf{P}, \mathbf{T}, \mathbf{F}, M_0, fu)$  its unfolded control flow graph. Let  $PS = (\mathcal{L}^{PS}, f_{chk})$  be a precondition system used by  $Sc$  and  $P$  a finite set of PS-preconditions. We overload the precondition checking function  $f_{chk}$  for a set of preconditions  $P$  as follows:*

$$f_{chk}(\mathcal{K}, P) = \begin{cases} \text{true} & \text{if } P = \emptyset \\ \bigwedge_{\varphi \in P} f_{chk}(\mathcal{K}, \varphi) & \text{if } |P| \geq 1 \end{cases} . \quad (4.19)$$

An execution state for an instance  $\dot{Sc}$  is a pair  $s = (M, \mathcal{K})$  where  $M$  is a marking for  $G_{cf}$  and  $\mathcal{K}$  is the corresponding knowledge base. A transition  $t \in \mathbf{T}$  is enabled in  $s$  iff

- (1)  $\forall p \in \mathbf{P}: p \in \bullet t$  implies  $M(p) \geq 1$  and
- (2)  $f_{chk}(\mathcal{K}, P) = \text{true}$

where  $P = fu(t).Pr.P$  (i.e., the set of preconditions in the profile  $Pr$  of the operation associated with  $t$ ). Besides, we say that  $t$  is token-enabled in  $s$  if *Item (1)* is satisfied (thereby disregarding *Item (2)*).

A transition  $t \in \mathbf{T}$  fires only if it is enabled and  $exec(fu(t)) = \text{succ}$ .

Given a marking  $M$  of  $G_{cf}$  and a transition  $t \in \mathbf{T}$ ,  $M'$  is the new marking resulting from firing of  $t$  if the following holds

$$\forall p \in \mathbf{P}: M'(p) = \begin{cases} M(p) - 1 & \text{if } p \in \bullet t \text{ and } p \notin t\bullet, \\ M(p) + 1 & \text{if } p \notin \bullet t \text{ and } p \in t\bullet, \\ M(p) & \text{otherwise.} \end{cases} \quad (4.20)$$

We write  $M \xrightarrow{t} M'$  to denote the transition from marking  $M$  to  $M'$  by firing of  $t$ . We write  $M_1 \xrightarrow{e} M_n$  to denote a firing sequence  $e = \langle t_1, \dots, t_n \rangle$  with  $M_i \xrightarrow{t_i} M_{i+1}$  for  $1 \leq i < n$ , leading from marking  $M_1$  to  $M_n$ . In this case we say that execution from  $M_1$  has reached  $M_n$ . Execution of the control flow graph completes only if, starting from the initial marking  $M_0$ , execution has reached the final marking  $M_f$  such that  $M_f(p_f) = 1$  and  $M_f(p) = 0$  for any other place  $p \in \mathbf{P} \setminus \{p_f\}$ .

Finally, we say that  $M_n$  is token-reachable from  $M_1$  if there is a firing sequence leading from  $M_1$  to  $M_n$  so that transitions are enabled and fire by disregarding precondition checking and execution of associated operations (i.e., a dry-run); token-prefixed terms are understood analogously (e.g., token-firing sequence).

In essence, a split place  $p_{\text{split}}$  models a choice between its output transitions  $p_{\text{split}}\bullet$ . Each output transition  $t \in p_{\text{split}}\bullet$  is an exclusive alternative, which is due to the fact that there can be at most one token in any place  $p \in \mathbf{P}$  of a control flow graph. If  $t$  fires



then it consumes the only token in  $p_{\text{split}}$ ; hence, all other transitions  $p_{\text{split}} \bullet \setminus \{t\}$  are no longer token-enabled. As a consequence, if there is more than one output transition for the initial place  $p_i$  (i.e.,  $|p_i \bullet| > 1$ ) then the first transition  $t \in p_i \bullet$  that fires consumes the only token of the initial marking  $M_0$  in  $p_i$ . In other words, in a control flow graph there is always exactly one output transition (which can be either an ordinary or a split transition) of the initial place  $p_i$  that fires. This will become important later in [Chapter 5](#) when discussing structural modifications of control flow graphs.

Note here that [Definition 4.13](#) abstracts from conditions determining one of the output transitions of a split place; hence, a nondeterministic choice is modeled. More precisely, not making conditions explicit in the process results in internal nondeterminism as opposed to external nondeterminism (see [Footnote 3](#) on [Page 44](#)). Indeed, if we want to ensure execution in a practical system to be uniquely determined then we need to avoid internal nondeterminism; hence, all choices must be explicit rather than abstract. As the way such choices are represented is not relevant subsequently, we have moved details to [Appendix A.2](#). In short, one way of extending [Definition 4.13](#) to represent deterministic processes in which choices at split places are made based on service-specific conditions is to (i) assign edges with a condition and (ii) extend the transition-enabling rule such that at most one output transition of a split place is enabled. Summing up, if we speak of a *deterministic process*, we refer to the case of no internal nondeterminism (while there can be external nondeterminism).

A split transition  $t_{\text{split}}$  spawns independent concurrent paths because all output places  $t_{\text{split}} \bullet$  receive a token if  $t_{\text{split}}$  fires. Conversely, a join transition  $t_{\text{join}}$  represents synchronization among concurrent paths because it becomes token-enabled only if all its input places  $\bullet t_{\text{join}}$  have a token. However, a join place  $p_{\text{join}}$  represents no synchronization because it receives a token as soon as one of its input transitions  $\bullet p_{\text{join}}$  fires. All its output transitions  $p_{\text{join}} \bullet$  become token-enabled as soon as a token arrives in  $p_{\text{join}}$ . This explains why the property of being well-structured is important for proper realization of synchronization. If two concurrent paths spawned by a split transition were joined by a join place then there is no synchronisation taking place among the paths at the join place: control proceeds as soon as control reaches the place on either path. In turn, if a choice among alternative paths spawned at a split place were joined by a join transition then control gets stuck at the transition because it never becomes token-enabled (i.e., synchronization is modeled where nothing can be synchronized since concurrency does not exist).

It follows from the syntactic restrictions that the *maximum degree of parallelism* within a control flow, which we denote with  $Q$ , corresponds to the number of unique start-to-end paths:

$$Q = |\{W \mid W = \langle p_i, \dots, p_f \rangle \text{ and } W \text{ is elementary}\}| \quad (4.21)$$

In summary, the structure defined for control flow graphs together with its execution semantics provides the possibility to represent processes that may include the most important *control flow patterns*, namely:

- *Sequential*. Subflows are executed one after the other (strict precedence order).
- *Conditional*. A choice is made for either of two or more subflows (xor).

- *Parallel*. Subflows are executed concurrently (partial precedence order).
- *Iteration*. Repeated execution of a subflow (do while).

It should be clear that these constructs might be nested (e.g., a sequence executed repeatedly).

In addition, the structural restrictions on control flow graphs ensure that they are *sound*. Soundness [vdA97] is a highly desired correctness property regarding the dynamic dimension of a process. More specifically, soundness of a Workflow net  $G$  is the property that its short-circuited net  $\circlearrowleft G$  is *live* and *bounded* [vdA97, Theorem 11]. Liveness is essentially the absence of deadlocks, meaning that for any marking that has been token-reached from the initial marking  $M_0$  there is a token-firing sequence that can token-fire any transition of the net. Boundedness in the context of Workflow nets refers to *proper termination*, that is, starting from the initial marking  $M_0$  it is always possible to token-reach the final marking  $M_f$ . Soundness of control flow graphs follows from [vdA97, Corollary 19]. In short, this corollary states that there exist four soundness-preserving expansion<sup>27</sup> rules:

- *Sequential expansion*: Replace a transition by two consecutive transitions.
- *Conditional expansion*: Replace a transition by two conditional transitions.
- *Parallel expansion*: Replace a transition by two parallel transitions.
- *Iteration expansion*: Replace a transition by an iteration of a transition.

Applying a sequence of these expansions, one can create any control flow graph starting from the smallest control flow graph with one transition only.<sup>28</sup> Consequently, soundness is a monotonic property for control flow graphs, which particularly implies that any subflow of a control flow graph is also sound.

To conclude this subsection on execution semantics, **Definition 4.13** describes when and how control in a process proceeds. What is not yet described is when and how updates to the representation of current state of affairs in a KB are made (i.e., when and how the execution state is advanced). This is addressed next.

### Advancement of Execution State

The firing of a transition  $t$  results in a new execution state, both in terms of an updated marking  $M$  (control state) and an updated KB  $\mathcal{K}$  (world state) resulting from the effects created by the operation invoked. Since these two parts are ultimately tight together representation of state-transition semantics of the discrete process model needs to include this tight relationship in a proper way. The advancement of the execution state in the course of execution is captured by the following definition.

<sup>27</sup>There are also four complementary and soundness-preserving reduction rules.

<sup>28</sup>Conversely, one can reduce any control flow graph to the smallest control flow graph by successively applying one of the complementary reduction rules.

**Definition 4.14** (Advancement of Execution State). Let  $Sc$  be a service and  $G_{cf} = (\mathbf{P}, \mathbf{T}, \mathbf{F}, M_0, fu)$  its unfolded control flow graph. Let  $ES = (\mathcal{L}^{ES}, f_{up})$  be the effect system used by  $Sc$  and let  $s = (M, \mathcal{K})$  be an execution state for an instance  $\dot{Sc}$ . The initial execution state is denoted with  $s_0 = (M_0, \mathcal{K}_0)$  where  $M_0$  is the initial marking of  $G_{cf}$  and  $\mathcal{K}_0$  is an initial knowledge base.

Given an execution state  $s = (M, \mathcal{K})$  of  $\dot{Sc}$  and a transition  $t \in \mathbf{T}$ ,  $s' = (M', \mathcal{K}')$  is the new execution state resulting from firing of  $t$  iff the following holds

- (1)  $M \xrightarrow{t} M'$  and
- (2)  $\mathcal{K} \Longrightarrow_U \mathcal{K}'$  where  $U = f_{up}(\mathcal{K}, fu(t).E)$ .

We write  $s \xrightarrow{t} s'$  to denote the transition from execution state  $s$  to  $s'$  by firing of  $t$ .

There are no further assumptions made on the initial KB  $\mathcal{K}_0$  except that it is consistent, which is a requirement for precondition checking and applicability of effects (see [Section 4.2.2](#)). In practice,  $\mathcal{K}_0$  would be populated with the relevant domain knowledge and an initial representation of the state of affairs in the domain of application.

Observe that [Definition 4.14](#) is a simplified description of an isolated execution of processes which does not take concurrency into account (i.e., it abstracts from interleaving state changes).

The global state  $\hat{s}$  of executions evolves as follows. First, given  $\hat{s} = (\hat{M}, \mathcal{K})$ , instantiation and termination of a service instance  $\dot{Sc}$  yields a new global state  $\hat{s}' = (\hat{M}', \mathcal{K})$  such that

$$\hat{M}' = \hat{M} \cup M_0 \text{ (instantiation)} \quad \text{resp.} \quad \hat{M}' = \hat{M} \setminus M_f \text{ (termination)}$$

and where  $M_0, M_f$  are the initial and final marking of  $\dot{Sc}$ . A local transition  $s \xrightarrow{t} s'$  propagates in the obvious way to the global level by a transition  $\hat{s} \xrightarrow{t} \hat{s}'$  such that  $\hat{M}'$  contains the updated local marking  $M'$  and  $\mathcal{K}'$  is the updated KB.

Since execution of an operation  $Op$  associated to a transition  $t$  ( $Op = fu(t)$ ) is considered indivisible, the application of the update to the KB is considered to be done in an atomic way; hence,  $Op$ 's effects become visible all at once upon completion. The correct handling of concurrent updates on the KB by applying a transactional model and serializability theory will be addressed in [Chapter 6](#).

Finally, we mention that in a practical system the marking part of the execution state is usually not materialized in this form but rather implicit in the program state of an execution engine that manages the execution of a service instance.

### 4.3.2 Data Flow

From a structural point of view, the data flow is seen as a directed overlay graph on top of the control flow graph. Although extensions of the PN formalism can in principle be used to also represent the data flow (e.g., coloured PNs), we decided to keep it separated from the control flow graph in order not to clutter it and unnecessarily complicate matters.

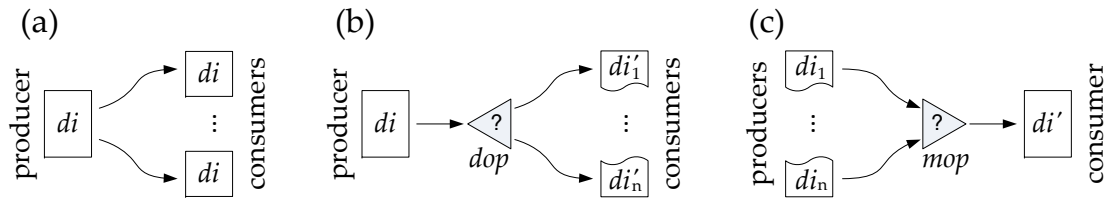


Figure 4.6: Data flow primitives: (a) fork; (b) divide where *dop* is a divide operator; (c) merge where *mop* is a merge operator.

Most of all, the data flow respects the directedness of the control flow graph: it cannot be counter-directional to the control flow; hence, it is not independent of the control flow. At the same time, the seamless compatibility of data producers with their consumers is a crucial requirement for automated executability. This becomes relevant when considering environments with syntactic, structural, and/or semantic data heterogeneities (see [She98] for this classification of different levels of heterogeneities). Therefore, in order to completely capture aspects of the data flow model considered in this thesis we find it necessary to address (i) its structure and execution semantics, (ii) to describe how it is linked to the control flow, and (iii) to provide and discuss a notion of data compatibility between the producer of some data item and its consumer(s).

Whereas the primitives in the flow of control are *split* and *join*, the primitives in the flow of data are *fork*, *divide*, and *merge*, which is depicted in Figure 4.6. A fork is essentially the use of a data item by multiple consumers (i.e., the use of the same data item multiple times). A divide as well as a merge of data ultimately requires the definition of a *divide operator* or a *merge operator*. A divide operator takes one incoming data item and divides it according to some instructions into multiple outgoing data items. Conversely, a merge operator takes two or more incoming data items and merges them according to some instructions to one outgoing data item. Depending on the input, typical operators involve selection, projection, join, and union. In the data flow model that we consider, divide and merge operators are not represented as first-class citizens. This is based on the observation that one can equally represent them by an operation within a service or a service itself (i.e., they can be realized in either way). Only a fork is an explicit part of the model.

## Sources and Sinks

Following common terminology, the producer of a data item is called the *source* and the consumer the *sink*. The data that “flows” from a source *connected* to a sink is a single data item (cf. Assumption 2). Sources and sinks map to input and output profile parameters. More precisely, we define the set of sources as the union of all outputs of sub services/operations a service is composed of plus the inputs of the service itself. Conversely, the set of sinks is the union of all inputs of sub services/operations a service is composed of plus the outputs of the service itself.

**Definition 4.15** (Source, Sink). *Let  $Sc$  be a service. The set of sources  $\mathbf{O}$  and sinks  $\mathbf{I}$  of  $Sc$  is defined as follows:*

$$\mathbf{O} = \left( \bigcup_{u \in Sc.U} u.Pr.O \right) \cup Sc.Pr.I \quad \text{and} \quad \mathbf{I} = \left( \bigcup_{u \in Sc.U} u.Pr.I \right) \cup Sc.Pr.O .$$

A source  $o \in Sc.Pr.I$  is called an *initial source*. A sink  $i \in Sc.Pr.O$  is called a *final sink*.

### Data Compatibility

In our data flow model, a connection between a sink and a source implies that both are *compatible*. The notion of data compatibility exists in one form or another in basically all service composition, workflow, or business process models/frameworks since it is imperative to define a data flow at all. Simply put, compatibility is understood as the possibility to forward the data item produced by the source to the sink so that it will be accepted and understood by the sink. Therefore, compatibility is a relation that has a syntactic (accept) and a semantic (understand) dimension. Compatibility at the syntactic level is a prerequisite for compatibility at the semantic level: While it is possible that a sink accepts a data item forwarded from a source but does not understand it, the opposite is impossible, intrinsically.

One can define the notion of data compatibility between sources and sinks in a data flow basically in two ways. Based on the methods employed, we classify them as *type-based* and the more general form of *mediator-based* compatibility; the latter building upon so-called *data mediators*. Data mediators are also not represented as first class citizens for the same reason than divide and merge operators: they can also be realized either as an operation within a service or an atomic service.<sup>29</sup>

*Type-based* compatibility is the direct form requiring that a source and a sink match semantically, at least. Using the machinery introduced in the service model this can be formulated as follows. A source profile parameter  $o$  is *type-compatible* with a sink profile parameter  $i$  if

$$type(o) \sqsubseteq type(i) . \quad (4.22)$$

This can be understood as a classical *plug-in match* [SWKL02, PKPS02] (see also [Section 5.4.1](#)). Observe that due to the unidirectional nature of the data flow a symmetric relation is not needed. Hence, it is not necessary to use the stricter *equivalence*  $type(o) \equiv type(i)$ . Conversely, the more permissive  $type(o) \sqsupseteq type(i)$ , which can be understood as a *subsume match*, turns out to be problematic under the assumptions described next.

**Condition (4.22)** is, however, not sufficient to ensure seamless compatibility. Without additional statements, it does not yet address compatibility at data level; that is, it lacks details on syntactical and structural data format requirements. One possibility that is (tacitly) made in most service frameworks/models is that type-compatibility implies syntactic and structural data-compatibility. Formulated in terms of our service

<sup>29</sup>A data mediator can actually be seen as a special form of a merge operator that has one input only.

model it is the assumption that the set of valid data values for a profile parameter coincides with the extension of the concept/data range so that [Equation \(4.4\)](#) and [Equation \(4.5\)](#), respectively, hold. Applied to  $o, i$ , this is achieved by requiring that  $o$  and  $i$  either use the same datatype<sup>30</sup> or that the datatype of  $o$  is derived from (is a restriction of) the datatype of  $i$ , which resembles an exact or plug-in match, respectively, at the level of data.<sup>31</sup> Consequently,  $o$  and  $i$  are seamlessly compatible under this assumption, both syntactically and semantically. Depending on the actual datatype system used in practice, this would be applicable equally for primitive as well as complex structured datatypes. For instance, the XML Schema type system includes the possibility to define a complex datatype  $d_1$  as the restriction over elements of another complex datatype  $d_2$ , which effectively makes the value space of  $d_1$  a subset of the value space of  $d_2$ . This also explains why extending the subsume match  $type(o) \sqsubseteq type(i)$  to data compatibility is problematic since there might be data items that are rejected by the sink because they are out of its value space.

At the technical execution level it is therefore indispensable that a source and a sink are compatible at the conceptual level as well as at the syntactic data level.

**Definition 4.16** (Source, Sink Execution Compatibility). *Let  $Sc$  be a service and  $\mathbf{O}, \mathbf{I}$  the set of its sources and sinks, respectively. A source  $o \in \mathbf{O}$  is execution compatible with a sink  $i \in \mathbf{I}$  iff [Condition \(4.22\)](#) holds and the data values produced by  $i$  are included in data values accepted by  $o$ .*

The notion of compatibility between sources and sinks becomes a more complex problem under structural and even more so under semantic data heterogeneities. This is the point where some form of data mediation is ultimately required. Though it is not the focus of this thesis to also cover this topic, we will briefly discuss the mediator-based approach next.

*Mediator-based* compatibility is more complex since it starts from the advanced cases where the data items produced by a source are not structural compatible with the accepted data items of a sink and/or where source and sink do not semantically match relative to some application domain conceptualization so that [Condition \(4.22\)](#) does not directly hold. Establishing compatibility under these circumstances involves solving a data integration problem (see [Len02] for an overview).

Probably the most prominent building block that has been proposed to this in the literature considers the use of mediators to achieve this [Wie92]. The basic principle has later been integrated as a core element in the WSMO service framework under the notion of *data level mediation* [SCMF06]. Data integration is one of the major and widely studied topics in databases and information systems with a large body of work on ontology based approaches [WVV<sup>+</sup>01, Noy04]. While the mediator-based approach is more of an architectural pattern that can be employed in principle to solve any syntactic, structural, and semantic data incompatibilities, it does not provide concrete methods

<sup>30</sup>Recap, information about the data format, which includes the datatype, is assumed to be included in the grounding of an operation (see [Section 4.1.3](#)).

<sup>31</sup>This type of data-compatibility corresponds to the notion of direct and indirect data type compatibility in [MBE03].

to achieve this. In fact, automated data integration (based on mediators) is still a challenging and not generally solved problem. For instance, a more recent review of the topic [BH08] states that “every step of the information-integration process requires a good deal of manual intervention”. Common approaches followed currently build on the idea of data schema mappings and/or structural transformation procedures. These are usually human-defined in an ad hoc manner for the data formats between which to mediate. Consequently, there is a certain degree of human involvement. An approach to estimate the effort in human involvement has been proposed in [GRR<sup>+</sup>08]. The authors define *mediatability* as a computable measure quantifying the effort of mediating between XML-based data schemata in terms of a similarity function.

Virtually all mediation-based approaches with humans in the loop have it that the data formats (schemas) among which to mediate are known in advance with only a few different formats involved. However, in open and possibly large-scale environments in which this cannot be assumed, different approaches are needed [Rah11].

### Structure

The structure of the data flow is defined using a consumer-pull style relation on sources and sinks (rather than producer-push style). In addition, we need a precedence relation on sources and sinks to represent that the data flow is not counter-directional to the control flow. Let  $Sc$  be a service and let  $G_{cf} = (\mathbf{P}, \mathbf{T}, \mathbf{F}, M_0, fu)$  be its control flow graph. Let  $pt: (\mathbf{O} \cup \mathbf{I}) \rightarrow (\mathbf{P} \cup \mathbf{T})$  be the mapping that returns for each source/sink of  $Sc$  its corresponding place/transition in the control flow graph; that is

$$pt(x) = \begin{cases} p_i & \text{if } x \in Sc.Pr.I \\ p_f & \text{if } x \in Sc.Pr.O \\ t & \text{if } x \in fu(t).I \text{ or } x \in fu(t).O . \end{cases} \quad (4.23)$$

A source  $o \in \mathbf{O}$  precedes a sink  $i \in \mathbf{I}$  w.r.t.  $G_{cf}$ , denoted with

$$o \prec_{G_{cf}} i , \quad (4.24)$$

iff there is a path  $W$  in  $G_{cf}$  such that  $pt(o)$  is the first and  $pt(i)$  is the last element in  $W$ .

**Definition 4.17** (Data Flow Graph). *Let  $Sc$  be a service and  $G_{cf}$  its control flow. A data flow graph (or data flow for short) for  $Sc$  is bipartite directed graph  $G_{df} = (\mathbf{O}, \mathbf{I}, \leftarrow--)$  where nodes are divided into the set of sources  $\mathbf{O}$  and sinks  $\mathbf{I}$  of  $Sc$  and  $\leftarrow-- \subseteq \mathbf{I} \times \mathbf{O}$  is the flow relation (edges) such that the following conditions hold:*

- (1) for each  $i \in \mathbf{I}$  there is a pair  $(i, o) \in \leftarrow--$  such that  $o$  must not be an initial source if  $i$  is a final sink,
- (2)  $(i, o) \in \leftarrow--$  implies that  $i, o$  are execution compatible, and
- (3)  $(i, o) \in \leftarrow--$  implies  $o \prec_{G_{cf}} i$ .

If  $(i, o) \in \leftarrow--$  then  $i$  is said to be connected to  $o$ . A source  $o \in \mathbf{O}$  is unused (or unconnected) if there is no pair  $(x, o) \in \leftarrow--$ .

Observe that **Item (1)** in **Definition 4.17** rules out open-sourced sinks; that is, all sinks are connected to a source. On the other hand, unused sources are permitted (i.e., an output data item produced by some operation need not necessarily be consumed), though this would rather rarely be the case in practice. In other words,  $\leftarrow$  is left-total and right-unique (i.e., it is a function) but neither injective nor surjective: different sinks may map to the same source (fork) and not all sources might be covered by  $\leftarrow$  (unused/unconnected source). Defining the flow relation in the dual form  $\rightarrow \subseteq \mathbf{O} \times \mathbf{I}$  renders the relation  $\rightarrow$  inverse-functional, which is the reason why we opted for the consumer-pull style.

As a last remark, observe that **Definition 4.17** precludes connecting an initial source with a final sink (**Item (1)**), thereby disallowing a data flow that simply and directly forwards a service's input to one or more of its outputs. The importance of this "restriction" lies in reasoning. Later in **Section 5.4.4** we will assume that every output is represented by a newly introduced constant. An output produced this way by looping it through from an input would actually imply being represented by an existing constant; hence, violate this assumption. From a purely data perspective, however, we could allow such a loop through data flow, though it is of little practical relevance anyway.

## Execution Semantics

The forwarding of data items in the course of execution is directly bound to the service lifecycle and transition firing as defined by the execution semantics of the control flow graph. More precisely, for each pair  $(i, o) \in \leftarrow$ , the data item becomes available at an initial source ( $pt(o) = p_i$ ) when the service gets instantiated (i.e., for the initial marking  $M_0$ ). If  $pt(o) = t$  then the data item is produced when  $t$  fires. The data item becomes available at a final sink ( $pt(i) = p_f$ ) when the service execution completes (i.e., for the final marking  $M_f$ ). If  $pt(i) = t$  then the data item is consumed when the operation or service  $fu(t)$  is invoked.

Since there are time gaps between production and consumption of data items in practice, it is the responsibility of an execution engine to keep a data item ready by temporarily storing it unless it was consumed by all sinks. This also includes cases of runtime invocation failures that make it necessary to keep data items for recovery purposes. Conversely, a data item that is never consumed (unconnected source) can in principle be discarded immediately once it was produced, except that it needs to be retained for other purposes (e.g., monitoring, recovery, rollback).

### 4.3.3 Well-formed Processes

As we have seen, both the data flow and the control flow require certain structural properties to be satisfied in order to be considered well-formed. In essence, these properties ensure that a process specification is correct in the sense that it can be executed in an automated way (assuming, of course, that all operations are implemented and necessary technical grounding details are available). Executability defined in toto in our process model includes two elements:



- *Soundness* – as defined in [vdA97] and embodied by the structural restriction to well-structured WorkFlow nets in [Definition 4.11](#). Soundness furthermore implies token-reachability of every transition.
- *Data compatibility* – as embodied by type-based datatype compatibility between connected sources and sinks.

We call a process specification that satisfies these properties *well-formed*. Verifying whether a process specification is well-formed is considered to be part either of the specification/design process or to be made at runtime as an initial step of the activation procedure (i.e., the instantiation) by a validation component part of the execution system.

There are other verifiable properties known in the literature that can extend the notion of a well-formed processes. One of them is *executability* [Rei01, BLM<sup>+</sup>05] which originates from action theory. Executability is the problem of determining whether an action or a sequence thereof is applicable in a particular state (i.e., whether preconditions are satisfied). Executability checking is in fact also done in our process model, although stepwise for single operations; embodied by [Item \(2\)](#) of the transition enabling rule in [Definition 4.13](#). Two more correctness properties are the absence of precondition and effect conflicts, which has been investigated in the context of semantic business process modeling in [WHM10].<sup>32</sup> Applied to our model, a *precondition conflict* exists between pairs of possibly concurrent operations  $Op_1, Op_2$  in a process if the effects of  $Op_1$  are inconsistent with the precondition of  $Op_2$  (i.e., execution of  $Op_1$  prior to  $Op_2$  cancels executability of  $Op_2$ ). An *effect conflict* exists between  $Op_1, Op_2$  if their effects are inconsistent w.r.t. domain constraints in a TBox.

## 4.4 Summary

While the overall approach in the system model follows existing Semantic Service frameworks in a number of respects, we have introduced several generalizations, namely the notion of representants and the precondition and effect systems. Moreover, we were able to seamlessly combine the different aspects of services: their execution, non-functional, change, and data semantics. The latter especially makes the data format aspect of services explicit by providing a characterization of the notion of data compatibility. Finally, we have taken great care to formulate it in a general way so as to separate practical choices from the conceptual basis.

The main components and layers that are combined in the system model are summarized in [Figure 4.7](#). Apart from the interpretation, all layers (vertical axis) are parametrized along one or more dimensions, which constitutes various degrees of freedom on how the model can be instantiated. On the other hand, we made simplifications at two places that account for the fact that we are concerned with the service execution task. These are the reduction to the single process to be executed (though a service might specify multiple different processes) and the one profile/grounding on each operation

<sup>32</sup>These two properties are known more generally in AI planning as action *dependencies* [GNT04].

to be invoked in the course of process execution. In this regard, the system model might be seen as tailored for the service execution task.

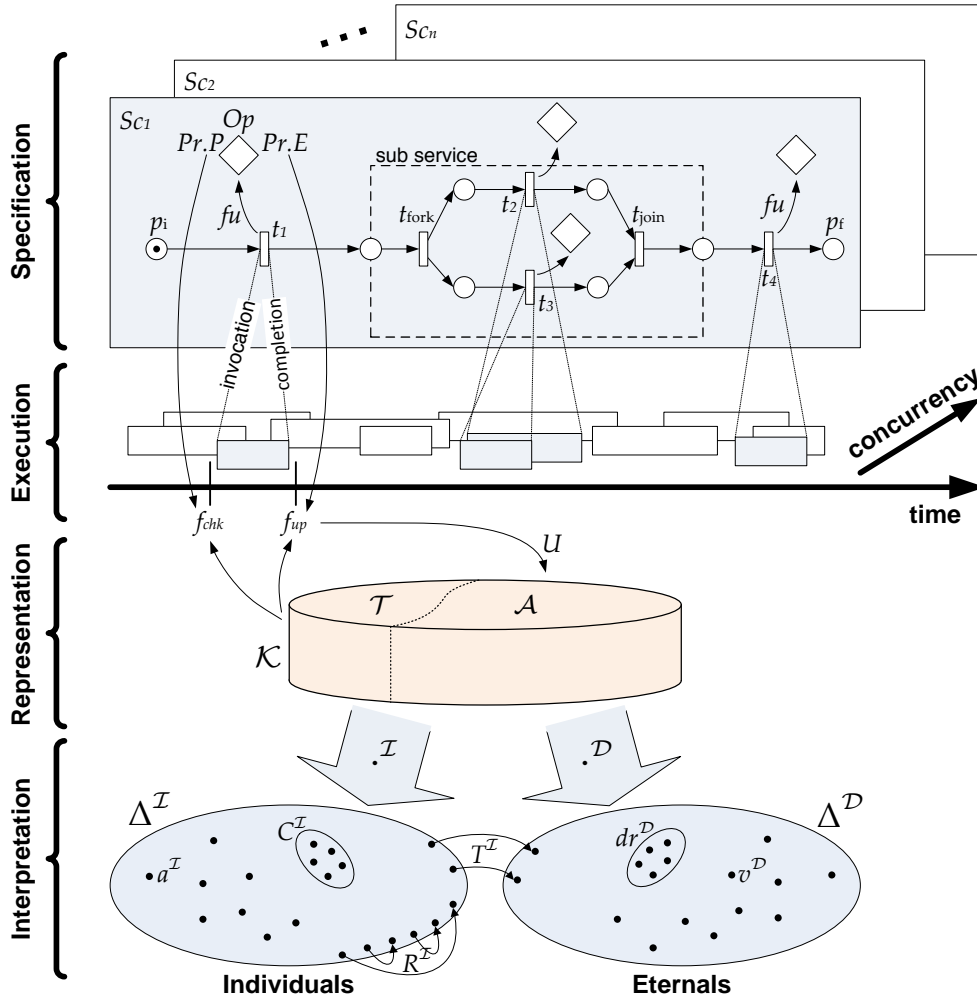


Figure 4.7: Summary of the system model depicting its main layers.

The representation and specification layer are parametrized along DLs. Leaving aside decidability, computational complexity, and determinism properties of reasoning especially over preconditions and effects, the model can be instantiated in principle with any DL that provides the basic subsumption inference. Yet it is precisely these aspects that limit the possibilities in theory as well as in practice. For example, a DL-based effect system that allows for disjunctive, existential, or universal restricted concepts as effects renders the change semantics of services/operations to be no longer deterministic in general (this would equally be introduced in the presence of TBoxes and the use of atomic concepts as effects that are defined possibly indirectly in terms of these three constructors). Furthermore, while for all “classical” DLs consistency is a prerequisite<sup>33</sup> for the subsumption inference, it is widely recognized that inconsistent

<sup>33</sup>Or, as [PT09] put it more succinctly: “the contemporary logical orthodoxy has it that, from contradictory premises, anything can be inferred”.

knowledge may naturally appear in many domains. Not surprisingly, reasoning under inconsistency recently gained considerable momentum in Semantic Web research (e.g., [MH09, FH10, NS10, ZLW10]). The system model is in principle amenable for such a paradigm shift; its potential and ramifications have not been explored, however, in this thesis. This applies equally to other reasoning paradigms such as statistical, probabilistic, inductive, or abductive reasoning.

Related to the choice of the actual DL is the choice for the OWA versus CWA. Also, whether the UNA is adopted or not. These are two more parameters of the model. In controlled and closed application environments where one can assume that all relevant information is available, adopting the CWA is fully satisfactory since negative information (e.g.,  $\neg\text{allergicTo}(\text{ALICE}, \text{PENICILLIN})$ ) does not need to be maintained then. The picture is different, however, in open environments such as the Web. Ultimately, the answer to the question whether OWA or CWA should be used in an open environment depends on what conclusions that are drawn from reasoning are used for. For instance, if a conclusion is the basis of a risky decision then the OWA (combined with the prudence principle) is supposedly a better choice. Consequently, adoption of OWA versus CWA should be decided on a case by case basis, rather than making a general decision. This calls for flexibility in an implementation, meaning that the decision for OWA versus CWA should be a query-specific rather than a global fixed parameter.

The system model is further parametrized with precondition and effect systems. This is mainly motivated by the fact that there is (still) no general purpose framework on the change semantics and reasoning about change.

To complete this summary, the system model is neutral to how execution is organized. In practice, an execution system can be centralized in the sense that a single instance is responsible for coordinating correct enactment of a service instance. Conversely, a distributed and possibly decentralized execution system can as well be used in which multiple peers cooperate on fulfilling the task. The consequence of distributed execution is the need to support shared access to the KB as there are multiple peers that query and update it. The demand for distributing the KB itself might “naturally” arise in this case in addition. Finally, the conceptual basis of the system model does not rule out concurrency, which is natural in almost any domain, as no simplifying assumptions are made in this regard.



# 5

## Forward Failure Handling using CFI

**B**ASED ON the system model for Semantic Service execution introduced in the previous chapter, in this chapter, we describe our method to achieve optimistic while forward-oriented failure handling to the service execution task. We call it *Control Flow Intervention (CFI)*, named after its main characteristic. The basic idea is to *intervene* in the control flow of a composite service execution in the presence of a failure by modifying its process specification so as to get a semantically equivalent execution and to finally resume execution using the modified control flow. The change that the control flow undergoes is consequently made in the midst of execution and can therefore be classified as dynamic.

Dynamic change within workflows or (business) processes is not a new research topic. Early works that address the problem on a conceptual level in the context of workflow systems date back to the mid-nineties (e.g., [EKR95]). What distinguishes CFI from conventional approaches is its second main characteristic of being optimistic and forward-oriented. Optimistic entails that what the change of a service in the presence of a failure will precisely be is determined ad hoc based on what failed rather than in advance. The change made to recover from a certain kind of failure is thus not pre-defined. CFI thereby provides an additional level of flexibility. Forward-oriented, on the other hand, means that one aims at finding a replacement that results in a semantically equivalent (or similar) outcome when executed instead. These two properties make CFI complementary to conventional failure handling methods such as transactional rollback and compensation or pre-defined exception and fault handlers. The overall context classified along (i) the time when recovery means are specified and (ii) what kind of recovery results can be modeled is depicted by [Figure 5.1](#) and shows where CFI is located.

A key element herein is the notion of a *replacement* that provides a semantically equivalent (or similar) execution. In order to achieve the goal of ad hoc creation of a replacement, mainly two problems need to be addressed. First, how to define a formal and decidable notion of equivalent (or similar) execution that meaningfully captures the intuition of humans. Second and closely connected, what techniques can be used to find or create a replacement. With respect to the former, we exploit the information on the semantics of services and their operations available through their profile, the process, and partly also the grounding. We propose two different notions of equivalent

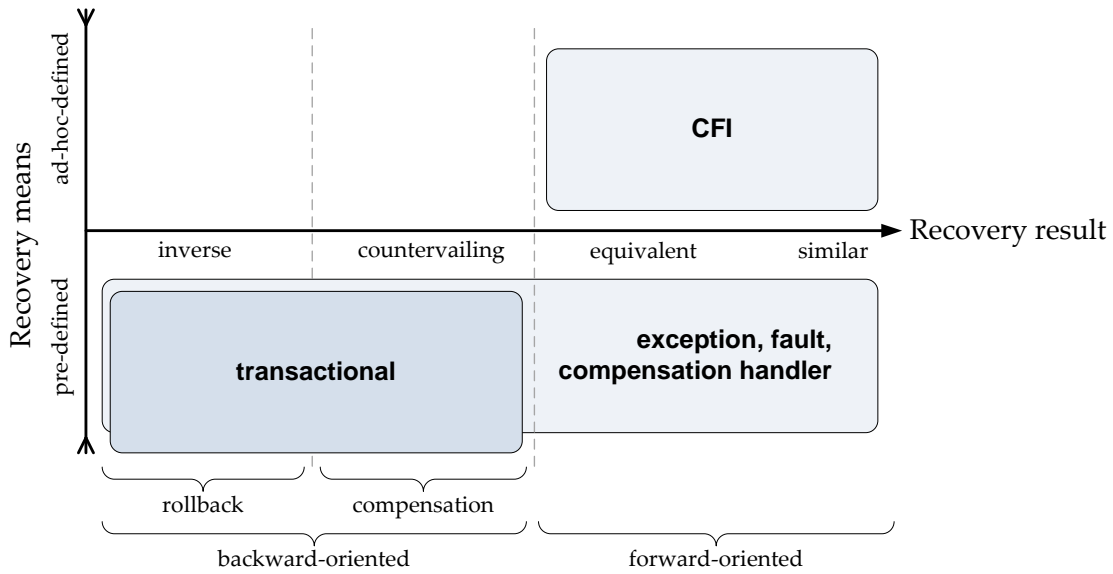


Figure 5.1: CFI in relation to conventional failure handling approaches.

execution that are close from a semantic point of view, but differ with respect to the techniques used to find/create replacements. More specifically, the first notion relies on service matchmaking techniques by formulating the process of finding a replacement as an iterative matchmaking problem. The second is strictly more general and relies on AI planning techniques by formulating it as a planning problem. While semantic matchmaking and planning are prominent on their own in semantic services research, we are focused on streamlining existing methods in these fields to provide an integrated approach with our service model, which has it that, amongst other aspects, the change semantics is modeled as a query answering and a belief update problem over a DL knowledge base.

We will furthermore discuss how the notion of equivalent execution can be broadened towards similar execution. This is worthwhile especially for application domains in which one cannot assume that the notion of equivalence is appropriate in the sense that it is too strict to be effective. Conceptually, we see a close link to the area of planning with soft goals that involves an optimization problem. We therefore sketch a solution by reducing it to so-called net-benefit planning problems. As a side effect, the notion of similar execution thereby defined also allows to include non-functional properties.

The remainder of this chapter is organized as follows. We start by providing a high level overview on how CFI would work in practice in [Section 5.1](#). The types of system environments and the types of failures to which CFI is applicable are detailed in [Section 5.2](#). In [Section 5.3](#) we then introduce the notion of a replacement from a syntactic point of view. The core part of this chapter is [Section 5.4](#) in which we address the problem at the conceptual level from a semantic point of view. The two notions of functionally equivalent execution are introduced and corresponding techniques are described to search for respectively synthesize replacements. In [Section 5.5](#) we clarify how the proposed techniques affect the correctness property of guaranteed termination put

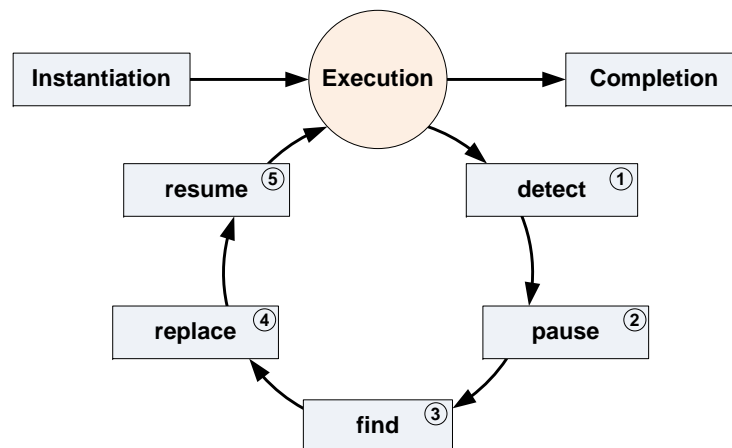


Figure 5.2: Integral activities forming the Control Flow Intervention cycle.

forward by transactional processes. In Sections 5.6 and 5.7 we discuss different related aspects. Finally, Section 5.8 concludes.

## 5.1 The Basic Control Flow Intervention Cycle

From a high level procedural point of view, CFI is divided into a cycle of actions depicted in Figure 5.2. Once a service has been instantiated by an execution engine, failure detectors monitor all relevant execution events (1). In case an event has been detected that is classified as a failure, the engine pauses execution at the earliest possible moment (2), meaning that it temporarily stops working off the control flow. While this is simple to implement for sequential control flows with a single execution thread, pausing a service instance with parallel flows involves a decision: pause all threads or just the one in which a failure event has been detected. The former is indicated if the recovery strategy includes the case that the entire service execution is to be aborted. The number of operations that would need to rollback (or compensate) in this case is thereby reduced to a minimum. On the contrary, if the recovery strategy is purely forward and guaranteed to succeed, all non-faulty execution threads can continue without the need to pause them. The decision is therefore a matter of the recovery strategies.

The following step (3) is the core part. It comprises all the actions that are started then by the engine to find a qualifying replacement for a part of the service, which might minimally be a single operation up to subflows. Once a replacement has been found, the engine then carries out the substitution by modifying the control and data flow accordingly (4). Whether a replacement found is reviewed and accepted by a user prior to substitution is an aspect not in the center of this work. The case in which step (3) terminates in failure because a replacement does not exist and how this case is further handled is left out from Figure 5.2. It should be clear that this is the event in which a conventional recovery strategy would kick in.

Execution finally resumes with the new replacement (5). This last step of the cycle is equally straightforward to pausing the execution. To conclude, steps (1), (2), (4), and

(5) are accompanying management steps of secondary relevance and are therefore not further addressed in this chapter.

## 5.2 Range of Application

CFI is applied to the system model introduced in [Chapter 4](#). While this model makes a couple of assumptions regarding the nature of services, it is not restricted to just one concrete type of environment. In particular, it includes distributed or non-distributed system environments as discussed in this section. By characterizing basic properties of these environments we can precisely identify those types of failures within these environments that are covered by CFI, which we will discuss after supported system environments and their basic properties have been described.

### 5.2.1 System Environments

Following distributed systems theory, we characterize concrete system environments in terms of three basic abstractions: different types of *processes* that run on different types of computing machines and *messages* exchanged among processes that rely on *links* that enable exchange of messages. Notably, there are three different types of processes:

- *Sub process*
- *Service instance process*
- *Server process*

First, a sub process reflects the execution of the implementation of an operation. Therefore, one can say that the process of a service instance  $\dot{S}c$  involves  $n = |Sc'.\mathcal{U}|$  sub processes ( $n \geq 1$ ) where  $Sc'$  is the unfolded service of  $Sc$ . Clearly, a sub process is activated with invocation and ends upon completion. Both the invocation and the sub process itself may fail as detailed below.

Second, a *server* is a computing machine at which sub processes run. Obviously, a server requires software that provides features to activate a sub process on invocation and to further manage its lifecycle. The execution of this software is reflected by a possibly infinitely running server process. A server may run any number of sub processes in parallel, virtually limited only by its technical computing resources. An *execution engine* is the software that activates and manages the process of a service instance (besides taking care on the actual execution task). Again, the execution of an engine itself is reflected by a possibly infinitely running process. For now it is not important whether one engine manages the process of a service instance exclusively or whether it cooperates with other execution engines (i.e., distributed service instance process). Analogous to a server, an execution engine may manage multiple processes of different service instances in parallel. Furthermore, an execution engine may run on a server or at a separate (and dedicated) computing machine.

Obviously, this setup results in a distributed system environment as soon as sub processes and processes of service instances run at different computing machines and



implies communication between them, which is assumed to be done by means of message exchange.

However, it should be noted that the applicability of CFI is independent of whether a concrete system environment is distributed or not. The reason is that the types of runtime failures that are considered in this work apply to both distributed and non-distributed environments. On the distributed end, this includes partially synchronous and synchronous system environments.<sup>1</sup> Since the step to real-time systems is “only” of a temporal nature, CFI would in principle also be applicable there, provided that appropriate means of ensuring timing constraints exist, which is, however, an aspect not addressed in this thesis. The main properties that we require to be provided by any concrete system environment are captured by the following assumption:

- (A7) There is an (eventually) perfect failure detector. Messaging among (sub) processes is reliable.

These two properties are reflected in the so-called *fail-stop* model [SS83] for synchronous distributed environments and its variant the *fail-noisy* model [CGL11a, Chapter 2] for partially synchronous distributed environments. The basic property shared by both models is that processes<sup>2</sup> may *crash* (i.e., halt prematurely). Moreover, processes can communicate with each other through point-to-point message-passing. The difference between the fail-stop and the fail-noisy abstraction lies in the *accuracy* of failure detectors and is inherent in the synchronous versus partially synchronous nature. Accuracy, in short, describes quality-of-service restrictions on the mistakes that a failure detector can make. In summary, the properties of these two models are as follows:

- A process that is not crashed follows its specification. It is called *correct* if it never crashes; otherwise it is called *faulty*.<sup>3</sup>
- Crashed processes do not perform anything. However, in the system environments that we consider they may recover as detailed in [Section 5.2.2](#).

---

<sup>1</sup>Recap, an asynchronous distributed system – which is not to be confused with asynchronous versus synchronous message sending as this refers to non-blocking versus blocking senders [BA06, Chapter 8] – is characterized by the following two properties:

- relative processor speeds and message transmission times are unbounded, which is introduced in practice when best-effort computing machines and networks are subject to by unpredictable loads;
- (sub) process’s local clocks are not synchronized (i.e., there may be arbitrary drifts) since there is no access to a global synchronized clock.

In contrast, a synchronous distributed system is characterized by the assumption that processing, communication delays and clock drifts have an upper bound that is known a priori. Partial synchrony [DLS88] captures the cases where either fixed bounds exist but are not known a priori or where the bounds are known but only hold after some unknown time. Finally, in a real-time system, upper bounds are not only known a priori but subject to strict constraints in the sense of deadlines that are to be met. For more information we refer to general text books such as [BA06, CGL11a].

<sup>2</sup>For the sake of convenience, we do not distinguish between sub processes, server processes, and processes of service instances here and just speak of processes.

<sup>3</sup>The formal definition of correct and faulty in [CT91] relates these terms to a *run*, which is understood as an infinite execution of a system.

- Every crashed process is (eventually) detected by each correct process based on a failure detector, which is accessible to each process (Completeness).
- Every correct process is (eventually) not erroneously suspected of having crashed by any correct process (Accuracy).
- A message sent to a correct process is eventually delivered. No message is delivered more than once (i.e., no duplicates). No message is delivered without having been sent (i.e., no creation).
- Messages sent between the same processes are delivered in the order sent (FIFO).

The feasibility of an (eventually) perfect failure detector necessitates that processes are not arbitrarily slow and latencies between message sending and delivery are not arbitrarily long. Finiteness of bounds is therefore indeed crucial since it would otherwise be impossible to distinguish a correct process from a crashed process [FLP85, DDS87]. An eventually perfect failure detector may erroneously suspect a correct process of having crashed, but there is a finite while unbounded time after which it *eventually* accurately detects a correct process as correct (i.e., it may make mistakes). It is well known that eventually perfect failure detectors can be implemented either using timeouts [CT91] or heartbeats [KACT97]; the latter being advantageous over the former since it does not rely on timeouts and is quiescent (i.e., eventually it stops sending messages). In contrast, a perfect failure detector is devoid of mistakes and detections are permanent (i.e., once a crashed process has been detected, a perfect failure detector will not change its mind). Finally, it should be evident that “applications that have timing constraints require failure detectors that provide a quality of service with some quantitative timeliness guarantees”, as addressed in [CTA02].

### 5.2.2 Failure types

CFI as a method for failure handling aims at covering runtime failures that can be further classified as either

1. invocation failures or
2. execution failures.

Such failures can occur for many reasons. Apart from software and hardware design errors made by humans, the main reasons are the following:

- Hardware systems are subject to various phenomena of a stochastic nature that may suddenly disrupt regular operation.
- Remote distribution of resources and the possibly large number of resources makes environments only partially observable. It is often impossible to have complete knowledge about the current state of all available resources.
- Even if one would have complete knowledge about the environment at some instant in time, it may be subject to arbitrary changes that cannot be anticipated beforehand.

Note here that detection and handling of Byzantine failures as first discussed in [LSP82] in the context of distributed systems is beyond the scope of this thesis. We see means to detect, mask, or protect systems from this class of failures – the importance of which are beyond question – as a separate matter. It is therefore assumed that systems do not expose malicious behavior, which is, in fact, implied by the correct versus faulty property stated earlier on the fail-stop model.

It should also be mentioned that we expect the crash of an execution engine to be transient; that is, processes of service instances that were active and affected by the crash, meaning that they have crashed as well, will eventually recover and resume. Consequently, implementations of execution engines need to provide appropriate means based on stable storage for correct recovery.

### Invocation Failures

There are mainly the following reasons for invocation failures:

1. A server that hosts implementations of operations is (temporarily) unavailable. This can be due to (i) network partitioning (infrastructure related failure), (ii) because the server is down (e.g., for maintenance purposes), or (iii) because it has crashed (local hardware or software fault).
2. The profile or implementation (grounding) of an operation has been changed in a backwards-incompatible way by its provider and some service specification that makes use of it still assumes the meanwhile outdated version; hence, a malformed invocation request is created on its execution (incompatibility fault).

Characteristic for an invocation failure is that a sub process was not activated; hence, no further assumptions on the failure behavior of the sub process need to be made. An invocation failure is detected by an execution engine based on a timeout in case of unavailability of a server, which means that guaranteed eventual delivery as ensured by reliable messaging is explicitly disabled in this case. The second type of invocation failure is detected by an execution engine based on an error reply message sent by the server. Finally, it should be clear that erroneously suspecting a server of having crashed by an eventually perfect failure detector increases the amount of invocation failures.

There is, however yet another type of runtime situation that can be considered an invocation failure though occurring ultimately before invocation: the case of unsatisfied preconditions. Given a transition  $t$  and an execution state  $s = (M, \mathcal{K})$  such that  $t$  is token-enabled in  $s$ , the operation  $Op = fu(t)$  is obviously not invocable if  $f_{\text{chk}}(\mathcal{K}, Op.Pr.P) = \text{false}$ .

### Execution Failures

Contrary to an invocation failure, an execution failure happens after an invocation was successful; that is, after a sub process has been activated and where the sub process is subject to a failure itself. In addition, we also subsume application-level execution failures under this category. By this we mean a prematurely ending but not crashing sub process whose functionality cannot be performed completely due to an unexpected

application-level constellation that prevents this. As an example, imagine an *order* operation within the *order & pay* service of the book seller scenario from [Section 2.1](#) that is successfully invoked but nevertheless fails because a book that the customer wants to order is out of stock. One might counterargue that such application level cases should all be modeled as preconditions so that precondition checking would already catch them. However, it is often impractical even infeasible to model *all* conditions required for successfully performing an operation (or service). In fact, this refers to the *qualification problem* – the insoluble dilemma we are faced with when trying to fully enumerate *all* requirements that may otherwise prevent successful use, as “anyone will still be able to think of additional requirements not yet stated” [McC90].

For execution failures, we need to further detail the fail-stop behavior of sub processes. As stated in [Section 5.2.1](#), a crashed process in the fail-stop and the fail-noisy model does not perform anything. However, it is not clear yet what this means regarding outputs and effects. The precise understanding of fail-stop is made explicit by the following assumption.

- (A8) Given a sub process  $sp$  that reflects execution of an operation  $Op$ , none of  $Op$ 's outputs  $O$  and effects  $E$  materialize if  $sp$  fails *permanently*, nor will any other erroneous side-effect be made permanent in the underlying sub system. If the functionality and/or effects of  $Op$  makes it infeasible to ensure this property then a failure must be *transient*.

A permanent failure means that  $sp$  does not recover. This does not exclude recovery of the server at which  $sp$  ran in case the failure (crash) of  $sp$  coincided with a crash of the server. A transient failure of  $sp$  means that  $sp$  recovers transparently and resumes execution with a sufficiently small delay. In fact, transient failures are not directly relevant for CFI since it is reasonable to assume that  $sp$  will eventually complete in any case, even if it was subject to repeated crashes.

Not materializing any effect, output, nor any other side-effect for a permanent failure essentially necessitates the *fail-safe* or *fail-fast* property. The transactional approach is one means to assure that implementations of operations are fail-safe: already created effects are undone in the presence of a failure (rollback). Fail-fast basically means that a process stops normal operation already before entering a flawed state and immediately reports an error. In both cases we assume that some form of an error message is reported back to the execution engine.

### 5.3 Replacements and their Structure

From the types of failures that CFI aims at (see [Section 5.2.2](#)) we can conclude that they all share the property that an ordinary transition  $t$  in a control flow graph  $G_{cf}$  cannot fire though it is token-enabled: either because already the preconditions of its associated operation (or service) are not satisfied (i.e., the second condition required to enable  $t$  does not hold) or because  $exec(fu(t)) = fail$ . In this section we start by viewing this problem merely from a structural point of view in the control flow. Treatment from a

semantic point of view to achieve a semantically equivalent execution is postponed for [Section 5.4](#).

If transition  $t$  cannot fire and having the basic idea of CFI in mind – where we aim at forward-handling this case – then one needs to *modify*  $G_{cf}$  so as to get an alternative execution. We see two possibilities of how such a modification can be done:

1. *Rebind*  $t$  to another service or operation that qualifies as an alternative to the original one.
2. *Replace* a subflow  $G_{cf}^e \trianglelefteq G_{cf}$  that starts with  $t$  by a different control flow  $G_{cf}^r$  that qualifies as a *replacement* for  $G_{cf}^e$ .<sup>4</sup>

Rebinding transition  $t$  is done by modifying the mapping  $fu$  for  $t$  so that  $fu(t)$  maps to an operation or service different from the original one. The second way of modifying a service can be seen as a *cut-and-replace* approach. By saying that  $G_{cf}^e$  starts with  $t$  we mean that  $t$  is preceded by the initial place of  $G_{cf}^e$ ; that is,  $\bullet t = \{p_i^e\}$ . The reason is obvious:  $G_{cf}^e$  need not include preceding transitions on a path from the initial place  $p_i$  of  $G_{cf}$  to  $t$  since they all have been executed successfully already (otherwise token-enabling would not have reached  $t$ ).

Considering the second option is more general because it allows even for structural changes to the original control flow  $G_{cf}$  (i.e., rebinding  $t$  does not change the structure of  $G_{cf}$ ). This is also motivated by the observation that in some application domains having the possibility for rebinding only might not be sufficient: rebinding  $t$  may imply the need to replace other subsequent transitions as well if they are functionally dependent.

The important property of a replacement  $G_{cf}^r$  from a structural point of view in the control flow is that  $G_{cf}^r$  seamlessly fits into  $G_{cf}$ . By this we mean that the substitution of  $G_{cf}^e$  by  $G_{cf}^r$  is control flow graph preserving. Second, that  $G_{cf}^e$  and  $G_{cf}^r$  are connected to the remaining part of  $G_{cf}$  through and only through their interface (i.e., their initial and final place). This implies the following property. If  $t$  is a starting transition in  $G_{cf}^e$  (i.e.,  $p_i^e \in \bullet t$ ) then any marking that token-enables  $t$  will also token-enable a starting transition  $t'$  in  $G_{cf}^r$ .

We are now defining such structural substitutions formally that ensures these properties.

**Definition 5.1** (Structural Substitution). *Let  $G_{cf}^e$  and  $G_{cf}$  be control flow graphs such that  $G_{cf}^e \trianglelefteq G_{cf}$ .  $G_{cf}^e$  can be structurally substituted by a another control flow graph  $G_{cf}^r$  if the following holds:*

- (1)  $\mathbf{T}^e \cap \mathbf{T}^r = \emptyset$ ; in words, the transitions in both  $G_{cf}^e$  and  $G_{cf}^r$  are different.
- (2)  $\mathbf{P}^e \cap \mathbf{P}^r = \{p_i, p_f\}$ ; in words,  $G_{cf}^e$  and  $G_{cf}^r$  coincide in their initial and final place  $p_i, p_f$  and the set of places are otherwise disjoint.

If  $G_{cf}^e$  can be structurally substituted by  $G_{cf}^r$  then  $G_{cf}^r$  is called a *replacement* for  $G_{cf}^e$ . We write  $G'_{cf} = G_{cf}[G_{cf}^e/G_{cf}^r]$  to denote the modified control flow graph  $G'_{cf}$  that is obtained by substituting  $G_{cf}^e$  with  $G_{cf}^r$  in  $G_{cf}$ .

<sup>4</sup>The superscript e and r are used to indicate the error (or exceptional situation) and the replacement context.

Given a control flow graph  $G_{cf}$ , let  $G'_{cf} = G_{cf}[G_{cf}^e/G_{cf}^r]$ . More precisely,  $G'_{cf} = (\mathbf{P}', \mathbf{T}', \mathbf{F}', M'_0, fu')$  where

$$\begin{aligned} \mathbf{P}' &= (\mathbf{P} \cup \mathbf{P}^r) \setminus \mathbf{P}^e \\ \mathbf{T}' &= (\mathbf{T} \cup \mathbf{T}^r) \setminus \mathbf{T}^e \\ \mathbf{F}' &= (\mathbf{F} \cup \mathbf{F}^r) \setminus \mathbf{F}^e \\ M'_0 &= M_0 \\ fu'(t) &= \begin{cases} fu^r(t) & \text{if } t \in \mathbf{T}^r \\ fu(t) & \text{otherwise.} \end{cases} \end{aligned} \quad (5.1)$$

Clearly, [Definition 5.1](#) implies that  $G_{cf}^r \sqsubseteq G'_{cf}$  and  $G_{cf}^e \not\sqsubseteq G'_{cf}$ . Moreover, it is not difficult to see that  $G'_{cf}$  is also a control flow graph; hence, it is sound, which can be proved as follows. Suppose  $G$  is the union of  $G_{cf}$  and  $G_{cf}^r$ . Then the initial and final place in which  $G_{cf}^e$  and  $G_{cf}^r$  coincide are a split and a join place, respectively. Since  $G_{cf}^e$  and  $G_{cf}^r$  have disjoint transitions and except for the initial and final place also disjoint places, their flow relations are also disjoint (i.e., their control flows are completely independent). Consequently, there is a choice between either of them in  $G$  and therefore also  $G$  is a control flow graph. Finally,  $G$  can be reduced to  $G'_{cf}$  by removing  $G_{cf}^e$ , which can be done by successively applying one of the soundness-preserving reduction rules, as mentioned on [Page 84](#). This allows us to formulate the following theorem.

**Theorem 5.2.** *Given a control flow graph  $G_{cf}$  and a replacement control flow graph  $G_{cf}^r$ , then replacing any subflow  $G_{cf}^e \sqsubseteq G_{cf}$  by  $G_{cf}^r$  as given by [Equation \(5.1\)](#) yields a sound control flow graph  $G'_{cf}$ .*

We classify substitutions into three types, characterized by an increasing level of alteration;  $n, m$  are the number of transitions (i.e.,  $n = |\mathbf{T}^e|, m = |\mathbf{T}^r|$ ):

1. *One-to-one (1:1)*: A single transition is replaced by another transition.
2. *One-to-many (1:n)*: A single transition is replaced by another subflow.
3. *Many-to-many (n:m)*: A subflow is replaced by another subflow.

Examples for these three types of substitutions are graphically depicted in [Figure 5.3](#). Clearly, a one-to-one substitution is the most simple form of alteration, which actually preserves the structure. In fact, a one-to-one substitution can equally be seen as a rebinding of a transition because the associated service or operation is what actually changes. It is also easily seen that a one-to-one substitution is a special case of a one-to-many substitution, and so is a one-to-many substitution a special case of a many-to-many substitution.

## 5.4 Semantically Equivalent Execution

Having introduced the structural properties of replacements in the previous section, we can now address, from a semantic point of view, the criteria that must be met for a

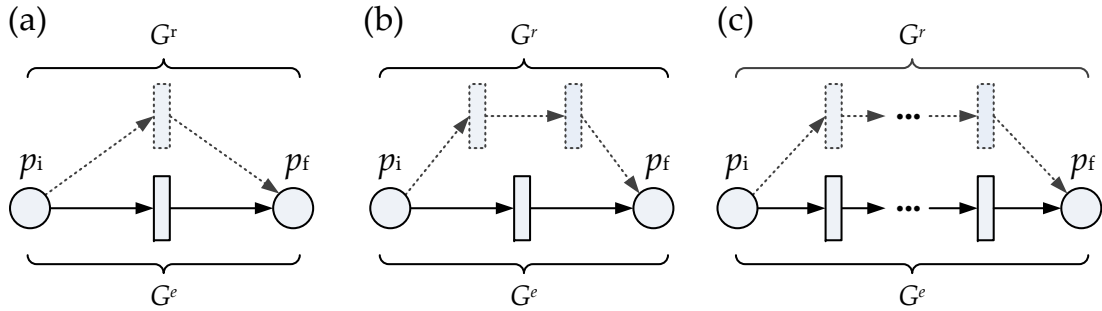


Figure 5.3: Examples for structural substitutions in control flow graphs. (a) One-to-one: Simple substitution of a single transition by another transition. (b) One-to-many: Substitution of a single transition by a sequential subflow. (c) Many-to-many: Substitution of a sequential subflow by another sequential subflow.

replacement  $G_{cf}^r$  to qualify as such. Intuitively,  $G_{cf}^r$  should provide the property that if it is executed instead of the original  $G_{cf}^e$  then it yields a *semantically equivalent execution*. But what *is* a semantically equivalent execution and what not? If one follows intuition then one would attribute a semantically equivalent execution as the property that, no matter whether  $G_{cf}^e$  or  $G_{cf}^r$  is executed, one gets equivalent outcomes. Clearly, this rather rough characterization needs to be formulated precisely. In fact, a detailed treatment should include two more aspects, at least:

1. Does the notion of a semantically equivalent execution consider the functional dimension only or does it also include the non-functional dimension?
2. Does the notion of a semantically equivalent execution preserve the original behavior as specified by  $G_{cf}^e$  or not?

Since the behavior of a service execution is implied by the structure of its control flow, the second question comes down to whether the structure is preserved by a replacement or not. The first question relates to whether a replacement would yield semantically equivalent outcomes only or whether it would also provide an equivalent quality-of-service level than the original. Both are, from a conceptual point of view, separate concerns. In order not to complicate matters, we will handle these two aspects separately. The main focus is on formulating the notion of *functional equivalent execution* in two different ways, which is done in [Section 5.4.4](#). Prior to that, in [Section 5.4.3](#) we introduce the notion of profile equivalence that we use, which is an ultimate prerequisite to the first type of functional equivalent execution. Afterwards in [Section 5.4.5](#) we sketch how the functional dimension can be combined with the non-functional dimension. More importantly, in this section we will also address how the notion of equivalent execution can be broadened towards a notion of similar execution.

Since the first type of functional equivalent execution builds on methods of service matchmaking while the second on service composition planning, we start by discussing them in [Section 5.4.1](#) and [Section 5.4.2](#) in order to provide basic background information. Readers familiar with these topics may skim through these two sections.

### 5.4.1 The Matchmaking Task

Determining whether a service (or an operation) semantically compares with another one is commonly known as *matchmaking*, which is a well known task at the core of service discovery. In fact, it can be understood as a special Information Retrieval problem [KLKR08]. Matchmaking has been widely considered in the literature on semantic services resulting in a variety of approaches. As pointed out in [Klu08], one dimension of classifying matchmaking approaches are the reasoning methods employed. One can distinguish three categories along this dimension: logic-based, non-logic-based, and hybrid. In short, logic-based matchmaking relies on possibly non-monotonic deductive rules of inference as available in (Description) Logics. Non-logic-based comprises all suitable methods that are not logic-based. One direction are approaches that apply (syntactic) similarity measures from Information Retrieval to quantify the semantic relatedness in terms of a distance measure. Another direction considers Machine Learning methods to find predictive patterns on the semantic relatedness based on the (meta) data available about services. Finally, hybrid approaches combine logic and non-logic-based approaches. For (comparative) overviews of these approaches the reader is referred to [Klu08, MIK<sup>+</sup>10, BB10].

#### Formalization

Independent of the actual approach, semantic matchmaking starts from setting up the notion of a *match*. The rationale behind a match is that an *advertised* service/operation (i.e., an offer) is of equivalent or similar value than a *requested* service/operation (i.e., a demand). Given a profile that semantically describes them, a match is defined in most cases exclusively on profiles; we shall distinguish advertised and requested profiles by denoting them with  $Pr^a$  and  $Pr^r$ , respectively. In mathematical terms a match is either formalized in terms of a binary relation or a binary predicate (that maps to true or false interpreted as match or no match). Oftentimes the notion of a match is asymmetric (hence irreflexive). Applied to profiles, it is usually intended to represent that an advertised profile  $Pr^a$  *matches with* a requested profile  $Pr^r$ , while the opposite –  $Pr^r$  matches with  $Pr^a$  – not necessarily holds as well. It is reasonable at least to consider the notion of a match as irreflexive because one might want to exclude the trivial case of no avail: every profile matches to itself per se. Finally, matchmaking is inevitably driven by domain (or background) information/knowledge based on which one concludes whether there is a match or not; that is, the domain knowledge *entails* a match or not. Altogether, we define matchmaking on profiles as follows.

**Definition 5.3** (Matchmaking Domain & Problem). *A matchmaking domain is a 4-tuple  $MD = (\mathcal{K}, \mathcal{P}, \sim, \models)$  where  $\mathcal{K}$  is a collection of domain knowledge<sup>5</sup>,  $\mathcal{P}$  is a finite set of profiles,  $\sim \subseteq \mathcal{P} \times \mathcal{P}$  is an irreflexive match relation, and  $\models$  is a consequence relation.*

*Given a matchmaking domain  $MD$  and a requested profile  $Pr^r \in \mathcal{P}$ , a matchmaking problem is a tuple  $MP = (MD, Pr^r)$ .  $ms \subseteq \mathcal{P}$  is a set of  $Pr^r$ -matches for  $MP$  iff  $\forall Pr^a \in ms: \mathcal{K} \models Pr^a \sim Pr^r$ .*

<sup>5</sup> $\mathcal{K}$  need not necessarily be a DL knowledge base herein. It can build on other formalisms of representing knowledge/information. In the same vein,  $\models$  need not realize deductive rules of inference.



The software that implements matchmaking in a specific matchmaking domain is usually called a *matchmaker*. A matchmaker inevitably needs to be closely integrated with a repository that stores profiles, which makes it a natural component of a service directory, integrated accordingly in the retrieval process (query answering).

As there can be a set of profiles that match some requested profile, one might additionally want to order them according to some *preferences*. There are basically two ways considered in the literature to this. First, by defining several matching relations that differ from each other in their *degree of match* (DoM). This induces a discrete rank over the DoM and needs to be combined with simple algorithmic processing: If the best match relation does not hold, try the second best, if this fails then try the next best, and so on. This approach has been considered in most cases on the functional dimension of profiles. Another way is to define, in addition to the notion of a match, a possibly strict order that models user preferences. For instance, one might want to order profiles that functionally match a requested profile according to their reliability, response time, throughput, usage costs, or the like. Of course, the order relation can incorporate both the functional and non-functional dimension. Preference-based matchmaking is consequently defined as a straightforward extension of basic matchmaking.

**Definition 5.4** (Preference-based Matchmaking Domain & Problem). *A matchmaking domain with preferences is a 5-tuple  $MD = (\mathcal{K}, \mathcal{P}, \sim, \succcurlyeq, \models)$  where  $\succcurlyeq \subseteq \mathcal{P} \times \mathcal{P}$  is a preference order.*

*Given a matchmaking problem  $MP = (MD, Pr^r)$ ,  $ms = \{Pr_1^a, \dots, Pr_n^a\}$  is a set of  $Pr^r$ -matches for  $MP$ , indexed by consecutive integers  $\{1, \dots, n\}$ , iff the following holds*

- (1)  *$ms$  is a set of  $Pr^r$ -matches for  $MP$  in  $MD = (\mathcal{K}, \mathcal{P}, \sim, \models)$ ,*
- (2)  *$\forall Pr_i^a, Pr_j^a \in ms: i < j$  implies  $Pr_i^a \succcurlyeq Pr_j^a$ .*

*$Pr_1^r$  is called the most preferred match and  $Pr_n^r$  the least preferred match.*

The preference order  $\succcurlyeq$  can be defined in many ways. One possibility are quantitative metrics such as distance measures.

### DL-based Matchmaking

The general principle common to all DL-based approaches known in the literature is to formulate the notion of a match, in one or another way, based on the set-theoretic subsumption relation. Reasoning on whether there is a match is thereby reduced to the standard subsumption inference task and derives its computational complexity properties (see Section 3.1.4). In the context of DL-based semantic services it has been first described in [PKPS02] and [LH03]. The former assumes structured semantic service descriptions such as a profile and defines the notion of a match on elements of the structure, which is why it can be classified as *structured matchmaking*. In contrast, the latter assumes that a service (or an operation) is semantically described by a single concept only that is a complex intersection

$$C_1 \sqcap \dots \sqcap C_n$$

where  $C_i$  are atomic or complex concepts. It is therefore classified as *monolithic match-making* [Klu08]. Part of the idea in [PKPS02] is inspired by earlier works on component theory in software engineering [ZW97] that similarly build on “more general”, “more specific” abstractions over the signature of components as well as parallel works in agent-based environments [SWKL02].

*Inputs and Outputs.* There are four prominent matching relations defined exclusively in terms of the subsumption relation [PKPS02, LH03]. Under structured matchmaking on the functional dimension of the profile they are utilized by pairwise matching elements in the input and output sets. More specifically, a profile  $Pr^a$  matches with profile  $Pr^r$  regarding the outputs if there is a matching output  $o^a \in Pr^a.O$  for every output  $o^r \in Pr^r.O$ . Note here that  $Pr^a$  may specify more outputs than  $Pr^r$  (i.e.,  $|Pr^a.O| \geq |Pr^r.O|$ ). We call such an additional output in  $Pr^a$  for which there is no matching output in  $Pr^r$  a *spare output*. Formally,

$$Pr^a \bowtie\text{-matches } Pr^r \text{ regarding } O \Leftrightarrow \forall o^r \in Pr^r.O \exists o^a \in Pr^a.O: type(o^r) \bowtie type(o^a) \quad (5.2)$$

where  $\bowtie$  can be one of

- *Exact*:  $type(o^r) \equiv type(o^a)$ ,
- *Plug-in*:  $type(o^r) \sqsupseteq type(o^a)$ ,
- *Subsume*:  $type(o^r) \sqsubseteq type(o^a)$ ,

and by a slight abuse of notation (the meaning should be clear)

- *Intersection*<sup>6</sup>:  $(type(o^r) \sqcap type(o^a)) \not\sqsubseteq \perp$ ,
- *Disjoint*:  $(type(o^r) \sqcap type(o^a)) \sqsubseteq \perp$ .

These relations constitute a ranked while discrete DoM, the order of which can be written as

$$Exact > Plug\text{-}in > Subsume > Intersection > Disjoint$$

where, stated informally,  $>$  means “stronger than”. The *Exact* match is clearly the most preferable as it is the strongest relation corresponding to extensional equality: According to the model-theoretic semantics in DLs, the concepts or data ranges of matching outputs have the *same* extension in every model  $\mathcal{I}$ . Given execution compatibility (see [Definition 4.16](#)), the value range of each output in  $Pr^r.O$  coincides with the value range of the matching output in  $Pr^a.O$ . The plug-in match is the second best and basically states that the outputs of the advertised suffice to fulfil the outputs of the requested (i.e., the advertised does not produce output values that also the requested would not produce). Conversely, the subsume match states that there might be output values produced by the advertised that would not be produced by the requested. The disjoint relation is at the lowest level. It is actually not a match since it shows that the advertised is incompatible with the requested as they have an empty intersection, which is why

<sup>6</sup>Also referred to as a *partial* match (e.g., [HBHP09]).

we were speaking of four matching relations. This distinguishes it from the intersection match which captures the case where both are not totally incompatible.

Contrary to outputs, a match is defined vice versa when applied to the inputs of a profile. More precisely, a profile  $Pr^a$  matches with profile  $Pr^r$  regarding the inputs if there is a matching input  $i^r \in Pr^r.I$  for every input  $i^a \in Pr^a.I$ . Observe that in this case  $Pr^r$  may specify more inputs than  $Pr^a$  ( $|Pr^r.I| \geq |Pr^a.I|$ ); which means that there can be *spare inputs* in  $Pr^r$  for which there is no input in  $Pr^a$ . However, it is reasonable to formulate a single condition rather than different degrees of match for inputs. This is justified by the consideration that the advertised should generally suffice to fulfil processing at least the range of input values that the requested does, but not less, in order to be considered a match. Formally,

$$Pr^a \text{ matches } Pr^r \text{ regarding } I \Leftrightarrow \forall i^a \in Pr^a.I \exists i^r \in Pr^r.I: type(i^r) \sqsubseteq_n type(i^a) \quad (5.3)$$

where  $n$  is the maximum *distance* between  $type(i^r)$  and  $type(i^a)$  in the concept/data range hierarchy. This means that the distance is calculated based on a graph-theoretic model in which vertices represent concepts and edges represent direct subsumption relations between them (e.g., given  $A \sqsubseteq B \sqsubseteq C$  and there is no  $D_1, D_2$  with  $A \sqsubseteq D_1 \sqsubseteq B$ ,  $B \sqsubseteq D_2 \sqsubseteq C$ , the corresponding graph contains three vertices  $A, B, C$  and two edges  $(A, B), (B, C)$ , but not  $(A, C)$ ). The simplest way is to take the *edge count* distance (e.g., given edges  $(A, B), (B, C)$ , the distance is 1 for  $A, B$  and 2 for  $A, C$ ). The basic assumption under the edge count distance measure is that subsumption represents uniform distance. As this might not be appropriate in general, another possibility is to assign weights in the interval  $[0, 1]$  to edges, thereby allowing for variability in the distance. It is further reasonable to combine this with standardization by requiring that the total sum of weights on the edges between a parent concept and its direct sub concepts is one. Determining appropriate weights can be done based on information-theoretic models in which one quantifies the semantic relatedness of concepts.

Limiting **Condition (5.3)** upwards using a distance is motivated by the case of profiles that are too generic and that should therefore be filtered. Otherwise, one would include profiles that match everything in the worst case: Imagine an input  $i^a$  of a very generic advertised profile with  $type(i^a) = \top$ . Clearly,  $i^a$  matches any input according to **Condition (5.3)** because  $\top$  is the universal concept. Limiting a match to direct parents ( $n = 1$ ) effectively avoids such matches, provided that the domain conceptualization is not flat (i.e., where  $\top$  is the direct parent). Such limits have also been applied conversely as lower bounds for matchmaking on outputs (e.g., [KFS09]). Finally, the distance can also be utilized for ordering. An advertised input  $i_1^a$  matches a requested input  $i^r$  *more closely* than another input  $i_2^a$  if its type is closer to that of  $i^r$ ; that is,

$$(type(i^r) \sqsubseteq_m type(i_1^a)) > (type(i^r) \sqsubseteq_n type(i_2^a))$$

if  $m < n$ , which orders  $i_1^a$  before  $i_2^a$  in this case.

**Preconditions and Effects.** Structured matching as embodied by **Condition (5.2)** and **(5.3)** can, in principle, be applied respectively to effects and preconditions that are DL-based (e.g., [BOI09]). As mentioned before, this principle, in fact, is inspired by previous work on precondition and effect matching on software components [ZW97].

Upon closer inspection we have found, however, that doing so is inappropriate for effects whose semantics is defined in terms of a belief update. We argue that **Condition (5.2)** does not appropriately capture the intuition of the plug-in and subsume match in this case. To explain this, recall that an inclusion  $C \sqsubseteq D$  or  $R \sqsubseteq S$  can be understood as an implication (see **Section 3.1.1**). In fact, an inclusion on effects describes a ramification (i.e., an indirect effect). For example, given a role inclusion  $\text{hasBoughtBook}(x, y) \sqsubseteq \text{ownsBook}(x, y)$ , an effect atom  $\text{hasBoughtBook}(x, y)$  implies  $\text{ownsBook}(x, y)$  as an indirect effect – the indirect effect of buying a book is ownership. Now, what is the intuition of the plug-in and subsume match regarding effects? According to the widely adopted view of [ZW97], the plug-in match<sup>7</sup> is defined as an implication. Expressed in terms of profiles it reads as follows:

$$Pr^a \text{ plugs into } Pr^r \text{ regarding } E \text{ iff the effects described by } Pr^r \text{ are implied by } Pr^a. \quad (5.4)$$

Alas, we see two problems in this definition. First, it would be possible that  $Pr^a$  describes additional effects besides the ones that imply the effects of  $Pr^r$ . Second, as the definition makes use of implication, it is sufficient that  $Pr^a$  specifies effects that indirectly imply  $Pr^r$ 's effects.

We view the plug-in and subsume match as follows. An advertisement subsumes a request regarding effects if the advertisement creates *at least all* the effects that the request creates. We might, more figuratively, say that the advertisement “does more” than the requested. If we further understand the plug-in match as the dual of the subsume match then this would mean that an advertisement plugs into a request if it creates *some* of the request's effects. It should now be apparent that **Condition (5.2)** does not represent this. We would actually accept that an advertisement plugs in if it creates *pre-indirect* effects and it subsumes the request if it creates *post-indirect* effects. The former means that the advertisement creates effects that indirectly imply (cause) the effects of the request ( $C^a \sqsubseteq D^r$ ), whereas the latter means that none of the effects of the request would be created ( $C^a \supseteq D^r$ ). Neither case does adequately represent our intuition. An alternative definition that represents it simply builds on *containment* on the effect sets and (mutual) *subsumption* between single effects. More precisely,

$$Pr^a \left\{ \begin{array}{l} \text{plugs into} \\ \text{subsumes} \end{array} \right\} Pr^r \text{ reg. } E \Leftrightarrow \left\{ \begin{array}{l} |Pr^a.E| \leq |Pr^r.E| \text{ and } \forall \varphi \in Pr^a.E: \varphi \bowtie m_p(\varphi) \\ |Pr^a.E| \geq |Pr^r.E| \text{ and } \forall \varphi \in Pr^r.E: \varphi \bowtie m_s(\varphi) \end{array} \right. \quad (5.5)$$

where  $m_p: Pr^a.E \rightarrow Pr^r.E$  and  $m_s: Pr^r.E \rightarrow Pr^a.E$  are injective mappings between the effect sets and  $\bowtie$  can be one of:

- *Weak* plug-in:  $\varphi \sqsubseteq m_p(\varphi)$ ; weak subsume:  $\varphi \supseteq m_s(\varphi)$ ;
- *Strict* plug-in/subsume:  $\varphi \equiv m_{p,s}(\varphi)$ .

We call the former *weak* and the latter *strict* because, depending on the actual effect semantics, a weak match might allow for indirect effects whereas a strict match does not. Observe that “ $\sqsubseteq$ ” is directed equally for weak plug-in versus weak subsume: in both cases the effects of  $Pr^a$  are more specific than those of  $Pr^r$ . Furthermore, “ $\sqsubseteq$ ” need not

<sup>7</sup>Actually called plug-in *post* match in [ZW97] if applied to effects only.

be interpreted strictly in the DL set-theoretic way. In particular, it need not be a transitive relation. Analogously, “ $\equiv$ ” is not necessarily understood in the strict mathematical sense (reflexive, symmetric, transitive) as one might want to rule out transitivity.<sup>8</sup>

From **Condition (5.5)** one can easily derive how weak/strict exact, intersection, and disjoint matches are defined, which we leave as an easy exercise.

To conclude, if we compare **Condition (5.4)** with our definition then we see that it corresponds to a weak subsume, which perfectly reflects the concerns raised above.

### Offline versus Online Matchmaking

Whether matchmaking can be done offline versus the need to do it online at runtime (e.g., as part of a CFI cycle) is merely determined by the temporal variability – thy dynamics – of information included in a match relation. For instance, DL-based subsumption matchmaking on *IO* profile parameters can be done offline since they are statically typed. The same applies to static preconditions and effects. Non-functional matchmaking on *N* profile parameters is more likely to be done online. The reason is that typical non-functional properties can be subject to possibly frequent dynamic changes (e.g., the response time that varies depending on the load). Clearly, offline matchmaking can be utilized for performance optimization by pre-computing matches between profiles and combining it with appropriate indexing or caching techniques for fast retrieval of matches at runtime (e.g., [SHF11]).

## 5.4.2 The Planning Task

The composition of services can be formulated as an AI Planning problem, which has been pioneered in particular by [McD02, MS02]. As a result of significant advancements in terms of theoretical foundations, scalability, available tools, and the fact that *actions* are well suited for modeling operations of services, AI Planning has come to be a primal approach to (semi-)automatic composition of (Web) services, evidenced by a series of surveys [RS04, KSKR05, Pee05, ACMS08, Klu08, MP09, GTSS11, SVV11]. Planning approaches range from functional, non-functional, and process-level composition, combinations of these, and composition under varying assumptions on the environment. Yet the prevalent model to planning – not only in the context of service composition – is that of a discrete state space that is to be searched for solutions.

### The Basic State Space Model to Planning

We briefly introduce the state space model, which follows mostly [GNT04, Gef11]. In its basic form the model contains:

- a finite set of *states*  $S$ , called the state space,
- a set of *actions*  $A$  where  $A(s) \subseteq A$  denotes the set of actions applicable (executable) in state  $s \in S$ , and

---

<sup>8</sup>Examples where this is the case are the *broader*, *narrower*, and *related* properties (roles) in the Simple Knowledge Organization System (SKOS) [MB09], which are not transitive (and neither defined as reflexive nor irreflexive).

- a *transition function*  $F: A \times S \rightarrow S$  such that  $a \notin A(s)$  implies  $F(a, s) = s$ ; in words,  $F$  associates to each current state  $s$  and action  $a$  a successor state  $s'$  that represents the result of applying  $a$  in  $s$  if  $a$  is applicable in  $s$ .

A *planning domain* is correspondingly represented by the 3-tuple

$$PD = (S, A, F) .$$

One can equally conceive this model as a directed graph in which a node is a state and an edge, labeled with an action, represents a transition between a state and its successor state resulting from the application of the action. The variety of this model lies in the actual definition of what a state and an action is, what the criterion for an action is to be applicable in a state, and how the transition function modifies a state.

Given a known *initial state*  $s_0 \in S$  and a *goal state*  $s_g \in S$ , a sequence of actions  $a_0, \dots, a_n$  such that

$$s_{i+1} = F(a_i, s_i), \quad 0 \leq i \leq n,$$

is a solution or *plan* in this model if

$$s_g = F(a_n, s_n) .$$

Planning is therefore the ability of a software (agent) to automatically synthesize a *plan* without being explicitly told the necessary steps that need to be performed to reach a *goal* from an initial situation (state). The 3-tuple

$$PP = (PD, s_0, s_g)$$

denotes a *planning problem* (or planning instance) in the planning domain  $PD$ . Clearly, a path between  $s_0$  and  $s_g$  in the graph forms a plan – a sequence of actions, which indicates that the process of planning can be reduced to (heuristic) search in the graph whether there exists a path leading from  $s_0$  to  $s_g$ . An *optimal* plan has minimum execution cost among all plans for a planning instance. Under the assumption that actions have uniform execution costs, a plan is optimal if there is no other plan that is shorter (i.e., the length of a plan represents its total execution costs).

A common assumption is that there is at least one action applicable in every state ( $\forall s \in S: A(s) \neq \emptyset$ ). While this assumption is necessary for liveness it is obviously not sufficient to guarantee that a plan exists for a planning problem. More relevant, in fact, are two properties concerning dependability of planning algorithms: *soundness* and *completeness*. A planning algorithm is sound if it generates plans that are correct, meaning that execution of the plan transforms  $s_0$  into  $s_g$ . A planning algorithm is complete if it is guaranteed to (eventually) find a plan if one exists.

The main reasoning task, which is performed by virtually all state space search planners either explicitly or implicitly when navigating through the space, is *plan checking*: given a planning problem  $PP$ , is a plan a solution for  $PP$ . The second prominent reasoning task is *plan existence*: given a planning problem  $PP$ , is there a solution for  $PP$ . Decidability of the latter is, however, not of utmost importance. Most planning tools assume the existence of a plan anyway and try to find one, instead of proving that none

exists; see also the discussion in [Hel02]. More relevant is therefore the efficiency of finding plans while not exhausting available resources such as time or memory.

The basic state space model captures restricted environments only. More specifically, the fact that the initial state is known assumes *full observability* of the environment, which essentially means that one has complete knowledge about the initial situation. Second, the fact that the transition function maps to a single successor state implies that actions are *deterministic*. Third, the system is *static*, meaning that it stays in a state unless an action is applied. Finally, time is *implicit* (i.e., abstracted away), which implies that actions are thought to be instantaneous and have no duration. Planning under these assumptions is commonly referred to as *classical planning*. The seminal and still common framework for encoding classical planning problems is the Stanford Research Institute Problem Solver (STRIPS) [FN71], which is introduced next.

### The STRIPS Framework for Encoding Classical Planning Problems

In short, a STRIPS planning problem is formulated as a 4-tuple  $(P, O, I, \Gamma)$  where:

- $P$  is a finite set of propositional variables (Boolean variables), called the *conditions* (a.k.a. *fluents* as their truth value can change from state to state);
- $O$  is a finite set of *operators* (actions) of the form  $(pre, add, del)$  where  $pre, add, del$  are each a subset of  $P$ , called the *precondition*, *add*, and *delete* sets, respectively;
- $I \subseteq P$  is the *initial state*; and
- $\Gamma \subseteq P$  are the *goals*.

There is one remark on actions versus operators in order here. Unlike stated, it is custom that an operator is understood as a *parametrized* action; that is,  $pre, add, del$  contain atoms (predicates) of the form  $p(x_1, \dots, x_n)$  where  $x_i$  is a variable that is implicitly existentially quantified. An operator thereby represents all actions that can be obtained by instantiating each variable from a finite set of given logical *constants*, which we denote with  $\mathcal{C}$ . These constants represent objects existing in the domain that are the subjects of planning – individuals in case of DLs. It is assumed that different constants denote different objects, that every object that exists is represented by a constant, and that the interpretation of constants does not change between states (i.e., standard names assumption together with a fixed interpretation). Notice that a ground atom therefore resembles a propositional variable. Formally, let  $\text{Var}(o)$  be the set of variables occurring in  $pre, add, del$  of an operator  $o$ . The set of actions that can be obtained by instantiating  $o$  based on  $\mathcal{C}$ , denoted with  $o[\mathcal{C}]$ , is

$$o[\mathcal{C}] = \{ o[\theta] \mid \theta: \text{Var}(o) \rightarrow \mathcal{C} \}$$

where  $o[\theta]$  denotes an action obtained by applying a substitution  $\theta$  to  $o$ .

A STRIPS instance encodes the planning domain as follows. Every state  $s$  is described as a subset of  $P$  ( $s \subseteq P$ ). States are interpreted under CWA:  $\varphi \in P$  is true in  $s$  if  $\varphi \in s$ ; otherwise  $\varphi$  is false in  $s$ . Hence, states are complete descriptions of the current situation, which matches the assumption of full observability. The initial state  $s_0$  is  $I$ . A

state  $s_g$  is a goal state if  $\Gamma \subseteq s_g$ . The set of actions applicable in a state  $s$ , denoted with  $A(s)$ , are those whose preconditions are a subset of  $s$ , formally

$$A(s) = \{ a \mid a \in \left( \bigcup_{o \in O} o[C] \right) \text{ and } pre(a) \subseteq s \} . \quad (5.6)$$

Finally, given an action  $a$  and a state  $s$ , the successor state is

$$s' = F(a, s) = (s \setminus del(a)) \cup add(a) . \quad (5.7)$$

Asymptotic computational complexity in the basic (propositional) STRIPS framework is intractable as it has been shown to be PSpace-complete [Byl94]. The reason is that there can, in general, be plans of exponential length in the size of the planning problem. Complexity drops to NP for plans bounded to polynomial length. Exceedingly long plans are more of a theoretical matter as the intuition especially in the area of service composition is that composite services are rather short.

As a result of intractability, a major part of planning research since then has focused on (i) encoding planning problems in a way that keeps the search space as small as possible and (ii) to devise search strategies that enable scaling up to possibly huge state spaces while being sound and complete, and ideally also optimal. A remarkable achievement in this respect is [HCZ10] where a translation of STRIPS planning problems into the framework of planning with multi-valued state variables is described, which is a special case of Functional STRIPS [Gef00], and which results in considerably smaller state spaces.

The Problem Domain Description Language (PDDL) [MGH<sup>+</sup>98] is the de facto standard machine-parsable format for representing STRIPS planning instances and instances expressed in successor languages of STRIPS. Newer versions of PDDL [FL03, GL06] have been extended in several ways; amongst others, to support expressing extended goals, which will be outlined next.

## Extensions to Cover Practical Domains

Since the basic state space model is not expressive enough to capture many real-world environments it is extended in several ways. Major dimensions along which extensions are made are summarized in Table 5.1, which we will go through briefly one by one.

**Goals.** Extensions regarding goals concern scenarios in which one wants to express more complex objectives than the specification of a final state to be reached. This includes two classes. First, constraints that describe states that must be traversed or, conversely, states that must be avoided by a plan (e.g., achieving a subgoal, avoiding a critical situation). Second, constraints that must be optimized or meet at some, any, or all time during plan execution. As both classes relate to a *state trajectory* in time<sup>9</sup>, such goals are referred to as *temporally extended goals*.

Another line of research is concerned with whether goals are regarded mandatory or not. In so-called *over-subscription planning* [Smi04] a.k.a. *partial satisfaction planning* [BNDK04] goals are no longer mandatory but desired, meaning that one is satisfied

<sup>9</sup>In short, a state trajectory is a sequence of pairs  $\langle (s_0, t_0), (s_1, t_1), \dots, (s_n, t_n) \rangle$  where each  $s_i$  is a state and  $t_i$  is a timestamp.



Table 5.1: Different dimensions of planning domains.

Dimension	Short Explanation
Goals	Whether a goal represents a state to be <i>reached</i> (no matter how) versus a desired <i>evolution</i> in the domain. Whether goals are regarded <i>mandatory</i> versus <i>optional</i> (desired).
Observability	Whether states of the domain are <i>partially</i> or <i>fully</i> observable.*
Controllability	Whether actions/operators are <i>deterministic</i> versus <i>nondeterministic</i> .
Dynamics	Whether the state of the domain changes only through actions or whether there can be other <i>events</i> that also effect state changes.
Agility	Whether plan generation and plan execution are <i>separated</i> versus <i>interleaved</i> , meaning that execution can take place while planning is ongoing.
Processing	Whether a plan is a strict versus a partial ordering, is conditional, iterative, or a mixture thereof.

\* Partial observability includes the special case of no observability at all.

with plans that achieve a subset of the goals, as opposed to classic planning which terminates in failure unless all goals can be achieved. Such goals are also referred to as *soft goals* as opposed to *hard goals*. They are mainly motivated by planning under limited resources available (e.g., time) or the presence of mutually exclusive goals. In order to assist a planner in choosing which goals to achieve, each goal has an *utility* value associated (i.e., one prioritizes goals). Alternatively, one can also penalize the violation of a goal, which is the approach considered in PDDL3 [GL06]. In addition, actions do no longer have uniform execution costs. The objective of a planner is then to maximize the utility while minimizing costs. Hence, planning involves a (combinatorial) optimization problem, also referred to as *net-benefit* problems [HDR08, KG09]. Soft goals, in fact, describe a simple model of *preferences*. More interestingly, soft goals<sup>10</sup> do not increase expressive power since they can be compiled into a STRIPS planning problem with action costs and hard goals [KG09], for which conventional cost-based STRIPS planning machinery can be used then.

**Observability.** In various real-world domains it is not practical even feasible (for technical reasons) to have complete information about states; that is, states are partially observable only. Planning under *incomplete information* about states is modeled by extending the basic model such that there is a set of initial states. A planner must then account for the fact that the system might be in any of these states, which is correspondingly referred to as *conformant planning* as a plan must work for all possible initial states. Extending STRIPS to allow for negative atoms, conditional effects, and adoption of OWA is one way of modeling partial observability. Not surprisingly, computational

<sup>10</sup>To be precise, a soft goal here is either a single fluent or a conjunctive or disjunctive formula over different fluents.

complexity becomes harder: plan existence is  $\text{ExpSpace}$ -complete for plans exponential in length [Rin04] and  $\Sigma_2^P$ -complete for plans bounded to polynomial length [Tur02]; plan checking is NP-hard. Similarly, a KB interpreted under OWA models partial observability since it may be satisfied by many interpretations, each representing a possible state of the domain.

*Controllability.* Apart from partial observability there can be one more source of uncertainty: actions that behave nondeterministically. Controlling involvement in this case necessitates taking all possible outcomes of actions into account, which is correspondingly referred to as *contingency planning*. Extending the transition function  $F$  to map to a set of successor states is one way of representing nondeterminism of actions, which renders the model conceptually close to a Kripke structure [Kri63] used mainly in the field of Model Checking [CGP01]. Another way leading to the theory of Markov Decision Processes [Put94] is to associate transitions with probabilities to capture the stochastic character of the domain. The general assumption, however, is that nondeterminism of actions is tractable, meaning that the number of different outcomes that an action may have is finite and all possible outcomes are known in advance.

*Dynamics.* Thus far planning domains were characterized by the absence of additional events transforming the system into a new state.<sup>11</sup> Extending the model to accommodate for such events can be done by (i) introducing a finite set of events  $E$  and (ii) formulating the transition function as  $F: A \times E \times S \rightarrow S$ . As there can be transitions solely caused by an action or an event, a neutral event and, symmetrically, a no-op action are introduced in addition. Events can also be used to model the concurrent execution of multiple plans (by different) engines in the domain.

*Agility.* An operational aspect not related to the underlying model concerns the way plan synthesis and plan execution are integrated. There are basically two possibilities to this. Under the paradigm of *static planning* (a.k.a. offline planning) both are strictly separated, meaning that the plan is not executed unless it has been completely generated. Conversely, under the paradigm of *dynamic planning* (a.k.a. online planning) both can be interleaved. In the most dynamic case planning and execution are interleaved in a step-by-step way: each new action added to a plan is executed immediately followed by planning for the next action, which is repeated unless the goal state is reached. Dynamic planning becomes relevant for (i) nondeterministic actions in order to react to their actual outcome and (ii) dynamic domains in which it is important to take events into account.

Similarly, a common technique to make planning under incomplete information practical is to execute *information-providing* actions<sup>12</sup> directly at planning time while execution of *world-altering* actions is simulated. However, this involves the so-called *Invocation and Reasonable Persistence Assumption* (IRP) [MS02]. Intuitively, IRP states that (i) information-providing actions can be executed at planning time (i.e., preconditions are satisfied) and that (ii) information persists once gathered until execution of world-altering actions, which includes that world-altering actions must not change gathered information even if the change is only simulated, see [Example 5.1](#).

<sup>11</sup>No matter whether these events are exogenous versus endogenous.

<sup>12</sup>Also known as sensing or callback actions.

**Example 5.1**

Imagine an information-providing action  $a_1$  that identifies an idle ambulance; say its execution returned the ambulance named  $A1$ . Imagine a world-altering action  $a_2$  that assigns a mission to  $A1$  by changing its state to busy. If  $a_1$  is executed again after simulating execution of  $a_2$  then it would report  $A1$  still as idle because the real world state is behind the simulated world state (i.e., re-executing  $a_1$  later during planning provides “outdated” information). Consequently, the simulated world state is incorrectly overwritten<sup>13</sup>, which might lead to incorrectly reassigning  $A1$ .

Conversely, if the state of  $A1$  changes in the real world between simulation and execution time of world-altering actions (due to dynamics such as concurrent executions) then the generated plan might become outdated relative to the real world state, which can lead to a runtime execution fault of  $a_2$ .

---

*Processing.* Plans need not necessarily be sequences of actions. Whenever two actions are not mutually exclusive (i.e., there is no causal dependency between them) then they can be arranged partially ordered; thus, their execution can be linearized in either order or even in a concurrent way. A *partial-order plan* consequently specifies only those orderings among actions that are necessary to achieve the goal, which is also referred to as the *least commitment strategy* [Wel94]. *Conditional plans* bring about another type of structure. They are mainly considered to cope with nondeterministic actions to choose the next action depending on the actual situation after a nondeterministic action has been executed. Similarly, *iterative plans* are a concise way of representing repeated execution of an action until a desired situation occurs. Execution of all these types of plans forms a process within the process model introduced in [Section 4.3](#).

### Linking Planning and Service Composition

The apparent connection between action planning and service composition is that a parametrized action corresponds to an operation or to an atomic service. A sequential plan corresponds to a composite service having a sequential control flow; analogously, a partially ordered plan corresponds to a control flow with parallel flows. Details of the correspondences vary depending on what types of semantics are represented by the action and service model (cf. [Figure 4.1](#)) and how it is precisely formalized.

### 5.4.3 Functional Profile Equivalence

We formulate the criteria for profile equivalence regarding the functional dimension in a similar way to structured matchmaking described in [Section 5.4.1](#). The main concept is straightforward: we consider inputs, outputs, and effects (*IOE*); hence, preconditions are not considered, the reasons of which will be discussed below. It is important to understand that we formulate a minimal criterion that comprises properties that are required at least to get a semantically equivalent execution: equivalent effects, required

---

<sup>13</sup>This can be seen as a *lost update* known from database transaction theory.

outputs, while satisfied with the same set of inputs available. It is therefore not contributing another approach to the service matchmaking, apart from the introduction of a revised criterion of effect matches (cf. [Condition \(5.5\)](#)) and revised match criteria described next.

We have found that the conventional approach reflected by [Condition \(5.2\)](#) and [\(5.3\)](#) cannot be utilized in that form for our purposes for two reasons. First, the actual data flow of a composite service also needs to be taken into account rather than just profiles themselves. Second, [Condition \(5.2\)](#) and [\(5.3\)](#) happen to form a surjective relation. This may lead to ambiguous situations that result in false positive matches; it should be noted that this was also recognized and described similarly in [KFS09]. The consequence of surjectivity is that an input (output) in one profile might match more than one input (output) in another profile, which is illustrated by [Example 5.2](#) and further explained afterwards. Therefore, a stricter relation is needed. Not surprisingly, it turns out that a one-to-one correspondence (i.e., a bijective relation) effectively disambiguates these situations. This in turn raises the question whether a match generally implies a single, unique bijection. Unfortunately, the answer is negative. As we will see, there is a special case, illustrated by [Example 5.3](#), in which more than one such bijection exists rather than a single unique one. As a result, more information is needed in order to decide which one to choose. On the other hand, since we can precisely identify the root of this case, we can formulate syntactic restrictions under which it does not occur, in general.

---

**Example 5.2**

Suppose a data range inclusion  $int \sqsubseteq long$  and two profiles  $Pr^r$  and  $Pr^a$  having the following output sets:

$$Pr^r.O = \{U:int, V:long\} \quad Pr^a.O = \{X:int, Y:date\}$$

According to [Condition \(5.2\)](#),  $Pr^a$  is a plug-in match for  $Pr^r$  regarding  $O$  because  $U$  is matched by  $X$  and  $V$  is also matched by  $X$ ;  $Y$  is a spare output.

This may happen analogously for inputs. Suppose  $Pr^r$  and  $Pr^a$  have the following input sets:

$$Pr^r.I = \{U:int, V:date\} \quad Pr^a.I = \{X:int, Y:long\}$$

According to [Condition \(5.3\)](#),  $Pr^a$  is a match for  $Pr^r$  regarding  $I$  because  $X$  is matched by  $U$  and  $Y$  is also matched by  $U$ ;  $V$  is a spare input.

---

The problem in [Example 5.2](#) is that though the outputs of  $Pr^r$  are obviously not the *same* (otherwise there would be just one output), they are not sufficiently semantically differentiable since *int is a long*. In general, this problem is the result of the following cooccurrence:

1. A reflexive, transitive relation  $\bowtie$  is used (e.g.,  $\sqsubseteq$  or  $\equiv$ ) for pairwise matching profile parameters.
2. A parameter in one profile either directly or transitively  $\bowtie$ -relates to more than one parameter in the corresponding set of another profile.

That is, if there is a sequence

$$\text{type}(Pa^1) \bowtie \text{type}(Pa_1^2) \bowtie \dots \bowtie \text{type}(Pa_n^2) \quad (n \geq 2)$$

where  $Pa^1, Pa_1^2, \dots, Pa_n^2$  are parameters from the same kind of set (e.g.,  $I$  or  $O$ ) of two profiles  $Pr^1, Pr^2$ . For example, in [Example 5.2](#) we have

$$\underbrace{\text{type}(x)}_{Pr^a.O} \sqsubseteq \underbrace{\text{type}(U) \sqsubseteq \text{type}(v)}_{Pr^r.O}$$

on the outputs and

$$\underbrace{\text{type}(U)}_{Pr^r.I} \sqsubseteq \underbrace{\text{type}(x) \sqsubseteq \text{type}(Y)}_{Pr^a.I}$$

on the inputs. Observe that transitivity does actually not come into play in this example; it can be easily modified such that it involves transitivity, which we leave as an exercise.

Clearly, what is required is a one-to-one correspondence on matching profile parameters: every parameter that is used should have a *unique* correspondent. Formulated in mathematical terms, we want to find a bijection  $m: X_1 \rightarrow X_2$  where  $X_1, X_2$  are sets of matching profiles parameters. Such a bijection obviously does not exist for [Example 5.2](#); we can find two surjective mappings  $m_1: Pr^a.I \rightarrow Pr^r.I$  and  $m_2: Pr^r.O \rightarrow Pr^a.O$  instead.

### Example 5.3

Again, suppose a data range inclusion  $int \sqsubseteq long$  and two profiles  $Pr^r$  and  $Pr^a$  having the following output sets:

$$Pr^r.O = \{U:int, v:int\} \quad Pr^a.O = \{x:long, Y:long\}$$

Then there exist two different bijections  $m_1, m_2$  each satisfying [Condition \(5.2\)](#):

$$\begin{aligned} m_1: \quad & m_1(U) = x, \quad m_1(v) = Y \\ m_2: \quad & m_2(U) = Y, \quad m_2(v) = x . \end{aligned}$$

Again, this may happen analogously for inputs.

[Example 5.3](#) illustrates the case in which more than one bijection  $m$  exists, rather than a single one. The triggering cause here is the existence of a sequence of four related parameters, of which two are in each profile:

$$\text{type}(Pa_1^1) \bowtie \text{type}(Pa_2^1) \bowtie \text{type}(Pa_1^2) \bowtie \text{type}(Pa_2^2) .$$

It is not difficult to see that one can avoid this case by requiring that profile parameters within one of the different types of sets of a profile  $Pr$  are not related among each other; that is, if

$$\forall Pa_1, Pa_2 \in Pr.X \text{ and } Pa_1 \neq Pa_2: \quad \text{type}(Pa_1) \not\sqsubseteq \text{type}(Pa_2) \quad (5.8)$$

where  $X$  is, for instance, the set of inputs or outputs. This raises the question whether this restriction is too confining. We leave a discussion of this question to [Section 5.7.1](#).

Effects are incorporated as follows. Given effect atoms of the form defined by [Equation \(4.10\)](#), we require a one-to-one correspondence  $m$  analogous to inputs and outputs and based on the equivalence relation ( $\equiv$ ), thereby being in accordance with [Condition \(5.5\)](#). The rationale behind using “ $\equiv$ ” is that two services (or operations) should be considered equivalent only if they create the *same* effects. Since effect atoms of DL-based effect systems are expressed using concept names and role names, this indeed means that only the same concepts/roles qualify as equivalent (since “ $\equiv$ ” is reflexive).

Interestingly enough, the constraint expressed by [Equation \(5.8\)](#) is natural for effects. In fact, the case of a set of effects  $E$  that violates [Equation \(5.8\)](#) should be regarded a modeling fault because redundancy is modeled. This is easily seen: Suppose the set of effects  $E$  of some profile contains two effect atoms  $A_1(x)$  and  $A_2(x)$ . Suppose there is an inclusion  $A_1 \sqsubseteq A_2$  in the domain (which implies that  $A_2$  is a ramification of  $A_1$ ). Then  $A_2(x)$  is redundant in  $E$  as it is implied by  $A_1(x)$  anyway; hence, it should be removed from  $E$ . The consequence is that, given profiles that adhere to [Equation \(5.8\)](#), either a bijection exists between two sets of effects or none exists, but never more than one.

We are now ready to introduce our notion of functional profile equivalence.

**Definition 5.5** (Functional Profile Equivalence). *Let  $Sc = (id, Pr, \mathcal{U}, G_{cf}, G_{df})$  be a service,  $Pr^r \in (\bigcup_{u \in Sc.\mathcal{U}} u.Pr) \cup Sc.Pr$  a requested profile of  $Sc$ . Let  $X$  be a subset of the sources available w.r.t.  $Sc$ 's data flow  $G_{df}$  as an input for  $Pr^r$ .<sup>14</sup> Let  $Y$  be the subset of outputs  $Pr^r.O$  that are actually consumed w.r.t.  $Sc$ 's data flow  $G_{df}$ .<sup>15</sup> We define the binary relation  $\overset{\sim}{\sim}$  on profiles<sup>16</sup> by setting  $Pr^r \overset{\sim}{\sim} Pr^a$  iff there exist bijective mappings  $m_I : X \rightarrow I^a$ ,  $m_O : Y \rightarrow O^a$ , and  $m_E : E^r \rightarrow E^a$  such that*

- $\forall x \in X: type(x) \sqsubseteq_1 type(m_I(x))$ ,
- $\forall y \in Y: type(y) \supseteq type(m_O(y))$ , and
- $\forall \varphi \in E^r: \varphi \equiv m_E(\varphi)$ .

*If  $Pr^r \overset{\sim}{\sim} Pr^a$  then the advertised profile  $Pr^a$  is said to be functionally equivalent to  $Pr^r$  w.r.t.  $Sc$ 's data flow  $G_{df}$ .*

Observe that outputs of the requested profile  $Pr^r$  that are not consumed anywhere in the overall data flow of  $Sc$  can be ignored. Conversely, since the data flow is defined to provide a value for every input that a profile lists, such cases cannot exist for inputs.

Another interesting observation regarding effects is that the approach expressed by [Definition 5.5](#) is independent of their change semantics ascribed by the actual effect system. The reason is not found in a general commonality between different effect systems. What matters in this regard is that the same effect system is used for all effects applied to a KB. Only if different effect systems are used for applying effects to the same KB then they need to be mutually compatible, whatever compatibility precisely means in this case. What is more, the approach is also not restricted to effect systems whose effect

<sup>14</sup>Formally,  $X \subseteq \{x \mid x \in \mathbf{O} \text{ and } x \prec_{G_{cf}} i\}$  where  $i \in Pr^r.I$ ,  $\mathbf{O}$  is the set of sources of  $G_{df}$ , and  $x \prec_{G_{cf}} i$  is defined according to [Equation \(4.24\)](#).

<sup>15</sup>Formally,  $Y = \{y \mid y \in Pr^r.O \text{ and } \exists x.(x, y) \in \leftarrow\leftarrow\}$  where  $\leftarrow\leftarrow$  is the flow relation of  $G_{df}$ .

<sup>16</sup>The symbol  $\overset{\sim}{\sim}$  is chosen such that it indicates the relation's asymmetry.

expression language  $\mathcal{L}^{ES}$  is DL-based. In principle, any framework can be used that defines an appropriate and decidable semantic generalization (subsumption) relation and where mutual generalization describes equivalence.

Finally, there are two reasons that justify the absence of preconditions for functional equivalence. First, the inclusion of preconditions is more of a dictate than a contribution to the decision whether an advertised profile describes equivalent outcomes. Requiring a plug-in or exact match for preconditions reflects the assumption that all candidates have similar requirements in order to be operable, which might not be the case. It is easily conceivable that services/operations have (completely) different preconditions yet are functionally equivalent. Including preconditions would therefore dictate conditions, which is more of an unnecessary restriction. In fact, one can actually not know what conditions one should require from an advertised profile to be operable. Second, it is even not essential to include preconditions bearing in mind that they need to evaluate to true anyway in order to be executable. What matters is that preconditions are satisfied at commencement of execution of the service/operation they describe.

#### 5.4.4 Functional Equivalent Execution

Upon closer inspection of the criteria for functionally equivalent execution, we have found that one should distinguish between two types: *structure-aware* and *structure-nescient* equivalence. Both have in common that two executions yield equivalent outputs and effects. Structure-aware equivalence reflects the additional property that two executions have the same structure of the control flow. Search for a replacement therefore needs to be aware of the control flow structure. Conversely, structure-nescience reflects the property that two executions yield equivalent outputs and effects independent of the structure of the control flow. Replacement search therefore can be nescient of the structure. In other words, the former calls for a mechanism that takes into account the process level while the latter need not.

As an example for structure-awareness versus structure-nescience, we return once more to the *book seller* service from [Section 2.1](#). Suppose the *order & pay* service provided by the original book seller is a sequence of four operations: first, check whether the book is on stock; second, check the credit card for sufficient credit; third, place the order; finally, charge the book price to the credit card. A structure-aware replacement would have the property that it preserves the sequence and the order in which outputs and effects are created. In contrast, a structure-nescient replacement may come along with a changed structure. In the example, there might exist, for instance, a replacement in which the latter two operations (place order, charge credit card) are arranged in a parallel flow thereby deviating from the original sequential behavior regarding these two operations.

Clearly, structure-awareness imposes stricter constraints on the notion of functional equivalent execution. As will be seen, structure-nescience is strictly more general than structure-awareness. The main reason for distinguishing between both types is that simple algorithmic processing is sufficient for automatically finding a structure-aware replacement, whereas finding a structure-nescient replacement is reduced to solving a planning problem, which is more involved. Both types are introduced in detail next.

## Structure-aware Functional Equivalent Execution

The general principle regarding a structure-aware equivalent execution is simple. Given a subflow  $G_{cf}^e$ , a replacement  $G_{cf}^r$  qualifies as such if

1.  $G_{cf}^r$  preserves the control flow of  $G_{cf}^e$  and
2. for each ordinary transition  $t$  in  $G_{cf}^e$  with its associated profile there is a functionally equivalent profile for the corresponding transition in  $G_{cf}^r$ .

Consequently, this type is characterized by the inclusion of a process' syntactic structure (as represented by its control flow) in the notion of equivalent execution.

Stating the first item in formally, the control flow is preserved if  $G_{cf}^e$  and  $G_{cf}^r$  are the *same* except for their mappings  $fu^e$  and  $fu^r$ . This directly explains what we mean by "corresponding transition" in the second item.

**Definition 5.6** (Structure-aware Functional Equivalent Execution). *Given a service  $Sc$ , let  $G_{cf}$  be its unfolded control flow graph and let  $G_{cf}^e, G_{cf}^r$  be unfolded control flow graphs such that (i)  $G_{cf}^e \trianglelefteq G_{cf}$  and (ii)  $G_{cf}^e$  and  $G_{cf}^r$  coincide in their set of places, transitions, and the flow relation. Then the execution of  $G_{cf}^r$  is said to be functionally equivalent to  $G_{cf}^e$  iff for each ordinary transition  $t \in \mathbf{T}^{e,r}$ :  $fu^e(t).Pr \overset{\sim}{\approx} fu^r(t).Pr$ .*

As a direct consequence of **Definition 5.6**, if  $G_{cf}^e, G_{cf}^r$  are functionally equivalent then not only the control flow structure is preserved but also the data flow structure. All inputs required within  $G_{cf}^r$  are available in a compatible way and all outputs from  $G_{cf}^e$  that are subsequently consumed are created in a compatible way by  $G_{cf}^r$ . This is a consequence of the mutual matches on elements in  $IO$  as required by the relation " $\overset{\sim}{\approx}$ ". The only missing piece for seamless replaceability is that all sources outside of  $G_{cf}^r$  that are connected to a sink inside  $G_{cf}^r$  need to be execution compatible (see **Definition 4.16**), which requires additional checks at the actual grounding level of operations.

It is easily seen that finding a structure-aware functional equivalent replacement reduces to rebinding each ordinary transition, which in turn can be reduced to solving a matchmaking problem for each transition. Therefore, a simple algorithm such as **Algorithm 1** is sufficient to realize structure-aware replacements. The algorithm iterates over all ordinary transitions in  $G_{cf}^e$ , updates the binding of transitions, and modifies the data flow according to replacements found. Searching and selecting matches is delegated to the algorithm *FindMatch*. As can be seen, we assume that *FindMatch* is parametrized with the relation capturing functional profile equivalence, " $\overset{\sim}{\approx}$ " in this case, and a preference relation " $\geq$ " for selecting the most preferred match. Obviously, the preference relation is not an ultimate requirement from a functional point of view. In **Section 5.4.5** we provide a way how the relation can be defined such that it incorporates non-functional properties based on a quantitative cost model. More details on how *FindMatch* can be implemented follow in **Chapter 7**.

If  $\mathcal{O}(f)$  is the asymptotic growth rate of *FindMatch* then the combined rate is  $\mathcal{O}(nf)$  where  $n$  is the number of ordinary transitions in  $G_{cf}^e$ ; note that  $n$  is not a parameter of  $f$ .

Finally, structure-aware replacements are applicable without restrictions on  $G_{cf}^e$ , meaning that  $G_{cf}^e$  can in principle be any subflow of a control flow  $G_{cf}$ . This is one



**Input:** service  $Sc = (id, Pr, \mathcal{U}, G_{cf}, G_{df})$ , subflow  $G_{cf}^e = (\mathbf{P}^e, \mathbf{T}^e, \mathbf{F}^e, M_0, fu^e)$  so that  $G_{cf}^e \preceq G_{cf}$ , knowledge base  $\mathcal{K}$ , set of advertised functional units  $\mathcal{U}^a$

- 1: **for each**  $t \in \mathbf{T}^e$  and  $t$  an ordinary transition **do**
- 2:      $u^e := fu^e(t)$ ,  $u^r := FindMatch(u^e, \mathcal{K}, \mathcal{U}^a, \lesssim, \gtrsim)$
- 3:      $\mathcal{U} := (\mathcal{U} \cup u^r) \setminus u^e$
- 4:     update  $fu$  so that it maps  $t$  to  $u^r$
- 5:     **for each**  $i^e \in u^e.Pr.I$  **do**
- 6:         update  $\leftarrow$  of  $G_{df}$  by replacing  $(i^e, o)$  with  $(m_I(i^e), o)$
- 7:     **end for**
- 8:     **for each**  $o^r \in u^r.Pr.O$  **do**
- 9:         update  $\leftarrow$  of  $G_{df}$  by replacing each pair  $(i, m_O(o^r))$  with  $(i, o^r)$
- 10:     **end for**
- 11: **end for**

Algorithm 1:  $Rebind(Sc, G_{cf}^e, \mathcal{K}, \mathcal{U}^a)$  where  $FindMatch(u, \mathcal{K}, \mathcal{U}^a, \lesssim, \gtrsim)$  is an implementation of preference-based matchmaking in the domain  $MD = (\mathcal{K}, \mathcal{P}, \lesssim, \gtrsim, \models)$  returning the most preferred matching functional unit for a requested functional unit  $u$ . Note that we tacitly assume a bijection between  $\mathcal{U}^a$  and  $\mathcal{P}$  (i.e., we can access the profile of each functional unit).

difference to structure-nescent replacements. Identifying the extent of  $G_{cf}^e$  is mainly a matter of dependencies between the functional units in the overall control flow. In practice it might often be the case that one just wants a 1:1 replacement such as replacing an atomic service as a result of an invocation failure by an alternative atomic service. This requires, of course, the possibility of replacing it independently of the rest. An example is the *identify person* atomic service that can be replaced independently of the other services in the overall *emergency assistance* service. The second typical case are  $n:n$  replacements of  $n$  dependent transitions; that is, where the need to replace one transition (due to a failure) implies the need to replace other transitions as well. An example here is the *select ambulance* operation that implies replacing also *trigger ambulance*.

### Structure-nescent Functional Equivalent Execution

If it is the outcome of an execution that matters foremost and behavior to achieve it comes second then one might as well tolerate different execution behavior in addition to the use of alternative services/operations. This is the underlying idea pursued by structure-nescent functional equivalent execution. The vantage gained is the possibility to formulate the notion of equivalent execution in a more general way by abstracting from the control flow; hence, the data flow as well. In particular, we want to formulate it such that it translates directly to a *planning problem*<sup>17</sup>: compose a terminating execution that finally satisfies a pre-defined *goal*, which represents the final outcome.

<sup>17</sup>A closely related alternative is translation into a *projection problem* [Rei01], which is a standard reasoning task in formal action theories: what holds after an action or a sequence of actions has been executed? Projection has been, to a large extent, studied independently from planning formalisms.

In our setting, the problem formulates as follows. Given a subflow  $G_{cf}^e$  whose execution yields a set of final outputs  $O$  and effects  $E$ , is there a replacement  $G_{cf}^r$  whose execution yields equivalent sets  $O$  and  $E$ . The goal (or target state) is therefore described in terms of  $O$  and  $E$ . A replacement  $G_{cf}^r$  is consequently predetermined only in terms of the target state and is otherwise up to be found (composed) from available services/-operations in the domain. What is more,  $G_{cf}^r$  should be satisfied with the same set of outputs (sources) as available for  $G_{cf}^e$ .

The aspect that remains to be clarified for a precise characterization is the extent of  $G_{cf}^e$ . By adopting the classic principle of planning in which the goal reflects the target state at the end of a plan execution, the final place of  $G_{cf}^e$  (hence,  $G_{cf}^r$ ) coincides either with the final place of the overall control flow  $G_{cf}$  or the final place of a sub service's control flow (if any). For instance, in the *emergency assistance* service there are two possibilities (cf. [Figure 4.5](#)): either the place after transition  $t_4$ , which reflects the end of the *trigger ambulance* sub service, or  $p_f$ , which reflects the end of *emergency assistance* itself. A subflow for which a replacement is to be found (composed) thereby always extends to a particular final place rather than to some reachable place. The reason is that this allows to take the profile of the (sub) service that ends with the final place as the specification of  $O$  and  $E$ . This is not to say that it is not possible to relax this. However, the main consequence of allowing  $G_{cf}^e$  to extend to some place  $p$  is the need for deriving the goal state absolute to  $p$  from the operation effects, which is obviously a matter of the effect semantics as defined by the actual effect system used; hence, a general mechanism of how to derive the goal state does not seem an easy undertaking if at all possible.

The following definition formally introduces the notion of structure-nescent functionally equivalent execution.

**Definition 5.7** (Structure-nescent Functional Equivalent Execution). *Given a service  $Sc = (id, Pr, \mathcal{U}, G_{cf}, G_{df})$ , let  $G_{cf}^e$  be a subflow  $G_{cf}^e \trianglelefteq G_{cf}$  such that  $p_f^e$  either coincides with the final place  $p_f$  of  $Sc$  or the final place  $p_f^u$  of a sub service  $Sc^u \in Sc.\mathcal{U}$  (if any); that is, either  $p_f^e = p_f$  or  $p_f^e = p_f^u$ . Let  $E$  be either  $Sc.Pr.E$  or  $Sc^u.Pr.E$  depending on whether  $p_f^e = p_f$  or  $p_f^e = p_f^u$ . Analogously, let  $O$  be either  $Sc.Pr.O$  or the subset of outputs  $Sc^u.Pr.O$  that are actually consumed w.r.t.  $Sc$ 's data flow  $G_{df}$  (cf.  $Y$  in [Definition 5.5](#)).  $E$  and  $O$  are called the goal effects and outputs, respectively. Finally, let  $s = (M, \mathcal{K})$  be the execution state such that  $M(p_f^e) = 1$ . Then the execution of a control flow graph  $G_{cf}^r$  is said to be functionally equivalent to  $G_{cf}^e$  iff*

- (1)  $G_{cf}^e$  can be structurally substituted by  $G_{cf}^r$ ,
- (2)  $\forall \varphi \in E: \mathcal{K} \models \varphi$  no matter whether  $G_{cf}^e$  or  $G_{cf}^r$  has reached marking  $M$ ,
- (3) there is a bijection  $m_O: O \rightarrow O'$  such that  $\forall o \in O: \text{type}(o) \sqsubseteq \text{type}(m_O(o))$  and where  $O'$  is the set of outputs produced in state  $s$  as the result of execution of  $G_{cf}^r$ .

Searching for a replacement under [Definition 5.7](#) is formulated as a planning problem along the lines of the STRIPS framework (see [Section 5.4.2](#)), yet there are considerable differences. A solution to a so-formulated planning problem – a plan – then constitutes a replacement. We put together a DL-based formulation inspired by [Mil08]

and message-based planning [HBHP09]. The latter considers the presence of domain constraints (called integrity constraints by the authors) that are, however, in the form of clauses that may contain universally quantified variables.<sup>18</sup> The main difference to our planning scheme is that we consider the effect system (ES1); hence, while there can also be domain constraints they are axiomatized as a TBox.<sup>19</sup> More specifically, the way it is defined features:

- (1) incomplete information (i.e., conformant planning under OWA)<sup>20</sup> and adoption of the UNA;
- (2) deterministic effect semantics defined as a belief update over an ABox w.r.t. a TBox;
- (3) message-based planning including on-the-fly creation of new individuals/values during planning;
- (4) goals may contain existentially quantified variables.

While the first item should be clear, the latter three deserve further explanation.

(2) Rather than modeling operations as nondeterministic to accommodate for execution errors, we do regard them as deterministic (see Section 4.1.1) modeling their *expected* (or normal) functionality. This spares us the cost of contingency planning. In this sense, a service is executed optimistically in the hope that it succeeds and only in case of a failure event the system reacts, which is possible since execution is monitored based on an eventually perfect failure detector (see Section 5.2.1).

Furthermore, Item (2) concerns the actual effect system used. As pointed out in [HBHP09], most planning-based service composition frameworks either (i) ignore the correspondence between the change semantics of services/operations and the belief update problem, (ii) make simplifying assumptions, or (iii) do not specify exactly how effects are applied, let alone conflict resolution. We will apply the DL-based effect system (ES1) introduced in Section 4.2.2.<sup>21</sup> An important observation is that for monotonic DLs an effect update (i.e., the update to the world state) can disregard conflict resolution as a heuristic, meaning that planning proceeds as if effects are never inconsistent with the ABox. Conflict resolution is thereby deferred until execution of a plan. However, effects that imply ramifications as entailed by the TBox are taken into account for planning. We will come back to this point later as we can precisely describe cases in which disregarding conflict resolution does not pose a problem.

<sup>18</sup>A clause therein is of the form  $\forall x_1, \dots, \forall x_k (l_1 \vee \dots \vee l_m)$  where  $x_1, \dots, x_k$  are the universally quantified variables and  $l_1, \dots, l_m$  are literals; that is, possibly negated  $n$ -ary predicates over constants and the variables  $x_1, \dots, x_k$ .

<sup>19</sup>It should be noted that there is an intersection between how the domain is represented in [HBHP09] and our DL-based planning scheme. Namely, the world state representations are conceptually close and an inclusion  $A \sqsubseteq B$  translates to a clause  $\forall x (\neg A(x) \vee B(x))$ . However, differences lie in the arity of predicates allowed (i.e., the arity in DLs is generally not larger than 2) and the constructors allowed (e.g.,  $A \sqsubseteq \exists R.C$  is not expressible in the form of a clause as it is defined in [HBHP09]).

<sup>20</sup>It follows from the state space planning model that it can be applied equally under CWA.

<sup>21</sup>Similar effect systems may as well be used. It follows from [Mil08] that this includes, for instance, effect systems that rely on the propositionally closed fragments of the DLs  $\mathcal{ALC}$  up to  $\mathcal{ALCQIO}$  and that use model-based update semantics described therein.

(3) Message-based planning essentially “ensures that the inputs of each service  $w$  [including operations in our setting] can always be provided by  $w$ ’s predecessors” [HBHP09]. What is more, it is recognized for some time [McD02] that introducing new objects on-the-fly at planning time is relevant especially in the area of service composition: it is natural that services/operations can have outputs, which makes them *information-providing*. More specifically, the information introduced by outputs corresponds to new planning objects. As the information was not known before and does not change the state of affairs, outputs are also referred to as *knowledge effects* (e.g., [Sir06]). Yet it is custom in planning that the set of planning objects  $\mathcal{C}$  is known a priori and fixed throughout planning. This has the consequence that services/operations would not be allowed to have outputs, as every output usually introduces a new constant. For instance, consider the *find book* service from Section 2.1. This service is information-providing: the ISBN number that it outputs is such a new object introduced as the result of its execution. Otherwise, we would need to assume that the ISBN number is already a member of  $\mathcal{C}$ , which implies that virtually all ISBN numbers need to be known – an assumption that is clearly not realistic in this case. A strategy that has been proposed to circumvent this is to execute information-providing actions before planning in order to gather additional constants and add them to  $\mathcal{C}$ . This strategy is, however, not practicable for operations that have inputs as each operation would need to be executed with each valid combination of inputs in order to gather all outputs, which can easily escalate in the number of executions that would need to be done.<sup>22</sup> Therefore, in accordance with [HBHP09], we assume that  $\mathcal{C}$  can grow in the process of planning. As pointed out therein, however, this also has a downside that might appear somewhat frightening at first: the plan existence problem for plans unbounded in length becomes undecidable, while decidability is retained for polynomially bounded plans. Notwithstanding the limitation, putting it into perspective makes it little intimidating: First, as discussed in Section 5.4.2, the use of planning normally goes along with the assumption that planning domains are such that a plan exists for a given planning problem and that it is more relevant to find it rather than to prove that there is none. Second, the intuition of composite services is that they are rather short, so are plans rather short.

(4) Variables in goals serve as placeholders that can be instantiated from the constants  $\mathcal{C}$  and therefore enable determining actual objects in the course of planning rather than a priori. Consider for instance the goal  $ambulanceTransport(x, ALICE)$  stating that there is an ambulance transport for Alice by an ambulance  $x$ . Determining which concrete ambulance this will be is thereby left open until planning time. Clearly, this involves the assumption that there *is* an ambulance represented by an individual name in  $\mathcal{C}$ , no matter whether pre-existing or learned on-the-fly (i.e., during planning). Observe that variables are thereby implicitly existentially quantified. It should be noted, however, that the possibility of using variables in goals constitutes no more expressive power. As pointed out in [GNT04, Section 2.4.1], they can be compiled away. This is achieved by adding a pseudo operation having the goal as its precondition and a unique grounded effect that is taken as the new goal instead.

<sup>22</sup>An operation that has  $n \geq 1$  inputs that can each be instantiated from a set of values with  $m_i$  elements results in  $\prod_{i=1}^n m_i$  combinations.

*A Planning Problem* for a replacement comprises five elements, which are informally described as follows. First, a set of initial constants containing at least the primary representatives of all outputs (sources) that have been produced already in the course of execution. Next, the TBox and ABox of a KB taken as a snapshot in some state at the moment execution was paused. The ABox makes up the initial planning state. Fourth, a set of functional units available in the domain. Finally, the goal that represents the effects in the profile that has been chosen as the final place. As will be seen, outputs are not directly part of the goal and are indirectly represented via the set of constants.

**Definition 5.8** (Replacement Planning Problem). *A planning problem for a replacement is a 5-tuple  $PP = (\mathcal{C}_0, \mathcal{T}, \mathcal{U}, \mathcal{A}, \Gamma)$  where*

- $\mathcal{C}_0$  is the set of initial constants (individuals names and lexical forms),
- $\mathcal{T}$  is a TBox such that those inclusions in  $\mathcal{T}$  obey the restrictions of (ES1) that involve a concept name  $A$  (role name  $R$ ) and where  $A$  ( $R$ ) occurs as an effect within the profile of a functional unit in  $\mathcal{U}$ ,
- $\mathcal{U}$  is a set of functional units<sup>23</sup>,
- $\mathcal{A}$  – the initial state – is an ABox based on  $\mathcal{C}_0$ , and
- $\Gamma$  – the goal – is a set of atoms in the form as given by Equation (4.10).

It is assumed that individual names and lexical forms occurring in the initial ABox  $\mathcal{A}$  are all in  $\mathcal{C}_0$ , which is expressed by the wording “based on  $\mathcal{C}_0$ ”.

The elements in Definition 5.8 are precisely as follows. Let  $s_c$  be a service instance,  $G_{cf}$  and  $G_{df}$  its control and data flow, respectively,  $t$  an ordinary transition of  $G_{cf}$ ,  $\mathbf{O}$  the sources of  $G_{df}$  according to Definition 4.15, and for  $i$  a sink such that  $pt(i) = t$  according to Equation (4.23), let  $\mathbf{o} = \{o \in \mathbf{O} \mid o \prec_{G_{cf}} i\}$  be the set of outputs preceding  $i$  (i.e., all outputs that  $i$  can possibly be connected to regardless of whether they are execution compatible). Then

$$\mathcal{C}_0 \supseteq \left( \bigcup_{o \in \mathbf{o}} Re[1](o) \right). \quad (5.9)$$

In addition, all constants in  $\mathcal{C}_i$  are *typed*, which we denote with  $type(c)$  for  $c \in \mathcal{C}_i$ . In case of the constants on the right-hand side of Equation (5.9) this is the type of the source (i.e., the type of the profile parameter). The type of all other initial constants  $c$  not in the set on the right-hand side is either a concept  $C$  of an assertion  $C(c)$  in  $\mathcal{A}$  if  $c$  is an individual (i.e.,  $type(c) = C$ ) or a data range  $dr$  if  $c$  is a lexical form (i.e.,  $type(c) = dr$ ).<sup>24</sup> One can therefore also see  $\mathcal{C}$  as a kind of a ABox containing knowledge effects (recall that outputs can be seen as knowledge effects). The main reason for representing them separately in  $\mathcal{C}$  rather than also in  $\mathcal{A}$  is that consistency conflicts (calling for a conflict resolution strategy) are, intrinsically, impossible since an output would match a so-called *forward effect* to which we return later. Another reason is that outputs do obviously not contribute

<sup>23</sup>As before, we tacitly assume the possibility to access the profile of each functional unit.

<sup>24</sup>This is similar to a typed RDF literal (e.g.,  $1.0 \hat{\wedge} xsd:float; c = 1.0$  and  $dr = xsd:float$ ).

to the world state, but rather to knowledge about what objects exist. Next,  $\mathcal{T}$  and in particular  $\mathcal{A}$  are the components of the KB  $\mathcal{K}$  taken as a snapshot from *some* global state  $\hat{s} = (\hat{M}, \mathcal{K})$  – preferably the most recent – such that  $\hat{M}$  contains  $G_{cf}$ 's marking  $M$  with  $M(t) = 1$ .  $\mathcal{U}$  is the set of functional units available in the domain, but  $fu(t) \notin \mathcal{U}$ . Consequently,  $\mathcal{U}$  may contain both operations and possibly complex services. The latter are treated equally to operations, meaning that planning abstracts from their structure by regarding them as indivisible black boxes via their profile. Finally, the goal  $\Gamma$  is the set of goal effects in [Definition 5.7](#) where some atoms may be instantiated partially from constants in  $\mathcal{C}_0$ . All remaining variables in  $\Gamma$  are instantiated in a final goal state from the final set of constants  $\mathcal{C}_\Gamma$ .

Not surprisingly, the states of the planning space and execution states are conceptually close. A planning state  $s$  herein is a pair  $s = (\mathcal{C}, \mathcal{K})$ . Including the marking  $M$  is not required at least from a planning point of view as it captures execution semantics. We shall therefore assume that it is implicit and functions as introduced, but for ease of notation we will include it subsequently only when necessary. In the initial planning state  $s_0$ ,  $\mathcal{K}$  is taken from the execution state as just described. Subsequent evolvment of  $\mathcal{K}$  in the process of planning is assumed to take place in a *detached* way, meaning that other changes made to  $\mathcal{K}$  in the background due to dynamics in the domain are not visible. We do not consider the combined evolvment of  $\mathcal{K}$  in this work. This is a topic for future work.

A functional unit  $u \in \mathcal{U}$  having the profile  $Pr$  is applicable in a plan state  $s = (\mathcal{C}, \mathcal{K})$  iff

1. for each  $i \in Pr.I$  there is a constant  $c \in \mathcal{C}$  such that  $type(c) \sqsubseteq type(i)$  and
2.  $f_{chk}(\mathcal{K}, Pr.P) = \text{true}$  (see [Equation \(4.19\)](#)).

Clearly, the first condition ensures concept compatibility in the data flow, but it does not necessarily ensure data compatibility; hence, not necessarily seamless execution compatibility as stated by [Definition 4.16](#). Consequently, an additional check is required at the technical grounding level.<sup>25</sup>

Applying  $u$  in plan state  $s = (\mathcal{C}, \mathcal{K})$  results in a new state  $s' = (\mathcal{C}', \mathcal{K}')$  such that the KB changes in the obvious way:

$$\mathcal{K} \Longrightarrow_U \mathcal{K}' \quad \text{where} \quad U = f_{up}(\mathcal{K}, Pr.E) .$$

The new set of constants is:

$$\mathcal{C}' = \mathcal{C} \cup \{c_1, \dots, c_n\}$$

such that for each  $o_i \in Pr.O$  there is a constant  $c_i = Re[1](o_i)$  (hence,  $n = |Pr.O|$ ) whose type is  $type(c_i) = type(o_i)$ . This implies that the constants  $c_i$  are pairwise different as it does not make sense to output the same constant twice. Furthermore, the fact that the constants  $c_i$  are assumed to represent newly introduced planning objects in the sense that they did not exist before implies  $\mathcal{C} \cap \{c_1, \dots, c_n\} = \emptyset$ . More subtly, an individual

<sup>25</sup>The only planning-based service composition framework we are aware of that includes such data level checks is [KG05].

denoted by a newly introduced constant did actually exist previously but were anonymous; hence, was not known, in a sense. This is a consequence of the assumption that the interpretation domain  $\Delta^{\mathcal{I}}$  is invariant. Recall, for (ES1), evolving interpretations  $\mathcal{I}, \mathcal{I}'$  share the same domains  $\Delta^{\mathcal{I}} = \Delta^{\mathcal{I}'}$ . Similarly, newly introduced lexical forms are new in the sense that they were just not used as planning objects before. Recall, also the datatype map  $\mathcal{D}$  does not change for evolving interpretations. Another subtlety is that if two new constants  $c_1, c_2$  are introduced such that  $type(c_1) \sqsubseteq type(c_2)$  then we might face the same kind of ambiguities described in Section 5.4.3.

A control flow graph  $G_{cf}^r$  is a solution to a so formulated planning problem if the application of all transitions according to its process semantics leads to a state  $s_{\Gamma} = (\mathcal{C}_{\Gamma}, \mathcal{K}_{\Gamma})$  in the final place such that  $\mathcal{K}_{\Gamma}$  satisfies all effects according to Item (2) in Definition 5.7 and there is a subset  $O' \subseteq \mathcal{C}_{\Gamma}$  satisfying Item (3) in Definition 5.7. Observe that this formulation is rather powerful as every kind of process would be possible. Many planning tools, however, synthesize sequential plans thereby restricting replacements to be sequential flows.

*Ways to Reduce Complexity.* Planning under the just introduced scheme is rather challenging. The main reason is the complexity introduced by modeling the change semantics as a query answering problem (preconditions) and a belief update problem (effects) as opposed to the more benign STRIPS semantics that is realized, respectively, by set inclusion and the add and delete lists (see Equation (5.6) and (5.7)). There are, however, ways to trade modeling expressivity for reduced complexity, which can be classified in two orthogonal categories:

- Avoid the possibility of conflicts, thereby, removing the need for conflict resolution strategies in order to preserve KB consistency.
- Restrictions that make the belief update problem modulo conflict resolution equally simple to propositional STRIPS effect semantics.

For DL-based effect systems in which effects are of the form as defined by Equation (4.10), if the TBox is empty or restricted to subsumption hierarchies over concept/role names<sup>26</sup> then the belief update problem modulo conflict resolution becomes equivalent to propositional STRIPS effect semantics. More precisely, for a possibly negated effect  $A(x)$  or  $R(x, y)$ , every inclusion in the TBox where  $A$  ( $R$ ) transitively occurs on the left-hand side is restricted to a concept name (role name) on the right-hand side. These kind of inclusions describe “straightforward” ramifications that, when taken into account, preserve the propositional character of STRIPS. To illustrate this, imagine a positive effect  $A(x)$ . Under STRIPS semantics  $A(x)$  would be in the add list; analogously, a negative effect  $\neg A(x)$  would be in the delete list. If there is an inclusion  $A \sqsubseteq B$  in the TBox where  $B$  is a concept name then this implies that the add (delete) list is simply extended (implicitly) by an effect  $B(x)$ . Analogously for roles. In contrast, if there is an inclusion  $A \sqsubseteq C$  in the TBox where  $C$  is a complex concept, say  $\exists R.\top$ , then we also have  $C(x)$  in the add list. The important difference is that an instantiation  $C(a)$  cannot be seen as a proposition whereas  $B(a)$  can; hence, the propositional machinery

<sup>26</sup>The presence of such subsumption hierarchies for service composition has been considered, for instance, in [CFB04].

sufficient for STRIPS is no longer sufficient in this case. On the other hand, as we have illustrated in [Example 4.2](#), the use of the existential concept constructor  $\exists R.C$  introduces a higher degree of nondeterminism. Since existential restriction is the only constructor available in  $DL-Lite_{\mathcal{FR}}^{++}$  for which instantiated effects cannot be seen as propositions, we conclude that (ES1) without them can be covered by propositional STRIPS effect semantics. Observe that the role axioms (Fun( $R$ ) and so on) available in  $DL-Lite_{\mathcal{FR}}^{++}$  are also not a problem since they do not construct complex roles (such as role composition  $R \circ S$ ).

Avoiding the possibility of conflicts can be achieved by the syntactic restriction on so-called *forward effects* [HBHP09]. Intuitively, a forward effect describes a change in the domain about which no information was previously represented in the KB. A forward effect therefore cannot, intrinsically, conflict with the existing world state representation in the KB. The notion of forward effects as introduced in [HBHP09] can be transferred easily to DL-based effect systems and our service model.

**Definition 5.9** (DL Forward Effect). *A possibly negated atom  $A(x)$  or  $R(x, y)$  is a forward effect if at least one of the variables  $x, y$  refers to a representative of an output.*<sup>27</sup>

[Definition 5.9](#) implies that an instantiated effect  $A(a)$  or  $R(a, b)$  involves at least one new constant and hence it cannot be inconsistent with the existing ABox. To explain this further we consider the effect system (ES1). Recap the possible conflict situations depicted in [Table 4.1](#). It is not difficult to see that in the presence of forward effects a conflicting combination can actually only occur if effects are inconsistent among each other and w.r.t. the TBox, which is not sensible anyway as it would be a modeling fault. As an example, let us take the combination where  $R$  is symmetric and there are assertions  $R(a, b)$  and  $\neg R(a, b)$ . Because  $a$  and/or  $b$  is new, neither of the assertions may have been already in the ABox. Hence, both must be effects, but they are inconsistent among each other w.r.t.  $\text{Sym}(R)$ .

The main downside of the restriction on forward effects is that one cannot express changes relating exclusively to pre-existing objects. For instance, one cannot model an operation *clear mission* with an effect  $\neg \text{assignedTo}(x, y)$  that shall represent the clearing of an ambulance  $x$  from the mission  $y$  and where  $x, y$  refer to pre-existing individuals. While the limited modeling power of forward effects might not pose a problem in some domains, we see it as a substantial restriction. Fortunately, effect system (ES1) has the property that if there is a conflict then it can be uniquely resolved, which makes planning under conflicts predictable.

An interesting possibility we see as a heuristic to circumvent the intricacy of conflict resolution for planning is to simply ignore it. Conflict resolution is thereby deferred until execution. This is supported by the following observation, which is a consequence of [Observation 3.1](#).

**Observation 5.1.** *In monotonic DLs and regardless of conflict resolution strategies defined by the actual effect system used, it is sufficient to delete assertions from the ABox in order to resolve a KB consistency conflict.*

<sup>27</sup>It should be noted that the definition assumes adoption of the UNA. Defining DL forward effects in the absence of UNA is possible but requires being more explicit about the interpretation of names.



This implies that conflict resolution does not involve adding assertions, at least not for preserving consistency.<sup>28</sup> Now, one of the most commonly used search heuristics in STRIPS-style planning, called *delete-relaxation*, is obtained by removing the delete lists of actions [BG01]. Consequently, ignoring the additional deletes of conflict resolution is an instance of this heuristic. We leave an investigation on ways to exploit this further as related future work since identifying or improving planning techniques is not a goal of this thesis.

### 5.4.5 Similar Execution and Non-functional Properties

Thus far we have concentrated on the functional dimension of service semantics. The two notions that we have introduced aim at equivalent execution and are therefore inevitably fairly strict and leave limited room for similarity, apart from the plug in match on inputs, outputs and the control flow nescience of the planning-based technique. In this section we sketch a basic approach to broaden the notion of equivalent execution towards similar execution. Conceptually, we see a close connection with *net benefit* problem solving (see Section 5.4.2). We will therefore follow this method, which means that it will also be reduced to a planning problem. We will again consider effect system (ES1). For the reason of higher level nondeterminism introduced by the existential restriction constructor, we will also assume that no effect is defined (indirectly) using  $\exists R.C$ . As discussed before, the belief update problem modulo conflict resolution described by (ES1) thus becomes equivalent to STRIPS effect semantics; hence, it becomes possible to use STRIPS machinery. However, applicability is still controlled by the actual precondition system used (i.e., applicability need not be defined in terms of STRIPS action applicability). For instance, if (PS1) is used then applicability reduces to conjunctive query answering. This constitutes a kind of hybrid planning scheme.

The formulation as a net benefit problem also allows us to add yet another layer of flexibility by including non-functional properties in this broader notion of similar execution and hence for replacement search. The main motivation is to optimize or at least enable comparing (ranking) the declared quality of service (QoS)<sup>29</sup> between replacements according to user *preferences*.

As a central step, two functions are introduced. A cost function that associates an operation with a value that represents its execution cost. A utility function that assigns an utility value to preferences. We will use a model where costs are a quantitative measure of quality – the lower the costs the higher the quality. Preferences take the place of goals and do either directly or in an extended way accommodate the goals  $\Gamma$ . If  $\mathcal{U}$  is a set

<sup>28</sup>It should be added that more advanced resolution strategies might make additions as part of higher-order conflict resolution (e.g., Example 4.2), but this is motivated from a higher application-specific level.

<sup>29</sup>One should keep in mind that QoS properties can be interpreted in a spectrum of guarantee levels. At one end is the firm commitment where the user is guaranteed a certain quality of service set by a non-functional property, which requires that the service level can be enforced accordingly by the underlying system. Towards the other end are soft guarantees in case the underlying system acts in a best effort manner. The (technical) aspects of service level negotiation and enforcement are outside the scope of this thesis.

of operations and  $\mathcal{F}$  is a set of preferences, the details of which are yet to be described, then the two functions are

$$f_{\text{cost}}: \mathcal{U} \rightarrow \mathbb{R}_0^+ \text{ and} \quad (5.10)$$

$$f_{\text{util}}: \mathcal{F} \rightarrow \mathbb{R}^+ . \quad (5.11)$$

Observe that  $f_{\text{cost}}$  models independent costs. Operation's costs have no interdependencies and are therefore additive. This holds equally for  $f_{\text{util}}$ . An extension to conditional costs (e.g., the cost of some operation is different depending on context such as whether another operation has been executed before, after, not at all) or conditional utility (e.g., the utility of some preference depends on whether another one has been achieved already, not achieved yet) is straightforward, at least from a conceptual point of view, but can easily lead to a non-linear optimization problem. In a similar though more benign way, the two functions can easily be revised to allow for negative values. While positive costs are expenses, negative costs are revenues. Conversely, a positive utility represents something sought, while negative utility stands for something to be avoided.

Next, we define the aggregated execution cost of a replacement  $G_{\text{cf}}^r$ . By overloading the cost function  $f_{\text{cost}}$  for replacements, aggregated costs are given by

$$f_{\text{cost}}(G_{\text{cf}}^r) = \sum_{Op \in \mathcal{U}} f_{\text{cost}}(Op) \quad (5.12)$$

where  $\mathcal{U}$  is the set of operations associated to ordinary transitions in  $G_{\text{cf}}^r$  (i.e., the codomain of  $f_{\text{cost}}$  minus no-op operations, if any). Considering the fact that in the most general case  $G_{\text{cf}}^r$  allows for several types of control flows, Equation (5.12) becomes an estimate in two cases. The result is a pessimistic upper bound regarding choice since costs of all choice paths are included. The result is an optimistic lower bound regarding iteration since potential repetitions are not included. Clearly, real costs cannot be known unless choices in the control flow are completely determined. Notice that Equation (5.12) can be used regardless of whether  $G_{\text{cf}}^r$  is unfolded or not since unfolding adds to the sum.

The objective is then to maximize the overall utility – the net benefit – of a replacement  $G_{\text{cf}}^r$  while taking into account its overall execution costs. By also overloading  $f_{\text{util}}$  the objective function is

$$f_{\text{util}}(G_{\text{cf}}^r) = \left( \sum_{\varphi \in \mathcal{F}} f_{\text{util}}(\varphi) \right) - f_{\text{cost}}(G_{\text{cf}}^r) \quad (5.13)$$

Clearly, a replacement  $G_{\text{cf}}^r$  is optimal if there is no other replacement  $G_{\text{cf}}^{r'}$  that has utility  $f_{\text{util}}(G_{\text{cf}}^{r'})$  higher than  $f_{\text{util}}(G_{\text{cf}}^r)$ . How  $f_{\text{util}}$  and  $f_{\text{cost}}$  are precisely defined over preferences and operations, respectively, is detailed in the following two subsections.

## Execution Costs

Execution costs of operations are defined over non-functional properties (NFPs) in the following way. Suppose there is a QoS ontology  $\mathcal{O}$  (e.g., [TTM09]) defining  $n$  named

QoS concepts (which are either general or domain specific). NFPs occurring in the profile of operations are typed using these concepts (i.e., if  $Pa$  is a NFP then  $type(Pa)$  is a concept declared in  $\mathcal{O}$ ). Every QoS concept in  $\mathcal{O}$  is associated with a *weight*  $w_i$  ( $1 \leq i \leq n$ ) that is a non-negative real value  $w_i \in \mathbb{R}_0^+$ . Weights are chosen so as to grade the relative importance of QoS concepts. Weights are zero by *default*. As will be seen, a weight  $w_i = 0$  functions as a mask, a value  $0 < w_i < 1$  has a suppressing effect, and  $w_i > 1$  has an amplifying effect.

Since NFPs are qualities expressing *how* something is supposed to be, we can assume that their value  $val$  can be mapped meaningfully onto a numeric value  $v$  from an interval  $[v_{\min}, v_{\max}]$ , even if the interval (i.e., the value range) is just binary. For instance, a binary representation of reliability: *unreliable*  $\mapsto 0$ , *reliable*  $\mapsto 1$ . Numeric values are furthermore normalized; that is, values are projected onto the interval  $[0, 1]$  (a.k.a. scaling). There are, however, two types of qualities: those that one wants to maximize such as reputation or processor speed and those that one wants to minimize such as response time or price (i.e., the higher the value the better versus the lower the value the better). They are respectively called *positive* and *negative* in the literature (e.g., [ZBN<sup>+</sup>04]). Hence, both types are normalized in different ways as follows:

$$\bar{v} = \begin{cases} \frac{v_{\max}-v}{v_{\max}-v_{\min}} & \text{if } v_{\max} - v_{\min} \neq 0 \\ 1 & \text{otherwise} \end{cases} \quad (\text{positive NFP}) \quad (5.14)$$

$$\bar{v} = \begin{cases} \frac{v-v_{\min}}{v_{\max}-v_{\min}} & \text{if } v_{\max} - v_{\min} \neq 0 \\ 1 & \text{otherwise} \end{cases} \quad (\text{negative NFP}) \quad (5.15)$$

where  $\bar{v}$  is the resulting normalized value in the interval  $[0, 1]$ .

The execution cost of an operation  $Op$  whose profile contains the set of NFPs  $N$  is then simply defined as the maximum out of the products of the normalized value of a NFP and the corresponding weight:

$$f_{\text{cost}}(Op) = \begin{cases} \max\{w\bar{v} \mid type(Pa) \mapsto w, val(Pa) \mapsto \bar{v}, Pa \in N\} & \text{if } N \neq \emptyset \\ \text{undefined} & \text{otherwise} \end{cases} \quad (5.16)$$

If the operation is not annotated by any NFP at all then  $f_{\text{cost}}$  is undefined. This is an unfavourable special case. Simply defining  $f_{\text{cost}} = 0$  for this case might not reflect well its real execution cost nor is  $f_{\text{cost}} = 1$  appropriate either. Setting it to zero would advocate laziness in providing information on non-functional properties over industriousness, which is apparently unfair. Conversely, setting it to one generally penalizes lack of information in an unfair way. There are many strategies conceivable to even out this case. Two rather simple ones are: The less information is available about non-functional properties the more an operation is demoted. An operation without any information is thereby considered in the last place (i.e., detailed information is preferred over lack of information). Second, the operation is assigned with average costs, calculated from data gathered from earlier executions or from data about functionally similar operations of which NFPs are known or which have also been executed previously.

This model of execution costs is simple but effective and has three main characteristics. First, we do not need to make the rather impractical assumption that every

operation (service) is annotated for each of a set of QoS concepts with a corresponding NFP, which is made in several works (e.g., [ZBN<sup>+</sup>04, AMM07, MBK<sup>+</sup>09, KMHJ10]). Except for controlled environments, one will rather find the situation that annotations are made freely, usually for a few QoS concepts only. Second, the user does not need to assign all weights. Instead, one can be lazy about QoS concepts that one deems unimportant. The default weight of zero will mask all QoS concepts anyway that the user does not care about. Third, the absolute value of weights is less important compared to the relative value. If the user considers QoS concept  $C$  more relevant than  $D$  then it suffices to ensure  $w_C > w_D$ . This supports the observation that humans are rarely willing nor able to express the importance of a QoS concept directly (and precisely) in terms of an absolute value. It is easier and arguably more natural to humans to decide, given two QoS concepts, which one they find more important in some context.

Finally, this model of assigning costs to operations is directly transferable to enrich both notions of equivalent execution with the possibility of taking non-functional properties into account. More specifically, the cost function as defined by Equation (5.12) and Equation (5.16) can be taken as the basis of the preference order considered in preference-based matchmaking (see Definition 5.4).

## Preferences

Contrary to the conventional interpretation of goals there is an important difference for preferences over target effects and outputs: they are interpreted as *desired* rather than mandatory (i.e., soft goals as opposed to hard goals). With respect to Definition 5.7, this means that the execution of a semantically similar replacement suffices to yield non-empty subsets  $E' \subseteq E$  and  $O' \subseteq O$ . In other words, the user is prepared to accept a non-empty difference between  $E'$  and  $E$  ( $O'$  and  $O$ ) that is not achieved.

Analogous to QoS properties, it is useful to give different utilities to preferences thereby indicating their relative importance. The intention is that one tries to satisfy those preferences first that have highest utility (and avoid those preferences that have negative utility). Therefore, every preference  $\varphi_i$  is associated with a positive weight  $w_i \in \mathbb{R}^+$ , analogous to QoS concepts.

Preferences can accommodate the effect goals in  $\Gamma$  and desired outputs  $O$  in different ways. One possibility is to establish a direct correspondence. A preference  $\varphi$  is thereby either an effect in  $\Gamma$  or an output in  $O$ ; hence,  $\mathcal{F} = \Gamma \cup O$ . Given a replacement  $G_{cf}^r$ , the utility of  $\varphi \in \mathcal{F}$  is then simply its associated weight  $f_{\text{util}}(\varphi) = w$  if execution of  $G_{cf}^r$  satisfies  $\varphi$ . Otherwise the utility is undefined and thus not included in Equation (5.13).

Another and strictly more expressive possibility is to follow the model put forward by PDDL3 [GL06], which has been considered in the context of service composition also in [SM10]. Preferences in PDDL3 are expressions of the form

$$(* w \text{ (is-violated } \varphi))$$

where  $w$  is a positive weight as before and  $\varphi$  is a preference formula.<sup>30</sup> The main difference to the first and rather simple form of preference expressions is that PDDL3 ex-

<sup>30</sup>To be precise,  $\varphi$  is actually a reference to a preference formula since the latter might be referred to by multiple preference expressions.

pressions can describe temporally extended preferences that are evaluated on a state trajectory that can minimally be a single state up to a trajectory that extends from the initial state up to a final state. This is achieved by including temporal operators ascribed with semantics of Linear Temporal Logic (LTL). Among the temporal operators available are: at-end, always, sometime, within, at-most-once, and a few more. For instance, consider the following two PDDL3 preference expressions:

```
(preference payment (always (or (payBy ?x VISA) (payBy ?x MASTERCARD))))
(preference order (sometime (hasBoughtBook ?x ?y)))
```

The former preference named *payment* states that whenever a payment is to be done for some item identified by *?x* the user pays either by VISA or MasterCard; *(payBy ?x VISA)* translates to a DL effect atom *payBy(x, VISA)* where *x* is a variable referring to a representative of either an input or output, *payBy* is a role name, and *VISA*, *MASTERCARD* are individual names. The second preference named *order* states that at some point the user identified by *?x* has bought a book identified by *?y*; *(hasBoughtBook ?x ?y)* translates analogously to an effect atom *hasBoughtBook(x, y)*.

The nested expression *(is-violated  $\varphi$ )* evaluates to a value equal to the number of times the preference  $\varphi$  is violated in the state trajectory described by  $\varphi$ . In other words, it calculates a penalty for violation of preferences, which is additionally either amplified or suppressed by the weight *w* expressing the relative importance. Notice that [Equation \(5.13\)](#) is thereby changed to a minimization problem since penalties as well as costs are to be minimized.

## 5.5 Integration with Transactional Processes

Composite service execution and the *correctness* thereof can be meaningfully defined and reasoned about in the theory of *transactional processes* [SABS02] that builds, in part, upon the concept of *flexible transactions* [ZNBB94, ZNB01]. Transferring the basic idea to the service model considered in this work, the execution of a composite service is conceived as a single transaction that encapsulates a number of sub transactions if the service is composite. Each sub transaction corresponds to the execution of one of its functional units. Eventually at the level of operations, a transaction corresponds to the execution of one operation in the underlying sub system, which may again unfold to sub transactions at the next lower level of the sub system. Altogether this results in a nested configuration – *nested transactions*.

The connection of transactional processes with the concept of transactions lies in the focus on two major correctness properties: *atomicity* and *isolation*. Transactional processes, in fact, set up a more general notion of atomicity, rather than simply transferring the conventional all-or-nothing understanding. More precisely, a transactional process is correct (or well-formed) in this regard if it provides *guaranteed termination*, meaning that execution ends in a well-defined state in any case. Potential failures in the course of execution are thereby handled in a way not leading to undefined or inconsistent states. State consistency criteria are normally defined from an application point of view; although application-independent generalizations are conceivable (e.g., avoid unreleased

resources that are subject to an exclusive access policy). Multiple services that are concurrently executed are correctly isolated from each other according to this model if the overall result is equivalent to a serial execution of them. Isolation is consequently a correctness property concerning the correct interaction of multiple processes.

It is the purpose of this section to discuss consequences of integrating CFI into this model, especially consequences on the notion of guaranteed termination. Moreover, we explore potential integration strategies. Isolation, on the other hand, is beyond the scope of this thesis. To this end, we would assume that a transactional manager is part of the system (execution engines) that enforces the precedence order as defined by a process over all the different system levels involved so as to avoid non-serializable inferences on shared resources.

### 5.5.1 Guaranteed Termination

Defining the notion of guaranteed termination starts from the assumption that the transaction representing the execution of a functional unit falls in either of three categories: *compensatable*, *retriable*, or *pivot* [MRKS92]. Without loss of generality, we can restrict ourselves in the following (for simplicity of expositions) to operations, which is possible because these characteristics can be preserved for nested configurations, see [Observation 5.2](#) below.

If  $Op$  is an operation then it is compensatable if there is an operation  $Op^{-1}$  that when executed either reverses or, more generally, compensates the execution of  $Op$ ; without resorting to execution of additional operations (i.e., cascade free).<sup>31</sup> An operation is retriable if its execution is guaranteed to complete successfully eventually, possibly after having been re-invoked multiple times. While an operation can be both compensatable and retriable, the opposite – neither being compensatable nor retriable – is equivalent to being pivot.<sup>32</sup> The existence of such an operation in a composite service is therefore “critical” in the sense that it marks a point of no return. Once it has been executed there is no way back since compensation is not an option anymore. Notice that compensating operations themselves are assumed to be retriable; hence, not pivot. Also, observe that compensatable, retriable, or pivot are non-functional characteristics that can be represented already by two Boolean-valued non-functional properties because compensatable is orthogonal to retriable and pivot matches the case of not compensatable and not retriable. By adopting the generalization put forward in [Sch01] these properties become real-valued and model *failure probability* and *compensation costs*, re-

<sup>31</sup>Reversible is the ultimate form of compensatable. An operation is reversible if all its results can be undone by executing  $Op^{-1}$ ; hence,  $Op^{-1}$  can be seen as the *inverse* of  $Op$  and execution of  $Op^{-1}$  at some point after  $Op$  has been executed yields a state as if execution of  $Op$  had never taken place. Compensatable, as it is understood here and in accordance with [KLS90], is more general as it extends to the case of compensating results in a countervailing and indirect way. The state after execution of  $Op^{-1}$  may not be identical to a state that would have been reached had  $Op$  never been executed. For instance, undoing an order by marking it as canceled (and issuing a credit note) instead of entirely deleting it. The general notion of compensation is therefore a matter of application-specific definition.

<sup>32</sup>We note that the notion of pivot has been used slightly different in the literature. While [MRKS92, ZNB01] set pivot equivalent to not compensatable and not retriable, a relaxed definition leaving off retriability is used in [Sch01, SABS02].

spectively. More precisely, a compensatable operation has finite compensation costs in the interval  $[0, \infty)$ . A retrievable operation has a failure probability of zero since it never fails eventually. A pivot operation has infinite compensation costs since compensation is not possible and a failure probability in the interval  $(0, 1)$  since it is not retrievable.

**Observation 5.2.** *The properties compensatable and retrievable are preservable for nested configurations, while pivot inevitably propagates upwards.*

This is easily seen. A functional unit is compensatable if all functional units at the next lower level are compensatable. In other words, compensation at the higher level is no longer possible as soon as one functional unit at the next lower level is not compensatable. This holds analogously for retrievability. As soon as there is one pivot functional unit at the lower level then the higher level is also pivot.

Clearly, the property of being pivot is a decisive factor on the notion of guaranteed termination. In the simple case with absence of pivot functional units one can guarantee termination if all functional units a service is composed of are either retrievable or compensatable, though in diametral states. All functional units being retrievable guarantees that, in case of a transient failure of a functional unit, one can eventually complete as expected. Conversely, all functional units being compensatable guarantees that, in case of transient or permanent failure of a functional unit, one can reach a ground level state in which everything that has been done has been compensated. The latter matches the concept of Sagas [GMS87, GGK<sup>+</sup>91]. In the presence of pivot functional units, guaranteed termination necessitates that each pivot that is not the very last in a control flow is associated with at least one *failure recovery path*, as we call it. A failure recovery path is basically a sub control flow that leads to a consistent state. All functional units on a recovery path must furthermore be retrievable; hence, it is guaranteed that execution of the failure recovery path eventually completes. Notice that not a failure of the pivot triggers its execution but a failure of a subsequent compensatable or pivot operation. What is more, there might be multiple failure recovery paths for each pivot thereby providing different alternatives. In this case an additional *preference order* is considered between recovery paths that ranks them. The idea is then to select the highest ranked recovery path first at runtime.

Now, how does CFI relate to the property of guaranteed termination? First of all, if we put CFI into perspective with what has been just explained then it should be clear that CFI can be used for dynamic and on demand creation of a recovery path, meaning that it has not been specified in advance. CFI is also more general in that it is not restricted to this case: The question at which transition a failure occurred is not relevant since an alternative – a replacement – can be created in principle starting from any place. Yet it is exactly the dynamic creation that is crucial for the property of guaranteed termination. The guarantee that a replacement exists is clearly a necessary requirement. In other words, guaranteed termination depends ultimately on guaranteed replacement existence. Unfortunately, the strong guarantee for the existence of a replacement in any case can only be made by design. In open environments in which available services and operations are not known in advance and are likely volatile one can only make weak guarantees, possibly supported by statistical evidence. This observation holds irrespective of how sophisticated the approach used for representing and reasoning about the

semantics of services is. The point herein is that the utility of CFI lies more in open environments. The reason is that in cases where it is ensured by design that replacements exist one can actually assume that they are known at the same time; hence, there is little sense in dynamically creating them. Another point is that existence of replacements is not sufficient for guaranteed termination. The requirement of guaranteed termination applies equally to replacements themselves. Replacements therefore either need to consist of retrievable operations (analogous to a recovery path) or the failure within the execution of a replacement must again be manageable in the same way.

All in all we have to accept that CFI alone cannot preserve the property of guaranteed termination in cases where one wants to use it. CFI rather provides a best effort service to the user. This result might leave a smack of dissatisfaction. Be that as it may, it appears to be the price to pay for not needing to anticipate and specify in advance failure handling means. We therefore see CFI as complementary to existing methods and discuss possibilities of integrating both in the next subsection.

### 5.5.2 Integration Strategies

Bearing in mind the best effort property of CFI regarding guaranteed termination, transactional recovery approaches that build on the concept of compensation<sup>33</sup> and CFI as a dynamic forward recovery approach should be reasonably combined. The general idea is to consider pre-defined transactional recovery means as a last resort. Then, two possible arrangements are:

1. CFI is attempted first. If a replacement does not exist then fallback to transactional recovery.
2. If there is a pivot and a failure occurs after the pivot has been executed then compensate backwards up to the pivot. CFI may then be attempted to find an alternative starting after the pivot. If this fails then fallback to transactional recovery.

The first strategy reflects the intention of trying forward recovery from failures first before falling back to compensation. It is further assumed that all operations of the original service are compensatable. One can therefore guarantee well-defined termination since one can always go back to a ground level state in case CFI fails. The application logic or user would then decide how to proceed from that state. This strategy will likely come to be used if a one-to-one replacement is anyway the number one choice. An example is the *book seller* service: if execution of *order & pay* (or *shipment*) fails then one would likely want to search for an alternative online seller (or shipping agency) first before rolling back as a last resort.

The second strategy considers taking a pivot as a savepoint. In case of a failure one would generally compensate backwards up to the pivot first and try to find a replacement from that place. Analogous to flexible transactions, a recovery path is assumed to be pre-defined and associated to the pivot. The recovery path serves as a last resort if finding a replacement fails. This strategy seems preferable if a service contains sub

---

<sup>33</sup>Recap, compensation herein is understood as a generalization that includes reverting results as a special case.



services and many-to-many replacements are intended. For example, suppose *identify person* in the *emergency assistance* service is marked as a pivot. If at execution time *trigger ambulance* fails then one would compensate *select ambulance* and search for a replacement of the sub service *activate ambulance*. However, the compensation of an operation  $Op$  and subsequent search for a replacement starting from a previous state not only involves executing the compensating operation  $Op^{-1}$ , it also implies additional efforts for undoing the effects in the KB. More precisely, if  $Op^{-1}$  is indeed the inverse then one can easily derive the inverse update  $U^{-1}$  from the update  $U$  that has been applied previously to the KB: every add  $(\mathcal{K} + \varphi)$  in  $U$  becomes a delete  $(\mathcal{K} - \varphi)$  in  $U^{-1}$  and vice versa. Otherwise, if  $Op^{-1}$  is a countervailing compensation rather than the inverse then it has to be regarded as a normal operation. Its set of effects  $E$  would then be applied in the standard way as defined by the effect system used.

On the other hand, if guaranteed termination is not of high importance, then an integration strategy in the opposite direction comes to mind. More specifically, the planning-based approach to replacement creation can take advantage of information whether functional units – operations or atomic services in this case – are compensatable, retrievable, or pivot. If  $\mathcal{U}$  is the pool of functional units available and  $u$  is a functional unit in a control flow  $G_{cf}^e$  that is to be replaced then:

- $u \in \mathcal{U}$  if  $u$  is retrievable,
- $u^{-1} \in \mathcal{U}$  if  $u$  is compensatable, and
- $u \notin \mathcal{U}$  if  $u$  is pivot.

Observe that both  $u$  and  $u^{-1}$  can end up in  $\mathcal{U}$  if  $u$  is both retrievable and compensatable, which in turn calls for a preference between compensation or retry, expressed, for instance, by different costs associated with them. If a planner adds  $u^{-1}$  to a plan then a kind of chronological rollback is performed. Compensation is thus integrated into planning.

## 5.6 Repeated Intervention

If a replacement  $G^r$  has been found and execution resumes using  $G^r$  then there is no guarantee that  $G^r$  will succeed; except that  $G^r$  is retrievable, which we shall leave out of consideration in this section. In particular, there can again be failures in the execution of  $G^r$ . This can, in theory, lead to the situation of endless interventions made by an execution engine without actually coming to an end: a failure occurs, a new replacement is found, execution is resumed using the replacement, a new failure occurs, and so forth. No matter whether the probability of such a situation is low in practice or not, there should be measures to deal with this issue. More specifically, there should be a decision procedure that determines when to give up starting new intervention cycles. Ideally, one should not give up too early nor try longer than it makes sense, meaning that it was still possible to come to an end versus an intervention was anyway in vain. We see three possibilities to this described in the following.

### 5.6.1 Threshold

Presumably the most apparent approach is to set a threshold as a maximum number  $\vartheta$  of interventions that are allowed to be made during the execution of a particular service instance. A counter is incremented for every new intervention cycle and if  $\vartheta$  is passed then one would give up intervening in case a new failure occurs. While this is simple to implement it has the downside that it introduces a new problem: What value should  $\vartheta$  be set to? It is evident that at least the actual service and the error frequency of the environment have an influence on the value. This means that  $\vartheta$  is a function of multiple parameters

$$\vartheta = f(p_1, \dots, p_n)$$

rather than a (global) constant that can be determined generally. As there seems to be no single obvious law for  $f$ , one would probably use more or less sophisticated heuristics. In case it is even difficult to come up with a suitable heuristic then  $f$  might simply realize a rule of thumb or an educated guess. More advanced strategies might try to learn  $f$  offline from training data or online as an evolutionary process. In any case the risk of giving up too early or too late remains as long as the value  $\vartheta$  cannot be proven to be correct in the sense that any smaller or larger value results in giving up too early/too late.

### 5.6.2 Progress

Another approach would be the definition and use of a notion of *execution progress*. Intuitively, this can be used if one would not want to give up as long as there is significant progress towards the end (i.e., execution does not stagnate). The question is how should such a notion of execution progress be defined? One possibility is to formulate it in terms of a difference between the current and the final state of an execution.

Suppose  $t$  is a transition in a control flow graph and  $dist$  is the distance between  $t$  and the final place  $p_f$ , measured as the length of the path  $\langle t, \dots, p_f \rangle$ ; recap, every transition  $t$  of a control flow graph is on a path between  $p_i$  and  $p_f$ . Formally,

$$dist(t, p_f) = |\langle t, \dots, p_f \rangle| .$$

Suppose  $t$  is a token-enabled transition that cannot fire and that it has been replaced in a first intervention cycle. Now, suppose that subsequently yet another failure occurs and that  $t'$  is the token-enabled transition that cannot fire this time. Then one could define progress as the condition that

$$dist(t, p_f) > dist(t', p_f) . \quad (5.17)$$

According to this condition there is progress between subsequent intervention cycles if the younger is for a transition closer to the end. Unfortunately, this condition works generally only for one-to-one but not in general for one-to-many and many-to-many replacements as the following example shows.

**Example 5.4**

Imagine a control flow graph  $G_{cf}$  that has a simple sequential control flow

$$\mathbf{F} = \{(p_i, t), (t, p), (p, t'''), (t''', p_f)\}$$

consisting of the consecutive transitions  $t$  and  $t'''$ . Suppose there was a one-to-many replacement  $G'_{cf} = G_{cf}[G_{cf}^e/G_{cf}^r]$  with

$$\mathbf{F}^e = \{(p_i, t), (t, p)\} \quad \text{and} \quad \mathbf{F}^r = \{(p_i, t'), (t', p'), (p', t''), (t'', p)\},$$

resulting in the flow relation

$$\mathbf{F}' = \underbrace{\{(p_i, t'), (t', p'), (p', t''), (t'', p)\}}_{\mathbf{F}^r \text{ of } G_{cf}^r}, (p, t'''), (t''', p_f) \} .$$

If there is subsequently a failure for  $t''$  after the operation associated with  $t'$  has been executed successfully, **Condition (5.17)** does not hold because  $dist(t, p_f) = dist(t'', p_f)$  though there was obviously progress. This is due the fact that the sequential control flow was extended under the replacement  $G_{cf}[G_{cf}^e/G_{cf}^r]$ , which is not taken into account.

**Example 5.4** shows that we need to account for structural changes. This can be done by taking into account the difference in the length of the subflow  $G_{cf}^e$  that starts with  $t$  and its replacement  $G_{cf}^r$ . Let  $|G_{cf}|$  denote the length of the longest elementary path  $\langle p_i, \dots, p_f \rangle$  in the control flow graph  $G_{cf}$ . Then, the difference  $\delta$  in length between  $G_{cf}^r$  and  $G_{cf}^e$  is

$$\delta = |G_{cf}^r| - |G_{cf}^e|$$

and we finally get

$$dist(t, p_f) > dist(t', p_f) - \delta . \quad (5.18)$$

Although the notion of progress expressed by **Condition (5.18)** can be used for one-to-one up to many-to-many replacements (notice that  $\delta = 0$  for one-to-one replacements), one can also define it in a converse way as described next.

### 5.6.3 Possibility to make Progress

Alternatively, one can view the notion of progress from another perspective. If there is no alternative replacement available anymore then one can obviously not make any progress anymore; except that one tries to execute a replacement again that has already been used in the hope that the failure was transient and does not occur again. In other words, one might not want to give up as long as there is a *possibility to make progress*.

Determining whether it is possible to make progress can be implemented based on how replacements are found. Under the use of matchmaking there is a possibility of making progress as long as there is at least one matching replacement found. Analogously, under planning there is a possibility of making progress if at least one alternative

plan is found. However, the downside is that in most cases (e.g., in open and/or dynamic environments) one cannot decide at the beginning of an intervention cycle prior to matchmaking/planning whether it is possible to make progress. In these cases it is only the search that can provide the answer. Otherwise, one would need prior information about the number of remaining options. Without this information, the approach thus always involves a possibly expensive search to know whether there is a possibility to make progress.

## 5.7 Discussion

All the way up to this point we have mainly been concerned with two problems. First, how to define a formal notion of equivalence (and similarity) that a replacement for a part of a service must meet to qualify as such and that is adequate from a practical point of view in the sense that it reflects the intuition of humans. Second, what are techniques that allow us to check whether we can find or compose a replacement given a set of candidate services and their operations. With hindsight, we shall return to issues that have been identified throughout the text but whose discussion has been postponed. There are also a few more reflections in order.

### 5.7.1 Disambiguating Profile Parameters

Coming back to the ambiguous cases illustrated by [Example 5.2](#) and [Example 5.3](#), the reason is that a profile  $Pr$  has two (or more) parameters  $Pa_1, Pa_2$  of the same sort for which  $type(Pa_1) \sqsubseteq type(Pa_2)$  holds (cf. [Equation \(5.8\)](#)). Obviously,  $Pa_1, Pa_2$  are not the same otherwise there would be just one of them in the set, but  $Pa_1$  “is a” specialization of  $Pa_2$  according to the backing domain conceptualization. In essence, what we want in order to eliminate the case that a unique mapping  $m$  cannot exist is a relation that tells two parameters, both of which fall into the same sort of profile sets, semantically apart unless they are the same in every respect.

One possibility is to enrich the match relation on parameters by additional means to further discriminate semantics of parameters. Several authors have made proposals in this direction that rely on statistic-driven or information-theoretic similarity measures defined usually over the names of parameters or the combination of the name and the type. While this approach has been shown to increase the precision to a certain extent, its characteristic is that it tries to live with the status quo, meaning the amount of information available from their description.

One can also take up another position that views this issue from the opposite end, which becomes clear when raising the following question. Should profiles that have parameters  $Pa_1, Pa_2$  of the same sort for which  $type(Pa_1) \sqsubseteq type(Pa_2)$  holds be considered deficient in the sense that they are not sufficiently precise conceptualized? If one agrees with this view then the conclusion would be that the semantic annotation should be revised so that it becomes sufficiently discriminating. How could this be done? As an example, imagine an information-providing service for retrieving departure and arrival

times of trains between two cities; hence, it is parametrized with two inputs for the start and destination. Suppose these inputs are:

$$Pa_1: \text{FROM:City} \quad \text{and} \quad Pa_2: \text{TO:City}$$

where *City* is a concept defined in some domain ontology. Hence, we have  $\text{type}(Pa_1) \sqsubseteq \text{type}(Pa_2)$  (and vice versa). One can make them fully discriminated by revising their conceptualization to:

$$Pa_1: \text{FROM:City} \sqcap \text{Start} \quad \text{and} \quad Pa_2: \text{TO:City} \sqcap \text{Target}$$

where *Start* and *Target* shall be additional concepts. This way we have broken the subsumption relation (provided that  $\text{Start} \not\sqsubseteq \text{Target}$ ) by adding the semantics carried actually by the parameter names to the conceptualization, which was not represented before. This example shows that it is not really the “fault” of the subsumption relation that ambiguities may arise but rather deficiencies in the semantic conceptualization.

### 5.7.2 To Plug-in Match or not to Plug-in Match

Both notions of equivalent execution are in essence defined based on the plug-in match regarding data and the more strict equivalence match regarding effects. This is a decision we have made in view of the question what humans would accept. We believe that this interpretation marks the border between a replacement that can be said to provide equivalent results and a broader form, as results created are not semantically “outside” the original. With the subsume match results can be created that one did not expect and therefore humans would likely want to be asked first for their approval before execution can resume with a replacement.

This aspect also relates to the question whether it is reasonable for a practical realisation to be fully automated in the sense that humans are not in the loop of decision making. Even with the current definition, this question is again mostly a matter of the application environment. We believe that a fully automated realisation appears reasonable in closed environments only, and with a strict definition of equivalent execution. Open environments in which one cannot assume complete knowledge about all options available rather make a semi-automatic and mixed-initiative approach more plausible where the system presents alternatives first and the user can then make a choice. In general, we conclude that the broader the notion is defined towards similarity the more it is natural that humans want to be in the loop. This conclusion should be taken as a design guideline for future systems.

### 5.7.3 Structure-aware versus Structure-nescent Replacements

Having the two techniques available for finding or composing replacements introduces a choice: applying either technique exclusively raises the question when to chose which. On the other hand, one can also ask the question whether they can be reasonably combined. Let us first identify situations in which the exclusive use of either technique is indicated.

### Exclusive use of Either Technique

The requirement of preserving the control flow structure can be found in industrialized manufacturing workflows or medical treatment workflows that consist of a sequence of steps from which one must not deviate because they are part of (factory-wide) production cycles, or because they are medically prescribed. Otherwise, structure-aware replacements are indicated in (controlled and rather small) environments in which it is known in advance or even deliberately the case per design that directly functional equivalent services/operations exist. A typical example is the *activate ambulance* service: in this domain the distribution of ambulance centers is intentionally so that an area is covered with sufficient redundancy.

In contrast, the main characteristic that indicates the use of planning-based structure-nescent replacements are domains in which it is likely that alternative services/-operations exist that can serve as a replacement only by combining them accordingly (i.e.,  $1:n$  and  $n:m$  replacements). The matchmaking-based structure-aware replacement technique would fail in this case as it does not support synthesizing combinations. For the process of finding combinations it is more important to enable efficiently exploring the possibly large combinatorial space for solutions, even if the use of heuristics to make this feasible implies that finding an optimal replacement is not guaranteed in general.

The discriminating feature for both techniques is therefore mainly the degree of functional standardization in the domain. The simplicity offered by the structure-aware replacement technique necessitates a high degree of standardization: without sufficient standardization one cannot assume the existence of directly functional equivalent services/operations.

### Combined use of Both Techniques

Both techniques can equally well be combined in the form of a chain. More specifically, bearing in mind that it is the situation that one transition cannot fire due to an error, one can always start simple by checking whether there exists a one-to-one replacement for the pending transition, which amounts to a single call to a matchmaker (*FindMatch*). Only if this fails then the more expensive planning-based technique is used to synthesize a composite replacement.

## 5.7.4 Replacement Composition Planning via Translation into PDDL

Various works consider formulating service composition planning problems via translation into a PDDL planning problem (e.g., [KG05, KK07, HMV<sup>+</sup>09, KC10]) or related/successor languages (e.g., [BFL<sup>+</sup>08]), which makes it possible to use off-the-shelf planning tools directly. In fact, the translation is rather straightforward if atomic services/operations are directly understood as parametrized actions (operators). Besides abandoning the correspondence between the change semantics and the belief update problem, there are two more concerns. The first one is rather technical. Translation into PDDL introduces a certain overhead of pre-processing. Complexity of pre-processing can be exponential in space because virtually all PDDL planning tools are implemented

based on a propositional representation; hence, require instantiating the variables of operators to get actions. To illustrate the consequences, suppose there are  $k$  operators each having  $l$  parameter that can in turn be instantiated from  $m$  constants. Then we have  $\sum_{i=1}^k \prod_{j=1}^l m_{ij}$  possible instantiations (actions). Additionally, the approach introduces post-processing overhead if synthesized PDDL plans need to be translated back into the process description language used by an execution engine. While pre-processing can in principle be eliminated by pre-translation prior to planning time, this is not possible analogously for post-processing unless planning can be done offline (e.g., classic planning in a static domain, or contingency planning taking into account dynamic changes in the domain). The amount of time required for translation to PDDL is supposedly (much) larger than back translation: (large) number of actions versus usually rather short plans.

The second concern is about the change semantics ascribed by the actual precondition and effect system used. Unless the concrete system matches the simple STRIPS-style change semantics as discussed above, the use of off-the-shelf PDDL planning tools is not directly possible as virtually all realize it. In particular, if the change semantics corresponds to a query answering problem and a belief update problem over a DL KB and both cannot be translated into a propositional representation then plan checking is more involved since it reduces to satisfiability checking of an ABox (w.r.t. a TBox) as opposed to satisfiability checking of propositional formulas. Similarly, the actual effect system may pose challenges regarding heuristic search functions pertaining to the expressivity of concepts and the types of TBox inclusions permitted.

## 5.8 Summary

The main value of the CFI approach is that failure handling and recovery measures do not need to be modeled explicitly at design time of a service. This simplifies the modeling process. Possible failure situations need not (all) be anticipated and represented appropriately by explicit failure handling primitives. As a result of formalisms where this is required, single services (activities in workflow modeling) are often associated with “mechanical” fault handlers in practice. A failure is simply propagated upwards in the application, often up to the highest level of human control. It is then up to the responsibility of the user to trigger appropriate actions.

Second, an execution system implementing CFI would normally not reverse or compensate the effects of service invocations that have been made already in the course of execution, thus, not rolling back to the initial state at commencement of execution. Therefore, it would also not require appropriate restart functionality. This is the approach taken by traditional backward recovery methods following the atomicity property of the transaction notion. CFI provides a solution to these situations when backward recovery is impossible – because a pivot has been executed successfully already – as it is inherently forward-oriented and, more importantly, an alternative but semantically equivalent execution is found dynamically. In fact, the property whether a service is compensatable or pivot becomes irrelevant at design time when CFI can be used at runtime. This solves a problem apparent in large scale environments such as the Inter-

net where termination properties of services (whether they are compensatable, retrievable, or pivot) are outside the control of a designer and might even be unknown (i.e., when service providers do not publish these properties as part of the service descriptions).

Third, only in the presence of a failure at runtime actions are taken by an execution system implementing CFI. This makes CFI an optimistic approach. In contrast to pre-defined failure handling, however, the approach relies on the availability of semantically equivalent services. This is why we see CFI as complementary to existing failure handling methods rather than a solution meant to replace any of such existing methods. Finally, with a solution that allows to implement CFI in a fully automatic way, failure handling can be done transparent to the user/application.



# 6

## Concurrency Control for Shared Knowledge Bases

OUR SYSTEM MODEL described in [Chapter 4](#) allows for two types of concurrency in the service execution task. First, *intra-service* concurrency due to the possibility of parallel paths in the control flow of a composite service. Second, *inter-service* concurrency as a matter of the fact that we did not impose restrictions on whether just one or multiple services can be executed at the same time in the system. Amongst other aspects<sup>1</sup>, concurrency implies the need for coordinating access (read and update operations) to a shared OWL knowledge base so that correct results are generated rather than inconsistencies, which should generally be avoided. This is the subject of this chapter.

Coordinating concurrent access is commonly referred to as *concurrency control* (CC) in the area of database research. CC essentially studies the means to maintain consistency of a shared database, or, as it is the case in the context of this chapter, a shared OWL knowledge base. In short, this is achieved for OWL knowledge bases following the ideas of (i) transactional information systems [WV02], (ii) semantically rich operations [VHBS98], and by (iii) applying an extended version of the Snapshot Isolation protocol [BBG<sup>+</sup>95] to coordinate concurrent access. More precisely, we set up a concurrency control model for OWL knowledge bases as follows:

1. We take the well known notion of a *transaction* and formulate it as a partial order of indivisible *read*, *add*, and *delete* operations over OWL syntactic instances. Conflicting concurrent access is thereby analyzed over these operations; that is, directly at the level of OWL syntactic instances.
2. Concurrent transactions that may otherwise interfere in an undesired manner are isolated using the *Snapshot Isolation* (SI) protocol. The protocol receives a simple algorithmic extension in order to provide protection from two higher level semantic conflicts specific to OWL and DLs in general.

The read, add, and delete operations are sufficient to realize knowledge base updates and for reading it. Snapshot Isolation, a multiversion concurrency control mechanism

---

<sup>1</sup>For instance, scheduling to share common computing resources such as a CPU.

(MVCC) introduced for databases, offers a higher degree of concurrency compared to classical inherently-blocking protocols such as strict two-phase locking (S2PL). This is due to the fact that multiple versions of data are used to provide non-blocking reads, at the tradeoff that reads may not necessarily see the most recent state of a knowledge base. As will be seen, snapshot management is simple in this model because multiple versions of data do actually not exist, resulting from the binary lifecycle of data items representing OWL syntactic instances.

The main advantage of applying conflict analysis at the level of OWL syntactic instances is that it allows to jointly consider

1. conflicting access on the data representing OWL syntactic instances and
2. two higher level types of conflicts owing to specifics of OWL and DLs in general. In short, the first type of conflict reflects the situation where semantically (logically) equivalent knowledge is described using different expressions, which is essentially introduced by syntactic redundancies in OWL. By considering the second type of conflict we can avoid that the updates of concurrent transactions lead to an unsatisfiable knowledge base.

Because of the specifics of OWL and DLs in general, these two conflict types have to be considered regardless of the actual CC protocol used.

The approach completely abstracts from the physical representation of OWL syntactic instances. In particular, we show that it is compatible with their representation in the form of RDF triples. It can therefore be used to build (or extend) RDF triple stores to provide correct concurrent access to OWL knowledge bases represented as RDF triples. Finally, the approach allows for efficient integration with reasoning engines.

We would also like to point out that correct concurrent access to a shared OWL knowledge base is of general relevance for applications where multiple clients (humans or software agents) need to concurrently read and update it. The approach is, therefore, described in a general way so as to be applicable beyond the service execution task. Prior to the detailed description, the next section briefly illustrates and motivates the need for CC in the service execution task.

## 6.1 Motivation

There are mainly two types of recurring events in the course of service execution using semantic services that trigger reading or updating a knowledge base. First, checking whether the current state of the knowledge base satisfies the preconditions of a service/operation whose invocation is due. Second, applying the effects upon (successful) completion of an invocation to incorporate its effects into an updated new world state.<sup>2</sup> As described in [Section 4.2.2](#), precondition checking requires submitting read queries to interrogate the knowledge base about the current world state. Application of effects is achieved by means of direct updates that insert new ABox assertions and/or delete

---

<sup>2</sup>One can easily find other events such as reading and updating context information that drives distributed execution, but they shall not interest us here.

existing ones. These reads and updates generate sequences of basic read and update (add and delete) operations over OWL syntactic instances in the KB.

Obviously, reads and updates will be submitted concurrently by an execution engine supporting intra-service concurrency. Therefore, the knowledge base must support correct coordination of these accesses so that anomalies resulting from interfering accesses can be avoided. Moreover, if execution engines support the concurrent execution of multiple services – inter-service concurrency – then the problem of correctly representing and maintaining a “global” world state that encompasses the effects of the concurrent executions and that evolves along them has to be considered: Executions likely happen all in the same domain and all might have interdependencies in this domain. If one would maintain an isolated world state for each execution then they would be even more incomplete if not incorrectly represent the “real world” since each would encompass only local changes of one execution. Consequently, the representation and management of a global world state inevitably requires a shared knowledge base equally accessible among concurrent executions. Altogether, this calls for a KBMS that:

1. provides the possibility to read and update the knowledge base concurrently,
2. is equipped with appropriate means to avoid concurrency phenomena and anomalies in the representation of the world state, and
3. where the chronology of knowledge base states reflects the temporal ordering in which the “real” world evolves so as to correctly represent happened-before and causal relations.

While the former two items should be clear, the latter needs further explanation. In essence, the concurrent accesses on the KB are partially ordered. The responsibility of the KBMS is to ensure that such a partial order of accesses that it receives is preserved and correctly materialized in memory, rather than subject to spurious transposition. If any specific order of accesses is requested then it is enforced independently of the KBMS. We will come back to this aspect in [Section 6.5](#) when detailing how CC is applied to the service execution task.

[Figure 6.1](#) shows when read and update queries are triggered in the course of service execution and that they may very well overlap in time as a result of the two types of concurrency. Consequently, anomalies and phenomena resulting from arbitrary interferences need to be avoided. One can easily find cases in the two application scenarios described in [Chapter 2](#) that one wants to avoid. For instance, for a concurrent execution of two instances of the emergency assistance service the system should ensure that activation and assignment of a particular ambulance to a mission can be done by either of the instances (whichever comes first) but it should be neither assigned to both nor that the representation of an assignment gets lost because it is overwritten.

On the other hand, achieving correctness for these read and update queries should not result in an approach in which correctness is achieved by a KB access policy such as *multiple-reader/single-writer* locking (MRSW), which is easy to enforce and implement but restricts the amount of concurrency that the system can handle and therefore the overall performance.

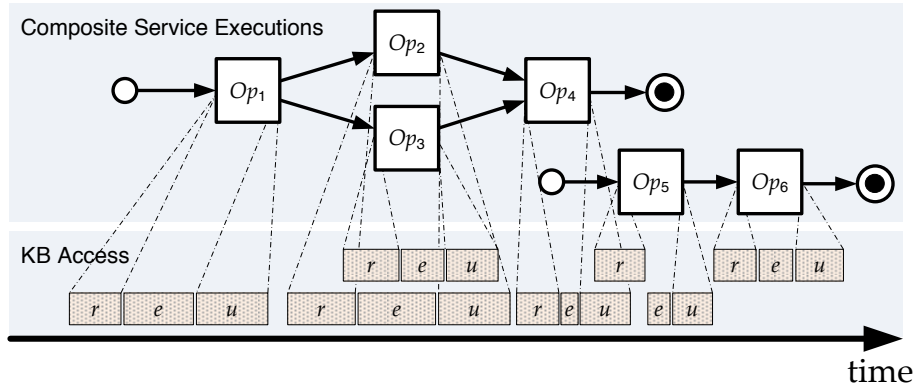


Figure 6.1: Concurrent execution of composite services with read ( $r$ ) and update ( $u$ ) queries on the KB, and actual execution time ( $e$ ) of invoked operations. Reads result from precondition checking, updates from application of effects. Read/update queries may overlap in time due to concurrent operation invocation within a service instance (intra-service concurrency) and due to concurrent service executions (inter-service concurrency).

## 6.2 CC Model for OWL Knowledge Bases

In the prevalent database read/write model [BHG87, Chapter 1] (a.k.a. page model [WV02, Chapter 2]), a database is understood as a finite set of named *data items*, each having a *value* over which two deterministic and total operations exist. One to *read* its current value and one to *write* the value, thereby mutating it. The size of data contained in a data item is called its *granularity*. In our model, an OWL knowledge base  $\mathcal{W}$  is also understood as a set of data items. The value of a data item, however, is exactly one possibly nested OWL syntactic instance. Consequently, CC will be applied directly at the level of OWL syntactic instances.

It should be noted that in our model data items are merely containers that do not have semantics themselves: It is the (possibly nested) OWL syntactic instance contained in a data item that is subject to be interpreted, thus, conveys semantics. This is one difference to the read/write model in which data items themselves are associated with semantics independent of their value. For instance, a record representing the balance of a bank account; no matter what the value is it is always to be interpreted as the balance of a certain bank account.

We start by defining the notion of OWL data items and the basic operations and their semantics required to realize KB updates. This is followed by formulating the notion of transactions, histories, and schedules in our model, introducing the SI-based access protocol, and two specific higher level conflicts. Finally, we discuss the properties that the SI-based protocol can guarantee.

### 6.2.1 OWL Data Items

Formally, an OWL *data item* whose value is an OWL syntactic instance  $\psi$  is denoted with  $di_{\psi}^{OWL}$ . In our model, however, there are two fundamental differences to the classical

read/write model stemming from the axiomatic way of knowledge representation in Description Logics.

First, since knowledge (stated by means of OWL syntactic instances) does either exist or not in an OWL KB  $\mathcal{W}$ , data items are immutable. Either they do not exist or if they do exist then they cannot be mutated. Consequently, the set of existing data items makes up the state of  $\mathcal{W}$ , as opposed to an assignment of values to data items that makes up a database state in the read/write model.

Second, OWL data items are distinguished only by their value rather than by their identifier as in the read/write model. Consequently, there cannot be distinct OWL data items that have the *same* OWL syntactic instance. This raises the question when are two OWL syntactic instances considered to be the same? This is defined based on the notion of *structural equivalence* [MPSP09, Section 2.1]. In short, since the meta model of OWL is defined in terms of UML, the notion of structural equivalence essentially considers (i) the recursive structure of syntactic instances and (i) type-based equality of strings and numbers occurring in syntactic instances.

**Definition 6.1** (Idem OWL Data Items). *Two OWL data items  $di_{\psi_1}^{OWL}$ ,  $di_{\psi_2}^{OWL}$  are the same iff the corresponding syntactic instances  $\psi_1, \psi_2$  are structurally equivalent. In this case we write  $di_{\psi_1}^{OWL} = di_{\psi_2}^{OWL}$ . Otherwise they are distinct.*

It is important to understand that the notion of idem OWL data items is purely syntactic. In other words, **Definition 6.1** does not define when two OWL data items should be regarded as semantically equivalent. As will be seen, this has consequences on conflict relations.

#### Example 6.1 (Idem OWL Data Items)

The OWL data items for the class axioms

$$\text{DisjointUnion}(:\text{Parent} : \text{Mother} : \text{Father})$$

$$\text{DisjointUnion}(:\text{Parent} : \text{Father} : \text{Mother})$$

are the same because the axioms are structurally equivalent: the order of the latter two individuals is irrelevant as they represent a set. Similarly, the class expressions

$$\text{ObjectIntersectionOf}(:\text{Book} : \text{OnStock})$$

$$\text{ObjectIntersectionOf}(:\text{OnStock} : \text{Book})$$

are structurally equivalent; hence, they have the same OWL data item.

We note that OWL syntactic instances can be nested. For example, the concept definition axiom

$$\text{EquivalentClasses}(A \text{ ObjectUnionOf}(C D))$$

contains the nested expression  $\text{ObjectUnionOf}(C D)$ , which defines an anonymous class in this case. Since the inner class expression cannot exist independently because it is anonymous, it does not represent its own OWL data item. Consequently, OWL data items may represent nested OWL syntactic instances.

## 6.2.2 Basic Operations

KB updates are understood as operations *directly* adding or deleting OWL syntactic instances according to [Definition 3.14](#). Due to the fact that OWL data items are immutable and can either exist or not, an *add* and a *delete* operation (as opposed to the write operation in the database read/write model) is sufficient to realize KB updates.

The add and delete operation (performed by the KBMS) are denoted with  $a(\psi)$  and  $d(\psi)$ , respectively. Their semantics is given by [Definition 3.14](#). In addition,  $r(\psi)$  shall denote the *read* operation that does not change a KB. Read, add, and delete are assumed to be indivisible (i.e., a KBMS is assumed to execute them in an atomic way). All operations have a return value. The read operation simply returns  $\psi$  if it exists and `null` otherwise.<sup>3</sup> The add and the delete operation have a Boolean return value. Let  $\mathcal{W}$  be an OWL KB and  $\mathbf{S}, \mathbf{S}'$  the set of OWL syntactic instances in  $\mathcal{W}$  before and after applying either an add  $a(\psi)$  or delete  $d(\psi)$  to  $\mathcal{W}$ . The execution of  $a(\psi), d(\psi)$  returns `true` only if  $\mathbf{S}' \neq \mathbf{S}$  as a result of their execution and `false` otherwise. In words, an attempt to add (delete) a syntactic instance which already exists (does not exist) does not modify  $\mathcal{W}$  and `false` is returned to inform the application about the futility of the operation. Hence, the return value allows applications to detect whether an add/delete actually caused a change and to handle this appropriately.

Essential to CC is the notion of commutativity of operations, which is defined in our setting as follows.

**Definition 6.2** (Operation Commutativity). *Let  $\mathbf{S}$  be the set of syntactic instances in an OWL KB  $\mathcal{W}$ . A combination of two add, delete, or read operations  $o_1, o_2$  is said to commute iff (i) the updated set  $\mathbf{S}'$  resulting from the execution of  $o_1$  and  $o_2$  on  $\mathbf{S}$  as well as (ii) the return values of  $o_1, o_2$  are the same regardless of their execution order.*

This notion of commutativity is the stricter of two forms because it is agnostic to the initial state of  $\mathcal{W}$ . It is therefore also referred to as *state-independent* commutativity [VHBS98]. In the weaker form,  $o_1, o_2$  *state-dependently* commute if there exists some  $\mathbf{S}$  such that  $\mathbf{S}'$  is the same and the return values are the same regardless of their execution order. In [VHBS98] it is even assumed that operation sequences and their return values are the only means to generate a database state and to detect an equivalence between any two database states. This can be very well transferred to our model: a KB can be a black box of which only an initial (possibly empty) state and the sequence of operations and their return values is known.

[Table 6.1](#) shows the commutativity matrix for all combinations of read, add, and delete operations. In addition, combinations that do actually not modify  $\mathcal{W}$  (i.e.,  $\mathbf{S}' = \mathbf{S}$ ) are also shown. These results are fully explained in [Appendix A.3](#).

A final remark about read operations is in order here. The basic read operation  $r(\psi)$  is supposedly not practicable for efficient implementation of all kinds of KB reads (e.g., query answering). One probably wants at least an additional operation to read through syntactic instances stored in a KB in an iterator-like manner. In case there are other types of read operations that cannot be mapped, at least in theory, to sequences of basic reads  $r(\psi)$  then we need to extend the commutativity matrix accordingly by

<sup>3</sup>This type of read operation can equally be understood as a *test* whether the KB contains  $\psi$  or not.

Table 6.1: Commutativity and set-preservation of read, add, and delete operations on OWL data items. An entry  $-/+$  means that the pair is not commutative (set-preserving) in general but state-dependently commutative in the constellation noted.

First/second Operation	Commutative			Set-preserving		
	$r(\psi)$	$a(\psi)$	$d(\psi)$	$r(\psi)$	$a(\psi)$	$d(\psi)$
$r(\psi)$	+	$-/+^a$	$-/+^b$	+	$-/+^a$	$-/+^b$
$a(\psi)$	$-/+^a$	$-/+^a$	-	$-/+^a$	$-/+^a$	-
$d(\psi)$	$-/+^b$	-	$-/+^b$	$-/+^b$	-	$-/+^b$

<sup>a</sup> For  $\psi \in \mathbf{S}$ .    <sup>b</sup> For  $\psi \notin \mathbf{S}$ .

the new types of read operations. On the other hand, there is an evident observation: different types of read operations apparently commute if they (i) are executed on the same state of the KB and (ii) whose results are functions of the KB state and nothing else. Consequently, if a combination of a read and a change operation does not commute then substituting the read by another type of such read operations would neither commute. This is important insofar as it allows to generalize commutativity matrices regarding such read operations.

### 6.2.3 Transactions

Adapted to our model, the intuition behind a *transaction* is to represent a finite sequence of accesses made by one and the same application to a shared OWL KB  $\mathcal{W}$ . The goal is to ensure *atomic* and *isolated* execution of concurrent transactions. Atomicity means that if a transaction terminates normally (commits) then all of its effects are made permanent in  $\mathcal{W}$  and become visible to others all at once. Otherwise, if it fails (aborts) it has no effect at all on  $\mathcal{W}$ . Isolation means that concurrent transactions access  $\mathcal{W}$  without interfering with each other (e.g., reading intermediate changes from a non-committed transaction).

**Definition 6.3** (Read/Update Transaction). *A read/update transaction (transaction for short) is a pair  $T = (O, <)$  where:*

- $O$  is a finite set of add, delete, and read operations containing, in addition, a terminating abort or commit operation, denoted with  $\textcircled{a}$  and  $\textcircled{c}$ , respectively, such that
  - (1)  $\textcircled{a} \in O$  iff  $\textcircled{c} \notin O$ ;
  - (2) if  $p$  is  $\textcircled{a}$  or  $\textcircled{c}$  (whichever is in  $O$ ) then  $\forall o \in O: o < p$ .
- $< \subseteq O \times O$  is a strict precedence order.

The two constraining items in [Definition 6.3](#) state that a transaction either has a commit  $\textcircled{c}$  or an abort  $\textcircled{a}$  as its final operation (but not both) to indicate whether it terminated successfully or not. The semantics of  $<$  is an ordering in time, meaning that for any  $o_1, o_2 \in O$  and  $o_1 < o_2$  then  $o_1$  is executed prior to  $o_2$ . This implies that there is no intra-transactional concurrency. Relaxing the strict precedence relation to a non-strict ordering  $\leq$  is possible for commuting operations since there is obviously no need to

impose an order on commuting operations. Doing so enables concurrency within a transaction, which is especially useful to model distributed transactions.

The *changeset* of a transaction  $T$ , denoted with  $\delta(T)$ , comprises all syntactic instances that are added or deleted. Formally, given a transaction  $T = (O, <)$  then

$$\delta(T) = \{\psi \mid o(\psi) \in O \text{ and } o \text{ is an add or delete}\} . \quad (6.1)$$

A *history* models an interleaved execution of multiple transactions, formally defined as follows.

**Definition 6.4** (History, Schedule). *Let  $\mathbb{T} = \{T_1, \dots, T_n\}$  be a finite set of transactions where every transaction  $T_i \in \mathbb{T}$  has the form  $T_i = (O_i, <_i)$  and  $1 \leq i \leq n$ . A history is a pair  $H = (O, <_H)$  where:*

- $O = \bigcup O_i$  is the union of all operations over all transactions, and
- $<_H \subseteq O \times O$  is a partial order, such that
  - (1)  $\forall T_i: <_i \subseteq <_H$  and
  - (2) each pair of conflicting operations  $o_1, o_2 \in O \setminus \{p_i \in O_i \mid p_i \text{ is } @_i \text{ or } \odot_i\}$  are ordered in  $<_H$  such that either  $o_1 <_H o_2$  or  $o_2 <_H o_1$ .  
Two operations  $o_1(\psi_1), o_2(\psi_2)$  conflict w.r.t.  $H$  iff they (i) do not commute, (ii) are from different transactions ( $o_1 \in O_i, o_2 \in O_j, i \neq j$ ), and (iii)  $\psi_1, \psi_2$  are structurally equivalent.

A schedule is a prefix of a history.

In plain language,  $H$  preserves the order of each transaction  $T_i$ , is complete in the sense that all operations of each transaction appear, and conflicting operations (i.e., non-commutative operations of distinct transactions over the same OWL syntactic instance) are ordered.

## 6.2.4 Correct Concurrent Access

The notion of a history (schedule) does not extend to *correctness*. Correctness, as it is understood in this context, essentially relates to the aforementioned isolation of transactions. In other words, the criterion for isolated transactions is not yet stated. A schedule might be correct, meaning that interleaved transactions are properly isolated; but it might as well be incorrect, meaning that transactions interfere.

The major correctness criterion in database transaction theory is *serializability*. It can be defined analogously in our model. For this, we have to introduce the notion of a serial history (schedule) first.

**Definition 6.5** (Serial History, Schedule). *A history (schedule) is serial if no transaction starts before another transaction ends.*



Since transactions in a serial schedule are ordered in time they can obviously not interfere in time (i.e., an ordering in time implies complete isolation).<sup>4</sup> This is precisely the idea underlying the notion of serializability: under the assumption that transactions are correct by themselves, meaning that each of them does not violate some integrity constraint  $IC$  by itself, then a serial execution of them is also correct since it neither leads to violation of  $IC$ . Serializability of a schedule is consequently the property that it is equivalent to a serial schedule.<sup>5</sup> One can also say that correctness is the absence of integrity constraint violations for a possibly interleaved execution of concurrent transactions. The role of integrity constraints is indeed important. Observe that serializability has it that integrity constraints do not need to be known; it all depends on equivalence to a serial schedule. This aspect will later come back into focus.

There are, in fact, three different formulations of serializability known in the literature [WV02]: *final state*, *view*, and *conflict* serializability. While conflict serializability has been shown to be a special case of view serializability, it is the one of practical relevance since checking whether a schedule is conflict equivalent to a serial schedule can be done in polynomial time in the size of the schedule, whereas the other two are NP-complete.

Enforcement of correctness is part of the protocol used to coordinate concurrent access. Different protocols may however deliberately offer different isolation levels (i.e., different levels of correctness), which is motivated mostly by trading correctness for performance. The protocol, that we will apply is introduced next.

### 6.2.5 Access Protocol

The access protocol considers transactions over OWL data items and operations as introduced in Section 6.2.1 to 6.2.3 and essentially follows the SI protocol [BBG<sup>+</sup>95]. The protocol enforces the following two rules:

1. Given an OWL KB  $\mathcal{W}$ , a transaction  $T_i$  sees a *snapshot* of  $\mathcal{W}$  from the point in time when  $T_i$  started – every new transaction is associated with a unique start-timestamp  $t_s$  larger than any timestamp that has been created before (i.e., strictly monotonic). Both read and update operations are executed on this snapshot. Updates submitted by other transactions that started after  $T_i$  are invisible to  $T_i$ .
2. When  $T_i$  is ready to commit, it gets a commit-timestamp  $t_c$  larger than any existing start and commit-timestamp.  $T_i$  commits successfully only if there is no other committed transaction  $T_j$  whose commit-timestamp is within the interval  $[t_s, t_c]$  and if the change sets are disjoint  $\delta(T_i) \cap \delta(T_j) = \emptyset$ . Otherwise,  $T_i, T_j$  *conflict* and  $T_i$  will be aborted. This is also referred to as *first committer wins*.

A transaction is said to be *active* starting from the point in time when it has been assigned with  $t_s$  until the point in time when it has been assigned with  $t_c$ .

<sup>4</sup>Note that any order among transactions is legitimate provided that there are no dependencies among them. If there are dependencies then it is assumed that an order is secured outside the KBMS.

<sup>5</sup>To be precise, serializability is usually defined over the so-called *commit-projection* of a schedule: a schedule  $S'$  obtained from a schedule  $S$  by removing aborted and active transactions (i.e.,  $S'$  contains the committed transactions of  $S$  only).

Observe that in this protocol add and delete operations need to be considered in order to determine whether a transaction can commit because they determine the content of the change sets  $\delta(T_i)$ ,  $\delta(T_j)$ . This makes commit analysis similar to SI applied to the database read/write model. Recap, in the presence of the write operation the write-sets are required to be disjoint. Read operations are not included in commit analysis because every transaction operates on its isolated snapshot. Reads can therefore be executed at their commencement. From this last aspect we can draw an important conclusion: Under the standard SI protocol it is irrelevant what type of read operations are available. This is important insofar as it allows adding specific query-like read operations (e.g., list all concept inclusions, list all assertions about a given individual). Such additional read operations are important for implementing efficient KB querying support, as opposed to mapping all reads to the basic read operation  $r(\psi)$ , which can easily become inefficient.

As a matter of the fact that the protocol is a MVCC mechanism (i.e., each transaction operates on its own snapshot), it is guaranteed that reads only present results of committed transactions. However, SI is known to allow for schedules that are not conflict equivalent to some serializable schedule; that is, there are some concurrency anomalies that can occur with this protocol. We will resume and detail the discussion on which properties are ensured in [Section 6.2.8](#). [Example 6.2](#) illustrates the situation of conflicting update transactions.

### Example 6.2

This example models the famous concurrent execution of a bank transfer and a withdraw from one account involved in the transfer. Suppose a KB  $\mathcal{W}$  that initially contains two OWL datatype assertions

$$\begin{aligned}\psi_1 &= \text{DataPropertyAssertion}(:\text{balance} :ACC1 2770) \text{ and} \\ \psi_2 &= \text{DataPropertyAssertion}(:\text{balance} :ACC2 200),\end{aligned}$$

representing knowledge about two accounts  $:ACC1$  and  $:ACC2$  having a balance of 2770 and 200, respectively (i.e.,  $:ACC1$  and  $:ACC2$  are names of two distinct individuals,  $:balance$  is the name of a data property, and 2770 and 200 are number values). Transaction  $T_{wd}$  shall withdraw an amount of 70 from  $:ACC1$  and transaction  $T_{tf}$  shall transfer an amount of 100 from  $:ACC1$  to  $:ACC2$ . This can be realized using the following sequence of operations per transaction

$$\begin{aligned}T_{wd} &= r(\psi_1) d(\psi_1) a(\psi_3) \\ T_{tf} &= r(\psi_1) r(\psi_2) d(\psi_1) d(\psi_2) a(\psi_4) a(\psi_5)\end{aligned}$$

where  $\psi_3, \psi_4, \psi_5$  would be

$$\begin{aligned}\psi_3 &= \text{DataPropertyAssertion}(:\text{balance} :ACC1 2700) \\ \psi_4 &= \text{DataPropertyAssertion}(:\text{balance} :ACC1 2600) \\ \psi_5 &= \text{DataPropertyAssertion}(:\text{balance} :ACC2 300) .\end{aligned}$$

Clearly, any interleaving of  $T_{wd}, T_{tf}$  such that

$$t_s^{T_{wd}} < t_c^{T_{tf}} < t_c^{T_{wd}} \quad \text{or} \quad t_s^{T_{tf}} < t_c^{T_{wd}} < t_c^{T_{tf}}$$

is not serializable; hence, according to the first committer wins rule,  $T_{tf}$  commits in the first case, whereas  $T_{wd}$  commits in the second.

**Example 6.2** contains conflicting read-add-delete transactions. The conflict would be detected because the change sets are not disjoint  $\delta_d(T_{wd}) \cap \delta_d(T_{tf}) = \{d(\psi_1)\}$ . This remains the same even if the applications submitting  $T_{wd}, T_{tf}$  would (magically) know the current balance of both accounts so that the reads  $r(\psi_1), r(\psi_2)$  do not appear (add-delete transactions), thus, the transactions still conflict.

Snapshots need to be created (for reads by other transactions) whenever an OWL data item  $di_\psi^{OWL}$  gets committed as deleted by a transaction  $T$  and there exist other active transactions that started before  $T$  started. These transactions must still be able to read  $di_\psi^{OWL}$ . In other words, a snapshot of a deleted data item  $di_\psi^{OWL}$  can be discarded as soon as there are no more active transactions in the system that have a start timestamp smaller than the delete-commit timestamp of  $di_\psi^{OWL}$ . Analogously, when a transaction  $T$  commits successfully its change set  $\delta(T)$  needs to be kept if there are other active transactions  $T_1 \dots T_n$  in the system because  $\delta(T)$  is required for conflict analysis against the change sets of  $T_1 \dots T_n$  to determine whether they are permitted to commit.

Testing for overlapping change sets can also be done more promptly (as opposed to doing it at the end). An algorithm for the read/write model called *first updater wins* that acts this way has been described in [FOO04]. Transferring this algorithm to our model is straightforward: The first active transaction  $T_1$  to add/delete a data item  $di_\psi^{OWL}$  is granted to execute the operation. If  $T_1$  goes on to commit, any other active transaction that has updated  $di_\psi^{OWL}$  immediately aborts. An active transaction  $T_2$  with  $t_s^{T_2} < t_c^{T_1}$  that attempts to update  $di_\psi^{OWL}$  after  $T_1$  has committed aborts upon that attempt.

### 6.2.6 Higher Level Conflicts

The notion of conflicting transactions, defined for our SI-based protocol in terms of disjointness of sets is the basis to coordinate concurrent access to OWL knowledge bases. In addition, there are two higher level types of situations that can impair correctness. In short, they are owing to syntactic redundancies in OWL and its DL-based semantics. The first indirectly represents a transaction conflict that can be detected only by taking into account the higher semantic level of OWL syntactic instances. By handling the second type of conflict one can ensure that satisfiability of a KB is preserved for interleaved transactions. We call them *Expression Conflict* (E) and *Satisfiability Conflict* (S). The conflict of operations as defined by **Definition 6.4** shall be (O) in this categorization.

It is important to understand that E and S conflicts are a consequence of the specifics of OWL, not the specifics of the SI-based CC protocol. Therefore, they have to be considered independent of the actual CC protocol used.

What is more, E and S conflicts cannot occur, in general, for reads and updates on annotations and entity declarations. The reason is that both are outside the underlying

DL theory nor are there syntactic redundancies on either of them. Consequently, we do not need to pay attention to them subsequently.

### Expression Conflict

There can be situations where two transactions operate on OWL syntactic instances, say  $\psi_1, \psi_2$ , which are structurally different but which are semantically (logically) equivalent. In other words, given two distinct data items  $di_{\psi_1}^{\text{OWL}}, di_{\psi_2}^{\text{OWL}}$  one cannot generally conclude that the syntactic instances  $\psi_1, \psi_2$  are not logically equivalent. Formally, there can be cases in which

$$\psi_1 \Leftrightarrow \psi_2 \quad \text{while} \quad di_{\psi_1}^{\text{OWL}} \neq di_{\psi_2}^{\text{OWL}}$$

and where  $\Leftrightarrow$  stands for logical equivalence. With the machinery introduced thus far, these transactions would not O-conflict because their OWL data items are distinct. The semantic equivalence of  $\psi_1, \psi_2$ , however, implies the need to treat them as the same. Such equivalences can exist for syntactic sugar constructs available in OWL, which are motivated by the wish to allow for more concise knowledge descriptions. Also, in OWL there is redundancy in the set of concept constructors (e.g., the negation constructor *ObjectComplementOf* allows to eliminate conjunctions in favor of disjunctions by using negation). **Example 6.3** illustrates these semantic equivalencies for different types of axioms, assertions, and concept expressions.

#### Example 6.3

(a) The following axiom

$$\psi_1 = \text{DisjointUnion}(:\text{Parent} : \text{Mother} : \text{Father})$$

and the axioms

$$\psi_2 = \text{EquivalentClasses}(:\text{Parent} \text{ ObjectUnionOf}(:\text{Mother} : \text{Father}))$$

$$\psi_3 = \text{DisjointClasses}(:\text{Mother} : \text{Father})$$

are clearly structurally different. Hence, they have distinct OWL data items and two transactions can never O-conflict. However, at the higher semantic level  $\psi_1$  is equivalent to  $\psi_2, \psi_3$  because  $\psi_1$  is a syntactic shortcut expressing the same knowledge than  $\psi_2, \psi_3$  together.

(b) This is analogously the case for the individual assertions

$$\psi_4 = \text{DifferentIndividuals}(:\text{OSIRIS} : \text{ISIS} : \text{HORUS})$$

$$\psi_5 = \text{DifferentIndividuals}(:\text{OSIRIS} : \text{ISIS})$$

$$\psi_6 = \text{DifferentIndividuals}(:\text{ISIS} : \text{HORUS}) .$$

$\psi_4$  is again a concise form of  $\psi_5, \psi_6$ .

(c) The following concept expressions are logically equivalent (De Morgan's law).

$$\psi_7 = \text{ObjectComplementOf}(\text{ObjectIntersectionOf}(A B))$$

$$\psi_8 = \text{ObjectUnionOf}(\text{ObjectComplementOf}(A) \text{ ObjectComplementOf}(B))$$

Again,  $\psi_7, \psi_8$  are not structurally equivalent, thus, would have distinct OWL data items. This can analogously occur for universally and existentially quantified concept restrictions (*ObjectAllValuesFrom*, *ObjectSomeValuesFrom*) or cardinality concept expressions, e.g., an exact cardinality restriction is a shortcut of a minimum and maximum cardinality restriction.

An E-conflict can be detected by pre-processing OWL syntactic instances that are parameters of the basic operations and transforming them into a syntactic normal form. The normalized forms are then used for standard O-conflict analysis while retaining, of course, the actual operation. This leads us to the following formal definition of expression conflicts between operations in a schedule.

**Definition 6.6** (Expression Conflict). *Let  $o_1(\psi_1), o_2(\psi_2)$  be two operations in a history  $H$  that would O-conflict according to [Definition 6.4](#) if  $\psi_1, \psi_2$  were structurally equivalent. Let  $Nf$  be a normalization procedure that transforms an OWL syntactic instance  $\psi$  into its normalized form  $Nf(\psi)$  and that preserves logical equivalence ( $\psi \Leftrightarrow Nf(\psi)$ ). Then,  $o_1, o_2$  E-conflict w.r.t.  $H$  iff  $o_1(Nf(\psi_1)), o_2(Nf(\psi_2))$  O-conflict w.r.t.  $H$ .*

Notice herein that we assume every pair of operations within a transaction not to E-conflict, which is a reasonable sanity assumption that one would want to make anyway on applications that generate transactions.

From [Definition 6.6](#) we can easily derive the criterion for an E-conflict under the SI protocol.

**Definition 6.7** (SI Expression Conflict). *Let  $T_1, T_2$  be two transactions and  $\delta(T_1), \delta(T_2)$  their change sets, respectively. Let  $Nf$  be a normalization procedure that transforms a change set  $\delta(T)$  into a normalized change set  $\delta^{Nf}(T)$  such that each OWL syntactic instance  $\psi$  occurring in  $\delta(T)$  is substituted by its normalized form  $Nf(\psi)$  in  $\delta^{Nf}(T)$ . Then,  $T_1, T_2$  E-conflict iff their normalized change sets are not disjoint  $\delta^{Nf}(T_1) \cap \delta^{Nf}(T_2) \neq \emptyset$ .*

There are several types of normal forms of which some of them can be used only for a subset of OWL. Concept expressions (using Boolean connectives, cardinality restrictions, or existential/universal quantifiers) can be normalized to *Negation Normal Form* (NNF). More precisely, a complex concept is in NNF iff negation ( $\neg$ ) occurs only in front of atomic concepts. NNF is obtained by applying De Morgan laws to push  $\neg$  inward (e.g.,  $\neg(C \sqcap D) \Leftrightarrow \neg C \sqcup \neg D$ ). Other candidates for concept expressions are the *Disjunctive Normal Form* (DNF), the *Conjunctive Normal Form* (CNF), or the *Prenex Normal Form*. A role is in *Inverse Normal Form* iff inverse applies only to the role name; that is,  $(R^-)^-$  is transformed to  $R$ . Finally, normalization of syntactic sugar constructs (e.g., [Example 6.3](#) (a) and (b)) is a simple rule-based rewriting function systematically transforming them into a set of equivalents. These rules can easily be derived from OWL's structural specification [MPSP09].

Normalization is rather cheap in terms of computational time complexity. For instance, NNF is linear in the size (number of terms) of concept expressions [HNSS90]. On the other hand, some normal forms can grow exponential in space. CNF and DNF may increase a concept expression of  $n$  terms in worst-case to  $2^n$  terms due to the distributive

law.<sup>6</sup> Normalization for some  $n$ -ary syntactic sugar constructs (e.g., *DifferentIndividuals*, *DisjointClasses*) is polynomial in space, otherwise it is linear (e.g., *SameIndividuals*, *EquivalentClasses*). Conversely, normalization can also involve simplifications, for instance, factoring out tautologies.

### Satisfiability Conflict

In general, if contradictory knowledge is added to a KB  $\mathcal{K}$  then it becomes *unsatisfiable*; that is, there exists no model  $\mathcal{I}$  that satisfies all axioms in the TBox and assertions in the ABox (see [Definition 3.6](#)). Recap, reasoning is no longer meaningful in this case because one can conclude anything from false premises. This is a general concern regardless of whether  $\mathcal{K}$  is updated concurrently or not. It can be avoided by a guard mechanism that analyzes and rejects updates which would result in an unsatisfiable KB (see [Section 3.1.5](#)). In other words, this mechanism preserves consistency at DL level. In case of concurrent access, however, it is no longer sufficient to analyze updates made by an application in isolation; that is to say, concurrent access necessitates combined analysis of concurrent transactions. [Example 6.4](#) illustrates this.

#### Example 6.4

Given an OWL KB  $\mathcal{W}$  and its core KB  $\mathcal{K}$ , consider the transactions

$$\begin{aligned} T_1 &= a(\text{DisjointClasses}(:A :B)) \odot \quad \text{and} \\ T_2 &= a(\text{ClassAssertion}(:I1 :A)) \ a(\text{ClassAssertion}(:I1 :B)) \odot . \end{aligned}$$

It is assumed that  $\mathcal{K}$  is initially consistent, that  $T_1, T_2$  start from the same state (snapshot) of  $\mathcal{W}$ , and that  $:A$  and  $:B$  are already declared as concepts in  $\mathcal{W}$ . Then,  $T_1, T_2$  do neither O-conflict nor E-conflict because syntactic instances are normalized and structurally different; hence, both can commit. If both commit then  $\mathcal{K}$  becomes inconsistent at DL level afterwards: the individual named  $:I1$  cannot be both an instance of  $:A$  and  $:B$  since the concepts were declared to be disjoint. This is due to the fact that the knowledge represented by the updates of  $T_1, T_2$  contradicts each other.

An S-conflict is consequently the situation where the execution of two or more operations of different transactions leads to inconsistency of the KB, which is formally expressed by the following definition.

**Definition 6.8** (Satisfiability Conflict). *Let  $o_1, \dots, o_n$ ,  $n \geq 2$  be each either an add or a delete operation in a history  $H$ . Then  $o_1, \dots, o_n$  S-conflict w.r.t.  $H$  iff the operations are from at least two different transactions  $T_1, T_2$  that are interleaved in  $H$  and commit of  $T_1, T_2$  leads to an unsatisfiable KB.*

A similar assumption to E-conflicts is made here: within a transaction there is no S-conflict. It is equally reasonable to assume that a transaction  $T$  generated by an application is itself consistent, meaning that if  $\mathcal{K}^T$  is the result of executing  $T$  on the empty KB then  $\mathcal{K}^T$  is satisfiable (consistent).

<sup>6</sup>For example, the following non-DNF concept expression having 2 terms is normalized in DNF to a concept expression having 4 terms:  $(C_1 \sqcup C_2) \sqcap (C_3 \sqcup C_4) \Leftrightarrow (C_1 \sqcap C_3) \sqcup (C_1 \sqcap C_4) \sqcup (C_2 \sqcap C_3) \sqcup (C_2 \sqcap C_4)$ .

Also, note that in case of OWL an S-conflict can only occur when new syntactic instances are added but not for deletion. This is a consequence of the fact that the DL  $SR\mathcal{OIQ}(\mathbf{D})$  underlying OWL 2 (and also any sublanguage such as the DLs underlying the profiles EL, QL, RL) is monotonic (see [Observation 3.1](#)). We could therefore safely drop the delete operation from [Definition 6.8](#). In other words, [Definition 6.8](#) is a generalization including non-monotonic DLs.

An S-conflict can be detected under the SI protocol by extending the guard mechanism to take into account local changes of a transaction  $T$  and the updates of meanwhile committed transactions, which essentially means to analyze changesets.

**Definition 6.9** (SI Satisfiability Conflict). *Let  $T_1$  be a transaction assigned with a start timestamp  $t_s^{T_1}$  and a commit timestamp  $t_c^{T_1}$ . Let  $\mathbf{S}$  be the initial set of syntactic instances in an OWL KB  $\mathcal{W}$  visible at the time when  $T_1$  starts (snapshot of  $T_1$ ) and whose core KB  $\mathcal{K}$  is satisfiable. Let  $A_{T_1}$  be the set of syntactic instances added by  $T_1$ . Let  $T_2$  be a transaction that committed in the interval  $[t_s^{T_1}, t_c^{T_1}]$  and  $A_{T_2} (D_{T_2})$  the set of syntactic instances that were added to (deleted from)  $\mathcal{W}$  by  $T_2$ . Then,  $T_1$  S-conflicts with  $T_2$  iff the resulting set  $\mathbf{S}' = (\mathbf{S} \setminus D_{T_2}) \cup A_{T_2} \cup A_{T_1}$  represents an OWL KB  $\mathcal{W}'$  whose core KB  $\mathcal{K}'$  is not satisfiable.*

If DL consistency needs to be preserved then  $T_1$  commits successfully only if the updated core KB  $\mathcal{K}'$  is satisfiable. In the general case there might be any amount of transactions  $T_i$  ( $i \geq 0$ ) which committed in the interval  $[t_s^{T_1}, t_c^{T_1}]$ . Let  $\mathbf{A} = \bigcup A_{T_i}$  and  $\mathbf{D} = \bigcup D_{T_i}$  be the set of added/deleted syntactic instances by these transactions in this interval. Then,  $T_1$  commits successfully only if  $(\mathbf{S} \setminus \mathbf{D}) \cup \mathbf{A} \cup A_{T_1}$  represents an OWL KB  $\mathcal{W}'$  whose core KB  $\mathcal{K}'$  is satisfiable.

Depending on the expressivity used in  $\mathcal{K}$ , satisfiability checking has worst-case polynomial complexity for tractable OWL profiles EL, QL, RL, exponential complexity for highly expressive OWL DL, up to the point – OWL Full – where it is undecidable in general (see [Section 3.3.3](#)). Checking for satisfiability conflicts can, therefore, be expensive compared to the other two types of conflicts.

### 6.2.7 Extended Commit Protocol

Jointly addressing E, S, and O conflicts requires an extension of the commit rule of the SI-based CC protocol, see [Section 6.2.5](#). From an algorithmic point of view, this involves essentially a sequence of three phases before a transaction is permitted to commit. We call this the commit pipe, depicted in [Figure 6.2](#). It works as follows.

For each transaction  $T$ , phase one starts with transaction begin ( $t_s^T$ ) and lasts until  $T$  is ready to commit. Updates (add and delete operations) submitted by  $T$  are normalized at their commencement and the normal form is kept in  $T$ 's changeset.

Phase two starts when a transaction  $T$  is ready to commit. A new commit timestamp  $t_c^T$  is assigned and its normalized change set is checked for disjointness against the normalized change sets of all transactions that committed in its lifetime interval  $[t_s^T, t_c^T]$ . If there is no O-conflict then this implies that there is also no E-conflict because of preceding normalization. Phase 3 can be started in this case. Otherwise, the  $T$  will be aborted.

In phase 3 the change set is checked for S-conflicts according to [Definition 6.9](#). This requires the use of a reasoning engine that is able to determine if applying the updates

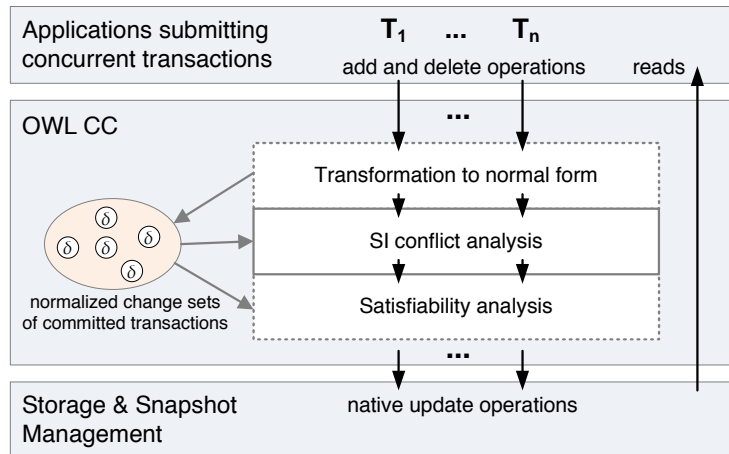


Figure 6.2: Commit Pipe for OWL Concurrency Control.

preserves satisfiability. Only if phase 3 also succeeds updates are sent irrevocably to the underlying storage layer to be made permanent. Otherwise, the system rejects the transaction and might return an appropriate error code to the application.

A change set of a committed transaction  $T_c$  needs to be kept as long as there is an active transaction  $T$  in the system which started earlier than the commit of  $T_c$  ( $t_s^T < t_c^{T_c}$ ).

Phase 3 is done at last because it can be much more expensive than phase 1 and 2 together. This is a performance optimization and is otherwise not conceptually presupposed. Phase 1 and 3 can, in fact, be optional depending on the consistency requirements of applications. Amongst other aspects, this is discussed in the next section.

### 6.2.8 Correctness of the Protocol

SI is known to guarantee the absence of a number of concurrency phenomena and anomalies, namely dirty reads and writes, lost updates, non-repeatable reads, and read skews [BBG<sup>+</sup>95]. Moreover, SI is known to generate recoverable schedules that are also strict [NØ10] (i.e., the class of schedules generated by SI are a proper subset of the class of schedules that are recoverable and avoid cascading aborts). These results can be transferred from the read/write model to our model because, based on commutativity properties of operations, we apply the standard SI commit rule. Since there can also be indirect conflicts owing to syntactic redundancies in OWL (i.e., data items that are distinct though the corresponding syntactic instances are logically equivalent), we take normalized change sets for conflict analysis, thereby ensuring that we do not miss such indirect conflicts.

It is well known, however, that using standard SI data is unprotected from so-called *write skews*. In essence, a write skew is the violation of an integrity constraint  $IC$  that exists between two or more data items (i.e., an invariant that *is* per definition or natural law). More precisely, a write skew is the situation in which the sum of the outcomes of two transactions lead to violation of  $IC$ , though each of them when viewed in isolation respects  $IC$ . The existence of this anomaly is plausible bearing in mind that transactions in a multiversion system each operate on their own snapshot; hence, they are isolated



from each other to the extent of being “blind” to what other transactions running in parallel do. The consequence of this anomaly is that SI permits schedules that would not occur for a serial execution, which means that SI does not comply with the ANSI/ISO SQL isolation level serializable.

Probably the most important observation concerning the write skew anomaly is the cooccurrence of two specifics:

1. A domain-specific integrity constraint is violated (as opposed to “core” integrity constraints such as referential and entity integrity).
2. More importantly, only multiple transactions, each of them being correct in the sense that each meets the integrity constraint on its own, can lead to violation of the integrity constraint, not a single one.

The prominent example of such a domain-specific integrity constraint (used to illustrate a write skew in [BBG<sup>+</sup>95, CRF09]) is that of two bank accounts  $x, y$  whose total balance must remain non-negative  $IC := x + y \geq 0$ . Write skews can analogously occur in case of add and delete operations on OWL data items because domain integrity constraints do obviously not disappear just because the domain is represented by means of an OWL KB, see [Example 6.5](#).

#### Example 6.5

Suppose there is a domain-specific integrity constraint  $IC(\psi_1, \psi_2)$ : either of  $\psi_1, \psi_2$  may exist but not both together. Also suppose that neither of them is initially contained in a KB  $\mathcal{K}$  being accessed. Then the transactions

$$\begin{aligned} T_1 &= r(\psi_1) r(\psi_2) a(\psi_2) \odot \quad \text{and} \\ T_2 &= r(\psi_1) r(\psi_2) a(\psi_1) \odot \end{aligned}$$

produce a write skew if both are executed on the same state of  $\mathcal{K}$  and both commit because  $IC$  is violated then. Observe that they do not O-conflict; we shall further assume that they neither E- nor S-conflict. The two transactions might be submitted by two programs that first test whether  $\psi_1, \psi_2$  already exist. Since this is not the case both conclude that they can add either of them; by doing so  $IC$  is not violated from their isolated perspective. Since  $T_2$  happens to add  $\psi_1$  and  $T_1$  happens to add  $\psi_2$  they cause  $IC$  to become violated.

Using our protocol in this form therefore means that write skews can be avoided only if considered at application level; that is, applications that access a KB would need to ensure to generate “safe” transactions (by careful programming) that do not provoke write skews. This is, of course, not ideal since enforcing correct access (by transparent handling of these kind of conflicts) should be a matter of the KBMS. However, there are several approaches known in the literature that establish the serializability property for SI. A general approach with low overhead has been proposed in [CRF09]. Equally important, this approach preserves the property that reads are non-blocking. Serializability of transactions is analyzed at runtime (rather than statically at development time) by

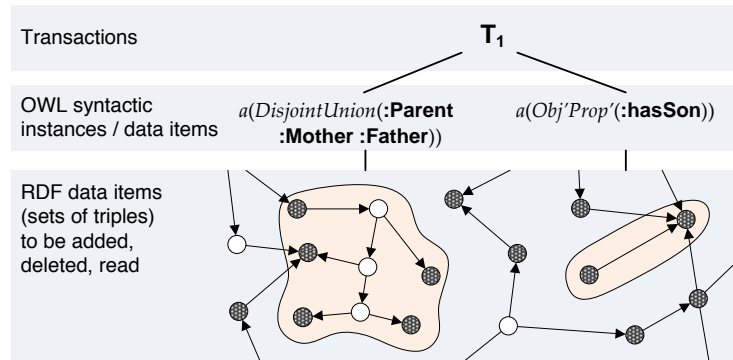


Figure 6.3: CC model levels for RDF triple store integration. Transactions consisting of operations over OWL syntactic instances result in operations over disjoint sets of RDF triples at lowest level.

extending conflict analysis with checks for distinctive non-serializable access patterns. These checks are essentially done based on an adapted multiversion serialization graph testing technique [Ady99, FLO<sup>+</sup>05]. However, the approach also has a downside: two transactions might be wrongly suspected to conflict when they actually do not (i.e., it produces false positives). Though the approach is transferable to our model, we have found that there is also a completely different approach that is devoid of false positives and therefore seems worth being explored, the basic idea of which will be sketched later on in the discussion [Section 6.6.1](#).

### 6.3 RDF Triple Store Integration

Applying CC at the level of OWL syntactic instances is compatible with their representation and storage as RDF triples. This means that our approach can be used to build (or extend existing) RDF triple stores to provide correct concurrent access to OWL knowledge bases if they should be represented as RDF triples. This relies on one main observation: Structurally different OWL syntactic instances are always mapped by the OWL-to-RDF mapping (see [Section 3.3.4](#)) into disjoint sets of RDF triples. This implies that it cannot happen that two update operations (add or delete) that operate over distinct OWL data items overlap/interfere at the level of RDF triples. Consequently, operations at the level of RDF triples are uniquely determined by the higher level operations over OWL data items. This results in the layered approach depicted in [Figure 6.3](#).

We use the notion of *RDF data items* as an ancillary tool to show that the sets of RDF triples resulting from the OWL-to-RDF mapping are mutually disjoint whenever they result from structurally different OWL syntactic instances. It should be mentioned that they would not find their way into an RDF triple store.

Given an OWL syntactic instance  $\psi$ , we denote its corresponding RDF data item with  $di_{\psi}^{\text{RDF}}$ . The RDF data item contains the transitive closure over blank subject nodes (a.k.a. *blank node closure*) of  $T(\psi)$  (see [Section 3.3.4](#)). This owes to the fact that OWL syntactic instances can be nested and triples resulting from the mapping of inner syntactic

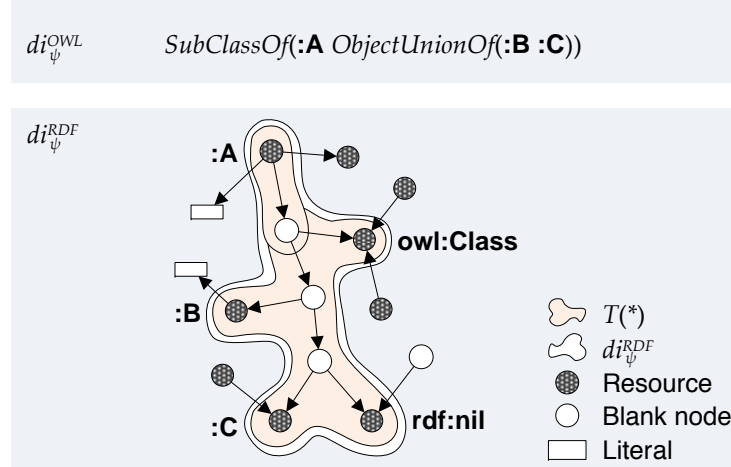


Figure 6.4: Data items at OWL and RDL level illustrated using a fictitious OWL syntactic instance.

instances need to be included if the inner syntactic instances are *anonymous*. The blank node closure (BNC) is recursively defined as follows. Let  $t = \langle s p o \rangle$  be a triple in an RDF graph  $\mathcal{G}$  where the subject node  $s$  is a resource (IRI) or a blank node. The blank node closure  $\text{bnc}(t, \mathcal{G})$  of  $t$  in  $\mathcal{G}$  is the smallest set containing (i) the triple  $t$  itself and (ii)  $\text{bnc}(u, \mathcal{G})$  where  $u \in \mathcal{G}$  is a triple matching the pattern  $\langle o * * \rangle$ ,  $o$  is a blank node, and  $*$  denotes any predicate and object node, respectively.<sup>7</sup> Then, an RDF data item is

$$di_{\psi}^{\text{RDF}} = \bigcup_{t \in T(\psi)} \text{bnc}(t, \mathcal{G}) \quad (6.2)$$

where  $\mathcal{G}$  is the RDF graph obtained from recursively mapping all syntactic instances contained in an OWL KB. Only if  $\psi$  is not nested, or if directly nested syntactic instances are non-anonymous, then the RDF data item encompasses exactly the set of triples to which it is mapped;  $di_{\psi}^{\text{RDF}} = T(\psi)$  in these cases.

It is easy to see that the number of triples contained in an RDF data item is not fix but varies depending on the actual syntactic instance  $\psi$  (cf. Table 3.4). Figure 6.4 illustrates how an RDF data item is determined by a nested OWL syntactic instance whose inner instance is anonymous.

**Theorem 6.10** (Disjointness of RDF Data Items). *Given an OWL KB  $\mathcal{W}$  containing two structurally different OWL syntactic instances  $\psi_1, \psi_2$ , represented by the RDF data items  $di_{\psi_1}^{\text{RDF}}$  and  $di_{\psi_2}^{\text{RDF}}$  being subsets of an RDF graph  $\mathcal{G} = T(\mathcal{W})$  then  $di_{\psi_1}^{\text{RDF}} \cap di_{\psi_2}^{\text{RDF}} = \emptyset$ .*

*Proof.* First, empty RDF data items which contain an empty set of RDF triples do not exist because there is no empty OWL syntactic instance. If two RDF data items  $di_{\psi_1}^{\text{RDF}}$ ,  $di_{\psi_2}^{\text{RDF}}$  contain just one triple then they are trivially distinct as long as the triples are distinct according to the RDF specification (otherwise they are the same  $di_{\psi_1}^{\text{RDF}} = di_{\psi_2}^{\text{RDF}}$ ).

<sup>7</sup>Note that the BNC is a subset of the so called *concise bounded description* [Pat05].

In any other case  $T(\psi_1) \subseteq di_{\psi_1}^{\text{RDF}}$  and  $T(\psi_2) \subseteq di_{\psi_2}^{\text{RDF}}$  are disjoint since the mapping  $T$  is unique. Finally, for all triples  $t \in T(\psi_1)$  and  $u \in T(\psi_2)$ ,  $\text{bnc}(t, \mathcal{G})$  and  $\text{bnc}(u, \mathcal{G})$  are mutually disjoint, also following from the uniqueness of  $T$ .  $T$  does create new blank nodes whenever necessary.  $\square$

The layered approach over OWL and RDF data items can be regarded as partly related to the concept of *Predicate Locks* (PL) [EGLT76]. This is justified by the fact that the definition of an RDF data item (see Equation (6.2)) can also be understood as a definition of a predicate satisfying exactly the set of triples that represent a given OWL syntactic instance.

Finally, we would like to point out that jointly addressing O, E, and S conflicts directly at the level of RDF triples is complicated if not infeasible in practice as it would introduce a considerable overhead: In order to check for E and S conflicts, one would inevitably need to reconstruct the semantics of OWL syntactic instances from triples first, for instance, by applying the inverse mapping  $T^{-1}$  from RDF to OWL.

## 6.4 Integration of Inferencing Engines

Integration with DL-based inferencing engines builds on the common ideas of (i) separating explicit from implicit knowledge and (ii) dynamic computation of implicit knowledge at query answering time versus materialization of implicit knowledge (pre-computation). We consider two types of system architectures depicted by Figure 6.5. Both have in common that explicit knowledge is maintained by a data store (which might be an RDF triple store) providing CC as discussed in Section 6.2. When materialization is used we propose, however, to internally partition the data store. This is done for performance reasons and to prevent the need for a more complex distributed transaction management. Altogether both architectures share the following properties:

1. Under concurrent updates it is important that inferencing engines see only the results of committed update transactions. This can be ensured by SI-based CC in general, no matter whether it is applied at the level of OWL syntactic instances or at the level of RDF triples.
2. The fact that reads to the data store are never delayed (by update transactions running in parallel) provides scalability only bounded by the technical access limits of the data store. More interestingly, it is even possible to use multiple inferencing engines in parallel. This allows for better scalability by concurrent query answering distributed over multiple inferencing engines.
3. The architectures are independent of the different types of reasoning algorithms that exist; notably, rule-based forward/backward chaining (e.g., OWLIM [KOM05]), Datalog engines (e.g., KAON2 [Mot06]), and tableau-based (e.g., Pellet [SPG<sup>+</sup>07], HermiT [MSH09], RacerPro [HM01b]).

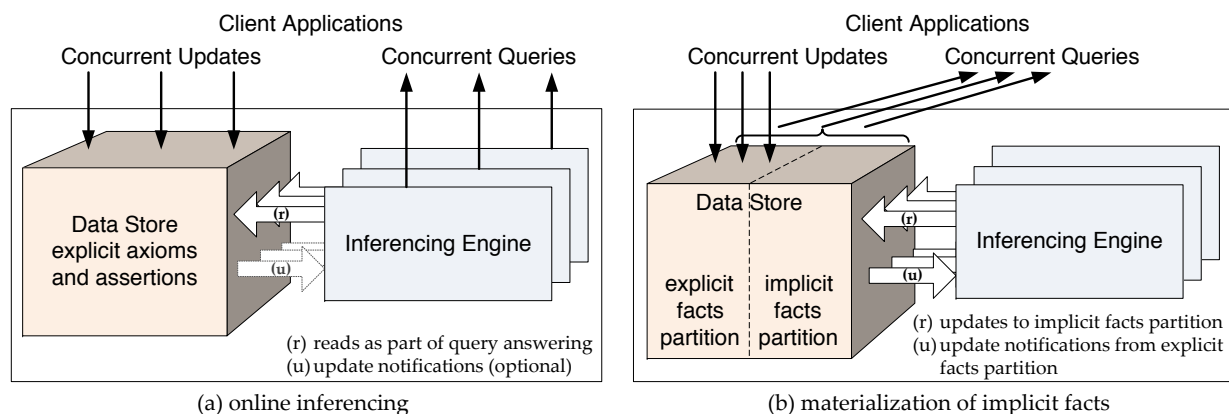


Figure 6.5: System Architecture Types for Integration of an OWL Data Store with Inferencing Engines.

### 6.4.1 Online Computation of Implicit Knowledge

The first type depicted by Figure 6.5 (a) considers inferences to be computed online as part of query answering requests. It is intended to be used when update frequency can become very high and computational complexity of reasoning is rather low, e.g., when a tractable OWL 2 profile is used (EL, QL, or RL).

Every update will be made directly to the data store and query answering (reads) will be generally handled via inferencing engines, assuming that an appropriate query interface is provided either directly by them or layered on top. Consequently, inferencing engines need to read from the data store in order to provide complete query answering over explicit and dynamically computed implicit knowledge. However, reasoning engines are purely read-only regarding the data store (i.e., they never need to update the data store). Since SI is used, they can read without additional delays and multiple instances can be used in parallel to distribute the load of concurrent query answering requests.

According to this architecture blueprint, an optional update notification mechanism exists from the data store directed towards inferencing engines. This is motivated by utilizing incremental reasoning capabilities provided by several inferencing engines. Whenever an update commits, its change set (added and deleted triples) would be propagated to inferencing engines. This allows them to keep internal caches up to date with changes to the data store. Finally, we note that with such a notification mechanism it would also make sense – for performance reasons of reasoning – to actually restrict the data store to maintain only the (possibly large) ABox and assume that the TBox and RBox is maintained internally by the inferencing engines.

### 6.4.2 Materialization of Implicit Knowledge

The second type of architecture blueprint addresses cases where update frequency is moderate and query requests (vastly) outnumber updates. In such cases it is often beneficial to precompute and materialize implicit knowledge and keep it in sync with up-

dates to the explicit knowledge. However, materialization is generally possible only if the expressivity of the underlying DL has the *finite model property*<sup>8</sup>; that is, if there cannot be cases in which one can infer an infinite number of axioms and/or assertions.

One might be tempted to directly apply a separation of explicit and implicit knowledge by using two independent data store instances. However, this imposes more complex CC in order to guarantee correct data access at a global level such that consistency spans both data stores. The reason is that in this case one has to apply distributed transaction management which would require dedicated coordination mechanisms between data stores to ensure atomicity (e.g., Paxos, Two-Phase Commit, or Commit Ordering). Our proposal still supports the separation of implicit and explicit facts and also the ability to distribute the load of inferencing to multiple engines. This is achieved, first, by using one data store which is internally partitioned in one for explicit and another for implicit knowledge. Second, by extending transactions such that updates to implicit knowledge *caused* by updates to explicit knowledge are committed all at once (or not at all), essentially making updates spanning both partitions atomic by combining them into one transaction. The partitioning also has the advantage that all implicit knowledge can be easily discarded (by clearing the partition).

We illustrate the need for combining updates to either of the partitions into one transaction using the following example.

#### Example 6.6

Imagine a transaction  $T_e$  making updates in the partition for explicit knowledge, i.e., that has a nonempty changeset  $\delta(T_e)$ . Furthermore, we assume that  $T_e$  does not conflict with another transaction. The successful commit of  $T_e$  implies updates in the partition for implicit knowledge, computed by an inferencing engine after  $T_e$  commits. Let us assume that these updates were applied to the partition for implicit knowledge by a transaction  $T_i$  having the nonempty changeset  $\delta(T_i)$ . Consequently,  $T_i$  does not start before  $T_e$  commits. In practice there would be a time interval between commit of  $T_e$  and commit of  $T_i$  in which another read-only transaction  $T_r$  might be executed. In this interval  $T_r$  sees the update  $\delta(T_e)$  but not yet  $\delta(T_i)$ ; that is, not the entailments of the update to the explicit knowledge. Obviously, this would be unsound w.r.t. to the underlying DL.

The missing knowledge anomaly described in [Example 6.6](#) can be avoided if  $T_e$  and  $T_i$  are combined into one transaction, essentially making application of explicit and implicit updates atomic. Since both updates become visible all at once in both partitions other transactions cannot see partial updates. This implies, of course, that the change sets  $\delta(T_e)$ ,  $\delta(T_i)$  need to be joined for conflict analysis against other active transactions.

<sup>8</sup>For instance, *SHOQ* is a rather expressive DL that has the finite model property.

## 6.5 CC applied to Semantic Service Execution

At the beginning of this chapter we have explained when semantic service execution requires read and update queries to be executed on shared KB (cf. [Figure 6.1](#)). In this respect, there are two important aspects to consider:

1. Temporal happened-before relations over changes in the domain need to be reflected accordingly in the KB. That is, a strict precedence in the control flow must be reflected exactly in the order in which updates made on the KB become visible. In contrast, a partial precedence should be reflected in a best effort manner, at least, meaning that the order in which changes are observed versus become visible may be different.
2. Since the KB is used for decision making, promptly updating it to changes in the domain and promptly checking whether required conditions hold is important insofar as it reduces the probability of suboptimal or even wrong decisions due to tardiness.

Let us begin with the second aspect. The point at which queries are issued (cf. [Figure 6.1](#)) naturally follows the aim of promptness: given the disposition of the service model, preconditions are checked as late as possible while effects are applied as early as possible. The concurrency control protocol that has been described in this chapter is applied to this in an equally natural way: precondition queries prior to invocation and effect update queries upon completion are encapsulated, respectively, by one transaction as depicted in [Figure 6.6](#). More precisely, precondition checking is encapsulated by one read-only transaction  $T_P$  possibly comprising multiple queries depending on how many preconditions a service has. This ensures reading the same state of the KB for each precondition. Effects are applied upon successful completion in one update transaction  $T_E$  possibly comprising multiple add, delete operations depending on the number of effects a service has. This ensures that effects are applied atomically.

The transactional model is thereby one with multiple transactions per service instance. This raises the question whether combining these transactions along the lines of multilevel transactions is worthwhile or even implied. More specifically, whether a model is used in which the execution of a service corresponds to a top-level transaction that encapsulates the transactions for precondition checking and effect updates as direct sub transactions (i.e., two transaction levels). We can immediately conclude that the concept of closed nested transactions [Mos85] is not appropriate. The reason is that the effects applied by update transactions would be made visible to other transactions only upon commit of the top-level transaction since commit of nested transactions is deferred until the commit of the top-level transactions. Hence, representation of changes in the domain is (more or less) delayed – a property that is clearly conflicting with the aim of prompt representation. The concepts of open nested transactions [WS92] and composite transactions [ABFS97] were introduced to overcome exactly this restriction, the latter of which being a generalization of the former. Both models are in principle capable of incorporating the two requirements made above. The commit of sub transactions is not deferred and the precedence order given by the control flow can

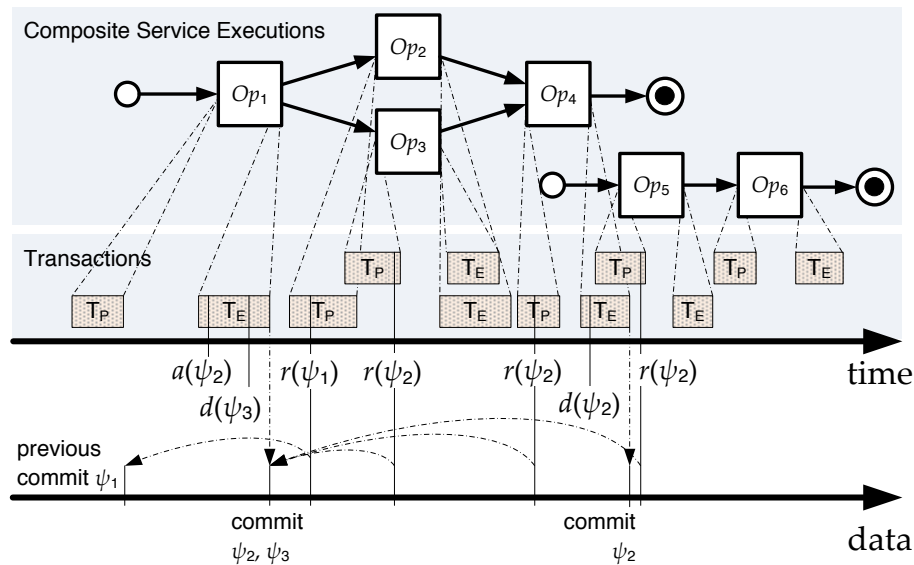


Figure 6.6: Mapping of read and update queries (see Figure 6.1) for service execution to read/update transactions and example operations over OWL syntactic instances.

be correctly enforced. Yet the main potential of these models lies in defining transactional semantics for services to enable reasoning about their transactional correctness, namely atomicity and isolation. In this regard, it is important to make a distinction between two related but different aspects: In this section we are concerned with coordinating access to a shared KB that is (primarily) used for representing the world state as it evolves. This should not be mixed up with the problem of how to meaningfully define and provide transactional semantics for the execution of services themselves. More specifically, advanced transaction models such as the concept of *flexible transactions* [ZNBB94], *Sagas* [GMS87, GGK<sup>+</sup>91], or *transactional processes* [SABS02] provide generalized notions of atomicity that guarantee termination in a well-defined state over a services' sub processes executed in the underlying sub systems. This aspect is not the subject of this section and has been discussed in Section 5.5.

Coming back to the first requirement stated above, correctly representing temporal happened-before relationships involves enforcing the temporal succession as specified by the control flow over the precondition checking and effect creation transactions. A strict precedence in the control flow therefore implies a strict precedence over the commit order of corresponding transactions. More precisely, if  $t_1, t_2$  are two ordinary transitions of some control flow such that  $t_1$  strictly precedes  $t_2$  (which is the case if there is a path  $W$  such that  $t_1$  is the first and  $t_2$  is the last element on  $W$ ) then  $T_P^{t_1} < T_E^{t_1} < T_P^{t_2} < T_E^{t_2}$  where  $<$  denotes the commit order.

If the SI protocol is applied for CC then precondition checking sees a snapshot of the latest domain state committed at this instant in time without being blocked nor blocking any other active transaction, which makes the protocol particularly suitable. Effects become visible in an atomic way all at once upon completion of each operation. Effects, however, do not necessarily have to be applied atomically all at once per operation. This actually owes to the service model with indivisible request-reply style invocation of op-



erations. Considering an advanced service model where effects of (very) long running operations can emerge any time during execution, it would be more appropriate to submit update transactions promptly with emergence of each single effect; thus, achieving an even more fine grained and prompt representation of effects in the knowledge base.

## 6.6 Discussion

What makes our approach that considers the non-blocking SI protocol particularly suitable for applying it as a concurrency control means to knowledge bases rather than the conventional read/write model or a locking-based protocol such as S2PL? To discuss this question we shall consider both the qualitative level along serializability properties (i.e., which isolation levels are guaranteed) and the quantitative level along runtime performance properties.

### 6.6.1 Correctness

A discussion of correctness properties should, first and foremost, ask the question which isolation level would be sufficient from an application point of view. That said, we have to accept that the required isolation level is completely determined by the concrete domain of application. Therefore, the pat answer is that there is no single answer. This is basically the same result that we get for classic database management. The one law that seems apparent here is that the larger the domain gets, the more likely it is that a relaxed correctness is the only feasible solution on the large scale (enforced by a protocol such as SI or one that provides *eventual consistency* [Vog09]), whereas a strict correctness notion such as serializability is required on the small scale (i.e., in subdomains).

Having discussed the isolation level currently provided by our protocol already in [Section 6.2.8](#), we sketch the basic idea of a completely different correctness notion to conventional serializability that is purely integrity constraint based. We call it *Integrity Isolation* ( $I^2$ ). The basic idea originates from the observation on write skews described in [Section 6.2.8](#). In fact, it can be seen as a generalization of S-conflict checking.

Recap, the leakage line impeding correctness is essentially the cooccurrence of an integrity constraint  $IC$  that gets violated only by the sum of the updates of multiple transactions. This gives raise to the idea of making  $IC$  *known* to the KBMS (or DBMS). If the KBMS is aware of  $IC$  and if  $IC$  is decidable then it could, in principle, transparently verify whether interleaved transactions conflict w.r.t.  $IC$ . Regarding the SI-based protocol this means that the change set of a transaction  $T$  that is ready to commit is checked regarding  $IC$  with the change set of every transaction that committed in its lifetime  $[t_s^T, t_c^T]$ . If  $IC$  is not violated (and there is also neither of an OES conflict) then  $T$  commits. Abstracting from the protocol, Integrity Isolation is defined as follows.

**Definition 6.11** (Integrity Isolation). *Let  $S$  be a schedule,  $S'$  its commit-projection, and let  $C$  be a finite set of integrity constraints. The non-aborted transactions in  $S$  are integrity isolated w.r.t.  $C$  if neither of them reads data that violates any of the constraints in  $C$  nor does the commit of transactions in  $S'$  leads to violation of any of the constraints in  $C$ .*

Algorithmically, integrity constraint checking can be integrated in the SI-based access protocol by proceeding in the same way as for S-conflict checking.<sup>9</sup> Likewise, S-conflict checking is a special case of Integrity Isolation because satisfiability can be seen as an integrity constraint.

Observe that Integrity Isolation is not equivalent to serializability in general since C might be incomplete (i.e., there might be integrity constraints in the domain that are not in C, whatever the reason for this may be).

OWL itself would be one candidate for specifying integrity constraints, especially as the use of OWL as an integrity constraint specification language has already been proposed and discussed [MHS09, TSBM10]. This appears attractive since the same formalism is homogeneously used. On the other hand, it is not possible to express all kind of integrity constraints in OWL. For example, one cannot express the prominent write skew example from the finance domain ( $x + y \geq 0$ , see Page 163) because addition does not exist in OWL. Therefore, one question that needs to be further explored for this approach is what would be practicable and sufficiently expressive integrity constraint specification languages/formalism. It seems evident that a Turing-complete language is required in order to be general enough. One possibility that comes to mind is to formulate integrity constraints directly as a Boolean query using, for instance, SPARQL.

Another question regarding Integrity Isolation concerns consequences on the runtime performance due to the fact that integrity constraint checking becomes part of transaction processing. The runtime performance change is presumably dominated by the computational complexity induced by the actual formalism used. What is more, the approach also comes at the cost of additional efforts at design time for modeling the constraints. The latter, on the other hand, might be desired anyway as it could be used, in addition, for checking soundness of the modeled domain at design time.

To the best of our knowledge, the idea of making integrity constraints explicit and integrating them for isolation reasoning has not been described nor implemented yet, though integrity constraint checking within single transactions is a standard and long standing part of database technology.

### 6.6.2 Performance

Contrasting our approach with the read/write model and other access protocols, particularly those that are blocking, we argue in the following pro (+) and contra (–) our approach on performance related concerns. There is one more point, which is inherent in the model rather than the access protocol, that seems disadvantageous at first sight but which can be eliminated by optimization ( $\pm$ ).

- + Reads are never delayed nor do they delay concurrent update transaction. They can be done in parallel by virtually any number of clients at the technical scalability limits of the system. Non-blocking reads are especially worthwhile for query

<sup>9</sup>It should be noted, in this context, that “integrity constraints have two flavours – static and dynamic” [Rei88]. Enforcement of the former does not involve taking previous states of a KB into account whereas this is the case for the latter. The example of a dynamic constraint mentioned in [Rei88] is that of employment salaries that must never decrease, which requires reading the previous value and comparing it with the new value in order to determine whether an update adheres to the constraint.

answering that involves possibly long-running reasoning. The time required to answer a read query does not have an impact on concurrent update transaction processing (i.e., both are completely decoupled). In contrast, the duration for which read locks need to be held in a blocking protocol directly affects the performance of concurrent update transactions in case they need to wait unless a lock is released.

- The longer a (read-only) transaction lasts in time, the more likely it is that its snapshot becomes outdated, provided that there is a considerable amount of updates taking place concurrently in this time. This becomes especially relevant to the case of representing a world state in the KB. It seems obvious that relying on an outdated world state can lead to problems (e.g., a decision made based on an outdated state that would not have been made based on the most recent state). On the other hand, considering a blocking protocol, the longer a transaction lasts the more likely it gets blocked because a data item that is to be read/updated has been accessed by another transaction. Therefore, it is more relevant to the case of world state representation not to let the proportion of transactions length and update frequency diverge too much; that is,

$$\frac{\text{global number of updates during } length(T)}{length(T)}$$

should not grow too large ( $length(T)$  is the duration of  $T$  in time).

- + Snapshot management of immutable data items is simpler compared to mutable data items because the former have a binary lifecycle (they can either exist or not) as opposed to the need of managing versions in case of mutable data items.
- Snapshot management of immutable data items can lead to higher space consumption because they are entirely deleted and added as opposed to mutable data items where usually just a fraction is updated (e.g., a record of which a single field is updated).
- ± Direct (naive) implementation of the add and delete operation leads to two accesses to (stable) memory for what is an update, as opposed to just one access in the read/write model. To illustrate this, let us come back to [Example 6.2](#). Updating the balance of the bank account :ACC1 by transaction  $T_{wd}$  is achieved by a delete of the obsolete assertion followed by an add of the current assertion ( $d(\psi_1) a(\psi_3)$ ). If they are directly executed on memory then this would result in two accesses: (i) return the space allocated by  $\psi_1$  to free memory; (ii) allocate space for  $\psi_3$  and store  $\psi_3$  in it. Considering the fact that  $\psi_1$  and  $\psi_3$  differ only in the number value (i.e., the subject in RDF terms), we can replace this by one memory access that writes  $\psi_3$  to the memory space that stores  $\psi_1$  (as it would be the case in the read/write model). In order to implement this optimization, we must be able to identify so-associated delete-add operation pairs, which can be achieved in two ways. Either the KBMS is explicitly told as an additional part of a transaction what are associated delete-add operations; hence, the application needs to provide this information, which is easily conceivable. Otherwise, the KBMS itself analyzes each transaction and checks whether there are pairs of operations that can be optimized this

way. The former appears favourable because it does not induce additional work in the KBMS, though it requires extension of applications and the transactional interface of the KBMS.

- + Integration of reasoning engines is simple, scalable, and (i) online computation of implicit knowledge at query answering time vs. (ii) materialization of implicit knowledge is supported (provided that an OWL profile is used which has the finite model property).

Among the different types of conflicts, S-conflict checking does certainly has the potential of dominating the overall performance as its complexity is determined by the complexity of basic satisfiability reasoning in the underlying DL actually used. On the other hand, S-conflict checking can be deactivated if (i) logical consistency is not a prerequisite for reasoning (e.g., paraconsistent reasoning) or if (ii) it is known that client applications behave such that they cannot create such update conflicts. Consequently, it should be seen as an optional feature that one might want to turn off deliberately. What is more, the presence of an S-conflict not necessarily requires rejection (abort) of one transaction. It is also possible to try to resolve the conflict. In fact, resolving an S-conflict corresponds to the belief revision/update problems (see [Section 3.1.5](#) and [Section 4.2.2](#)). This means that a conflicting transaction would be extended by additional delete and/or add operations which resolve the conflict.

Likewise, E-conflict checking is not required if all clients agree on submitting transactions in which syntactic instances are always in the same normal form. On the other hand, the overhead induced by normalization might not be wasted. Reasoning engines usually require pre-processing and normalization anyways. Consequently, they can take advantage by directly reusing the normalized forms, provided that a normal form is used that is the same as the one used by a reasoning engine. Whether this suggests storing normalized forms rather than (possibly more concise) unnormalized originals is an aspect beyond the scope of this thesis.

## 6.7 Summary

Taking OWL syntactic instances as first class citizens for applying concurrency control directly on them is the basic principle that allows for combining the notion of data consistency with the higher level notion of logical consistency for coordinating concurrent access to OWL knowledge bases. This idea can in principle be transferred to other axiomatic knowledge representation formalisms in which axioms are immutable (i.e., when revising/updating knowledge is accomplished by removing obsolete axioms and adding current axioms). As we have seen, the basic principle includes a departure from the prevalent database read/write model with the basic read and write operations towards a model in which the basic operations are read, add, and delete. We have shown how CC is applied in this model. It turned out that the paradigm shift is almost straightforward. Applying this model generally appears natural to us whenever data items are immutable.

Moreover, the basic principle of read/add/delete operations over OWL syntactic instances can be used regardless of the actual CC protocol, provided that the protocol relies on the notion of commutativity for determining whether a transaction is permitted to commit. Likewise, the specifics of OWL necessitates E- and S-conflict checking irrespective of the CC protocol used. In fact, this observation can also be generalized: every (axiomatic) knowledge representation framework that comes with syntactic redundancies and in which a notion of consistency exists as a prerequisite for reasoning procedures necessitates E- and S-conflict checking for concurrent access on a shared knowledge base. We therefore expect these to become standard parts of future KBMSs.

Motivated by the specifics of practical applications that rely on OWL as their information representation framework, we have chosen to adopt the non-blocking SI protocol rather than a classical locking-based protocol such as S2PL, even though “standard” SI is known to be not serializable in general. As we have discussed, we expect better performance for read-dominated workloads and if satisfiability checking is activated. The choice for SI is also not problematic bearing in mind that several techniques have been reported that enforce serializability. Yet this remains to be integrated in the overall approach.

Finally, we have also shown that the method is independent of the (physical) representation of OWL syntactic instances provided that (physical) data items are distinct whenever their corresponding OWL syntactic instances are distinct. As a result, one can use representation formats other than OWL’s abstract syntax. In particular, it is possible to use RDF triple stores underneath. An implementation of the method will be described in [Chapter 7](#) and results of a quantitative evaluation follow in [Chapter 8](#).



# 7

## Implementation

**T**HIS TECHNICAL CHAPTER focuses on the software developed as part of this thesis. Our contribution is threefold. First, we describe OSIRIS NEXT – an infrastructure for peer-to-peer style service execution in which CFI introduced in [Chapter 5](#) has been prototyped. OSIRIS NEXT works with OWL-S as the service description and process specification language, which implies that the underlying knowledge representation is based on OWL. OSIRIS NEXT further includes an execution strategy that allows for on-demand migration of ongoing service executions among remote peers. We explain the basic principle and how it is implemented. Afterwards, we describe two performance optimization techniques that have been implemented to speed up sub-tasks that may recur frequently for service execution using semantic services. While we use these techniques in the context of service execution, they are not limited to this use case, and are applicable in general whenever the same parts of a KB are frequently accessed/queried. Third, we describe an OWL store that implements the CC method introduced in [Chapter 6](#). This chapter is correspondingly divided into three sections. Finally, it should be mentioned that all software is written in Java.

### 7.1 OSIRIS NEXT

We have developed OSIRIS NEXT<sup>1</sup> as a prototype to implement our view of a distributed, modular, and semantic-aware service execution infrastructure. It combines a rich set of features from Distributed Systems, Database Technology, Process Management, and Semantic Technologies. It also provides the basis on which our method for flexible failure handling introduced in [Chapter 5](#) has been implemented and evaluated.

OSIRIS NEXT builds upon two lines of previous work. First, the *Open Service Infrastructure for Reliable and Integrated process Support* (OSIRIS) that has been originally developed at ETH Zurich [SWSS03, SWSS04, SST<sup>+</sup>05]. Second, OSIRIS-SE (Stream Enabled) – a successor that has been developed in the context of research work on reliable

---

<sup>1</sup>Available Open Source (LGPL) at <http://on.cs.unibas.ch>.

data stream processing at University of Basel [BS07, BS11].<sup>2</sup> While the commonality between these three systems is that they feature distributed and decentralized execution, the execution system implemented in OSIRIS NEXT has been developed independently of the original OSIRIS process execution system (which is also used in OSIRIS-SE). In fact, both are different regarding the way in which execution is distributed, which is further explained in the next section and [Section 7.1.2](#).

### 7.1.1 Architectural Overview

At its bottom layer, OSIRIS NEXT is essentially a component-based and message-oriented middleware that supports reliable point-to-point (e.g., one-way, request-reply) as well as publish/subscribe message exchange patterns. At the higher level, *application services* as well as *system services* can be deployed. The latter provide basic infrastructure-related services, and they may also be used by application services. Much of the design at these two levels goes back to the original OSIRIS system and is conceptually still the same, though it has been extensively refactored and streamlined.

Application services are either *standard* Web Services accessible via SOAP or *native* application services, which differ from the former in that they are accessible via OSIRIS NEXT-specific messaging. Both system and native application services are designed at programming level as managed components that undergo a basic lifecycle. The execution engine implemented in OSIRIS NEXT is realized as one such component.

### Distributed System Organization

The underlying distributed system model considers a set of *peers* (which may be synonymously called nodes).<sup>3</sup> Peers can reliably exchange *messages* (i.e., FIFO, no duplicates, eventual delivery) in both an asynchronous and synchronous way.<sup>4</sup> Publish/subscribe message exchange is available in addition and is implemented as a system service on top of the point-to-point message exchange primitive. The set of peers is not static, meaning that peers can join and leave at any time (i.e., peers may become disconnected).

[Figure 7.1](#) provides a high level organizational overview of OSIRIS NEXT. The main property is that there is no hierarchy; hence, it can be seen as a peer-to-peer structure. Execution engines are deployed at peers. Our design considers at most one execution engine instance per peer, which is not a limitation since each engine is designed to allow for concurrent execution of multiple service instances. Execution engines interact with each other in order to execute composite services in a distributed way. More precisely, *distributed execution* in OSIRIS NEXT means that multiple peers may be involved,

---

<sup>2</sup>Apart from the data stream processing unique to OSIRIS-SE, it is a reimplementations of major parts of OSIRIS in Java, whereas OSIRIS is written in Microsoft C++ and is thus runnable on this platform only – a drawback resolved by the Java implementation.

<sup>3</sup>The system supports a one-to-many relationship between physical machines (hosts) and peers, meaning that a host may run multiple peers in parallel. This implies that peers are not necessarily remote in the sense that they run on different hosts. Peers running on the same host are, however, in different JVM instances, which in turn means that messages go through a local network layer.

<sup>4</sup>Asynchronous versus synchronous message sending is not to be confused with an asynchronous versus synchronous model of time, see also [Footnote 1](#) at [Page 99](#).



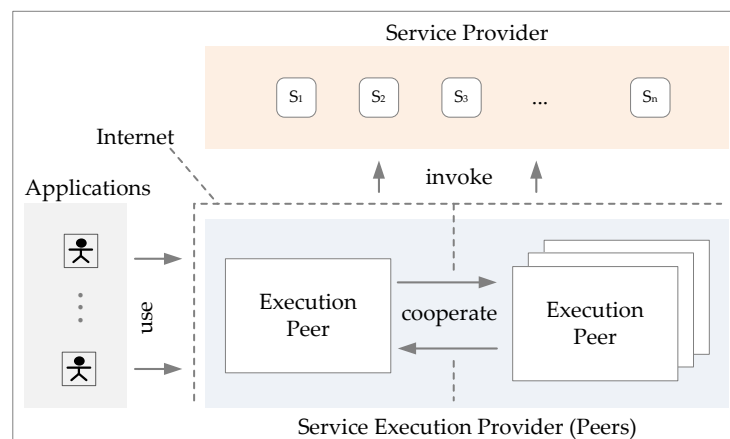


Figure 7.1: High level organization of service execution engines, application services, and client applications, and the main interactions between them.

but not concurrently.<sup>5</sup> Control is consequently not distributed at the same time for a concurrent process; hence, there is no need for a distributed synchronization protocol. Application services are either deployed outside of the OSIRIS NEXT network or on peers within the network. The former are invoked by execution engines via their standard protocols (e.g., SOAP) whereas the latter are invoked via OSIRIS NEXT-specific message sending. For this, we have specified an additional OWL-S grounding, see the subsection on grounding service operations below.

Active engines wait for execution requests sent by client applications. These requests are either self-contained, meaning that a request includes (i) the service specification (in OWL-S) and (ii) required input values. In addition, we have implemented a service specification repository (not depicted in Figure 7.1) by means of which clients can request the execution of a service that has been uploaded to a repository.

### General Peer Architecture

The internal design and the functional decomposition of a peer is depicted in Figure 7.2. In short, the design is comparable to the *staged event-driven architecture* (SEDA) [WCB01]. Incoming and outgoing message processing is divided into stages that are connected by FIFO queues. Queues can be configured to be backed by a persistent store; hence, being crash-failure safe.

<sup>5</sup> From a conceptual point of view, distributed execution is understood herein as follows. Let  $Q$  be the maximum degree of parallelism within the control flow of a service  $S_c$  (see Equation (4.21)), let  $[t_i, t_j]$  ( $i < j$ ) be the interval between execution start and end of an instance  $s_c$ , and let  $P_k$  be the set of peers at which the locus of control dwells at some moment in time  $t_i \leq t_k \leq t_j$ . Then, distributed execution means that  $|\bigcup_{i \leq k \leq j} P_k|$  is not limited to 1 and that  $\forall t_k: |P_k| \leq Q$  (i.e., the possibility that control spreads over multiple peers, not necessarily at the same time, and never over more peers at the same time than the maximum degree of parallelism). As a counterexample, a client-server system in which one server (peer) executes a service on behalf of a client does not perform distributed execution, even if operations are invoked remotely. In case of OSIRIS NEXT it holds that  $\forall t_k: |P_k| = 1$  irrespective of  $Q$ . In case of the original OSIRIS system it holds that  $\exists t_k: |P_k| = Q$ .

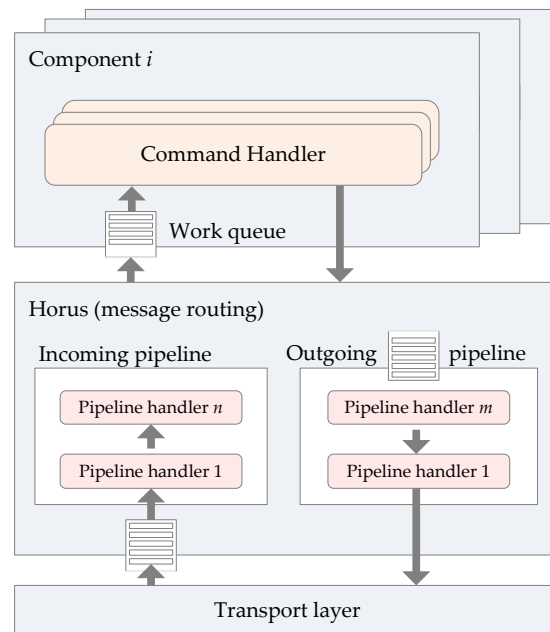


Figure 7.2: Internal design and functional decomposition of an OSIRIS NEXT peer.

Each stage is backed by a thread pool in order to allow for concurrent message processing per stage. There are basically two stages: one at front-level for message dispatching and pre- and post-processing, and a rear-level one at which application/system logic is located. They are respectively integrated in the *Horus* and *components*. Although the *Horus* is, from a software-technical point of view, also a component, it is the central message dispatcher. All incoming and outgoing messages pass through an incoming and outgoing pipeline, respectively. Both pipelines are a configurable chain of pipeline handlers, each implementing specific message pre- and post-processing. For instance, publishing (sending) a message to all subscribers of a certain topic is implemented as a dedicated outgoing pipeline handler. In addition, the *Horus* manages the lifecycle of components (i.e., instantiation, initialization, deactivation).

Components are the basic programming abstraction to implement system or application services. A peer may run any number of components in parallel, though components are singletons. Each component, in turn, may run any number of *command handler* that implement specific system or application logic. An incoming message is eventually processed by a command handler, and it may send any number of outgoing messages in return, or designate the behavior for the next message that it receives.<sup>6</sup>

A *message* is structured like a key-value associative array. Values are essentially the payload containers and may be any serializable Java object. Keys, on the other hand, are strings that, in addition, may form a tree structure (e.g., the keys “A.B” and “A.C” form a tree in which “A” is the root and “B”, “C” are leaves). Messages are furthermore typed. Together with a component’s address, this uniquely determines a command handler that is responsible for handling an incoming message.

<sup>6</sup>A command handler is thus also comparable with the notion of an *actor* in the actor model [HBS73, Hew11], which has recently gained interest in the area of massively concurrent computing.

## Global System Services

There are a mainly two global system services available to all peers: the so-called *subscriptions repository* (SubRep) and the *load repository* (LoadRep).<sup>7</sup> The LoadRep holds the recent CPU load of active peers. For this, each peer sends its current load whenever there was a change larger than a pre-configured threshold, which avoids sending a message if the load has not considerably changed.

The SubRep holds meta data and replicates it partially on every peer. Notably, this includes the current set of subscribers of a particular publish/subscribe topic and the current set of components (i.e., services) available per peer, the latter of which therefore establishes a very simple service registry. The SubRep furthermore tracks the current status of peers that were once known, which we call the *peers agency*. The algorithm to detect whether a peer is *online* or *offline* has been extended in OSIRIS NEXT. It is based on periodic “ping” and “pong” messages and outputs a list of peers suspected offline (which makes it different to *heartbeats* [KACT97]). More precisely, the interval in which a ping is sent is successively increased up to a maximum value, and provided that the status of a peer did not change as a result of the last ping. This gradually reduces the frequency of messages sent, which is motivated by the observation that peers usually either have a long uptime (e.g., a server) or a short uptime (e.g., a mobile device). In other words, the longer the time a peer is known to be online (offline), the more likely it is that it remains online (offline) for an even longer period. A change of the status from online to offline, or vice versa, goes along with a reset of the interval to the smallest value. Also, a peer that starts up (again) registers itself at the peers agency; hence, it will be promptly marked online rather than at the time when the next ping is sent.

## Grounding Service Operations in OSIRIS NEXT

As stated, we have extended OWL-S with a dedicated grounding for OSIRIS NEXT. The grounding is based upon the following correspondences with the service model:

- A component corresponds to a service.
- A command handler of a component corresponds to an operation.
- A key-value pair of an incoming or outgoing message corresponds to an input and output, respectively; the key corresponds to the identifier *id* and the value to the data value *val*.

Addressing relies upon the standard mechanism in OSIRIS NEXT. More precisely, the grounding includes an address (syntactically represented as a URI) that provides all the necessary details to address a particular command handler of a component.

Related to the grounding is the way how a concrete assignment function  $\sigma$  can be realized. This is achieved by a programmatic abstraction called `OWLTransformer`. The name indicates that it is not only used for this purpose. In fact, an `OWLTransformer` combines determining representatives with transforming service-specific

---

<sup>7</sup>The original OSIRIS system has additional types of repositories, but they are not used within OSIRIS NEXT.

Java data objects (a.k.a. POJOs) to OWL ABox assertions and vice versa, thereby providing a mechanism conceptually equivalent to *data lowering* and *lifting* considered in SAWSDL.

### Automated Semantic Service Description Generation

Another feature that we have implemented is automated generation of semantic service descriptions in OWL-S. Every deployed application component thus publishes complete service descriptions for every command handler registered, and which is accessible to any peer via standard messaging as well as via HTTP (provided that the Web server component is deployed). For this, the developer of a component annotates command handlers such that every message element (i.e., every input respectively output) gets associated with (i) an identifier *id*, (ii) its *type*, and optionally a (iii) a human readable short description and (iv) a custom `OWLTransformer` to be used. The *type* is essentially a reference to an OWL concept or data range.

#### 7.1.2 Peer-to-Peer Execution

The execution system implemented in OSIRIS NEXT features composite service execution that is distributed on demand while decentralized, which provides another means of flexibility in the service execution task.

Distributing the execution of composite services invariably requires means of organizing and coordinating how participating peers share this task. Such means are built upon three core properties:

- How to partition the composite service into *sections* that can be executed by different execution peers. Within a section, execution is performed by the same peer.
- A *strategy* to decide when and where sections are distributed to execution peers.
- A *protocol* that coordinates control among the participating peers such that correct execution is guaranteed.

The strategy that has been developed for OSIRIS NEXT is designed particularly for ad hoc services (which supposedly have been composed automatically). As a result, these kind of services are executed usually a few times only, possibly just once. This calls for a strategy that involves minimal initial overhead to get all peers prepared that might later be involved in the execution of a particular service. The strategy is furthermore designed for flexibility in dynamically changing environments. For example, environments with mobile devices on one hand and computing, storage, and other types of resources delivered as “elastic” services on the other (of which the recent Cloud Computing shift is a prominent representative). Both objectives – ad hoc services and dynamic environments – are addressed by the possibility to *migrate* ongoing executions between peers. What is more, the decision whether to migrate or not is made dynamically at runtime as opposed to a decision determined in advance (which thus cannot take dynamics at runtime into account).

Table 7.1: OWL-S control constructs classified according to whether migration can be performed immediately versus possibly delayed, and whether migration is possible at all inside them.

Immediately	Delayed	Indivisible
Any-Order	Split	Choice
For-Each	Split-Join	If-Then-Else
Repeat-While		Perform
Repeat-Until		Set
Sequence		Produce

## Sections

The question how a composite service can be partitioned into sections can be answered by analyzing the different control flow patterns that can be formed in the process model. With an abstract point of view, however, sections that encompass exactly one ordinary transition  $t$  naturally come to mind, which is at the same time the smallest sensible granularity. Using sections of this size, a migration becomes possible in principle before and after  $t$ ; leaving aside technicalities, at  $t$ 's input and output place.

Upon inspection of the control constructs available in OWL-S we have found, however, that additional aspects need to be taken into account. Control constructs in OWL-S can actually be classified into three categories: (i) collection and iteration constructs that can be partitioned into sections and for which migration is permitted without further constraints; (ii) collection constructs that can be partitioned, but where additional conditions have to be met to permit migration; (iii) constructs that cannot be sensibly partitioned as they should be considered indivisible. The categories and the constructs that fall into them are listed in [Table 7.1](#). All constructs in the left column are innocuous in the sense that a migration inside them can be done without further considerations as they do not fork concurrent execution threads. This is apparently different for the constructs in the middle column. A migration of multiple concurrent execution threads becomes possible only after all threads are in a state in which this is possible, which means that one has to pause and wait for all threads to be paused. Since a decision to migrate is made at some point in either thread, the point in time when the migration can actually be performed is likely delayed. Finally, constructs in the right column are always executed by one peer since a migration inside them is not sensible. *Choice* and *If-Then-Else* are, however, somewhat special in the sense that only the (conditional) choice is made in an indivisible way, and subsequent constructs are not affected.

Notice, however, that the model of OWL-S has it that control constructs are nested (rather than chained) to form complex services; hence, a process forms a tree in which leaves are atomic service/operation invocations. Nesting implies that an outer (higher-level) *Split* or *Split-Join* dominates nested control constructs regarding when migration becomes possible. As illustrated by [Figure 7.3](#), if a *Split-Join* encloses two (or more) control constructs – a *Sequence* and a *Perform* in this case – then a migration within the sequence is temporally governed by the other thread (i.e., it may be delayed until the point in time when it becomes possible in the other parallel thread).

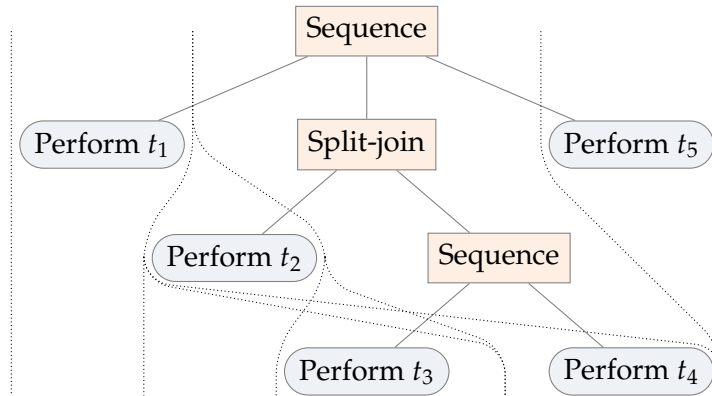


Figure 7.3: *Emergency Assistance* process (see Figure 4.5) depicted as OWL-S constructs (nesting implies hierarchy). Dotted lines point out stages at which a migration is permitted. As can be seen, there are seven stages altogether. For example, the rightmost line symbolizes a migration upon completion of  $t_1, \dots, t_4$ , but before  $t_5$ .

## Strategy

Figure 7.4 illustrates the migration strategy on the example of a composite service structured as a sequence of three atomic service invocations ( $S_1, S_2, S_3$ ). It is also assumed that the composite service is split up into sections equal to the atomic services. The strategy works as follows:

1. The client sends the OWL-S composite service description together with the input data to one available execution peer,  $EP_1$  in this case (step 1 in Figure 7.4).<sup>8</sup>
2.  $EP_1$  parses the service description and instantiates the service by initializing a new execution state for storing intermediate results and the control flow state.
3. A decision step follows to determine whether execution of the next section should be made by  $EP_1$  or whether the execution should migrate to another peer and continue there. In the example we assume that execution stays at  $EP_1$ . Consequently, it executes the first section which maps to service  $S_1$  (2).
4. The decision step is repeated prior to each section.
5. In the example we assume that subsequent to invocation of  $S_2$  by  $EP_1$  (3) execution migrates to another peer, say  $EP_2$  (4). In short, this is achieved by serializing and transferring the execution state from  $EP_1$  to  $EP_2$ .
6. Finally, after invocation of  $S_3$  (5) the last section has finished and the result is returned to the calling client by  $EP_2$  (6).

<sup>8</sup>The client uses the SubRep to get a list of available execution peers first and then selects one by itself. Alternatively, the client can send the execution request as a publish message having a corresponding “service execution” topic. The infrastructure will then select a peer on its own to which the request is forwarded.

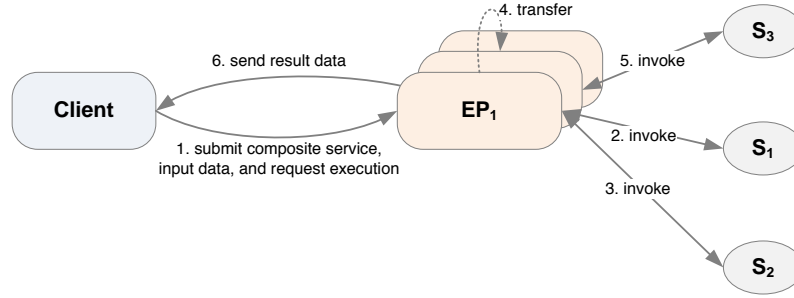


Figure 7.4: Execution strategy illustrated for three available execution peers  $EP_1$ ,  $EP_2$ ,  $EP_3$  and a composite service made up of a sequence of three atomic services  $S_1$ ,  $S_2$ ,  $S_3$ .

Migration from one peer to another comes at the expense of additional communication (compared to no migration) because it is essentially the current execution state that needs to be transferred to the new peer. A migration therefore becomes beneficial if the utility of executing the remaining part at another peer outweighs the communication overhead and the value of staying at the same peer. Having said that, a migration can even be imperative if the current peer is simply not capable of executing an entire service by itself (e.g., a mobile device whose resources such as storage, bandwidth, battery charge are not sufficient). Yet another example is offloading ongoing executions from a peer to newly added peers as a means of scaling out to match an increasing global workload; analogously possible in the opposite direction as a means of scaling in.

A simple cost model underlies the decision whether a migration is beneficial. Specifically, let  $P$  be a finite set of online peers and  $p^c, p^n \in P$ ,  $p^c \neq p^n$  the current and new execution peer, respectively; synonymously called source and target peer. We define the *benefit*, denoted with  $B$ , as the function

$$B : P \times P \rightarrow \mathbb{R}$$

$$B(p^c, p^n) = cost_x(c_{p^c}) - cost_x(c_{p^n}) - cost_m(p^c, p^n)$$

where  $cost_x$  and  $cost_m$  model execution and migration costs, respectively. Execution costs  $cost_x$  is a function of a *context information set*  $c_{p^c}$  ( $c_{p^n}$ ) representing any kind of information about the environment of the current (new) peer. The context information set might include various information about (i) the infrastructure or (ii) the application domain. The former includes, for instance, available computation and memory resources, load, and other runtime related information. Application related information includes meta data about services such as whether a service is memory, data, or computing intensive. The function  $cost_m$  estimates the costs for migrating execution from  $p^c$  to  $p^n$ . The value of  $cost_m$  mostly depends on the network bandwidth and the overhead to serialize and de-serialize the current execution state. Assuming that costs are positive (i.e.,  $cost_x > 0$  and  $cost_m > 0$ ), the result of  $B$  can be characterized as follows:

- $B < 0$ : execution is adverse at the new peer;
- $B = 0$ : execution can continue at either peer;
- $B > 0$ : there is a benefit in migrating to the new peer.

Finding the optimal target peer  $p^n$  requires calculating the benefit for all online peers:

$$p^n = \arg \max_{p^n \in P} B(p^c, p^n) .$$

Practicability therefore depends mainly on (i) the size of  $P$  and (ii) whether the information used by  $cost_x$  and  $cost_m$  is locally available or has to be gathered remotely first. If  $P$  may become large then an incomplete approach that not necessarily finds the optimum may be the only practical approach. Also, reducing the amount of information exchanged for the calculation to a minimum is clearly valuable. Taking into account the possibility of replicating information, there are many arrangements conceivable rather than a general solution. Our implementation therefore does not come with a fixed implementation but allows to plug-in different implementations.

## Protocol

The migration-based strategy requires a protocol that guarantees *atomic commitment* between two peers, meaning that migration happens either completely or not at all, but avoids duplication (i.e., source and target peer both continue execution the service instance) and loss (i.e., execution abruptly ends as no peer continues). Technically, this can be achieved based on reliable messaging asserting exactly once message delivery.

There are two more factors that influence the design of the migration protocol. First, is calculation of the benefit integrated into the protocol and if so does it involve information exchange (because calculation is distributed or involves remote information gathering)? Second, do candidate target peers have the option to refuse a migration request (versus being obliged to accept a request in any case)? An answer of yes to these questions makes a two-phase migration protocol favourable. Both, information exchange and obtaining approval, can be achieved then in an opening phase. Otherwise, if information exchange does not take place and target peers neither have the option to refuse a request then a single-step request/reply message exchange is sufficient and preferable.

In our implementation, we have decided that peers do have the option of refusing a migration request. What is more, since one can plug-in an implementation of the benefit function, calculation may or may not involve information exchange depending on how it is designed. The protocol is therefore divided into two phases. Yet another factor to this is that a migration involves essentially transferring the service specification and the execution state. While the former is usually moderate in size, the latter may be large depending on the size of intermediate results and the part of the local KB that needs to be transferred.<sup>9</sup> A two phase-protocol also avoids transferring the possibly large amount of data if a target peer refused becoming the new execution peer. The two phases are as follows.

- **Opening Phase:** The source peer preselects a set of candidate peers. A peer is preselected if it is online, runs an execution engine, and its current load is below a certain threshold.<sup>10</sup> The source peer then sends a migration inquiry to all peers in

<sup>9</sup>This is essentially the ABox and references to ontologies that have been loaded into the TBox.

<sup>10</sup>The first two pieces of information are available from the SubRep, the latter from the LoadRep.



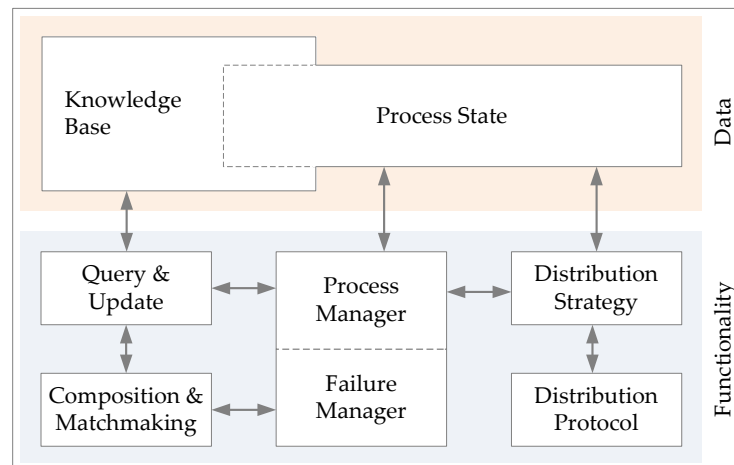


Figure 7.5: Internal structure and main components of an execution peer.

the set. Target peers reply by sending either an accept or refuse message. A negative refuse ends the protocol between the two peers. This happens analogously if the source peer does not receive a reply after a timeout (i.e., sending a negative reply is optional, but not replying may increase the delay after which the source peer considers an inquiry as failed for a particular peer). A positive accept communicates a commitment limited in time, meaning that the target peer will take over execution of the service, provided that the source peer reacts before the timeout set by the target. The positive accept may additionally contain target-specific context information that the source peer may use to calculate that value of  $B$  for the replying peer.

- **Final Phase:** Upon selecting a particular target peer, the source peer sends a final migration request message to the selected peer that contains intermediate results and the part of the local KB that needs to be transferred. Provided that this message is received by the target peer before the timeout it resumes execution and replies with an acknowledge. Otherwise, it can still reply with a negative timeout passed message informing the source peer about the too large delay. It is therefore important that the timeout is set sufficiently ahead of time, which might involve taking latency of the network layer into account in case it cannot be neglected.

### 7.1.3 Control Flow Intervention

Implementing CFI in the execution engine requires at least built-in procedures to (i) dynamically modify the control flow at execution time, to (ii) find respectively compose replacements, and (iii) to redirect the data flow. The additionally required procedures to pause and resume execution are mostly reusable from the implementation of execution migration as they are essentially the same for both. As a natural consequence, the OSIRIS NEXT execution engine is functionally divided into several modules as depicted in Figure 7.5.

The central part is the *Process Manager*. It mainly acts as an interpreter for OWL-S and invokes operations, by delegating request and reply processing to implementations of the different supported grounding types. The process manager further provides a monitoring interface. A monitor register itself with the process manager and specifies an event filter that determines the types of events it will get notified about.

Closely integrated with the process manager is the *Failure Manager*. Failure detectors are, however, part of the process manager and the failure manager is triggered in the presence of a failure for which it provides recovery support. Specifically, failures detected by the process manager are handed over to the failure manager and it is responsible for managing the procedure of finding a replacement, modifying the control flow, and redirecting the data flow. In order to do that it interacts with the *Composition & Matchmaking* module. This module, in turn, relies on a service repository to retrieve available service and operation profiles, which is implemented as another component not depicted in [Figure 7.5](#). Internally, the service repository is currently realized as an RDF triple store that can be queried using SPARQL. What is more, the failure manager is not designed to be restricted to CFI only. One could, for instance, integrate conventional rollback and/or compensation strategies based on transactional support.

All the functionality required to distribute the execution among peers is implemented by the *Distribution Strategy* and *Distribution Protocol*. Finally, the *Query & Update* component is tightly integrated with the KB. One function is to provide a SPARQL query interface to the KB integrated with an inference engine for deductive DL based reasoning.<sup>11</sup> Its other function is to realize a concrete precondition and effect system.

Each composite service execution instance has associated its local process state (representing the control state) and a partly shared KB. The KB is initially populated with required domain ontologies, the OWL-S service specification to execute, and pre-existing assertions about individuals, which represent the world state. The part that is shared among different execution instances is the TBox. This does not pose a problem regarding correct concurrent access since the TBox is a protected part (see [Assumption 5](#)) and is thus read only at execution time. Each service instance currently has its own ABox. The extension to a fully shared KB by integrating our KB store implementation that enforces correct concurrent access (see [Section 7.3](#)) has been left as future work.

Each execution peer has, in addition, a background KB that acts as a cache for domain ontologies that have already been used. Ontologies accessed for the first time will be fetched from the Web and loaded into it. If a domain ontology that already exists in the background KB needs to be loaded into a local execution KB of a composite service instance, only a reference needs to be set (i.e., no physical copying is required).

## 7.2 KB Access Optimization Techniques

In contrast to traditional workflow engines that mainly act as interpreters of the process specification (expressed in a language such as BPEL [JE07]), service execution using semantic services is more involved. Illustrated in part by [Figure 6.1](#), it is characterized

<sup>11</sup>We have used the OWL-DL reasoner Pellet [SPG<sup>+</sup>07].

by recurring sub-tasks that all produce queries to the KB. The main types of read and update queries are:

- Read the control and data flow constructs from the process specification so that the engine can interpret and execute them (according to their operational semantics).
- Read inputs for operation invocations from the KB. Write outputs produced by operation invocations (intermediate results) back to the KB so that they are available for subsequent use.
- Read operation grounding details from the KB in order to prepare service invocation messages and process replies.
- Read preconditions of services from the KB and check if they are satisfied w.r.t. to the current state of the KB.
- Materialize effects as a result of operation invocations in the KB to correctly represent the current world state.

As a consequence, the KB is interrogated almost permanently and updated frequently. The overall performance is hence influenced to a large extent by the runtime efficiency of query answering and reasoning services in the KBMS.

We apply two optimization techniques to the service execution task: *prepared queries* and a caching strategy called *frame caching*. Both are, however, of general utility beyond this application.

Prepared queries actually transfer a concept well known in database programming to knowledge base querying. One advantage is that costs for repeated query translation are reduced. The main value, however, is a reduction of the number of KB updates, which in turn improves the performance of queries that involve inferencing. The reason is that reasoning engines need to re-classify and re-realize the KB after every update, though some of them (e.g., Pellet [SPG<sup>+</sup>07]) implement incremental approaches to reduce the effort.

Frame caching, on the other hand, aims at reducing the total number of KB accesses made for composite services that contain iterative control constructs by keeping materialized views of frequently accessed individuals and data values in data structures that are similar to *frames* [Min81]. Frame caching is in fact a technique that can be used for reducing the conceptually implied overhead if an RDF graph-based data model is used at the lower level and OWL is used at the higher level.

Both techniques have been implemented in the OWL-S API<sup>12</sup>, which is a high level programming library written in Java that has been initially developed by the Mindswap research group at University of Maryland [SP04]. Although the OWL-S API has been designed to support RDF as well as OWL programming frameworks underneath, it is currently implemented on top of Jena, a prominent Semantic Web programming framework [Jena]. The ground-level programming abstraction is therefore organized as an RDF graph, which has the consequence that an OWL KB is accessible at the lowest level as a set of RDF triples (no matter how the backing data store is actually organized).

<sup>12</sup>Available Open Source at <http://on.cs.unibas.ch/owl-s-api>.

## 7.2.1 Prepared Queries

In our implementation, a prepared query is a re-usable and pre-compiled SPARQL statement that allows late binding of variables at query execution time. Prepared queries are therefore comparable with *prepared statements*, a well-known abstraction provided by programmatic access interfaces to relational DBMS (e.g., JDBC) that aim at similar improvements of efficiency.

Prepared queries are used in our implementation of the OWL-S API to increase efficiency of checking (i) preconditions and (ii) conditions of conditional process control constructs that are expressed using SWRL [HPB<sup>+</sup>04], which, despite being a quasi-standard, has achieved widespread adoption.<sup>13</sup> Such a SWRL (pre-)condition, subsequently referred to as a condition for shortness, is a conjunction of SWRL atoms analogous to a conjunctive ABox query. A SWRL atom can be either of the form shown in Table 7.2 or from a subset of built-ins. SWRL built-ins are binary comparison relations such as *lessThan*, *greaterThanOrEqual* or basic mathematical functions such as *add*, *multiply* over XML datatypes. This results in a precondition system whose expressivity is between (PS1) and (PS2).<sup>14</sup> Analogous to (PS1) and (PS2), a variable either refers to a representative of a profile parameter or is a existentially quantified solution set variable (i.e., distinguished), supposed to be referred to by an effect. Finally, a condition  $q$  is satisfied (evaluates to true) iff every atom  $\alpha \in q$  is entailed by the KB (which is equivalent to Equation (4.8)) and where entailment of single atoms is defined as shown in Table 7.2; built-ins are not listed here for reasons of space, but their semantics is mostly apparent, see [HPB<sup>+</sup>04, Section 8]. The empty condition is trivially satisfied.

The approach used to check whether a SWRL condition is satisfied is to translate it to a SPARQL query, executed thereupon on the current state of the KB.<sup>15</sup> This is possible because the formal semantics of a conjunction of SWRL atoms can be preserved if they are translated to SPARQL *basic graph patterns* (BGP), and assuming that the resulting SPARQL query is answered under the OWL 2 Direct Semantics Entailment Regime [GO10, Section 6]. Table 7.2 shows how SWRL atoms translate to BGPs. For example, the following condition in abstract syntax

$$\text{Class}(x, \text{InsuredPerson}) \wedge \text{Class}(x, \text{PhysicalObject})$$

where  $x$  shall refer to a representative of an input, say, an individual named *:BOB*, and *:InsuredPerson*, *:PhysicalObject* shall be concepts of some domain ontology, translates to

```
SELECT *
WHERE { :BOB rdf:type :InsuredPerson ;
        rdf:type :PhysicalObject . }
```

<sup>13</sup>Note, however, that OWL-S does not mandate the use nor support of particular formalisms.

<sup>14</sup>This precondition system is more expressive than (PS1) due to built-ins and the two additional types of atoms to check individual (in)equality, and less expressive than (PS2) mainly because one cannot use variables in the place of concepts or roles.

<sup>15</sup>A condition is thus *uniquely* satisfied if evaluation of the translated SPARQL query yields exactly one result, it is not uniquely satisfied if there is more than one result, and it is not satisfied for no result.

Table 7.2: SWRL atoms, their semantics, and mapping to SPARQL BGP.

Abstract Syntax	Semantics SPARQL BGP Triple Form
$Class(x, C)$	$x^{\mathcal{I}} \in C^{\mathcal{I}}$ $\langle x, \text{rdf:type}, C \rangle$
$IndividualProperty(x, R, y)$	$(x^{\mathcal{I}}, y^{\mathcal{I}}) \in R^{\mathcal{I}}$ $\langle x, R, y \rangle$
$DataProperty(x, T, v)$	$(x^{\mathcal{I}}, v^{\mathcal{D}}) \in T^{\mathcal{I}}$ $\langle x, T, v \rangle$
$SameIndividual(x, y)$	$x^{\mathcal{I}} = y^{\mathcal{I}}$ $\langle x, \text{owl:sameAs}, y \rangle$
$DifferentIndividuals(x, y)$	$x^{\mathcal{I}} \neq y^{\mathcal{I}}$ $\langle x, \text{owl:differentFrom}, y \rangle$

Using standard interpretation  $\mathcal{I} = (\Delta^{\mathcal{I}}, \Delta^{\mathcal{D}}, \cdot^{\mathcal{I}})$ ;  $C$  a concept;  $R$  an abstract role;  $x, y$  an individual or individual variable;  $T$  a concrete role;  $v$  a data value (lexical form) or data variable.

In this case, the condition is satisfied if the query has an empty result (which is not to be confused with no result) as there is no variable to project to (i.e., there is no distinguished variable).

The simple and rather naive approach to condition checking starts with replacing each atom that contains variables that refer to inputs with a new atom in which these variables have been substituted by the corresponding value. Since conditions are in the majority of cases expressed using input variables, this implies additional work in all of these cases; nota bene an insertion of new axioms in the KB (triples in case the KB is represented in RDF), which is required for the subsequent translation to a query. The insertion is also a result of the fact that SWRL conditions are part of the service description; hence, they are also stored in the KB. While these insertions are fairly cheap from a data management point of view, they have severe consequences if an reasoning engine is attached to the KB, which is almost always the case since reasoning is a primal feature. Unfortunately, today's reasoning engines do not yet perform well under (frequent) KB updates since they need to exhaustively re-perform consistency checks, classifications, and realizations. The consequence is that such updates to the KB provoke (more or less) high delays for subsequent queries; thus, reducing the overall performance.

Yet there is another weakness when it comes to repeated checking of the same condition; for instance, if a conditional control construct in a composite service is executed multiple times (e.g., as part of a loop). As shown in the evaluation chapter, a considerable amount of time is spent just for the creation of the SPARQL query from a SWRL condition (creation of ground atoms and translation to SPARQL). This can result in cases in which creation time exceeds query execution time by a factor greater than two, which calls for an optimization.

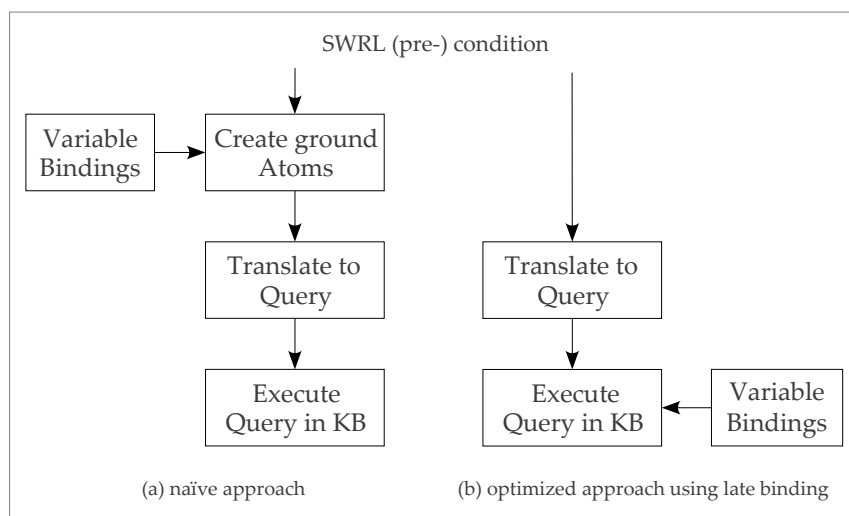


Figure 7.6: Comparison of (pre-) condition evaluation procedure for conventional and optimized approach using prepared queries.

Optimizing SWRL condition checking thus has to address these problems by factoring out the overhead induced by the query creation process. The proposed *prepared queries* simplify and optimize the process of condition checking by avoiding the creation of new ground atoms in the KB. This is achieved by three changes. First, provide the possibility of translating the original SWRL condition directly to a SPARQL query. Second, allow for late binding of variables occurring in the query at query execution time. Third, having late binding of variables enables reusing queries. A condition must therefore be translated only once and can be executed as often as needed thereby also supporting conditions that need to be checked multiple times.

This results in a simplified procedure for SWRL (pre-)condition evaluation, which is depicted in Figure 7.6. Yet its most important advantage is that insertion of additional ground atoms in the KB is eliminated; thus, preventing that reasoning engines need to (exhaustively) re-check consistency, re-classify, and re-realize the KB.

## 7.2.2 Frame Caching

Frame caching is used in the context of the OWL-S API to gain rapid access to the process specification (i.e., the control and data flow) and to operation grounding information stored in the KB as part of the overall service description/specification. The basic idea is to exploit locality in repeated KB read accesses whose results match and can be cached in frame-like data structures. Caching these information improves the performance for repeated execution of the same service as well as the execution of loop control constructs such as *Repeat-Until* and *For-Each*<sup>16</sup> that iterate over their body. Otherwise,

<sup>16</sup>It should be noted that OWL-S specifies two control constructs – *Any-Order* and *Repeat-While* – that are not covered by the process model presented in Section 4.3. There are furthermore two control constructs – *Produce* and *Set* – that are actually constructs for controlling the flow of data rather than the flow of control. Since the OWL-S API aims at being a complete implementation of the OWL-S framework, they are included here.

an execution engine has to fetch information repeatedly from the KB (by submitting the same queries over and over again). The same performance gain is achieved for groundings if services are repeatedly invoked.

However, finding an appropriate caching solution is not straightforward as it has to take into account the following aspects:

- *Location* where cached data is stored: Either close to the KB store (probably inside the KB store) versus close to the application, which acts as client to the KB and which uses the query results.
- *Granularity* of cached data: Limiting the amount of cached data to exactly the query result versus more advanced look-ahead strategies where data that is likely to be read in the future is pre-fetched and cached in advance.
- *Representation* of cached data: Graph-based, essentially in the same representation as stored in the KB versus a differently structured representation that fits more closely with the access patterns of the client application.
- *Cache coherence*: Invalidation of cached data in the presence of updates to the original data in the KB due to concurrent access by multiple clients.
- *Implicit information* inferred by reasoning engines on the fly at query execution time.

The *frame caching* approach we have developed addresses all the aspects summarized above. This is achieved by combining: (i) materialized views of proximate triples that form sub graphs of a KB, using (ii) frame-based data structures, which, at the same time, (iii) realize a simple form of a look-ahead cache, (iv) local to the place where they are used, and (v) possibly contain inferred information.

The notion of a *frame* was introduced in frame-based systems [Min81] as an alternative to logic-oriented knowledge representation systems. More formally, a frame  $F$  contains a finite set of *slots*, similar to entries in a record. A *slot filler* is the value of a slot and can be a data value or again a frame, thereby allowing nested frames. Now, the basic idea of frame caching is to use a frame to provide a record-like view of triples  $\langle s, *^p, *^o \rangle$  in the KB. The subject  $s$  corresponds to the frame  $F$ , a property  $*^p$  corresponds to a slot of  $F$ , and an object  $*^o$  is the filler of the corresponding slot  $*^p$ . If a filler is again a frame, one can represent tree-like sub graphs by nested frames. A frame will always be created (successively) from the results returned by KB read queries. Using object-oriented languages, frames can be represented one-to-one as objects. In doing so, one gets rapid access to the fillers of a frame. Consequently, one gets rapid access to (all) objects of some subject once a corresponding frame was filled.

Frames may include inferred information which thus does not need to be recomputed on every access. Creation (filling) of frames is very cheap as it essentially amounts to allocating an object instance in memory and assigning references. Frames can be implemented with moderate additional memory requirements provided that slots can be implemented as direct references. This is possible anyway if the filler is again a frame. Otherwise, a close integration with the KB store is required such that slots are references to the values in the KB store.

Finally, in our execution engine implementation, we apply frame caching for grounding information and the process specification only. The reason is that only these are the parts that may be repeatedly accessed. Specifically, each operation grounding is cached by one frame and the entire process specification (which essentially includes the control and data flow) is cached by a single nested frame.

## Cache Coherence

As with any caching strategy that is supposed to provide cache coherence, there is one reason that causes (partly) invalidating a cache: if data in the backing data store has (partly) changed as the result of a write performed by another application in the background. If the backing data store is however not concurrently accessed (i.e., if it is not shared) then only an update of the accessing client needs to be handled correctly, which can be done in the usual way either by a write-through or write-back strategy.

Since we apply frame caching in our execution engine for the grounding information and the process specification only, we can use the following approach that does not require a cache coherence protocol. Every service execution instance has its private cache being a set of possibly nested frames. Among these, the process frame does not need to be invalidated as long as the process is not subject to dynamic modifications at execution time since the process specification does otherwise not change. However, a dynamic change as part of a successful CFI cycle has the consequence that obsolete parts of the frame are invalidated. This is sufficient because the change is service instance local. The first access to parts of the replacement not yet in the cache induces fetching the information from the KB into new sub frames of the process frame.

## Integration with Snapshot Isolation Data Store

We finally discuss, how the caching technique integrates with a data store, such as the one described in [Section 7.3](#), that offers a transactional interface together with Snapshot Isolation. To answer this, we need to clarify first whether the cache is transaction-local, meaning that every transaction has its private cache, or whether it has a broader scope. In the former case, the situation is simple. Cached data never has to be invalidated within a transaction since it operates on its own snapshot. An update to data in the cache by the transaction can be handled by a write-back strategy integrated into transaction processing: an update writes to the cache first and is written back if the transaction is permitted to commit. The downside is that it is not decidable right away whether the cache can be retained beyond the transaction end for a subsequent transaction. This requires information whether the cached data was updated in the backing store meanwhile by another transaction, which calls for a synchronization strategy that could be based on an active notification mechanism.

The situation is different if the cache scope spans multiple concurrent transactions. Since every transaction needs to see its own snapshot, one would need to ensure that also the cache correctly reflects this, which means that one would need to implement a snapshot management also for the cache. This makes its implementation considerably more complex.



## 7.3 Snapshot Isolation OWL Data Store

We have implemented a prototype of our SI-based concurrency control approach as a main memory (hence, transient) OWL 2 store. It comes as an alternative data binding for the OWL API [HB09]. We first address how the OWL store interfaces with the OWL API, which requires providing some basic background information. Afterwards the implementation of the data store itself is detailed.

### 7.3.1 Interfacing with the OWL API

In short, the OWL API is an Open Source library written in Java that “includes first class change support, general purpose reasoner interfaces, validators for the various OWL 2 profiles, and support for parsing and serialising ontologies in a variety of syntaxes” [HB09]. It is considered a reference implementation for OWL 2, designed in close correspondence with the OWL standard, used by prominent applications (e.g., Protégé-OWL Editor [HT06]), and is supported by major reasoning engines such as FaCT++ [TH06], HermiT [MSH09], Pellet [SPG<sup>+</sup>07], Racer Pro [HM01b], or TrOWL [TPR10].

The flexible design of the OWL API includes a service provider interface, named *Internals*, by means of which it can be extended with third-party storage mechanisms. This provides a clean programming abstraction that our data store implements. Since the OWL API, however, lacks a transaction programming abstraction, we had to extend it with programmatic means for transaction end demarcation (commit, rollback). Transaction begin demarcation, on the other hand, is implicit with the first change or read (after a rollback or commit).

At its core, the Java object model of the OWL API is a structured set of Java interfaces whose names are aligned with the names of syntactic constructs in the OWL 2 structural specification [MPSP09] (i.e., the different types of axioms, assertions, and annotations). The resulting one-to-one abstraction of OWL syntactic instances as Java objects, rather than an RDF-centric abstraction, is ideal from the perspective of our concurrency control model. First, these Java objects become the unit of concurrency control. Second, these Java objects are immutable by design; that is, they can be created using factories only and do not provide object state mutating methods. In addition, the OWL API has built-in support for determining structural equivalence of syntactic instances and implements transformation to NNF, which we use for O- and E-conflict checking. The latter means that a transaction’s change set contains OWL concept expressions (if any) that have been brought to NNF.

The fact that Java objects representing OWL syntactic instances are immutable implies that an `OWLontology`<sup>17</sup> can be modified only through applying add and remove changes, which correspond to the basic add and delete operations. More precisely, changes are applied via an `OWLontologyManager` associated with each `OWLontology`, which also manages the lifecycle of a set of `OWLontology` objects.

---

<sup>17</sup>`OWLontology` is the basic Java abstraction of an OWL 2 ontology in the API, which is comparable with the notion of an OWL knowledge base, see [Definition 3.16](#).

### 7.3.2 Data Structures and Snapshot Management

The main (global) data structures of the store are concurrent sets and multi-maps. The latter are used as index structures for fast lookup tables to answer simple queries. For example, one can get the set of named individuals ( $V_I$ ) directly from one of the maps (key set) and all assertions about some individual (value set per key). Both sets and maps are implemented using hash functions for fast lookup. The hash function maps two syntactic instances to the same hash value if they are structurally equivalent. Collisions are resolved in the obvious way by falling back to more costly checking for structural equivalence.

The main set and map data structures are thread-safe, meaning that all public methods that they provide can be invoked safely by multiple threads in parallel. In addition, they provide snapshot and transaction management. Snapshot management is implemented in a transparent and fully hidden way, meaning that the interface for programmatic access does not contain any methods by means of which one becomes aware nor has control over snapshot management internals. Snapshot management is internally based on a special set data structure, which we call `StatusSet`. In short, it is a standard set in which each element – a Java object representing a syntactic instance – is additionally associated with either of two types of timestamps. A committing transaction associates added objects with a timestamp of type *current* and deleted objects with a timestamp of type *obsolete*. This is sufficient for determining whether an object is visible to (can be read by) a transaction or not. Recap, the SI protocol has it that an object is visible to a transaction  $T$  if it existed at  $t_s^T$ . Obsolete objects can be discarded if there are no more active transactions in the system that started earlier than the obsolete timestamp. Cleanup of discardable objects is done automatically by a garbage-collection like background thread. In order to make cleanup an operation of  $\mathcal{O}(n)$  complexity where  $n$  is the number of discardable objects rather than the overall number of objects in a `StatusSet`, we use an inverse lookup table: a timestamp of type *obsolete* maps to all objects that are associated with this timestamp to allow for fast collection.

### 7.3.3 Transactions and Conflict Checking

Transactions are represented implicitly by a start timestamp. Timestamps are essentially 64-bit integers and are assigned in strictly monotonic order.<sup>18</sup> Transactions are thread-confined; that is, each thread can have at most one active transaction at a time. Updates made by a transaction are kept entirely thread-local until a transaction commits. This has two advantages: (i) additional synchronization means are not needed when transactions make updates since they are not accessed concurrently, (ii) they can be used at the same time for representation of the change set. Only if a transaction does not conflict, thread-local changes are applied irrevocably to the global data structures. Otherwise, all thread-local changes can be discarded at almost no cost.

Conflict analysis takes place as described in Section 6.2.7. More precisely, the normalized change set is updated instantly with every add and delete, but OES-conflict

<sup>18</sup>A rough calculation shows that even for a quite high and constant transaction rate of 50000 Tx/sec a system can run for about 5.8 M years until an overflow occurs.

checking is performed finally at transaction end, as opposed to incremental checks performed instantly on each new operation submitted (which is done by the *first updater wins* strategy [FOO04]). One reason is that the incremental strategy requires a larger number of checks if there are no conflicts, which is made clear by [Example 7.1](#).

**Example 7.1**

Imagine two transactions  $T_m, T_n$  that consist of  $m, n > 1$  add/delete operations, respectively, such that the corresponding change sets have  $|\delta(T_m)| = m, |\delta(T_n)| = n$  elements. Assume change sets are implemented as hash sets, with containment checking complexity usually  $\mathcal{O}(1)$ . As long as change sets are disjoint, incremental while instant disjointness checking amounts to  $m + n$  containment checks in total: a check is performed whenever an update operation is submitted. It is not difficult to see that if disjointness checking is done once at the end then only  $\min\{m, n\}$  checks are needed.

From [Example 7.1](#) we conclude that the incremental strategy is favourable only for high conflict rates (because there is a higher chance that less checks are actually done in case a conflict is detected early), whereas the once-only strategy is favourable for low conflict rates and for large transactions.

A similar effect is achieved for S-conflict checking. Bearing in mind the possibly high computational cost of satisfiability reasoning, and unless the actual reasoner used provides an incremental satisfiability feature (e.g., [HPS06, GHKS10]), it is cheaper to check satisfiability once only at the end instead of repeatedly after each update operation. S-conflict checking is implemented currently in a primitive way: (i) apply the changes tentatively first, (ii) check whether the KB is still satisfiable, and (iii) revoke the changes in case the result was negative.

Finally, we have implemented a strategy that allows non-conflicting transactions to commit concurrently while ensuring commit atomicity (recap, updates of a committing transaction must become visible globally all at once). The former is desired for performance reasons while the latter is necessary also because a non-trivial commit requires making changes to multiple of the global data structures thereby inducing a set of sub operations that must be observed from the outside in a non-divisible and atomic way. The basic idea is to synchronize commit with creation of start timestamps: a new start timestamp can be created at any time unless a commit is going on. Hence, a new transaction cannot start unless a commit has completed, which avoids that a new transaction reads partial results. Finally, since all global data structures are thread-safe by themselves, concurrently executing commits is not a problem and relies on the correctness of the data structures.



# 8

## Experimental Results

**D**IFFERENT EXPERIMENTS have been conducted to evaluate the practical utility of the methods introduced throughout previous chapters. The results of these experiments are presented in this chapter. For the most part, the objective in the experiments was to demonstrate feasibility in practice on problems of a realistic size and to verify that the qualitative gain of the methods comes at an acceptable expense.

In order to consider the different factors separately, the experiments are divided into four groups that correspond to main methods presented: CFI, the execution engine implemented in OSIRIS NEXT, the KB access optimization techniques, and the Snapshot Isolation OWL data store. The subsequent presentation is ordered accordingly.

### 8.1 Control Flow Intervention

Evaluation of CFI is targeted to investigate the time it takes (i) to search for a replacement and (ii) to perform the substitution of a failed subflow by its replacement. This has been combined with different experiments that further aim at investigating the impact of the following parameters: (i) the size of profiles, (ii) the amount of advertised profiles that exist, and (iii) the fraction of profiles that are semantically equivalent.

The evaluation is focused on 1:1 replacements using the matchmaking-based technique (i.e., structure-aware replacements). An evaluation of the planning-based approach for structure-nescient replacements has been left as future work. Our main goal in this regard was it to formulate how planning methods can be applied and seamlessly integrated into the overall approach. Different aspects regarding its practicability have been discussed however in [Section 5.7](#). For the matchmaking-based technique, we can show that, with the commodity hardware used, search time remains sufficiently small to realize the approach by an interactive human-computer interface<sup>1</sup> up to approximately 3000 advertised services. Furthermore, the cost of substitution is almost negligible compared to search and grows linearly at a flat increase.

---

<sup>1</sup>The common rule of thumb is that humans tolerate approximately 3 to 10 seconds of delay in the system response to keep their attention focused on the dialogue [Nie94].

### 8.1.1 Experimental Setup

Rather than conducting the experiments in a production environment, we have created a dedicated testbed based on the *emergency assistance* composite service. Required services were implemented as native services deployed on OSIRIS NEXT peers (i.e., invocation is done via the OSIRIS NEXT messaging system). Services are decidedly mockups that do nothing more than creating synthetic outputs and thus have almost zero reply time. This allows to disregard their execution times and to focus exclusively on the CFI cycle. Service failures are simulated by message timeouts (i.e., the execution engine fails to invoke a service).

All experiments were conducted on the hardware mentioned in [Section 8.3.1](#). Descriptions of advertised services have been uploaded to our simple service repository implementation, which is essentially a main memory RDF triple store based on Jena [Jena] that provides a SPARQL query interface for retrieval and has the Pellet reasoner [SPG<sup>+</sup>07] attached. The repository and the execution engine were deployed at the same peer.

A randomized generator has been implemented that is used throughout the experiments to generate synthetic OWL-S service descriptions. The generation process is parametrized in three ways. First, one can specify a set of domain ontologies, the concepts and data ranges (if any) of which are selected randomly (with a uniform distribution) as the type of inputs and outputs in generated profiles. Second, the number of inputs/outputs generated per profile can be controlled in two ways: uniform number of inputs/outputs versus a random number limited by an upper bound. Third, one can control whether generated descriptions are placed each in its own ontology or all into one. Finally, generated service descriptions are complete in the sense that they describe an atomic service grounded to a WSDL Web service; its implementation being nonexistent, though.

### 8.1.2 Results

#### Minimal Setting

In the first experiment the *emergency assistance* composite service has been executed 40 times with a simulated service failure in a “minimal setting”. This means that the KB is reduced to that knowledge required minimally, which amounts to 11 domain ontologies representing concepts used by the services and 10+4 OWL-S ontologies in this case. There was also just one semantically equivalent service stored in the service repository. The average execution time without a failure was measured with 114 ms. When simulating a failure, the execution time was 250 ms on average.<sup>2</sup> The increase induced by CFI breaks down to 133 ms for querying the service repository (“search”) and 4 ms for modifying the control and data flow (“substitution”). The significant difference indicates that the runtime performance of CFI is dominated largely by the query and reasoning performance (search), which is supported also by subsequent results.

---

<sup>2</sup>The value is adjusted for an additional delay caused by a constant timeout for failure detection.

Table 8.1: Search and substitution times for service profiles of different size (varying number of inputs, outputs)

Inputs+Outputs in Profile (No. of concepts, data ranges)									
Search time [ms]									
Substitution time [ms]									
(1,1)	(2,0)	(1,2)	(2,1)	(3,1)	(4,2)	(5,3)	(6,4)	(6,6)	
111	105	107	128	133	188	370	837	1171	
1.6	1.5	2.2	2.5	4.0	5.2	6.1	7.9	9.1	

### Increasing Profile Size

In a second experiment we have analyzed to what extent search performance depends on the size of a service profile for which an equivalent service is to be found. [Table 8.1](#) compares results for a selection of different service profiles, varying in their number of inputs and outputs and whether they are typed to a concept or a data range. The results show that (i) search time is more bound to the number of concepts than data ranges and that (ii) substitution time is comparably small with an almost negligible increase.

### Increasing Number of Available Services

Finally, we measured search and substitution times as a function of the number of available services in the services repository. In the first experiment, synthetically generated service specifications have been uploaded to the repository that are ensured to be non-equivalent to any sub service of the initial *emergency assistance* service. Each of the generated service profiles has two inputs and two outputs, each typed to a randomly selected concept of the LUBM ontology [GPH05]. Results are shown in [Figure 8.1a](#). The second experiment considers synthetically generated service specifications that are all ensured to be equivalent to a failed service, see [Figure 8.1b](#). The results show a linear increase of search for non-equivalent services. On the contrary, search in case of equivalent services takes slightly more time, but remains almost constant up to approximately 200 equivalent services. The subsequent increase appear to be caused by the reasoning involved in query answering.

In order to determine whether the increase of search is dominated solely by SPARQL query processing or by reasoning, both experiments have been repeated in a slightly modified setting: domain ontologies were extended by explicit concept equivalence assertions that would have been deduced by the reasoner otherwise (when evaluating a query); thus, making it possible to deactivate the reasoner. The results are depicted by the Search No Reasoning curves in [Figure 8.1](#). Compared to the search with activated reasoner, this gives a performance gain by a factor of  $\approx 3$  for non-equivalent services, which shows that the increase of search is due to SPARQL query processing in this case. However, in case of equivalent services, this results in a constant search time, which can be explained by the fact that the query is limited to select the first matching service found.

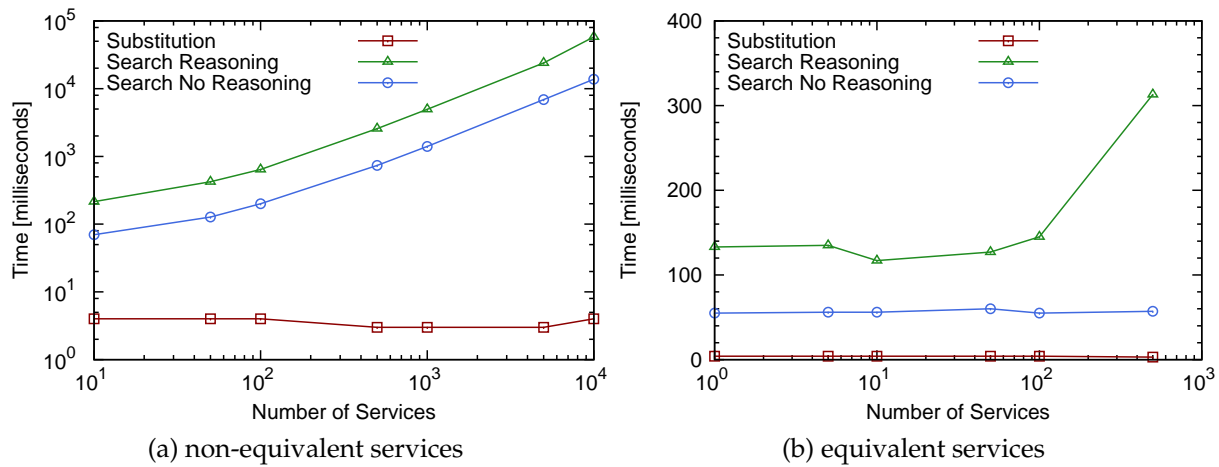


Figure 8.1: Search and substitution times for increasing number of available services.

## 8.2 Execution Engine

Evaluation of the execution engine that runs on peers was done with the objective of examining the characteristics of the internal queued and multi-threaded architecture as a function of increasing local load, generated by an increasing number of concurrent execution requests. We show that the internal architecture scales well until a resource-determined saturation point is reached.

### 8.2.1 Experimental Setup

Again, all experiments were conducted on the hardware mentioned in [Section 8.3.1](#). Live Web services deployed in the Internet have been applied to this evaluation in order to include practical influences. For this, an exemplary composite service called *DictionaryTranslator* has been used, built upon a sequence of three real SOAP-based Web services: First, an input term is translated to English from another language. Afterwards, the translated term is looked up via an online English dictionary service returning a short definition of the term. Finally, the definition is translated back to the original language using the translator service again.

A client peer that runs 1 up to 40 threads (step size 5) issued the requests containing the service specification together with varying input terms to the execution peer. Each group was repeated ten times. Migration has been suppressed as this is apparently not relevant for this particular evaluation.

### 8.2.2 Results

[Figure 8.2](#) shows the tendency and dispersion in total execution time under increasing local load. The median value shows that the engine nearly scales ideally until a saturation point is reached, which is around 35 concurrent executions. The increase afterwards is not necessarily caused by exhausted CPU resources on the execution peer,



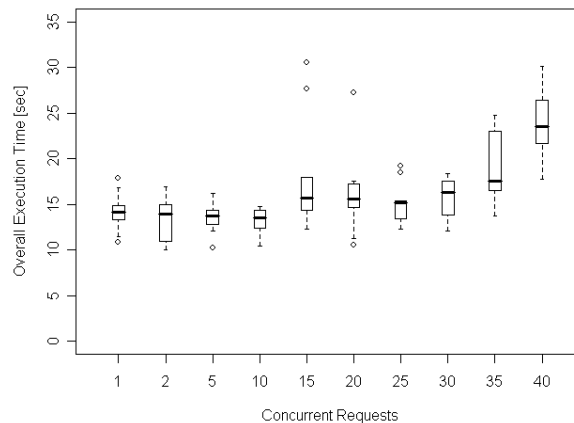


Figure 8.2: Fife-number summary of total execution time for *Dictionary* service as a function of increasing number of concurrent execution requests per peer.

but can also be a matter of the network and/or the Web services used. The visible outliers and skews, even for this moderate composite service, are a matter of the dynamics of the Internet environment and the rather low number of measurements per group.

## 8.3 KB Access Optimization Techniques

The main objective in the evaluation of the two KB access optimization techniques – Prepared Queries and Frame Caching – was to measure the speedup<sup>3</sup> in the execution as a function of (i) the types of services used and (ii) the size and shape of the KB. The latter has been made to confirm the presumption that execution performance using frame caching is independent of the KB size and its structure. In addition, we have verified the presumption that the overhead of filling frames of typical size is negligible.

### 8.3.1 Experimental Setup

In order to quantify the performance gain of Prepared Queries and Frame Caching for service execution, we have implemented them in the OWL-S API such that they can be turned on and off; subsequently we will refer to this as *optimized* (on) versus *conventional* (off) configuration. This provides the flexibility to easily compare any constellation (i.e., for any kind of invocable service and using differently sized and shaped KBs).

Various service specifications have been created that are designed specifically for testing and benchmarking purposes. The intention in the design was to cover a wide range of possible OWL-S process models and types of (pre-)conditions. As a result, they simulate different characteristic cases such as short running services or services

<sup>3</sup>The factor  $k$  to which an optimized procedure is faster than an unoptimized one, given a particular load or problem; that is,  $k = t_u/t_o$  where  $t_u$  and  $t_o$  are the execution times of the unoptimized and optimized procedure, respectively.

Table 8.2: Exemplary services used for evaluation purposes.

Name	Short Description
<i>Any-Order</i>	Composite service that uses an <i>Any-Order</i> control construct consisting of three elements, each being the same atomic service that logs the value of one out of three inputs. Has no preconditions.
<i>For-Each</i>	Composite service that uses a <i>For-Each</i> control construct whose loop body is an atomic service that plays a given MIDI note. This is combined such that one can play an input list of MIDI notes. Uses an SWRL precondition that verifies that each note can actually be played.
<i>If-Then-Else</i> (1, 2)	Two composite services, each consisting of an <i>If-Then-Else</i> control construct with different SWRL branching conditions (built-in <i>less-than</i> and class atom).
<i>MathPow</i>	Atomic service that calculates the power of two numbers, both provided as inputs. Has no precondition.
<i>Repeat-While</i>	Composite service that uses a <i>Repeat-While</i> control construct whose loop condition is expressed in SWRL. The loop body is an atomic service incrementing a given input number. The while loop ends if a target value is reached.
<i>Translator</i>	Atomic service realizing language translation of words provided as an input. Uses an SWRL precondition asserting that the source and target language are supported.

with many iteration cycles. Some of these services are listed and briefly described in [Table 8.2](#).<sup>4</sup>

All experiments started from a completely populated KB, meaning that it contains the service specification under evaluation and required domain ontologies. Furthermore, a consistency check, classification, and realization was initially done on the KB. We used Pellet [SPG<sup>+</sup>07] as the reasoner attached to the KB. Note that KB was always kept entirely in main memory.

Finally, all experiments were conducted on commodity hardware: Win XP; x86 Hyper-Threading CPU, 32 bit, 3.4 GHz; 2 GB RAM; 1 Gb/s Ethernet network connection; Java 6, maximum heap size  $\approx 1.5$  GB.

## 8.3.2 Results

### Speedup as a Function of Service Structure and Conditions

In the first round of experiments, an evaluation run for each service has been performed in the conventional and the optimized configuration to measure the difference in execu-

<sup>4</sup>The services including those not listed in [Table 8.2](#) are available at the OSIRIS NEXT Web site.

Table 8.3: Execution speedup of exemplary services.

Test Case	Conventional	Optimized	Speedup
<i>Any-Order</i>	40 ms	39 ms	1.02
<i>For-Each</i>	5410 ms	448 ms	12.08
<i>If-Then-Else 1</i>	180 ms	33 ms	5.45
<i>If-Then-Else 2</i>	27 ms	3 ms	12.33
<i>MathPow</i>	502 $\mu$ s	42 $\mu$ s	11.96
<i>Repeat-While</i>	2821 ms	828 ms	3.41
<i>Translator</i>	6625 ms	3631 ms	1.82

tion times and calculate the speedup. An evaluation run is the execution of each service at least ten times, with arbitrary breaks between, and the average time was taken. [Table 8.3](#) shows the results. Unlike the second round of experiments, the KB is always “minimal”, meaning that it contains not more information than ultimately required.

The execution time of the *Any-Order* service cannot be improved because it neither has a precondition nor is any part of the process model accessed more than once; hence, also frame caching has no effect. This service was included in order to analyze whether creation of frame cache objects introduces a considerable overhead. As the values show, the execution times differ in what is seemingly a measurement inaccuracy. A more fine grained analysis showed that creation of the frame that represents the entire grounding (having 8 slots) takes 80 ns on average. This confirms that creating and filling a frame is time-wise cheap and therefore negligible. Memory-wise, the overhead of a frame is estimated analytically as

$$\text{Memory overhead per frame} \approx 8 \text{ bytes} + (n \times [rsize|dsize])$$

where  $n$  is the number of slots,  $rsize$  is the number of bytes for an object reference on the CPU architecture used,  $dsize$  is the number of bytes taken by a data value (e.g., an `int`, `double`, `String`), and the leading 8 bytes is the size of a Java object header. This is explained by the fact that a frame is represented as a Java object, and every slot is either a data value or an object reference.

The *For-Each* service is interesting insofar as the speedup is mostly due to the optimized condition evaluation process; that is, the elimination of additional KB inserts. In the conventional setting, the execution time is dominated by the need of repeated classification and realization by the reasoner after a KB update.

The evaluation run of the *MathPow* service is actually different. It is executed 3000 times in succession thereby mimicking frequent re-execution of the same service. [Table 8.3](#) lists the average value for one execution. The speedup here is solely due to frame caching since the service has been deliberately designed not to involve reasoning.

Execution of the *Repeat-While* service benefits from both techniques because of the while loop condition and the repeated execution of the loop body. [Figure 8.3b](#) further details the values listed in [Table 8.3](#) in the fraction measured for (i) condition evaluation and (ii) the execution itself. Apart from the reduction of execution time due to frame caching, the numbers for condition evaluation show that by activating prepared queries

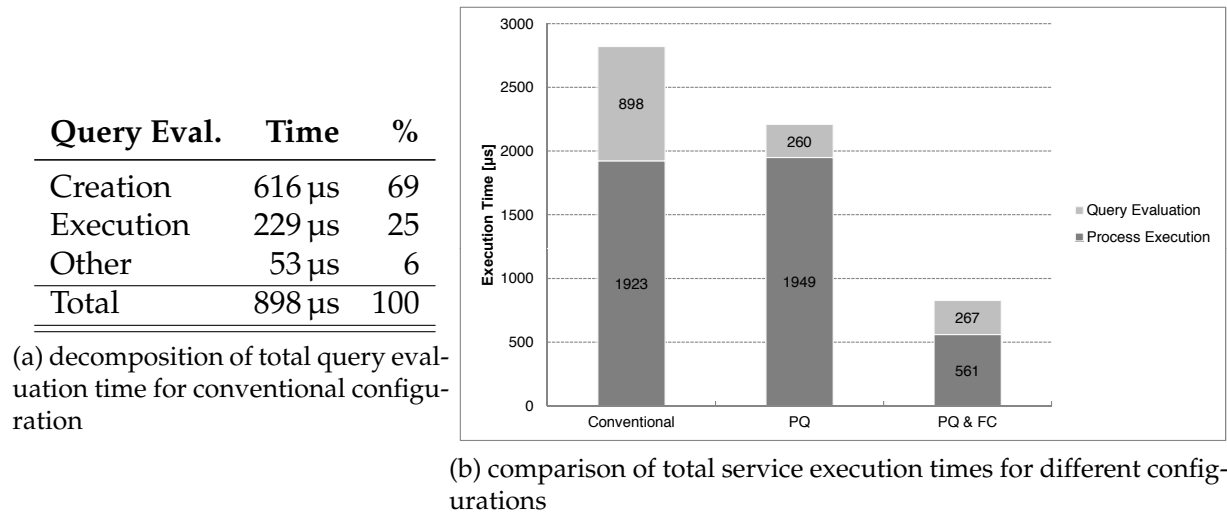


Figure 8.3: *Repeat-Until* service executed with different configurations (PQ = Prepared Queries on; FC = Frame Caching on).

the overhead of repeated translation from SWRL atoms to queries is almost eliminated, cf. Figure 8.3a.

Finally, the values listed for the *Translator* service are filtered measurements of pre-condition evaluation not including execution time. The reason is that frame caching has no effect since neither the process specification nor the grounding is accessed more than once.

### Speedup as a Function of KB Size and Shape

In the second round we repeated the *MathPow* evaluation run, but with an incrementally growing ABox of the KB. The KB has been enlarged in two ways. First, by adding synthetic OWL-S service descriptions. For this, we have implemented a randomized generator that is parametrized in several ways to control the complexity of generated service descriptions; more details about it can be found in Section 8.1.1. Second, by adding randomly generated individuals and assertions about them using concepts and roles of the LUBM ontology [GPH05]. In case of adding more OWL-S services, the KB is enlarged by adding new assertions about *new* individuals. Specifically, the number of triples  $|\langle s, *^p, *^o \rangle|$  for any subject  $s$  remains constant. In the second case, however, the KB is enlarged by adding new assertions about *existing* individuals, which means that  $|\langle s, *^p, *^o \rangle|$  is proportional to the KB increase for those subjects  $s$  about which new assertions are added. As a result, the shape of the KB significantly differs in the maximum branching factor (i.e., the maximum number of assertions about individuals). It should also be noted that in this round execution times are not distorted by reasoning because the *MathPow* service has neither (pre-)conditions nor effects; hence, its execution does not involve query answering nor updates.

Figure 8.4a and Figure 8.4b show that execution times remain constant when using frame caching, no matter what the size of the KB is. In contrast, the numbers for the

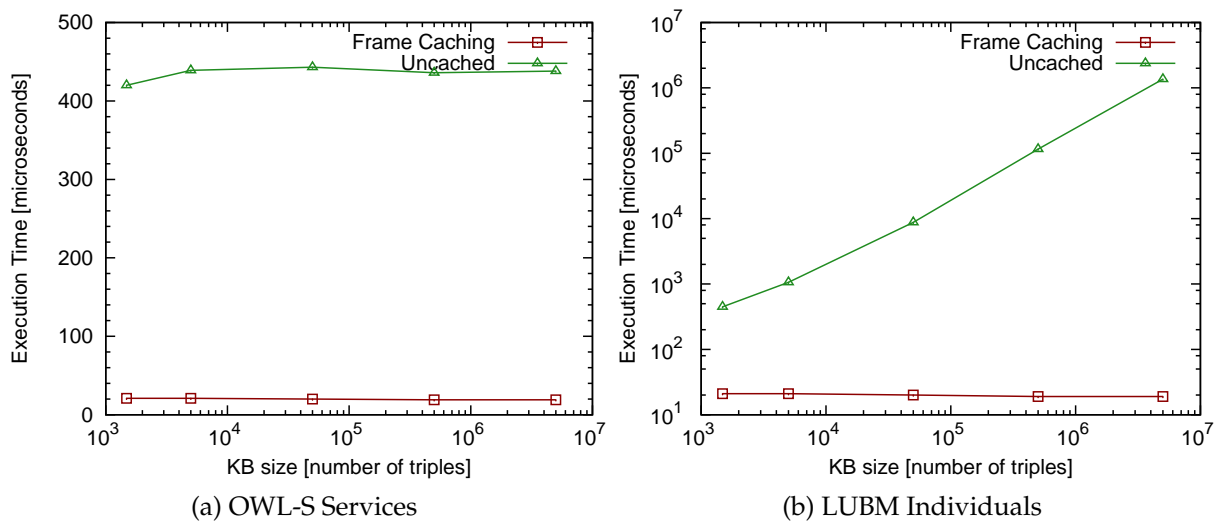


Figure 8.4: Total execution time as a function of KB size for conventional and optimized configuration.

configuration in which caching is deactivated feature significant differences. Whereas [Figure 8.4a](#) shows that execution times also remain constant in case of adding new service descriptions, but at a significantly higher level compared to the cached case, times increase linearly with the KB size when new individuals and assertions are added ([Figure 8.4b](#)). This difference is exactly a result of the shape of the KB. It can be explained by looking at how indexing is designed for graph-based data structures in the underlying Jena graph implementation. In fact, the two cases decidedly model the best and worst case regarding the index structure. In the former case, access takes advantage of the index whereas in the latter case sequential scans are performed in addition, which explains the linear increase.

To conclude, both techniques show considerable performance gain even though the KB is maintained in main-memory. Since both techniques aim at avoiding repeatedly performing the same tasks, the increase is in every way determined by the actual case, which is also evident in the results. Generally, the advantage especially of Frame Caching becomes the larger the higher the latency of direct KB accesses becomes.

## 8.4 Snapshot Isolation OWL Data Store

The evaluation of the SI-based concurrency control method is targeted to measure transaction execution times and transactions per second for different benchmarking workloads. The evaluation is centered around comparing results of our implementation with two contestants in order to demonstrate its competitiveness. Another important aspect that has been investigated is how the contestants behave under an increasing amount of concurrency. Finally, a brief investigation on the time-wise overhead of normalization (induced by E-conflict checking) confirms the presumption that normalization adds little to overall costs.

Table 8.4: Workloads used for the performance analysis and their characteristics.

Workload	#Tx <sup>1</sup>	Min/Max <sup>2</sup>	R/W <sup>3</sup>	A/D <sup>4</sup>	C/R <sup>5</sup>
Base Transactions	100	50/100	1	1	100
Mixed Transactions	100	5/500	10	1	100
Many Transactions	1000	50/100	1	1	100
Few Transactions	10	50/100	1	1	100
Large Transactions	100	250/500	1	1	100
Small Transactions	100	5/15	1	1	100
Read-Centered	100	50/100	100	1	100
Write-Centered	100	50/100	0.01	1	100
Add-Centered	100	50/100	1	100	100
Delete-Centered	100	50/100	1	0.01	100
Medial Commit Ratio	100	50/100	1	1	1
Rollback-Centered	100	50/100	1	1	0.01

<sup>1</sup> Number of transactions per thread. <sup>2</sup> Minimum/maximum number of application operations per transaction. <sup>3</sup> Read/write ratio per transaction. <sup>4</sup> Add/delete ratio per transaction. <sup>5</sup> Commit/rollback ratio per transaction.

## 8.4.1 Experimental Setup

### Benchmark Workloads

As there is no established benchmark for OWL updates, we have designed a benchmark ourselves. It consists of several workloads and aims at exposing candidates to diverse access patterns. These workloads imitate typical application scenarios such as low up to moderate update ratios, bulk loading in which add transactions dominate, and erroneous environments with frequent aborts. We also drive parameters towards corner cases such as large transactions, high abort rates, and update-only transactions in order to analyze whether runtime properties considerably change in these cases. [Table 8.4](#) lists the different workloads and provides an overview on how parameters are set.

Each workload consists of fixed sequences of transactions that are generated in advance for each thread (i.e., one sequence per thread) according to the parameters in [Table 8.4](#). This ensures that for each workload each competitor receives exactly the same transactions arriving in the same order. Transaction length is evenly distributed in the given interval. All ratios are average values. For instance, a commit/rollback ratio of 100 means that 1 out of 100 transactions aborts on average.

### Dataset, Update & Read Queries

A run of any workload starts from a populated KB containing initially (i) the LUBM benchmark ontology [GPH05] and (ii) approximately 10000 axioms and assertions created with its data generator. Updates are made more frequently to the ABox than to the TBox. Irrespective of the workload characteristics, this induces bias towards the ABox (i.e., accesses are not evenly distributed over the KB). Updates add or delete concept

expressions, data ranges, properties, concept (property) inclusion axioms, individual declarations, assertions about individuals, or individual (in)equalities. Reads are in fact simple queries that may have large result sets such as getting all class assertions, all object property assertions, all axioms in the TBox, or getting particular entities. The number of operations per transaction listed in Table 8.4 thus corresponds to typical application operations, but not to the basic add, delete, and read operations introduced in Section 6.2.2. Except for the delete-centered and read-centered workloads, all other workloads imitate a growth of the KB, which is natural in many application domains.

### Competitors

We ran each workload with our (i) implementation, (ii) a base line, and (iii) under *multiple-reader/single-writer* locking, subsequently called *SICC*, *Base*, and *MRSW*, respectively. *Base* does not provide any correctness guarantees because basic add/delete operations are executed at their commencement. However, *Base* uses locking at the level of global set and multi-map data structures in order to make the basic add/delete operations atomic. *MRSW* is enforced by a shared read-write lock, thus, update transactions are exclusive while read-only transactions can execute concurrently. We have also experimented with OWLDB [HKGB09], which was configured to use an in-memory RDBMS (H2 and HSQLDB). Unfortunately, we encountered significant performance limitations probably due to the object-relational mapping. The tests using OWLDB also suffered from runtime exceptions causing workload runs to end prematurely.

### Other Settings

All benchmark runs have been repeated at least three times and the average runtime has been taken. Transaction think time is generally short, starting from 20  $\mu$ s up to 2.5 ms. We generally did not attach a reasoning engine in order not to disturb results by reasoning that takes place in the background. This also implies that S-conflict checking is not performed throughout the experiments, as this would be more of an evaluation of the particular reasoning engine used.

An initial ramp-up phase in which the Base Transactions workload is executed precedes every benchmark run in order to make sure that the Java just-in-time compiler threshold is exceeded so that compiled code is executed rather than byte code in the slower interpreted mode.

Finally, all tests have been conducted on a standard server machine: x86-64 dual QuadCore CPU (16 cores are reported to the OS), 2.26 GHz; 12 GB RAM; Linux.

## 8.4.2 Results

The comparison between the three competitors *Base*, *MRSW*, and *SICC* is shown in Figure 8.5 and 8.6 for every workload. *SICC* performs nearly as good as *Base* on most workloads. In some cases *SICC* is even faster when transaction think time is short; see Base, Small Transactions, and Add-Centered workload. The reason could be that keeping all changes thread-local until they are applied all at once to the global data structures (in case of a commit) results in lower memory contention than performing every change

Table 8.5: Average time to normalize  $n$ -ary axioms/assertions ( $2 \leq n \leq 10$ ).

Axiom/Assertion	Time [ $\mu$ s]
<i>DifferentIndividuals</i>	96
<i>DisjointClasses</i>	98
<i>DisjointUnion</i>	115
<i>EquivalentClasses</i>	30
<i>SameIndividuals</i>	31

directly on the thread-safe global data structures as it is done in the *Base* implementation. Low level profiling would be needed to further evidence this, which is seemingly difficult because of the small time scale that makes it prone to inaccuracy due to instrumentation that is required. The global locking of *MRSW* shows the expected strong increase for increasing transaction think time due to increasing lock hold time. On the other hand, *SICC* is about 0.8s slower than *Base* for the Read-Centered workload, see [Figure 8.5c](#). This result is furthermore independent of the amount of concurrency, see [Table 8.6](#). More detailed analysis revealed that there is an implementation specific extra cost of filtering obsolete data items, which can probably be further optimized.

The Many Transactions workload (see [Figure 8.6c](#)) is interesting insofar as the competitors are exposed to a constant load for a rather long time (compared to the other workloads). We included this workload with the original intention of analyzing whether runtime properties are subject to drifts over time. Because of the side effect that the size of the KB is considerably enlarged, we realized that this workload should rather be considered an evaluation of scalability in KB size.

The results in [Figure 8.6d](#) to [8.6f](#) are for the rather academic workloads of short system up time (Few Transactions), high frequency of delete operations (Delete-Centered) and high frequency of aborts (Rollback-Centered). These results support the results of other workloads. The overall runtime behavior does not change substantially in these cases.

We have repeated each workload run under an increasing amount of concurrency. This is done by doubling the number of concurrent threads from 4 up to 32. In addition, transaction think time was evenly distributed in the interval from 20  $\mu$ s to 500  $\mu$ s for Many, Large, and Mixed Transactions workloads and 100  $\mu$ s to 2.5 ms for all other workloads. Results are shown in [Table 8.6](#). As the load increases, *SICC* is often faster than *Base* (starting from 16 threads, in 7 out of 12 workloads). For high load with 32 threads, *SICC* is even up to 2 times faster for the Small, Base, and Add-Centered workload. Since these are the workloads where *SICC* also performs better for small transaction think time, we attribute this again to lower contention due to keeping changes thread-local until they are committed.

Finally, we have measured the overhead induced by normalization for frequently used  $n$ -ary axioms and assertions, with a range of values for  $n$  that appears most likely in practice. The results shown in [Table 8.5](#) suggest that normalization introduces only a small delay in the range of microseconds.



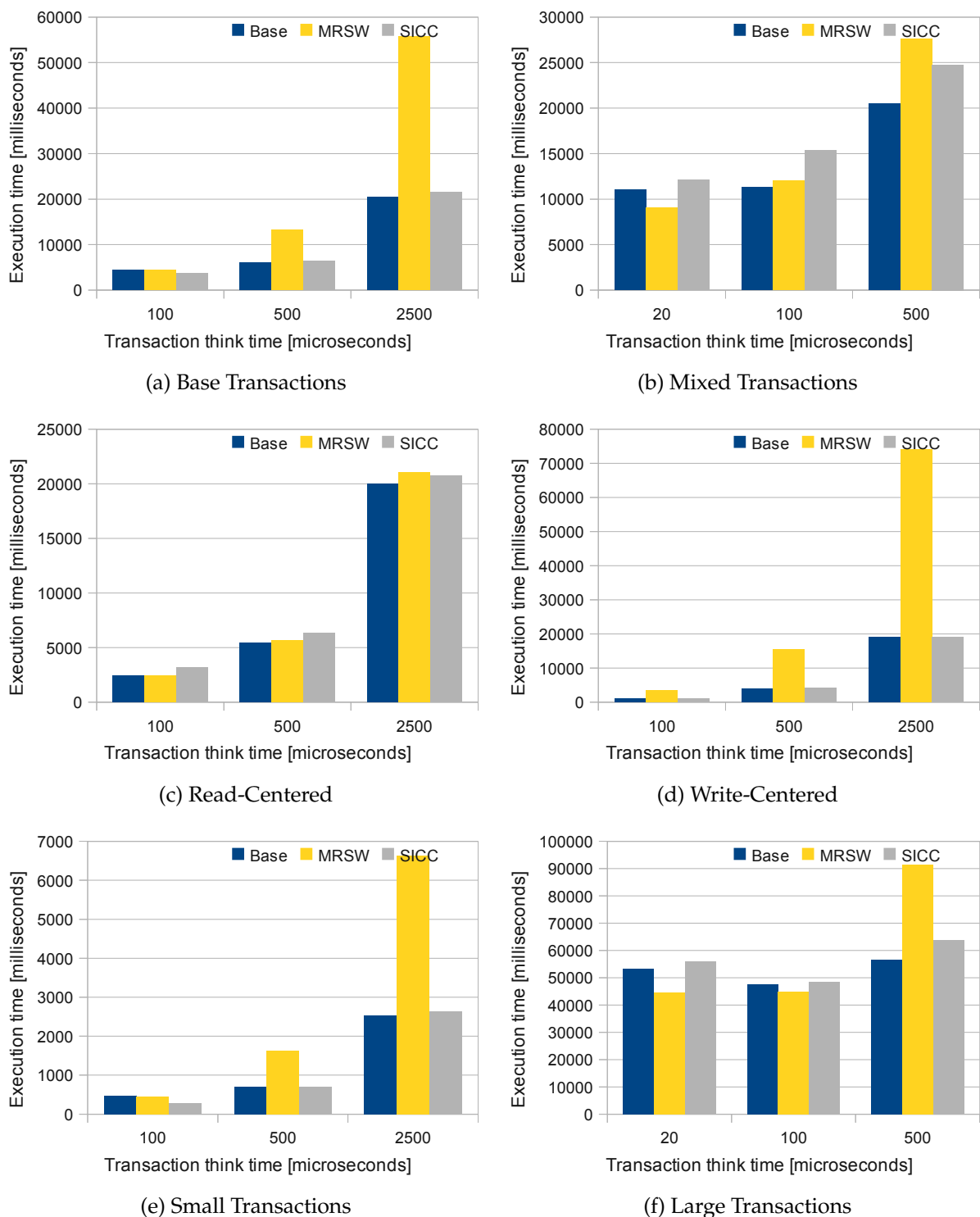


Figure 8.5: Execution times in comparison for basic workloads and 4 concurrent threads. Note the shorter transaction think time for Mixed and Large Transactions workloads.

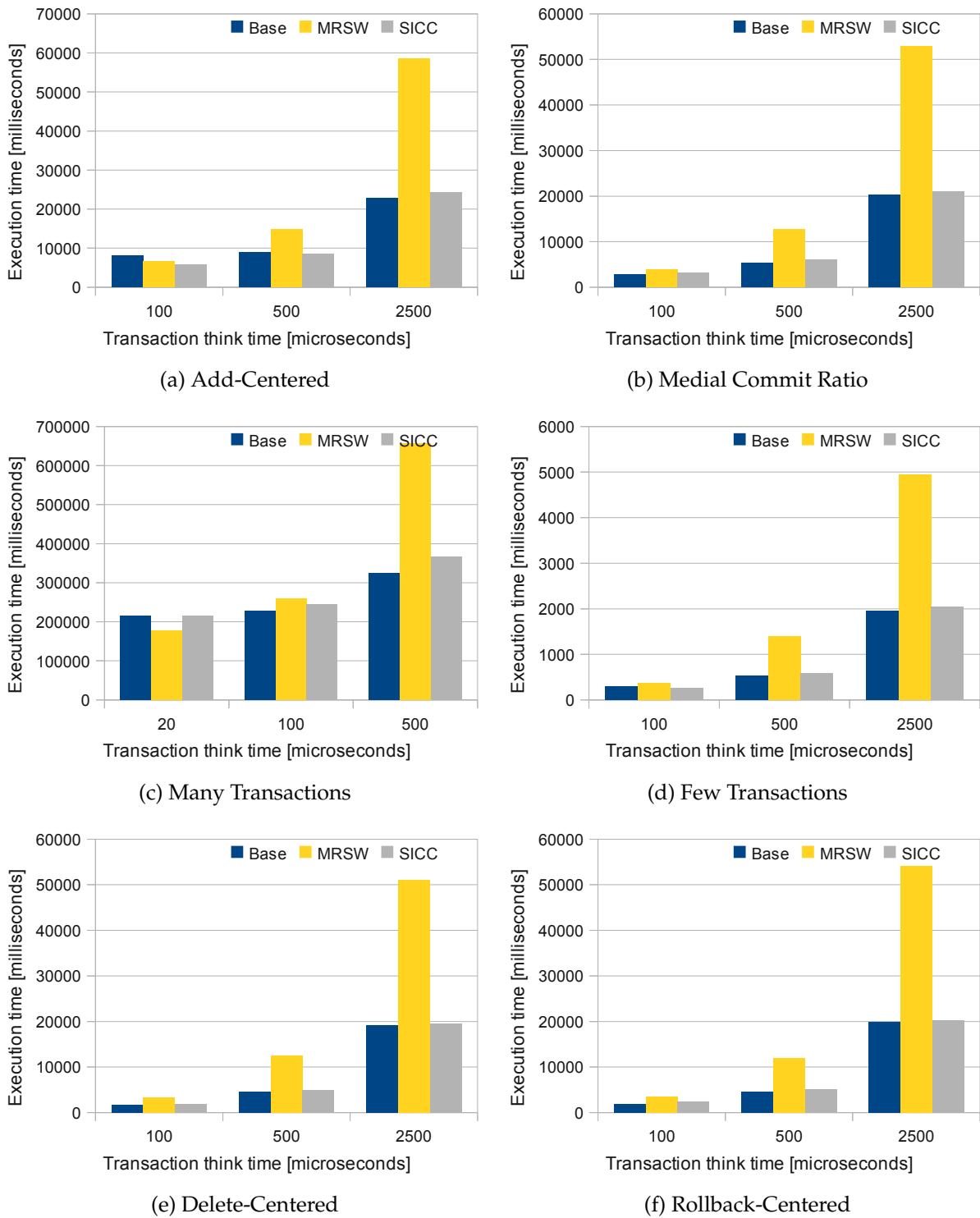


Figure 8.6: Execution times in comparison for additional workloads and 4 concurrent threads. Note the shorter transaction think time for Many Transactions workload.

Table 8.6: Comparison of execution time  $t_{ex}$  in seconds and transactions per second Tx/s metrics among the competitors as a function of increasing concurrency and workload patterns. Final number of OWL axioms, assertions and entity declarations in the KB at the end of each workload run is also listed.

		4 Threads		8 Threads		16 Threads		32 Threads	
		$t_{ex}$	Tx/s	$t_{ex}$	Tx/s	$t_{ex}$	Tx/s	$t_{ex}$	Tx/s
Base Transactions	Base	12	33.3	15	53.3	33	48.5	101	31.7
	MRSW	30	13.3	57	14.0	127	12.6	294	10.9
	SICC	13	30.8	15	53.3	21	76.2	56	57.1
	Number of Data Items	26368		42308		76261		143823	
Mixed Transactions	Base	13.8	30.0	27.7	28.9	59.8	26.8	167.9	19.1
	MRSW	17.4	23.0	32.0	25.0	77.2	20.7	220.9	14.5
	SICC	17.8	22.5	27.8	28.8	60.1	26.6	185.9	17.2
	Number of Data Items	21245		34973		54323		91659	
Many Transactions	Base	250	16.0	781	10.2	2903	5.5	11596	2.8
	MRSW	410	9.8	1087	7.4	3607	4.4	13435	2.4
	SICC	284	14.1	616	13.0	1664	9.6	7227	4.4
	Number of Data Items	174630		343149		677788		1344838	
Few Transactions	Base	1.3	30.8	1.4	57.1	1.6	100.0	3.0	106.7
	MRSW	3.0	13.3	7.3	11.0	12.9	12.4	24.9	12.8
	SICC	1.3	30.8	1.4	57.1	1.7	94.1	1.9	168.4
	Number of Data Items	12261		14232		18124		24419	
Large Transactions	Base	48	8.3	180	4.4	646	2.5	2369	1.4
	MRSW	65	6.1	194	4.1	701	2.3	2753	1.2
	SICC	54	7.4	143	5.6	432	3.7	1643	1.9
	Number of Data Items	89991		191716		358980		706525	
Small Transactions	Base	1.4	285.7	1.7	470.6	2.1	761.9	4.4	727.3
	MRSW	3.8	105.3	7.5	106.7	14.8	108.1	30.0	106.7
	SICC	1.5	266.7	1.7	470.6	1.7	941.2	1.9	1684.2
	Number of Data Items	11616		13017		15758		20388	
Read-Centered	Base	11.0	36.4	12.0	66.7	12.1	132.2	13.8	231.9
	MRSW	11.1	36.0	13.2	60.6	16.0	100.0	21.6	148.1
	SICC	11.8	33.9	12.7	63.0	12.9	124.0	14.8	216.2
	Number of Data Items	10381		10700		11695		13193	
Write-Centered	Base	10.9	36.7	11.0	72.7	12.5	128.0	20.4	156.9
	MRSW	40.0	10.0	77.2	10.4	157.0	10.2	315.2	10.2
	SICC	11.1	36.0	11.1	72.1	11.0	145.5	14.3	223.8
	Number of Data Items	43754		76129		143909		276602	
Add-Centered	Base	14.8	27.0	23.1	34.6	66.9	23.9	232.4	13.8
	MRSW	31.7	12.6	68.4	11.7	157.4	10.2	406.2	7.9
	SICC	15.6	26.6	18.9	42.3	38.4	41.7	126.1	25.4
	Number of Data Items	45703		79655		152402		285901	
Delete-Centered	Base	11.1	36.0	11.0	72.7	11.5	139.1	12.2	262.3
	MRSW	27.8	14.4	57.3	14.0	109.3	14.6	224.3	14.3
	SICC	11.5	34.8	11.2	71.4	11.8	135.6	12.3	260.2
	Number of Data Items	10318		10332		10360		10377	
Medial Commit Ratio	Base	10.9	36.7	12.0	66.7	17.9	89.4	48.8	65.6
	MRSW	28.7	13.9	56.7	13.9	121.0	13.2	254.1	12.6
	SICC	11.6	34.5	12.7	63.0	15.3	104.6	31.0	103.2
	Number of Data Items	18347		27049		43023		78533	
Rollback-Centered	Base	10.9	36.7	10.8	74.1	11.2	142.9	12.1	264.5
	MRSW	29.1	13.7	55.4	14.4	109.0	14.7	225.0	14.2
	SICC	11.3	35.4	11.3	70.8	11.6	137.9	12.5	256.0
	Number of Data Items	10410		10723		10757		12354	



# 9

## Related Work

THIS CHAPTER reviews related work in three different areas. First, CFI introduced in [Chapter 5](#) relates to adaptation and exception handling in the field of workflow management and process-aware information systems. Second, the peer-to-peer style execution realized in OSIRIS NEXT and introduced in [Chapter 7](#) relates to distributed execution of workflows and processes. Finally, the knowledge base concurrency control model and protocol introduced in [Chapter 6](#) relates to transaction theory and management in the field of databases.

### 9.1 Adaptation and Exception Handling

The field of workflow, process, or service adaptation as a means to flexibility and exception handling is too broad to be discussed here in general. This is illustrated further by different taxonomies that have been proposed (e.g., [SMR<sup>+</sup>08, BHB<sup>+</sup>10]) with the aim of better understanding the different dimensions. For instance, the taxonomy proposed in [SMR<sup>+</sup>08] identifies four ways of achieving flexibility, namely by:

- *Design* – anticipated changes in the operating environment are handled by strategies that are defined at design-time.
- *Deviation* – unforeseen changes in the operating environment are handled by deviating from the expected behaviour such that differences are minimal (e.g., re-ordering of activities).
- *Underspecification* – anticipated changes in the operating environment are handled by strategies that cannot be defined at design-time, because the final strategy is not known in advance or is not generally applicable (e.g., *late modeling* or *late binding*).
- *Change* – unforeseen changes in the operating environment are handled by modifying the process specification at execution time.

CFI falls into the last category; notice its difference from flexibility by deviation due to the fact that parts of the control flow are replaced. We therefore concentrate in the

following on approaches that also belong to this category. For instance, we do not take account of approaches to flexibility and exception handling realized in systems such as FLOWer [AWG05] and YAWL [AHAE07] as they focus on flexibility by deviation respectively by design and underspecification. The authors of [SMR<sup>+</sup>08] further point out variability in the *effect* of change and the *moment* when changes are allowed. The former defines whether a change is performed on an instance or on the specification, thereby affecting all new instances. These two types are correspondingly called *momentary* and *evolutionary* change in [SMR<sup>+</sup>08]. The moment at which a change is introduced is either at *entry time* or *on the fly*, meaning either at instantiation time or in the midst of execution. CFI is thus further classified as momentary and on the fly.

Research around the ADEPT1 system [RD98, HRRD03] covers the topic of workflow adaptation at various levels, including runtime deviation (e.g., move an activity for the purpose of reordering or postponement), workflow schema evolution, and ad hoc changes (e.g., insert, delete an activity). The latter has been considered particularly for the purpose of exception handling at runtime in case of process activity failures [RBR06]. These works differ from CFI in that they focus on (i) identifying conditions that need to be satisfied in order to apply a change, (ii) factors that influence the choice of changes, and (iii) how to correctly perform a change operation. However, they do not target automating the search for semantically equivalent or similar replacements. The methods presented therein and in subsequent works [LRD06, LRD08] consider the semantic aspect to the extent of describing application-specific mutual exclusion and dependency constraints among activities, which therefore make them more related to the declarative paradigm of constraint-based workflow specification [PSSA07]. Research along correctness and reasoning about workflow adaptation is continued in the successor system ADEPT2 [RRD09] (and its commercial offshoot AristaFlow). While research in the context of ADEPT1 focused on the control flow perspective, a relaxed notion of correctness for workflow instance changes that additionally takes the data flow into account is presented in [RRW08].

The declarative workflow system DECLARE [PA06, PSSA07] also features momentary and on the fly workflow change. The fundamentally different principle of defining a workflow as a set of activities together with a set of constraints over them (as opposed to an explicit specification of the control and data flow in the prevalent while imperative paradigm) greatly facilitates flexibility since all kinds of executions are anyway admitted that satisfy the constraints (i.e., a set of activities and an empty set of precedence-implying constraints admits any sequential and/or parallel combination).

The major difference of CFI to the methods of change realized in DECLARE as well as both versions of ADEPT is that their objective does not lie in finding respectively composing failure handling means on demand depending on what has failed. Their focus lies rather on investigating conditions under which a process instance can be modified such that it complies with its changed schema. The important assumption is that the change (i.e., what and how) is determined externally usually by humans, not by the system itself. All these works should therefore be considered complementary.

Probably the closest relative to the techniques introduced with CFI has been presented in [VWS08, WVKS08]. The authors introduce *replace-by* recovery actions to recover from a failure in the execution of a service in a forward-oriented way, which

is virtually the same idea that underlies CFI. The authors also consider exploiting semantic service descriptions (which is based on OWL-S) in order to search for recovery actions. However, these works stay at a coarse level of detail, do neither describe an implementation nor an evaluation.

Another close relative is [FFM<sup>+</sup>10]. The authors propose an approach to handle exceptions at execution time by repair plans. Generation of a repair plan takes into account constraints posed by the process structure, dependencies among data, and available repair actions. A model based theory is used to reason about these three elements. Generation of a plan is formulated as a planning problem and the search is based on disjunctive logic programming. The authors also discuss the notion of *repairability* as a property of a composite service that should be verifiable at design time. The important assumption in this work is that repair actions are defined at design time together with rules for applying them. This assumption is the basis that makes repairability analysis possible at design time.

Finally, the idea of applying AI planning as a means to adaptivity and exception handling in workflow systems and service-oriented architectures is not new. An early work in this direction is [JMS<sup>+</sup>99]. Among subsequent and similar works are [GMM<sup>+</sup>05, FF06, DBCO07, MMR11]. The single overarching principle in all these works including the planning-based technique put forward by this work is that execution and planning is interleaved along the lines of dynamic planning. However, what distinguishes our framework from these works is that we consider a DL-based world state representation and change semantics that corresponds to a query answering respectively a belief update problem.

## 9.2 Distributed Execution

Before discussing related work in this field, we identify key classifying characteristics that can be used to compare execution systems. Most of all, the essential criterion required of an execution system to qualify as distributed is that the locus of control may be distributed at execution time in location and time as detailed in [Footnote 5 at Page 179](#). The subsequent discussion concentrates on systems that meet this criterion. Yet the selection is representative rather than exhaustive as there are numerous proposals in the literature. Other classifying features related to distributed execution are:

- Close versus remote coupling of execution nodes and atomic services/operations: the former allows for local invocations whereas the latter necessitates remote invocations.
- Direct forward of data being processed between execution engines versus indirect via overlay networks or (centralized) middleware components (e.g., via a broker or queueing system).
- Static versus dynamic decision how control is forwarded between nodes. Static means that all decisions are made prior to execution and can thus not be changed at execution time, whereas dynamic means that decisions are made on the fly (e.g., to allow for late binding).

Historically, research on system support for distributed workflows and the distributed execution thereof started in the 1990s, mainly as a means to serve the, at that time, increasing interest in availability, reliability, and scalability. Among these works are [BMR94, AMG<sup>+</sup>95, WWWD96, BD97, BD00]. Except for [WWWD96] they all share the property of close coupling.

The INCAS system [BMR94] considers workflows that execute under the control of autonomous nodes and presents a computational model that already considers (i) dynamics in the workflow (i.e., the control and data flow may change at execution time), (ii) partial automation (i.e., some activities in a workflow may be manual), and (iii) partial connectivity (i.e., nodes may be transiently disconnected). Distributed execution is coordinated based on so-called *information carriers*, which are essentially objects that carry information relevant for (i) the local execution by a node and (ii) routing of control between the nodes involved in a workflow. Information carriers are forwarded directly (end-to-end) between nodes. Another specific property of INCAS is that the forward of control as well as the data flow is encoded by means of *event-condition-action* rules, which means that every decision that is to be made in the course of execution needs to be representable in this scheme. Rules of this kind are similarly used in the EVE system [GT98].

The Exotica/FMQM architecture [AMG<sup>+</sup>95] belongs to the first proposals in which complete distribution of execution on individual nodes is considered on one hand, and in which the authors deal with the aspect of failure-resilience; to be precise, the crash of single nodes. The latter is achieved by means of persistent and transactional messaging between nodes. Messages are therefore exchanged indirectly via messaging middleware. The approach requires all activities within a workflow to be reentrant (i.e., they can be safely restarted after a crash) and assumes no inter-workflow dependencies. The authors also do not discuss whether control forwarding decisions are static or not.

The execution system presented in [BD97, BD00] is conceptually similar to OSIRIS NEXT in that it also considers the migration of ongoing execution instances between nodes (which may have been added as a result of a growing overall load). The difference is, however, that the authors focus on optimizing communication costs only. A later paper in this line of work analyzed the aspect whether dynamic migration of execution state from one node to another justifies the additional (communication) effort that it creates [BR04]. It is shown that it is not reasonable as a reaction to an overload of the communication system and/or the current node since it would add even more load to them.

In [NCS04], the authors present a compiler-inspired analytical optimization technique. Based on a cost model that analyzes the dependencies in the control and data flow, a composite service (specified in BPEL) is partitioned into sections that can be executed independently. The goal is to minimize communication costs while maximizing the throughput of multiple concurrently executed service instances, which is achieved by reordering sections. The underlying assumption is therefore frequent re-execution of services, which makes it different from the objective of OSIRIS NEXT that targets ad hoc services.

While the objective of the OSIRIS system [SWSS03, SWSS04, SST<sup>+</sup>05] is a true decentralized peer-to-peer approach in which no central components exist, its design also



makes it well suited for frequent re-execution, which is mainly due to close coupling and an approach similar to [AMG<sup>+</sup>95]. Specifically, the process is partitioned into sections equal to the activities, which are mapped to atomic services. The architecture considers that nodes at which atomic services are deployed are equipped with an additional so-called *hyperdatabase layer*, which can be understood as a local execution engine. Prior to execution of a process, a node that provides a service within the process needs to be prepared by providing it with all information necessary to locally forward control and data to subsequent nodes according to the process' control flow. The initial distribution of this information is insignificant if the process is executed many times (since it is done once only). However, it turns out to be an overhead for ad hoc processes (ad hoc composite services) that are executed a few times only, possibly just once; which is the reason why OSIRIS NEXT is designed not to involve such an initial step.

Execution of ad hoc composite services has also been addressed in the AMOR system [BCF<sup>+</sup>06]. Similarly to OSIRIS, its architecture also relies on meta data repositories, but it follows a mobile agent approach, meaning that a process instance is organized as a mobile agent that moves directly between peers (end-to-end) at which services are deployed. This approach therefore also assumes close coupling, and requires an agent runtime layer at service provider peers.

Finally, another agent-based approach has been realized in the NIÑOS architecture [LMJ10]. One of its main characteristics is that it relies on publish/subscribe message exchange. Specifically, peers communicate indirectly via a message broker overlay network. The control flow of a process is deployed indirectly to so-called *activity agents* by establishing a set of subscriptions. More precisely, in order to represent a precedence relation between two activity agents  $A < B$  (i.e.,  $A$  precedes  $B$ ), agent  $A$  defines a topic to which agent  $B$  subscribes. If  $A$  completes execution it sends a message to the topic which is then received by  $B$ , thereby handing over control to  $B$ . Messages sent are furthermore used to forward data according to the data flow. This shows that the approach also involves an initial deployment phase, which therefore makes it inappropriate for ad hoc composite services.

## 9.3 Concurrent Access to Knowledge Bases

While research in database concurrency control has been around nearly as long as database research (with the result that the topic is well understood today and backed by thorough theory), the study of correctness in concurrent access to shared axiomatic knowledge bases by developing an analogous transaction theory is in its infancy. Not surprisingly therefore, the literature on the topic is rare, at best. One reason is perhaps the practice “to use a database system to store the information in a DL knowledge base representation system [...] so as to piggyback on the facilities for concurrent access provided by the database system” [BCM<sup>+</sup>07, Section 7.3.3]. One example in this direction is [HKGB09] in which the authors consider the use of a relational DBMS together with an object-relational mapping framework as the underlying data store for the OWL API. Other examples that are all backed by relational database technology are [BHT05, ZML<sup>+</sup>06, AJPS10, CGL<sup>+</sup>11b]. These works, however, are more focused on

efficient mappings between OWL and the relational model as well as efficient query answering. The aspect of concurrent updates and reads is barely discussed explicitly. In some cases the rather restrictive policy of multiple-reader/single-writer (MRSW) is assumed.

However, the consequence of using conventional database technology is that the notion of consistency is restricted to the data level. The higher level notion of consistency at the semantic (logical) level is not considered. Jointly addressing both notions by transaction processing has, to the best of our knowledge, not yet been described in the context of axiomatic knowledge representation.

The picture of concurrency control is similar when it comes to RDF triple stores. While one direction also considers the use of relational database technology underneath, there are also “native” RDF triple stores that are designed particularly for RDF’s graph based data model. The *x-RDF-3X* system [NW10] is one example in this regard in which the access protocol is tailored for RDF specifics. The authors consider the use of Snapshot Isolation combined with predicate-locking for updates to circumvent the overhead of fine-grained locking over RDF triples. Other native triple stores such as *Bigdata* [SYS09] and *AllegroGraph*<sup>1</sup> [Fra] essentially use Snapshot Isolation, but it is not clear from the documentation whether they implement additional means to avoid write skews. The *Jena Tuple Database* (TDB) [Jenb] started out with the limited MRSW access policy. Only recently the authors have extended it with transactional means providing serializable access through the use of a still rather limited policy of single writer *plus* multiple reader, which should not be confused with MRSW that is defined as single writer *xor* multiple reader.

Concerning the transactional model, close relatives to our work are the unified transaction model with semantically rich operations [VHBS98] and the concept of multilevel (or nested) transactions [Mos85] and later generalizations. In fact, our work draws from the theory established by these works, particularly [VHBS98]. Multilevel transactions, on the other hand, consider transaction management at different levels of abstraction independently, albeit not separately. Data items at a higher level are subject to a 1:n mapping to the next lower level (e.g., the mapping from  $di_{\psi}^{\text{OWL}}$  to  $di_{\psi}^{\text{RDF}}$ ). Similarly, a higher level operation is implemented as a sub transaction, by a sequence of lower level operations. Whenever a conflict is detected and handled at a higher level, this needs to be handled accordingly at the next lower level; thus, making sure that an execution at lower level does not violate a scheduling decision that has been made further up. Several such approaches exist that differ in the degree of parallelism that can be achieved. Closed nested transactions [Mos85] restrict the visibility of each sub transaction completely to the scope spanned by its top-level transaction, which severely impacts the degree of concurrency. In open nested transactions [WS92], in turn, sub transactions are allowed to commit prior to the commit of the associated top-level transaction, which leads to an increased level of concurrency. The composite systems theory, finally, [ABFS97] considers conflicts at lower levels at finer granularity and even allows to execute conflicting higher level operations concurrently at the next lower level.

Works on concurrency control in the context of deductive databases and Frame-based knowledge representation are furthermore worth noting. In [CM95], a locking-

---

<sup>1</sup>Version 4.4

based protocol called *Dynamic Directed Graph policy* has been presented and analyzed which supports graph-based KBs that contain cycles. In the area of Frame-based KBs, an optimistic concurrency control algorithm has been presented in [KCP99]. An approach similar to multi-granularity locking with enhanced lock modes capturing the abstraction relationships' semantics has been presented in [RH95]. None of these approaches is generally blocking-free for reads because they use locking-based protocols.

Finally, somewhat further away are works on collaborative while concurrent ontology development, which is especially relevant for large ontologies such as SNOMED CT [Int] that are developed and maintained in larger teams. A recent proposal to this has been made in [EGHB11]. The authors adapt concurrent versioning techniques that are usually used in software engineering. Furthermore, the authors have defined notions of structural equivalence and difference between different states of an ontology. They are used to enable tool-based resolving of structural as well as semantic (logical) conflicts.



# 10

## Conclusions and Future Work

**A**MONG THE TASKS in the lifecycle of service-oriented applications, service execution, as it has been viewed in this thesis, is of equal standing to its siblings service discovery and composition. In fact, the means of flexibility investigated herein involved an interwoven treatment of the three of them. There is no doubt that the increased flexibility comes at a price: an additional layer of complexity. In this concluding chapter, we recapitulate the main contributions vis-à-vis the additional complexity and point out research directions that can extend and improve our work.

### 10.1 Summary

In this thesis, we have extended the computational basis of automated service execution support in two ways: First and foremost, we have presented a complete framework to deal with runtime failures in an optimistic and forward-oriented way that spares one the effort of anticipating and pre-defining recovery means. Second, we have put forward a method for distributed and decentralized composite service execution. These two features bring about two dimensions of flexibility that account for challenges which we have identified in new application domains that adopt the service-oriented computing paradigm, namely: ad hoc composed services, distributed setting, variety of devices (mobile, stationary, embedded), and possibly large amount of semantically related services out of which composite services can be synthesized.

As an inceptive step, we have laid the basis by setting up a formal system model that views service execution as a discrete process and that incorporates those dimensions of service semantics that are relevant to our objectives. While the overall strategy of modeling service semantics follows prior work on semantic services in a number of respects, we were able to generalize and extend it as well as to combine the different dimensions of service semantics in a coherent way. Namely, the immanent relationship between change and execution semantics has been combined seamlessly and in a general way through the notion of an execution state and its advancement as determined by the concrete precondition and effect system used. The possibility to instantiate the system with

different precondition and effect systems provides a way to vary the tradeoff between expressivity and computational complexity depending on practical needs.

For the failure recovery framework, the essential step was to formulate a notion of semantically equivalent execution that reflects the intuition of humans. Specifically, we have formulated two notions of functionally equivalent execution, which are reduced respectively to a matchmaking and a planning problem. The reason to formulate two rather than a single notion is motivated both technically and with application domains in mind. The matchmaking-based replacement search technique trades narrower scope of practical applicability for lower algorithmic complexity. The planning-based technique, on the other hand, is strictly more general since it abstracts from the control flow, but comes at a higher algorithmic complexity.

The planning-based replacement composition technique that we have presented builds directly upon the DL knowledge base part of the execution state, which is motivated precisely by getting to the aforementioned seamlessness. The consequence is that a translation into a propositional planning framework such as STRIPS is not generally given, depending on the concrete precondition and effect system used. Other researchers may prefer translatability into a propositional planning framework in order to reduce implementation efforts by using off-the-shelf planning tools.

We have furthermore described a way how the planning-based notion of equivalence can be broadened towards similarity as well as to take into account the non-functional dimension. This follows essentially the shift from viewing goals as mandatory towards viewing them as desired and formulating preferences over goals. The consequence of this extension is however that combinatorial search performed during planning is extended by an optimization problem.

The central concept in the system model – the use of the symbolic approach based on Description Logics to represent and reason about semantics – implies that reasoning in the notions of equivalence that we have defined essentially reduces to the basic deductive subsumption inference. After pondering over the usefulness of relying on subsumption, it turned out that a sufficiently accurate conceptualization of the domain is more important in order to avoid ambiguities in profiles. Yet it is precisely this accuracy that is often hard to achieve – developing an accurate and consistent domain conceptualization is still time-intensive expert work. The problem is also due to the fact that humans are subject to beliefs. Beliefs may change over time and different people may have different beliefs. This calls for future improvements and we will come back to this aspect below.

Regarding the way how execution is organized, we have presented a self-contained strategy that is distributed, decentralized, and that is designed particularly for ad hoc services and devices of a diverse range of computing, memory, and network resources. As the use of semantic services involves frequent access to a KB at execution time, regardless of how it is organized, we have also presented two optimization techniques geared towards reducing the number of accesses, avoiding unnecessary updates, and simplifying query evaluation. Both techniques can greatly improve performance and are furthermore applicable generally beyond the service execution task.

Another problem investigated in this thesis is coordinating concurrent access to shared knowledge bases such that incorrect inferences are avoided. This is a problem of

general relevance to shared knowledge management. It occurs in our system model due to the fact that we allow for concurrent execution of multiple service instances. Specifically, we have considered the axiomatic knowledge representation paradigm that comes with a higher level semantic (logical) notion of consistency as a prerequisite to reasoning over knowledge. The approach presented is the first to combine the conventional notion of consistency at the data that represents the knowledge with the higher level semantic notion of consistency. This is achieved by taking axioms (syntactic instances, as we also call them) as the unit of concurrency control. Since they are furthermore immutable, it appeared natural to us to depart from the prevalent read/write model towards a model in which the basic operations are add, delete, and read. The access protocol is essentially Snapshot Isolation with a simple algorithmic extension. Though Snapshot Isolation does not satisfy the correctness notion of serializability, it has been chosen mainly because query answering might involve intractable reasoning, which does not pose a problem as reads are non-blocking in this protocol. We have finally proposed an alternative notion of correctness that is exclusively based on integrity constraints and sketched a way how it could be realized for Snapshot Isolation.

Finally, the conclusion from the implementation and the quantitative evaluation is that the different methods present an ensemble capable of handling practical problems of a realistic size, thereby demonstrating the potential of the overall approach.

## 10.2 Future Work

Not looking through a researcher's eyes for a moment, perhaps the premier question a practitioner might raise is what are the costs of implementing the approach in practice and what are its risks. While the results obtained from our quantitative evaluation demonstrate the basic feasibility, a thorough study in a real practical setting is needed. Setting off such a study is still difficult because a key element in the overall approach – the use of profiles describing the semantics of services and their operations – turns out to be still critical in this regard: the until now unresolved cold start problem of semantic services [BB10]. To date, it is by far not standard that profiles are available for real existing services. Part of the problem is also due to the fact that developing domain conceptualizations is expensive expert work. The biggest challenge is therefore to make the process of creating these annotating profiles less costly while retaining accuracy. In fact, we believe that the success of semantic services in general depends strongly on automation support for this process. What is needed at least is tool support that makes the process easier for humans and faster in total. Perhaps it is also necessary to become rewarding for service providers. Ideally, the process is automated to an extent that humans will be in the position of final assessment and revision of profile proposals created by computers. Approaches that seem most potential to this are (statistic-driven) machine learning methods. Though this problem is related it has a direct influence on the future success of the work presented herein.

Coming back to the research perspective, there are several directions in which our work can be further extended and improved, some of which are connected with related areas.

Two topics for future work are readily visible from the simplifying assumptions (A2), (A8) that we have made. Transferring optimistic failure-handling to data stream processing services yields questions about how to represent and reason about the semantics of data streams and processing operators. Approaches might, however, build upon theoretical and technical groundwork for handling operator failures presented in [BS11]. Relaxing the assumption on the failure behavior is closely related to the extension of the service and process model towards prompt representation of effects. The current model has it that application of effects to the world state representation in the KB is considered to be made at once upon completion of operation invocation. This is the less appropriate the longer an operation runs. One could therefore further investigate a paradigm shift towards a model in which effects might occur any time during the invocation of an operation and are immediately applied to the KB rather than upon completion. We see such an extension involved both on the theoretical side and on the technical side. What is required technically is that operations need to be either inspectable so that they can be observed for their effects (i.e., monitoring), or provide an event mechanism for instant notification about creation of effects (i.e., callbacks).

Another interesting question is how to extend CFI to include ad hoc defined roll-back and compensation. In other words, how can the vacant quadrant in Figure 5.1 be filled. While we have outlined different possibilities already in Section 5.5.2, a formal underpinning is certainly left to be done that particularly includes an investigation of the property of guaranteed termination. More generally, a framework defining a notion of recoverability and means to reason about it such that it can be verified prior to execution is worth being developed.

Finally, Integrity Isolation, the alternative notion of correctness for concurrent transactions discussed in Section 6.6.1 is worth being further investigated. Since the all-important part in this approach is identification of integrity constraints in practical domains and whether their quantity is manageable, the next logical step before further technical considerations should be an investigation whether identification is reasonably possible.



# Appendix

## A.1 Effect System Algorithms

The following two algorithms are based on [CKNZ10] and have been extended to accommodate the additional features of the DL considered by the Effect System (ES1). Given a TBox  $\mathcal{T}$ , let  $cl(\mathcal{T}) = \{\phi \mid \mathcal{T} \models \phi\}$  be the deductive closure of  $\mathcal{T}$  (i.e., all TBox axioms entailed by  $\mathcal{T}$ ). Analogously, let  $cl_{\mathcal{T}}(\mathcal{A})$  be the deductive closure of  $\mathcal{A} \cup \mathcal{T}$ . Algorithm *FastEvol* takes as its input a KB  $\mathcal{K}$ . Furthermore, given a primal update  $U_p$ , it takes the set  $\mathcal{A}_+ = \{\phi \mid (\mathcal{K} + \phi) \in U_p\}$  (i.e., the assertions added by  $U_p$ ) and the set  $\mathcal{A}_- = \{\phi \mid (\mathcal{K} - \phi) \in U_p\}$  (i.e., the assertions deleted by  $U_p$ ). *FastEvol* computes the set of all conflicting assertions  $D$ . Algorithm *Weeding* takes the KB  $\mathcal{K}$  and the set  $D$  and deletes every assertion  $\phi \in D$  if  $\phi \in \mathcal{A}$  (i.e., if it is explicitly asserted). Otherwise, it deletes all assertions from  $\mathcal{A}$  that  $\mathcal{T}$ -entail  $\phi$ .

```
Input: consistent KB  $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ , set of assertions  $D$  to be deleted from  $\mathcal{A}$ 
Output: concomitant update  $U_c$ 
1:  $U_c := \emptyset$ 
2: for each  $C_1(c) \in D$  do // first, pick concept assertions
3:   if  $C_1(c) \in \mathcal{A}$  then  $U_c := U_c \cup \{\mathcal{K} - C_1(c)\}$  end if
4:   for each  $B \sqsubseteq C_1 \in cl(\mathcal{T})$  do
5:     if  $B(c) \in \mathcal{A}$  then  $U_c := U_c \cup \{\mathcal{K} - B(c)\}$  end if
6:     if  $B$  equals  $\exists R.C_2$  then
7:       for each  $R(c, d) \in cl_{\mathcal{T}}(\mathcal{A})$  do
8:         if  $C_2(d) \in cl_{\mathcal{T}}(\mathcal{A})$  then  $D := D \cup \{R(c, d)\}$  end if
9:       end for
10:    end if
11:  end for
12: end for
13: for each  $\neg C_1(c) \in D$  do // second, pick negated concept assertions
14:   if  $\neg C_1(c) \in \mathcal{A}$  then  $U_c := U_c \cup \{\mathcal{K} - \neg C_1(c)\}$  end if
15:   for each  $B \sqsubseteq \neg C_1 \in cl(\mathcal{T})$  do
16:     if  $B(c) \in \mathcal{A}$  then  $U_c := U_c \cup \{\mathcal{K} - B(c)\}$  end if
17:     if  $B$  equals  $\exists R.C_2$  then
18:       for each  $R(c, d) \in cl_{\mathcal{T}}(\mathcal{A})$  do
19:         if  $C_2(d) \in cl_{\mathcal{T}}(\mathcal{A})$  then  $D := D \cup \{R(c, d)\}$  end if
20:       end for
21:    end if
22:  end for
23: end for
```

Algorithm 2, Part 1 of 2: *Weeding*( $\mathcal{K}, D$ )

```

24: for each  $R_1(a, b) \in D$  do // third, pick role assertions
25:   if  $R_1(a, b) \in \mathcal{A}$  then  $U_c := U_c \cup \{\mathcal{K} - R_1(a, b)\}$  end if
26:   for each  $R_2 \sqsubseteq R_1 \in cl(\mathcal{T})$  do
27:     if  $R_2(a, b) \in \mathcal{A}$  then  $U_c := U_c \cup \{\mathcal{K} - R_2(a, b)\}$  end if
28:   end for
29: end for
30: for each  $\neg R_2(a, b) \in D$  do // finally, pick negated role assertions
31:   if  $\neg R_2(a, b) \in \mathcal{A}$  then  $U_c := U_c \cup \{\mathcal{K} - \neg R_2(a, b)\}$  end if
32:   for each  $R_2 \sqsubseteq R_1 \in cl(\mathcal{T})$  do
33:     if  $\neg R_1(a, b) \in \mathcal{A}$  then  $U_c := U_c \cup \{\mathcal{K} - \neg R_1(a, b)\}$  end if
34:   end for
35: end for

```

Algorithm 2, Part 2 of 2: *Weeding*( $\mathcal{K}, D$ )

**Input:** consistent KB  $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ , ABox  $\mathcal{A}_+$  consistent with  $\mathcal{T}$ , ABox  $\mathcal{A}_-$

**Output:** concomitant update  $U_c$

```

1:  $\mathcal{A}_0 := cl_{\mathcal{T}}(\mathcal{A} \setminus \mathcal{A}_-) \cup cl_{\mathcal{T}}(\mathcal{A}_+)$ ,  $\mathcal{A}_+ := cl_{\mathcal{T}}(\mathcal{A}_+)$ ,  $D := \emptyset$ 
2: for each  $C(a) \in \mathcal{A}_+$  do
3:   if  $\neg C(a) \in cl_{\mathcal{T}}(\mathcal{A})$  then
4:      $D := D \cup \{\neg C(a)\}$ 
5:   end if
6: end for
7: for each  $R$  occurring in  $\mathcal{T}$  do
8:   if  $\{R(a, b), \neg R(a, b)\} \subseteq \mathcal{A}_0$  then
9:     if  $R(a, b) \in \mathcal{A}_+$  then  $D := D \cup \{\neg R(a, b)\}$ 
10:    else  $D := D \cup \{R(a, b)\}$  end if
11:   end if
12: end for
13: for each  $R^-$  occurring in  $\mathcal{T}$  do
14:   if  $\{R(a, b), \neg R^-(b, a)\} \subseteq \mathcal{A}_0$  then
15:     if  $R(a, b) \in \mathcal{A}_+$  then  $D := D \cup \{\neg R^-(b, a)\}$ 
16:     else  $D := D \cup \{R(a, b)\}$  end if
17:   end if
18:   if  $\{R^-(a, b), \neg R(b, a)\} \subseteq \mathcal{A}_0$  then
19:     if  $R^-(a, b) \in \mathcal{A}_+$  then  $D := D \cup \{\neg R(b, a)\}$ 
20:     else  $D := D \cup \{R^-(a, b)\}$  end if
21:   end if
22: end for

```

Algorithm 3, Part 1 of 2: *FastEvol*( $\mathcal{K}, \mathcal{A}_+, \mathcal{A}_-$ )

```

23: for each  $\text{Dis}(C_1, C_2) \in \text{cl}(\mathcal{T})$  do
24:   if  $\{C_1(a), C_2(a)\} \subseteq \mathcal{A}_0$  then
25:     if  $C_1(a) \in \mathcal{A}_+$  then  $D := D \cup \{C_2(a)\}$ 
26:     else  $D := D \cup \{C_1(a)\}$  end if
27:   end if
28: end for
29: for each  $\text{Dis}(R_1, R_2) \in \text{cl}(\mathcal{T})$  do
30:   if  $\{R_1(a, b), R_2(a, b)\} \subseteq \mathcal{A}_0$  then
31:     if  $R_1(a, b) \in \mathcal{A}_+$  then  $D := D \cup \{R_2(a, b)\}$ 
32:     else  $D := D \cup \{R_1(a, b)\}$  end if
33:   end if
34: end for
35: for each  $\text{Fun}(R) \in \text{cl}(\mathcal{T})$  do
36:   if  $\{R_1(a, b), R_2(a, c)\} \subseteq \mathcal{A}_0$  then
37:     if  $R_1(a, b) \in \mathcal{A}_+$  then  $D := D \cup \{R_2(a, c)\}$ 
38:     else  $D := D \cup \{R_1(a, b)\}$  end if
39:   end if
40: end for
41: for each  $\text{Asy}(R) \in \text{cl}(\mathcal{T})$  do
42:   if  $\{R_1(a, b), R_2(b, a)\} \subseteq \mathcal{A}_0$  then
43:     if  $R_1(a, b) \in \mathcal{A}_+$  then  $D := D \cup \{R_2(b, a)\}$ 
44:     else  $D := D \cup \{R_1(a, b)\}$  end if
45:   end if
46: end for
47: for each  $\text{Sym}(R) \in \text{cl}(\mathcal{T})$  do
48:   if  $\{R_1(a, b), \neg R_2(b, a)\} \subseteq \mathcal{A}_0$  then
49:     if  $R_1(a, b) \in \mathcal{A}_+$  then  $D := D \cup \{\neg R_2(b, a)\}$ 
50:     else  $D := D \cup \{R_1(a, b)\}$  end if
51:   end if
52: end for
53:  $U_c := \text{Weeding}(\mathcal{K}, D)$ 

```

Algorithm 3, Part 2 of 2:  $\text{FastEvol}(\mathcal{K}, \mathcal{A}_+, \mathcal{A}_-)$

## A.2 Conditional Choice for Control Flow Graphs

One way of extending the notion of a control flow graph to also model processes in which deterministic choices are made based on service-specific conditions is as follows.<sup>1</sup> Given a control flow graph  $G_{cf} = (\mathbf{P}, \mathbf{T}, \mathbf{F}, M_0, fu)$ , let  $\mathbf{F}_{split} = \{(p, t) \mid p \text{ a split place and } t \in p\bullet\}$  be the set of all output edges of split places in  $G_{cf}$ . Let  $\mathcal{L}$  be a formal condition expression language where a condition  $\gamma \in \mathcal{L}$  is understood as a Boolean-valued function (i.e., it is either true or false). First, a partial mapping

$$cond: \mathbf{F}_{split} \rightarrow \mathcal{L}$$

is introduced that assigns a condition  $\gamma \in \mathcal{L}$  to a pair  $(p_{split}, t) \in \mathbf{F}_{split}$  (i.e., only an outgoing edge of split place can have a condition assigned). Notice that *cond* is partial; hence, a condition is possibly assigned but not necessarily. Conditioning transition enabling merely on the evaluation of conditions is therefore not sufficient to ensure that at most one output transition  $t$  of a split place  $p_{split}$  becomes enabled because (i) the edge  $(p_{split}, t)$  might not be associated with a condition and (ii) there can be multiple edges  $(p_{split}, t_i)$  whose associated condition is true. Consequently, additional means are required that enforce this property. There are two ways to achieve this:

1. For every split place  $p_{split} \in \mathbf{P}$  the conditions possibly assigned to its outgoing edges are required to be mutually exclusively true and every split place has at most one outgoing edge that does not has a condition assigned. Then, a transition  $t \in \mathbf{T}$  is enabled iff **Item (1)** and **Item (2)** of **Definition 4.13** hold and, in addition,

$$\forall p_{split} \in \mathbf{P}: p_{split} \in \bullet t \text{ and } \begin{cases} cond(p_{split}, t) = \gamma & \text{implies } \gamma \text{ is true} \\ cond(p_{split}, t) = \text{undefined} & \text{implies} \\ \forall t' \in p_{split}\bullet \text{ and } t \neq t': cond(p_{split}, t') \text{ is false.} \end{cases}$$

This rule says that if  $t$  is the output transition of a split place  $p_{split}$  then  $t$  is enabled either if the condition on the edge  $(p_{split}, t)$  is the one that is true among all the conditions on outgoing edges of  $p_{split}$ , or if there is no condition assigned to  $(p_{split}, t)$  then the conditions on all other outgoing edges of  $p_{split}$  are false.

2. For each split place  $p_{split} \in \mathbf{P}$  a priority mapping  $prty_{p_{split}}$  is introduced that defines a precedence order on its output transitions. Formally,

$$prty_{p_{split}}: \{t \mid t \in p_{split}\bullet\} \rightarrow \{1, \dots, |p_{split}\bullet|\}$$

such that  $prty_{p_{split}}$  is a bijection and the value 1 is interpreted with highest and  $|p_{split}\bullet|$  with lowest priority. The priority is employed by the transition-enabling rule such that it uniquely determines one transition out of the output transitions of  $p_{split}$  in cases where the condition on more than one outgoing edge of  $p_{split}$  is true or where more than one outgoing edge does not has a condition associated.

<sup>1</sup>There are also alternative ways to achieve the same based on coloured PNs.

Formally, a transition  $t \in \mathbf{T}$  is enabled iff **Item (1)** and **Item (2)** of **Definition 4.13** hold and, in addition,

$$\forall p_{\text{split}} \in \mathbf{P}: p_{\text{split}} \in \bullet t \text{ and} \\ \text{cond}(p_{\text{split}}, t) \text{ is undefined or true implies } t = \arg \min_{t' \in p_{\text{split}} \bullet} \text{prty}_{p_{\text{split}}}(t')$$

where  $\arg \min$  denotes the *argument of the minimum*; that is, in this case, the transition  $t' \in p_{\text{split}} \bullet$  for which  $\text{prty}_{p_{\text{split}}}(t')$  is minimal.

### A.3 Properties of Read and Update Operations

In the following, we explain the commutativity and set-preservation results given in **Table 6.1** for those combinations which do not commute and are not set-preserving in general. Let  $\mathbf{S}$  be the initial set of syntactic instances in an OWL knowledge base  $\mathcal{W}$ , let  $\mathbf{S}'$  be the resulting set after any combination of two read, add, or delete operations has been applied to  $\mathcal{W}$ , and  $\psi$  a OWL syntactic instance. The return value of an operation is indicated as follows.  $a(\psi) = \mathbf{t}$ ,  $a(\psi) = \mathbf{f}$  shall denote that adding  $\psi$  returned `true` or `false`, respectively. Indices 1, 2 are used if necessary to distinguish among different invocations of the same operation.

It is plainly apparent that the read/read combination is commutative and set-preserving. The return value is the same regardless of the execution order and neither of the reads causes a change of  $\mathcal{W}$  ( $\mathbf{S} = \mathbf{S}'$ ). Consequently, it does not matter which read is executed first.

For all other combinations two cases need to be considered:  $\psi \in \mathbf{S}$  and  $\psi \notin \mathbf{S}$ . Depending on these cases and the actual combination either add and/or delete may be futile or the read may return `null` as shown in the following:

Read/Add	$\psi \in \mathbf{S}$ :	$r(\psi) = \psi,$	$a(\psi) = \mathbf{f}$	$\mathbf{S}' = \mathbf{S}$
		$a(\psi) = \mathbf{f},$	$r(\psi) = \psi$	$\mathbf{S}' = \mathbf{S}$
	$\psi \notin \mathbf{S}$ :	$r(\psi) = \text{null},$	$a(\psi) = \mathbf{t}$	$\mathbf{S}' = \mathbf{S} \cup \psi$
		$a(\psi) = \mathbf{t},$	$r(\psi) = \psi$	$\mathbf{S}' = \mathbf{S} \cup \psi$
Read/Delete	$\psi \in \mathbf{S}$ :	$r(\psi) = \psi,$	$d(\psi) = \mathbf{t}$	$\mathbf{S}' = \mathbf{S} \setminus \psi$
		$d(\psi) = \mathbf{t},$	$r(\psi) = \text{null}$	$\mathbf{S}' = \mathbf{S} \setminus \psi$
	$\psi \notin \mathbf{S}$ :	$r(\psi) = \text{null},$	$d(\psi) = \mathbf{f}$	$\mathbf{S}' = \mathbf{S}$
		$d(\psi) = \mathbf{f},$	$r(\psi) = \text{null}$	$\mathbf{S}' = \mathbf{S}$
Read/Delete	$\psi \in \mathbf{S}$ :	$r(\psi) = \psi,$	$d(\psi) = \mathbf{t}$	$\mathbf{S}' = \mathbf{S} \setminus \psi$
		$d(\psi) = \mathbf{t},$	$r(\psi) = \text{null}$	$\mathbf{S}' = \mathbf{S} \setminus \psi$
	$\psi \notin \mathbf{S}$ :	$r(\psi) = \text{null},$	$d(\psi) = \mathbf{f}$	$\mathbf{S}' = \mathbf{S}$
		$d(\psi) = \mathbf{f},$	$r(\psi) = \text{null}$	$\mathbf{S}' = \mathbf{S}$

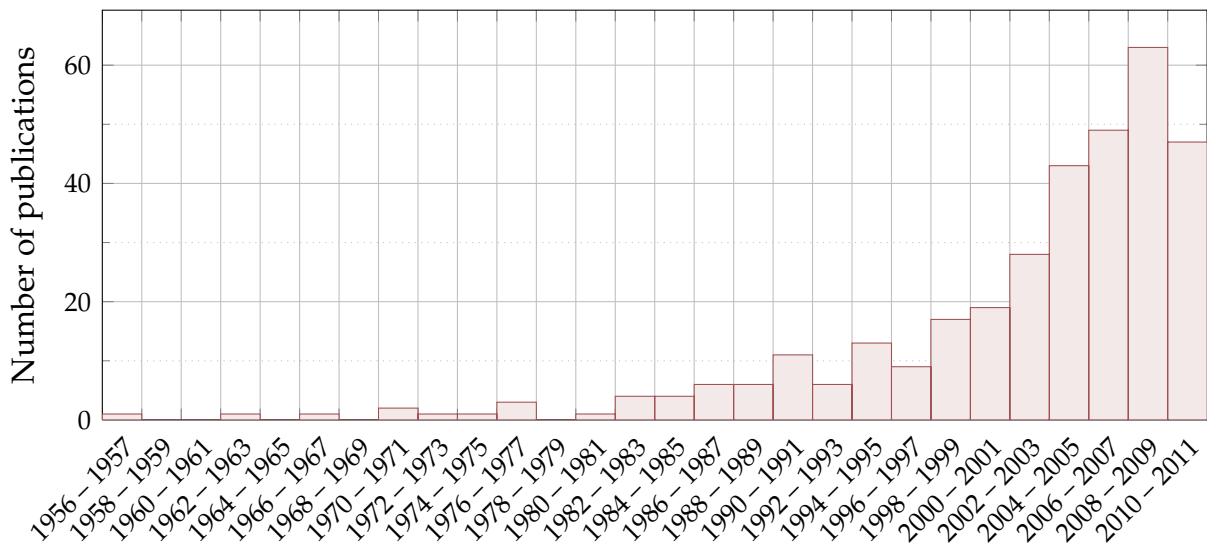
Add/Add	$\psi \in \mathbf{S}$ :	$a_1(\psi) = \mathbf{f},$	$a_2(\psi) = \mathbf{f}$	$\mathbf{S}' = \mathbf{S}$
		$a_2(\psi) = \mathbf{f},$	$a_1(\psi) = \mathbf{f}$	$\mathbf{S}' = \mathbf{S}$
	$\psi \notin \mathbf{S}$ :	$a_1(\psi) = \mathbf{t},$	$a_2(\psi) = \mathbf{f}$	$\mathbf{S}' = \mathbf{S} \cup \psi$
		$a_2(\psi) = \mathbf{t},$	$a_1(\psi) = \mathbf{f}$	$\mathbf{S}' = \mathbf{S} \cup \psi$
Delete/Delete	$\psi \in \mathbf{S}$ :	$d_1(\psi) = \mathbf{t},$	$d_2(\psi) = \mathbf{f}$	$\mathbf{S}' = \mathbf{S} \setminus \psi$
		$d_2(\psi) = \mathbf{t},$	$d_1(\psi) = \mathbf{f}$	$\mathbf{S}' = \mathbf{S} \setminus \psi$
	$\psi \notin \mathbf{S}$ :	$d_1(\psi) = \mathbf{f},$	$d_2(\psi) = \mathbf{f}$	$\mathbf{S}' = \mathbf{S}$
		$d_2(\psi) = \mathbf{f},$	$d_1(\psi) = \mathbf{f}$	$\mathbf{S}' = \mathbf{S}$
Add/Delete	$\psi \in \mathbf{S}$ :	$a(\psi) = \mathbf{f},$	$d(\psi) = \mathbf{t}$	$\mathbf{S}' = \mathbf{S} \setminus \psi$
		$d(\psi) = \mathbf{t},$	$a(\psi) = \mathbf{t}$	$\mathbf{S}' = \mathbf{S}$
	$\psi \notin \mathbf{S}$ :	$a(\psi) = \mathbf{t},$	$d(\psi) = \mathbf{t}$	$\mathbf{S}' = \mathbf{S}$
		$d(\psi) = \mathbf{f},$	$a(\psi) = \mathbf{t}$	$\mathbf{S}' = \mathbf{S} \cup \psi$

Clearly, these combinations do neither commute nor are they set-preserving in general. However, some of them *state-dependently* commute [VHBS98], meaning that they commute depending on the initial state; that is, depending on whether  $\psi \in \mathbf{S}$  or not. Read/add and add/add commute for  $\psi \in \mathbf{S}$  (but not for  $\psi \notin \mathbf{S}$ ); if the add operation is futile because the syntactic instance  $\psi$  is in  $\mathcal{W}$  anyway already. Analogously, read/delete and delete/delete commute for  $\psi \notin \mathbf{S}$ ; if the delete operation is futile because  $\psi$  is not in  $\mathcal{W}$ .

This raises the question whether it is worth the extra effort of optimizing for state-dependent commutativity. The main motivation is that if the change set of concurrent transactions are additionally checked whether they contain such combinations then the number of conflicts that lead to aborts might be reduced. It seems to be evident that the decision pro or against spending the effort depends on the workloads. More specifically, it depends on the absolute occurrence frequency of state-dependently commuting combinations in a workload and, more importantly, on the relative frequency of cases where a transaction would conflict under “standard” commutativity (1) whereas it would not conflict under state-dependent commutativity (2). The higher the ratio between (1) and (2) the more worthwhile the additional effort becomes. However, we believe that optimizing for state-dependent commutativity is forlorn hope because cases of state-dependent commuting operations are rare in most workloads. What is more, considering applications that are “conscious”, meaning that they try to avoid futile read, add, and delete operations anyway (since it is natural to avoid doing futile things), the frequency of state-commuting cases becomes fairly small.

# Bibliography

Publication year histogram of this bibliography.



- [Aal98] Wil M.P. van der Aalst. The Application of Petri Nets to Workflow Management. *Journal of Circuits, Systems, and Computers*, 8(1):21–66, 1998.
- [ABFS97] Gustavo Alonso, Stephen Blott, Armin Feßler, and Hans-Jörg Schek. Correctness and Parallelism in Composite Systems. In *Proceedings of the 16th Symposium on Principles of Database Systems (PODS)*, pages 197–208, New York, NY, USA, 1997. ACM.
- [ABHM03] C. Areces, P. Blackburn, B. M. Hernandez, and M. Marx. Handling Boolean ABoxes. In *In Proceedings of the 2003 International Workshop on Description Logics (DL-2003)*, 2003.
- [ABM04] Serge Abiteboul, Omar Benjelloun, and Tova Milo. Positive active xml. In *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, PODS '04*, pages 35–45, New York, NY, USA, 2004. ACM.
- [ACMS08] Vikas Agarwal, Girish Chafle, Sumit Mittal, and Biplav Srivastava. Understanding Approaches for Web Service Composition and Execution. In *Proceedings of the 1st Bangalore Annual Compute Conference*, pages 11–18, New York, NY, USA, 2008. ACM.
- [Adr00] Carlos Areces and Maarten de Rijke. From Description to Hybrid Logics, and Back. In Frank Wolter, Heinrich Wansing, Maarten de Rijke, and Michael Zakharyashev, editors, *Advances in Modal Logic*, pages 17–36. World Scientific, 2000.

- [Ady99] Atul Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1999.
- [AH02] Wil M.P. van der Aalst and Kees M. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT Press, 2002.
- [AHAE07] Michael Adams, Arthur ter Hofstede, Wil van der Aalst, and David Edmond. Dynamic, Extensible and Context-Aware Exception Handling for Workflows. In Robert Meersman and Zahir Tari, editors, *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS*, volume 4803 of *LNCS*, pages 95–112. Springer, 2007.
- [AHW03] Wil M.P. van der Aalst, Arthur H.M. ter Hofstede, and Mathias Weske. Business process management: A survey. In Mathias Weske, editor, *Business Process Management*, volume 2678 of *LNCS*. Springer, 2003.
- [AJPS10] Lina Al-Jadir, Christine Parent, and Stefano Spaccapietra. Reasoning with Large Ontologies Stored in Relational Databases: The OntoMinD Approach. *Data & Knowledge Engineering*, 69(11):1158–1180, 2010.
- [ALRL04] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11–33, 2004.
- [AMG<sup>+</sup>95] Gustavo Alonso, C. Mohan, Roger Günthör, Divyakant Agrawal, Amr El Abbadi, and Mohan Kamath. Exotica/FMQM: A Persistent Message-Based Architecture for Distributed Workflow Management. In *Proceedings of the IFIP Working Conference on Information Systems for Decentralized Organization*, pages 1–18, Trondheim, Norway, 1995.
- [AMM07] Eyhab Al-Masri and Qusay H. Mahmoud. QoS-based Discovery and Ranking of Web Services. In *Proceedings of 16th International Conference on Computer Communications and Networks (ICCCN)*, pages 529–534, 2007.
- [AS11] Wil van der Aalst and Christian Stahl. *Modeling Business Processes: A Petri Net-Oriented Approach*. MIT Press, May 2011.
- [AWG05] Wil M.P. van der Aalst, Mathias Weske, and Dolf Grünbauer. Case handling: a new paradigm for business process support. *Data & Knowledge Engineering*, 53(2):129–162, 2005.
- [BA06] Moti Ben-Ari. *Principles of concurrent and distributed programming*. Addison-Wesley, 2nd edition, 2006.
- [BB10] Shalini Batra and Seema Bawa. Review of Machine Learning Approaches to Semantic Web Service Discovery. *Journal of Advances in Information Technology*, 1(3), 2010.
- [BBD<sup>+</sup>02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, PODS '02*, pages 1–16, New York, NY, USA, 2002. ACM.



- [BBG<sup>+</sup>95] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O’Neil, and Patrick E. O’Neil. A Critique of ANSI SQL Isolation Levels. In Michael J. Carey and Donovan A. Schneider, editors, *SIGMOD Conference*, pages 1–10. ACM Press, 1995.
- [BBL08] Franz Baader, Sebastian Brandt, and Carsten Lutz. Pushing the EL Envelope Further. In *Proceedings of the OWLED 2008 DC Workshop on OWL: Experiences and Directions*, Washington, DC, USA, 2008.
- [BBL11] David Beckett and Tim Berners-Lee. *Turtle - Terse RDF Triple Language*. W3C Team Submission, March 2011. Available at <http://www.w3.org/TeamSubmission/turtle/>.
- [BCF<sup>+</sup>06] Walter Binder, Ion Constantinescu, Boi Faltings, Klaus Haller, and Can Türker. A Multiagent System for the Reliable Execution of Automatically Composed Ad-hoc Processes. *Autonomous Agents and Multi-Agent Systems*, 12:219–237, 2006.
- [BCI09] Antonio Brogi, Sara Corfini, and Stefano Iardella. From OWL-S Descriptions to Petri Nets. In Elisabetta Nitto and Matei Ripeanu, editors, *Service-Oriented Computing - ICSOC 2007 Workshops*, pages 427–438. Springer, Berlin, Heidelberg, 2009.
- [BCM<sup>+</sup>07] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2nd edition edition, 2007.
- [BD97] Thomas Bauer and Peter Dadam. A Distributed Execution Environment for Large-Scale Workflow Management Systems with Subnets and Server Migration. In *Proceedings of the Int’l Conference on Cooperative Information Systems (CoopIS)*, pages 99–108. IEEE, June 1997.
- [BD00] Thomas Bauer and Peter Dadam. Efficient Distributed Workflow Management Based on Variable Server Assignments. In *Proceeding 12th Int’l Conference on Advanced Information Systems Engineering (CAiSE)*, number 1789 in LNCS, pages 94–109. Springer, June 2000.
- [BDS08] D. Benslimane, S. Dustdar, and A. Sheth. Services Mashups: The New Generation of Web Applications. *Internet Computing, IEEE*, 12(5):13–15, sep. 2008.
- [BFL<sup>+</sup>07] Eric Bouillet, Mark Feblowitz, Zhen Liu, Anand Ranganathan, Anton Riabov, and Fan Ye. A semantics-based middleware for utilizing heterogeneous sensor networks. In James Aspnes, Christian Scheideler, Anish Arora, and Samuel Madden, editors, *Distributed Computing in Sensor Systems*, volume 4549 of LNCS, pages 174–188. Springer, 2007.
- [BFL<sup>+</sup>08] Eric Bouillet, Mark Feblowitz, Zhen Liu, Anand Ranganathan, and Anton Riabov. A Tag-based Approach for the Design and Composition of Information Processing Applications. In *Proceedings of the 23rd ACM SIGPLAN*

- conference on Object-oriented Programming Systems Languages and Applications, OOPSLA '08*, pages 585–602, New York, NY, USA, 2008. ACM.
- [BG01] Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1-2):5–33, 2001.
- [BH91] Franz Baader and Philipp Hanschke. A Schema for Integrating Concrete Domains into Concept Languages. In *Proceedings of the 12<sup>th</sup> International Joint Conference on Artificial Intelligence (IJCAI)*, pages 452–457, Sydney, 1991. A long version appeared in the Technical Report RR-91-10, DFKI, Germany, 1991.
- [BH93] Gerhard Brewka and Joachim Hertzberg. How to Do Things with Worlds: on Formalizing Actions and Plans. *Journal of Logic and Computation*, 3(5):517–532, 1993.
- [BH08] Philip A. Bernstein and Laura M. Haas. Information integration in the enterprise. *Communications of the ACM*, 51:72–79, September 2008.
- [BHB<sup>+</sup>10] Sören Balko, Arthur H.M. ter Hofstede, Alistair P. Barros, Marcello La Rosa, and Michael J. Adams. Business process extensibility. *Enterprise Modelling and Information Systems Architectures Journal*, July 2010.
- [BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
- [BHM<sup>+</sup>04] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferrisand, and David Orchard, editors. *Web Services Architecture*. W3C Working Group Note, February 2004. Available at <http://www.w3.org/TR/ws-arch/>.
- [BHT05] Sean Bechhofer, Ian Horrocks, and Daniele Turi. The OWL Instance Store: System Description. In Robert Nieuwenhuis, editor, *Automated Deduction – CADE-20*, volume 3632 of *LNCS*, pages 177–181. Springer, 2005.
- [BKM99] Franz Baader, Ralf Kusters, and Ralf Molitor. Computing Least Common Subsumers in Description Logics with Existential Restrictions. In *IJCAI'99: Proceedings of the 16th international joint Conference on Artificial Intelligence*, pages 96–101, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.
- [BL85] Ronald J. Brachman and Hector J. Levesque, editors. *Readings in Knowledge Representation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1985.
- [BLF<sup>+</sup>06] Federico Bergenti, Heimo Laamanen, Alberto Fernandez, Sascha Ossowski, Heikki Helin, Oliver Keller, Matthias Klusch, Heimo Laamanen, Antonio Lopes, Sascha Ossowski, Heiko Schuldt, and Michael Schumacher. Context-aware Service Coordination for Mobile e-Health Applications. In *European Conference on eHealth (ECEH06)*, 2006.
- [BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, 2001.

- [BLL10] Franz Baader, Marcel Lippmann, and Hongkai Liu. Using Causal Relationships to Deal with the Ramification Problem in Action Formalisms Based on Description Logics. In Christian G. Fermüller and Andrei Voronkov, editors, *LPAR-17*, volume 6397 of *LNCS (subline Advanced Research in Computing and Software Science)*, pages 82–96, Yogyakarta, Indonesia, October 2010. Springer.
- [BLM<sup>+</sup>05] Franz Baader, Carsten Lutz, Maja Miličić, Ulrike Sattler, and Frank Wolter. Integrating description logics and action formalisms: first results. In *AAAI-20*, pages 572–577. AAAI Press, 2005.
- [BML<sup>+</sup>05] Franz Baader, Maja Miličić, Carsten Lutz, Ulrike Sattler, and Frank Wolter. Integrating Description Logics and Action Formalisms for Reasoning about Web Services. Technical Report LTCS-05-02, Chair of Automata Theory, Institute of Theoretical Computer Science, Dresden University of Technology, Germany, 2005. Long version of [BLM<sup>+</sup>05].
- [BMR94] Daniel Barbara, Sharad Mehrotra, and Marek Rusinkiewicz. INCAS: A Computation Model for Dynamic Workflows in Autonomous Distributed Environments. Technical report, Matsushita Information Technology Laboratory, 1994.
- [BNDK04] Menkes van den Briel, Romeo Sanchez Nigenda, Minh Binh Do, and Subbarao Kambhampati. Effective Approaches for Partial Satisfaction (Over-Subscription) Planning. In Deborah L. McGuinness and George Ferguson, editors, *AAAI*, pages 562–569. AAAI Press / The MIT Press, 2004.
- [BOI09] Ayse B. Bener, Volkan Ozadali, and Erdem Savas Ilhan. Semantic match-maker with precondition and effect matching using SWRL. *Expert Systems with Applications*, 36(5):9371–9377, 2009.
- [BR04] Thomas Bauer and Manfred Reichert. Dynamic Change of Server Assignments in Distributed Workflow Management Systems. In *Proceedings of the 6<sup>th</sup> Int'l Conference on Enterprise Information Systems (ICEIS)*, pages 91–98, Porto, Portugal, 2004.
- [BS07] Gert Brettlecker and Heiko Schuldt. The OSIRIS-SE (stream-enabled) infrastructure for reliable data stream management on mobile devices. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data, SIGMOD '07*, pages 1097–1099, New York, NY, USA, 2007. ACM.
- [BS11] Gert Brettlecker and Heiko Schuldt. Reliable distributed data stream management in mobile environments. *Information Systems*, 36(3):618–643, 2011. Special Issue on WISE 2009 - Web Information Systems Engineering.
- [Byl94] Tom Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1-2):165–204, 1994.
- [CFB04] Ion Constantinescu, Boi Faltings, and Walter Binder. Large Scale, Type-Compatible Service Composition. In *Proceedings of the International Confer-*

- ence on Web Services (ICWS), pages 506–513, San Diego, California, USA, June 2004. IEEE.
- [CGL98] Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. On the Decidability of Query Containment under Constraints. In *PODS*, pages 149–158, Seattle, Washington, June 1998. ACM Press.
- [CGL<sup>+</sup>07] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Tractable Reasoning and Efficient Query Answering in Description Logics: The *DL-Lite* Family. *Journal of Automated Reasoning*, 39:385–429, 2007.
- [CGL11a] Christian Cachin, Rachid Guerraoui, and Rodrigues Luís. *Introduction to Reliable and Secure Distributed Programming*. Springer, 2nd edition, 2011.
- [CGL<sup>+</sup>11b] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, Mariano Rodriguez-Muro, Riccardo Rosati, Marco Ruzzi, and Domenico Fabio Savo. The MASTRO System for Ontology-based Data Access. *Semantic Web*, 2:43–53, January 2011.
- [CGLN01] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Daniele Nardi. Reasoning in Expressive Description Logics. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, chapter 23, pages 1581–1634. Elsevier Science Publishers, 2001.
- [CGP01] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 2001.
- [Che76] Peter Pin-Shan Chen. The Entity-Relationship Model—Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.
- [CKNZ10] Diego Calvanese, Evgeny Kharlamov, Werner Nutt, and Dmitriy Zheleznyakov. Evolution of DL-Lite Knowledge Bases. In Peter Patel-Schneider, Yue Pan, Pascal Hitzler, Peter Mika, Lei Zhang, Jeff Pan, Ian Horrocks, and Birte Glimm, editors, *The Semantic Web – ISWC 2010*, volume 6496 of *LNCS*, pages 112–128. Springer, 2010.
- [CKO92] Bill Curtis, Marc I. Kellner, and Jim Over. Process modeling. *Commun. ACM*, 35:75–90, September 1992.
- [CM95] Vinay K. Chaudhri and John Mylopoulos. Efficient Algorithms and Performance Results for Multi-user Knowledge Bases. In *IJCAI'95*, pages 759–766, San Francisco, CA, USA, 1995. Morgan Kaufmann.
- [CMRW07] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana, editors. *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*. W3C Recommendation, June 2007. Available at <http://www.w3.org/TR/wsdl20/>.
- [Cod70] Edgar F. Codd. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM*, 13(6):377–387, 1970.
- [Cod90] Edgar F. Codd. *The relational model for database management: version 2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

- [CRF09] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. Serializable isolation for snapshot databases. *ACM Trans. Database Syst.*, 34(4):1–42, 2009.
- [CS06] Jorge Cardoso and Amit P. Sheth, editors. *Semantic Web Services, Processes and Applications*, volume 3 of *Semantic Web And Beyond Computing for Human Experience*. Springer, 2006.
- [CT91] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for asynchronous systems. In *Proceedings of the tenth annual ACM symposium on Principles of distributed computing*, PODC '91, pages 325–340, New York, NY, USA, 1991. ACM.
- [CTA02] Wei Chen, Sam Toueg, and Marcos Kawazoe Aguilera. On the Quality of Service of Failure Detectors. *IEEE Transactions on Computers*, 51:561–580, 2002.
- [CWT08] Bin Cheng, Xingang Wang, and Weiqin Tong. Ontology-Based Semantic Method for Service Modeling in Grid. In Jian Cao, Minglu Li, Min-You Wu, and Jinjun Chen, editors, *Network and Parallel Computing*, volume 5245 of *LNCS*, pages 339–348. Springer, 2008.
- [DBCO07] María R-Moreno Dolores, Daniel Borrajo, Amedeo Cesta, and Angelo Oddi. Integrating Planning and Scheduling in Workflow Domains. *Expert Systems with Applications*, 33(2):389–406, 2007.
- [DDS87] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34:77–97, January 1987.
- [DLS88] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35:288–323, April 1988.
- [DS05] M. Duerst and M. Suignard. *Internationalized Resource Identifiers (IRIs) – Request for Comments: 3987*. The Internet Society, January 2005. Available at <http://tools.ietf.org/html/rfc3987>.
- [Dvt05] Marlon Dumas, Wil M.P. van der Aalst, and Arthur H.M. ter Hofstede, editors. *Process-Aware Information Systems: Bridging People and Software through Process Technology*. Wiley-Interscience, Hoboken, NJ, 2005.
- [EF95] Heinz-Dieter Ebbinghaus and Jörg Flum. *Finite model theory*. Perspectives in mathematical logic. Springer, 1995.
- [EGHB11] Jiménez-Ruiz Ernesto, Bernardo Cuenca Grau, Ian Horrocks, and Rafael Berlanga. Supporting concurrent ontology development: Framework, algorithms and tool. *Data & Knowledge Engineering*, 70(1):146–164, 2011.
- [EGLT76] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.
- [EKR95] Clarence Ellis, Karim Keddara, and Grzegorz Rozenberg. Dynamic Change Within Workflow Systems. In *Proceedings of Conference on Organizational Computing Systems (COOCS)*, pages 10–21, New York, NY, USA, 1995. ACM.

- [ES07] Jérôme Euzenat and Pavel Shvaiko. *Ontology Matching*. Springer, Berlin, 2007.
- [FF06] Hugo M. Ferreira and Diogo R. Ferreira. An Integrated Life Cycle for Workflow Management Based on Learning and Planning. *Journal on Cooperative Information Systems*, 15(4):485–505, 2006.
- [FFM<sup>+</sup>10] Gerhard Friedrich, Mariagrazia Fugini, Enrico Mussi, Barbara Pernici, and Gaston Tagni. Exception Handling for Repair in Service-Based Processes. *IEEE Transactions on Software Engineering*, 36(2):198–215, 2010.
- [FFST11] Dieter Fensel, Federico Michele Facca, Elena Simperl, and Ioan Toma. *Semantic Web Services*. Springer, Berlin, Heidelberg, 2011.
- [FH10] Jun Fang and Zhisheng Huang. Reasoning with inconsistent ontologies. *Tsinghua Science & Technology*, 15(6):687–691, 2010.
- [FHH<sup>+</sup>01] Dieter Fensel, Frank van Harmelen, Ian Horrocks, Deborah L. McGuinness, and Peter F. Patel-Schneider. OIL: An Ontology Infrastructure for the Semantic Web. *IEEE Intelligent Systems*, 16(2):38–45, 2001.
- [FL03] Maria Fox and Derek Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20(1):61–124, 2003.
- [FL07] Joel Farrell and Holger Lausen, editors. *Semantic Annotations for WSDL and XML Schema*. W3C Recommendation, August 2007. Available at <http://www.w3.org/TR/sawSDL/>.
- [FLO<sup>+</sup>05] Alan Fekete, Dimitrios Liarokapis, Elizabeth O’Neil, Patrick O’Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30:492–528, June 2005.
- [FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32:374–382, April 1985.
- [FMK<sup>+</sup>08] Giorgos Flouris, Dimitris Manakanatas, Haridimos Kondylakis, Dimitris Plexousakis, and Grigoris Antoniou. Ontology change: classification and survey. *The Knowledge Engineering Review*, 23(02):117–152, 2008.
- [FN71] Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189–208, 1971.
- [FOO04] Alan Fekete, Elizabeth O’Neil, and Patrick O’Neil. A Read-Only Transaction Anomaly under Snapshot Isolation. *SIGMOD Record*, 33:12–14, September 2004.
- [Fra] Franz Incorporation. AllegroGraph RDFStore Web 3.0’s Database. <http://www.franz.com/agraph/allegrograph/>. Visited on July 27, 2012.
- [FUV83] Ronald Fagin, Jeffrey D. Ullman, and Moshe Y. Vardi. On the semantics of updates in databases. In *Proceedings of the 2nd ACM SIGACT-SIGMOD*

- symposium on Principles of database systems*, PODS '83, pages 352–365, New York, NY, USA, 1983. ACM.
- [FWL02] Dieter Fensel, Wolfgang Wahlster, and Henry Lieberman, editors. *Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential*. MIT Press, Cambridge, MA, USA, 2002.
- [Gef00] Héctor Geffner. *Functional STRIPS: A More Flexible Language for Planning and Problem Solving*, pages 187–209. Kluwer Academic Publishers, Norwell, MA, USA, 2000.
- [Gef11] Hector Geffner. Planning with Incomplete Information. In Ron van der Meyden and Jan-Georg Smaus, editors, *Model Checking and Artificial Intelligence*, volume 6572 of *LNCS*, pages 1–11. Springer, 2011.
- [GGK<sup>+</sup>91] Hector Garcia-Molina, Dieter Gawlick, Johannes Klein, Karl Kleissner, and Kenneth Salem. Modeling Long-running Activities as Nested Sagas. *Data Engineering*, 14:14–18, March 1991.
- [GHKS10] Bernardo Cuenca Grau, Christian Halaschek-Wiener, Yevgeny Kazakov, and Boontawee Suntisrivaraporn. Incremental Classification of Description Logics Ontologies. *Journal of Automated Reasoning*, 44:337–369, 2010.
- [GHVD03] Benjamin N. Grosz, Ian Horrocks, Raphael Volz, and Stefan Decker. Description Logic Programs: Combining Logic Programs with Description Logic. In *Proceedings of the 12th International World Wide Web Conference*, pages 48–57, Budapest, Hungary, 2003.
- [Gin86] Matthew L. Ginsberg. Counterfactuals. *Artificial Intelligence*, 30:35–80, October 1986.
- [GL06] Alfonso Gerevini and Derek Long. Plan Constraints and Preferences in PDDL3. In *ICAPS Workshop on Soft Constraints and Preferences in Planning*, pages 7–13, 2006.
- [GLHS08] Birte Glimm, Carsten Lutz, Ian Horrocks, and Ulrike Sattler. Answering conjunctive queries in the *SHIQ* description logic. *Journal of Artificial Intelligence Research*, 31:150–197, 2008.
- [GLPR09] Giuseppe De Giacomo, Maurizio Lenzerini, Antonella Poggi, and Riccardo Rosati. On Instance-level Update and Erasure in Description Logic Ontologies. *Journal of Logic and Computation*, 19:745–770, October 2009.
- [GMM<sup>+</sup>05] Michal Gajewski, Mariusz Momotko, Harald Meyer, Hilmar Schuschel, and Mathias Weske. Dynamic failure recovery of generated workflows. In *Sixteenth International Workshop on Database and Expert Systems Applications*, pages 982–986, 2005.
- [GMP06] Stephan Grimm, Boris Motik, and Chris Preist. Matching Semantic Service Descriptions with Local Closed-World Reasoning. In York Sure and John Domingue, editors, *The Semantic Web: Research and Applications*, volume 4011 of *LNCS*, pages 575–589. Springer, 2006.

- [GMS87] Hector Garcia-Molina and Kenneth Salem. Sagas. *SIGMOD Record*, 16:249–259, December 1987.
- [GNT04] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, Amsterdam, 2004.
- [GO10] Birte Glimm and Chimezie Ogbuji, editors. *SPARQL 1.1 Entailment Regimes*. W3C Working Draft, October 2010. Available at <http://www.w3.org/TR/sparql11-entailment/>.
- [GPH05] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics*, 3(2-3):158–182, 2005.
- [GR10] Birte Glimm and Sebastian Rudolph. Status QIO: Conjunctive Query Entailment is Decidable. In *Proceedings of the 12th International Conference on the Principles of Knowledge Representation and Reasoning (KR-10)*. AAAI Press/The MIT Press, 2010.
- [GRL<sup>+</sup>08] Levent Gurgen, Claudia Roncancio, Cyril Labbé, André Bottaro, and Vincent Olive. Sstreamware: a service oriented middleware for heterogeneous sensor data management. In *Proceedings of the 5th international conference on Pervasive services, ICPS '08*, pages 121–130, New York, NY, USA, 2008. ACM.
- [GRR<sup>+</sup>08] Karthik Gomadam, Ajith Ranabahu, Lakshmith Ramaswamy, Amit P. Sheth, and Kunal Verma. Mediatability: Estimating the Degree of Human Involvement in XML Schema Mediation. In *IEEE International Conference on Semantic Computing*, pages 394–401, August 2008.
- [GRS10] Karthik Gomadam, Ajith Ranabahu, and Amit Sheth, editors. *SA-REST: Semantic Annotation of Web Resources*. W3C Member Submission, April 2010. Available at <http://www.w3.org/Submission/SA-REST/>.
- [Gru93] Thomas R. Gruber. A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.
- [GS88] M. L. Ginsberg and D. E. Smith. Reasoning About Action I: A Possible Worlds Approach. *Artificial Intelligence*, 35:165–195, June 1988.
- [GT98] Andreas Geppert and Dimitrios Tombros. Event-based Distributed Workflow Execution with EVE. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 427–442, London, UK, 1998. Springer.
- [GTSS11] Georg Grossmann, Rajesh Thiagarajan, Michael Schrefl, and Markus Stumptner. Conceptual Modeling Approaches for Dynamic Web Service Composition. In Roland Kaschek and Lois Delcambre, editors, *The Evolution of Conceptual Modeling*, volume 6520 of *LNCS*, pages 180–204. Springer, 2011.
- [Har06] Frank van Harmelen. Where does it break? or: Why Semantic Web research is not just “Computer Science as usual” – Keynote at ESWC. <http://www.eswc2006.org/>



- [//www.eswc2006.org/keynote-frank-van-harmelen.pdf](http://www.eswc2006.org/keynote-frank-van-harmelen.pdf), 2006. Visited on July 27, 2012.
- [Hay04] Patrick Hayes, editor. *RDF Semantics*. W3C Recommendation, February 2004. Available at <http://www.w3.org/TR/rdf-mt/>.
- [HB09] Matthew Horridge and Sean Bechhofer. The OWL API: A Java API for Working with OWL 2 Ontologies. In Rinke Hoekstra and Peter F. Patel-Schneider, editors, *OWLED*, volume 529 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2009.
- [HBHP09] Jörg Hoffmann, Piergiorgio Bertoli, Malte Helmert, and Marco Pistore. Message-Based Web Service Composition, Integrity Constraints, and Planning under Uncertainty: A New Connection. *Journal of Artificial Intelligence Research*, 35:49–117, May 2009.
- [HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial Intelligence*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann.
- [HCZ10] Ruoyun Huang, Yixin Chen, and Weixiong Zhang. A Novel Transition Based Encoding Scheme for Planning as Satisfiability. In *Proceedings of the 24th Conference on Artificial Intelligence (AAAI)*, pages 89–94, Atlanta, Georgia, USA, 2010. AAAI Press.
- [HDR08] Malte Helmert, Minh Do, and Ioannis Refanidis, editors. *IPC Deterministic Competition*, 2008. Available at <http://ipc.informatik.uni-freiburg.de/Results>.
- [Hel02] Malte Helmert. Decidability and Undecidability Results for Planning with Numerical State Variables. In *Proceedings Workshop Planen und Konfigurieren (PuK)*, 2002.
- [Her96] Andreas Herzig. The PMA Revisited. In *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning*, pages 40–50. Morgan Kaufmann, November 1996.
- [Hew11] Carl Hewitt. Actor Model of Computation: Scalable Robust Information Systems. In *Inconsistency Robustness*, pages 16–28, Stanford, USA, 2011.
- [HHL99] Jeff Heflin, James Hendler, and Sean Luke. SHOE A Knowledge Representation Language for Internet Applications. Technical Report CS-TR-4078 (UMIACS TR-99-71), Dept. of Computer Science, University of Maryland, 1999.
- [HKGB09] Jörg Henss, Joachim Kleb, Stephan Grimm, and Jürgen Bock. A Database Backend for OWL. In Rinke Hoekstra and Peter F. Patel-Schneider, editors, *Proceedings of the 5th International Workshop on OWL: Experiences and Directions (OWLED)*, volume 529, 2009.
- [HKRK09] Mohamed Hamdy, Birgitta König-Ries, and Ulrich Küster. Non-functional Parameters as First Class Citizens in Service Description and Matchmaking – An Integrated Approach. In Elisabetta Di Nitto and Matei Ripeanu,

- editors, *Service-Oriented Computing - ICSOC 2007 Workshops*, volume 4907 of *LNCS*, pages 93–104. Springer, 2009.
- [HKS06] Ian Horrocks, Oliver Kutz, and Ulrike Sattler. The even more irresistible *SROTQ*. In *Proc. of the 10<sup>th</sup> Int. Conf. on Principles of Knowledge Representation and Reasoning (KR2006)*, pages 57–67. AAAI Press, 2006.
- [HLP08] Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter, editors. *Handbook of Knowledge Representation*. Foundations of Artificial Intelligence. Elsevier Science, 1st edition, 2008.
- [HM01a] Volker Haarslev and Ralf Möller. High Performance Reasoning with Very Large Knowledge Bases: A Practical Case Study. In Bernhard Nebel, editor, *IJCAI*, pages 161–168, Seattle, Washington, USA, August 2001. Morgan Kaufmann.
- [HM01b] Volker Haarslev and Ralf Müller. RACER System Description. In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *Automated Reasoning*, volume 2083 of *LNCS*, pages 701–705. Springer, 2001.
- [HM05] Peter Haase and Boris Motik. A mapping system for the integration of OWL-DL ontologies. In *Proceedings of the first international workshop on Interoperability of heterogeneous information systems, IHIS '05*, pages 9–16, New York, NY, USA, 2005. ACM.
- [HM08] Volker Haarslev and Ralf Möller. On the Scalability of Description Logic Instance Retrieval. *Journal of Automated Reasoning*, 41(2):99–142, 2008.
- [HMV<sup>+</sup>09] Ourania Hatzi, Georgios Meditskos, Dimitris Vrakas, Nick Bassiliades, Dimosthenis Anagnostopoulos, and Ioannis Vlahavas. PORSCE II: Using planning for semantic web service composition. In *In Proceedings of the International Competition on Knowledge Engineering for Planning and Scheduling in Conjunction with ICAPS*, pages 38–45, 2009.
- [HNSS90] Bernhard Hollunder, Werner Nutt, and Manfred Schmidt-Schauß. Subsumption Algorithms for Concept Description Languages. In *Proceedings of the 9th European Conference on Artificial Intelligence*, pages 348–353, Stockholm, Sweden, 1990.
- [Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, New York, 1985.
- [Hor98] Ian Horrocks. Using an Expressive Description Logic: FaCT or Fiction? In *Principles of Knowledge Representation and Reasoning*, pages 636–649, 1998.
- [HPB<sup>+</sup>04] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosz, and Mike Dean. *SWRL: A Semantic Web Rule Language Combining OWL and RuleML*. W3C Member Submission, May 2004. Available at <http://www.w3.org/Submission/SWRL/>.
- [HPS06] Christian Halaschek-Wiener, Bijan Parsia, and Evren Sirin. Description Logic Reasoning with Syntactic Updates. In Robert Meersman and Zahir Tari, editors, *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA,*

- GADA, and ODBASE*, volume 4275 of *LNCS*, pages 722–737. Springer, 2006.
- [HPS09] Matthew Horridge and Peter F. Patel-Schneider. *OWL 2 Web Ontology Language Manchester Syntax*. W3C Working Group Note, October 2009. Available at <http://www.w3.org/TR/owl2-manchester-syntax/>.
- [HPSH03] Ian Horrocks, Peter F. Patel-Schneider, and Frank van Harmelen. From SHIQ and RDF to OWL: The Making of a Web Ontology Language. *Journal of Web Semantics*, 1:2003, 2003.
- [HPSK06] Christian Halaschek-Wiener, Bijan Parsia, Evren Sirin, and Aditya Kalyanpur. Description Logic Reasoning for Dynamic ABoxes. In Bijan Parsia, Ulrike Sattler, and David Toman, editors, *Description Logics*, volume 189 of *CEUR Workshop Proceedings*, Windermere, Lake District, UK, 2006. CEUR-WS.org.
- [HR99] Andreas Herzig and Omar Rifi. Propositional belief base update and minimal change. *Artificial Intelligence*, 115:107–138, November 1999.
- [HRRD03] Arthur ter Hofstede, Manfred Reichert, Stefanie Rinderle, and Peter Dadam. ADEPT Workflow Management System. In Mathias Weske, editor, *Business Process Management*, volume 2678 of *LNCS*, pages 370–379. Springer, 2003.
- [HS01] Ian Horrocks and Ulrike Sattler. Ontology reasoning in the *SHOQ(D)* description logic. In *Proc. of the 17th Int. Joint Conf. on Artificial Intelligence (IJCAI 2001)*, pages 199–204. Morgan Kaufmann, 2001.
- [HS10] Steve Harris and Andy Seaborne, editors. *SPARQL 1.1 Query Language*. W3C Working Draft, October 2010. Available at <http://www.w3.org/TR/sparql11-query/>.
- [HST99] Ian Horrocks, Ulrike Sattler, and Stephan Tobies. Practical Reasoning for Expressive Description Logics. In Harald Ganzinger, David A. McAllester, and Andrei Voronkov, editors, *LPAR*, volume 1705 of *LNCS*, pages 161–180. Springer, 1999.
- [HT00] Ian Horrocks and Sergio Tessaris. A Conjunctive Query Language for Description Logic Aboxes. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 399–404. AAAI Press, 2000.
- [HT06] Matthew Horridge and Dmitry Tsarkov. Supporting Early Adoption of OWL 1.1 with Protégé-OWL and FaCT++. In Bernardo Cuenca Grau, Pascal Hitzler, Conor Shankey, and Evan Wallace, editors, *OWLED*, volume 216 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2006.
- [Int] International Health Terminology Standards Development Organisation. SNOMED CT: Systematized Nomenclature of Medicine-Clinical Terms. <http://www.ihtsdo.org/snomed-ct/>. Visited on July 27, 2012.

- [JB96] Stefan Jablonski and Christoph Bussler. *Workflow Management: Modeling Concepts, Architecture and Implementation*. International Thomson Computer Press, September 1996.
- [JE07] Diane Jordan and John Evdemon. *Web Services Business Process Execution Language Version 2.0*. OASIS Standard, April 2007. Available at <http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html>.
- [Jena] Jena Developer Team. Jena - Semantic Web Framework. <http://openjena.org>.
- [Jenb] Jena Developer Team. Jena Tuple Database. <http://incubator.apache.org/jena/documentation/tdb/>. Visited on July 27, 2012.
- [Jen87] Kurt Jensen. Coloured Petri nets. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Central Models and Their Properties*, volume 254 of LNCS, pages 248–299. Springer, 1987. 10.1007/BFb0046842.
- [JMS<sup>+</sup>99] Peter Jarvis, Jonathan Moore, Jussi Stader, Ann Macintosh, and Paul Chung. Exploiting AI Technologies to Realise Adaptive Workflow Systems. In *Proceedings of the AAAI Workshop on Agent-Based Systems in the Business Context*, 1999.
- [Jos07] Nicolai M. Josuttis. *SOA in Practice: The Art of Distributed System Design*. O'Reilly, 2007.
- [KACT97] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Heartbeat: A timeout-free failure detector for quiescent reliable communication. In Marios Mavronicolas and Philippas Tsigas, editors, *Distributed Algorithms*, volume 1320 of LNCS, pages 126–140. Springer, 1997.
- [Kal06] Aditya Kalyanpur. *Debugging and repair of OWL Ontologies*. PhD thesis, University of Maryland at College Park, College Park, MD, USA, 2006. AAI3222483.
- [Kaz08] Yevgeny Kazakov. RIQ and SROIQ Are Harder than SHOIQ. In Gerhard Brewka and Jérôme Lang, editors, *KR*, pages 274–284, Sydney, Australia, September 2008. AAAI Press.
- [KBR<sup>+</sup>05] Nickolas Kavantzias, David Burdett, Greg Ritzinger, Tony Fletcher, Yves Lafon, and Charlton Barreto, editors. *Web Services Choreography Description Language Version 1.0*. W3C Candidate Recommendation, November 2005. Available at <http://www.w3.org/TR/ws-cdl-10/>.
- [KC10] Mehmet Kuzu and Nihan Cicekli. Dynamic Planning Approach to Automated Web Service Composition. *Applied Intelligence*, pages 1–28, 2010.
- [KCP99] Peter D. Karp, Vinay K. Chaudhri, and Suzanne M. Paley. A Collaborative Environment for Authoring Large Knowledge Bases. *Intelligent Information Systems*, 13(3):155–194, 1999.
- [KFS09] Matthias Klusch, Benedikt Fries, and Katia Sycara. OWLS-MX: A hybrid Semantic Web service matchmaker for OWL-S services. *Web Semantics: Science, Services, and Agents on the World Wide Web*, 7(2):121–133, 2009.

- [KG05] Matthias Klusch and Andreas Gerber. Semantic Web Service Composition Planning with OWLS-XPlan. In *In Proceedings of the 1st Int. AAAI Fall Symposium on Agents and the Semantic Web*, pages 55–62, 2005.
- [KG09] Emil Keyder and Hector Geffner. Soft Goals Can Be Compiled Away. *Journal of Artificial Intelligence Research*, 36:547–556, September 2009.
- [KGV08] Jacek Kopecký, Karthik Gomadam, and Tomas Vitvar. hRESTS: An HTML Microformat for Describing RESTful Web Services. In *Proceedings of the 2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology - Volume 01*, pages 619–625, Washington, DC, USA, 2008. IEEE Computer Society.
- [KK07] Hyun-Sik Kim and In-Cheol Kim. Mapping Semantic Web Service Descriptions to Planning Domain Knowledge. In R. Magjarevic, J. H. Nagel, and Ratko Magjarevic, editors, *World Congress on Medical Physics and Biomedical Engineering*, volume 14 of *IFMBE Proceedings*, pages 388–391. Springer, 2007.
- [KKRM05] Michael Klein, Birgitta König-Ries, and Michael Mussig. What is needed for semantic service descriptions – a proposal for suitable language constructs. *International Journal Web Grid Serv.*, 1:328–364, December 2005.
- [KLKR08] Ulrich Küster, Holger Lausen, and Birgitta König-Ries. Evaluation of Semantic Service Discovery—A Survey and Directions for Future Research. In Marius Walliser, Stefan Brantschen, Monique Calisti, Thomas Hempfling, Thomas Gschwind, and Cesare Pautasso, editors, *Emerging Web Services Technology, Volume II*, Whitestein Series in Software Agent Technologies and Autonomic Computing, pages 41–58. Birkhäuser, 2008.
- [KLS90] Henry F. Korth, Eliezer Levy, and Abraham Silberschatz. A Formal Approach to Recovery by Compensating Transactions. In *Proceedings of the sixteenth International Conference on Very Large Databases (VLDB)*, pages 95–106, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.
- [Klu08] Matthias Klusch. Semantic Web Service Coordination. In Michael Schumacher, Heikki Helin, and Heiko Schuldt, editors, *CASCOM: Intelligent Service Coordination in the Semantic Web*, chapter 4, pages 59–104. Birkhäuser, 2008.
- [KM91] Hirofumi Katsuno and Alberto O. Mendelzon. On the Difference between Updating a Knowledge Base and Revising It. In *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning*, pages 387–394. Morgan Kaufmann, April 1991.
- [KM09] Sebastian Ryszard Kruk and Bill McDaniel, editors. *Semantic Digital Libraries*. Springer, 2009.
- [KMHJ10] Zhang Kun, Xu Manwu, Zhang Hong, and Xu Jian. Agent Service Matchmaking Algorithm for Autonomic Element with Semantic and QoS Constraints. *Knowledge-Based Systems*, 23(2):132–143, 2010.

- [KOM05] Atanas Kiryakov, Damyan Ognyanov, and Dimitar Manov. OWLIM – A Pragmatic Semantic Repository for OWL. In Mike Dean, Yuanbo Guo, Woonchun Jun, Roland Kaschek, Shonali Krishnaswamy, Zhengxiang Pan, and Quan Sheng, editors, *Web Information Systems Engineering – WISE Workshops*, volume 3807 of *LNCS*, pages 182–192. Springer, 2005.
- [KRH07] Markus Krötzsch, Sebastian Rudolph, and Pascal Hitzler. Conjunctive queries for a tractable fragment of OWL 1.1. In *Proceedings of the 6th international The semantic web and 2nd Asian conference on Asian semantic web conference, ISWC’07/ASWC’07*, pages 310–323, Berlin, Heidelberg, 2007. Springer.
- [Kri63] Saul Kripke. Semantical Considerations on Modal Logic. *Acta Philosophica Fennica*, 16:83–94, 1963.
- [KSKR05] Ulrich Küster, Mirco Stern, and Birgitta König-Ries. A Classification of Issues and Approaches in Automatic Service Composition. In *First International Workshop on Engineering Service Compositions (WESC)*, 2005.
- [LA90] P.A. Lee and T. Anderson. *Fault tolerance, principles and practice*. Dependable computing and fault-tolerant systems. Springer, 2nd edition, 1990.
- [Law97] Peter Lawrence. *Workflow Handbook 1997*. John Wiley, New York, 1997.
- [LB87] H.J. Levesque and R.J. Brachman. Expressiveness and Tractability in Knowledge Representation and Reasoning. *Computational Intelligence*, 3(1):78–93, 1987.
- [Len02] Maurizio Lenzerini. Data Integration: A Theoretical Perspective. In *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 233–246, New York, NY, USA, 2002. ACM.
- [LH03] Lei Li and Ian Horrocks. A software framework for matchmaking based on semantic web technology. In *Proceedings of the 12th international conference on World Wide Web, WWW ’03*, pages 331–339, New York, NY, USA, 2003. ACM.
- [Liu10] Hongkai Liu. *Computing Updates in Description Logics*. PhD thesis, Technical University of Dresden, January 2010. Available at <http://lat.inf.tu-dresden.de/research/phd/Liu-PhD-10.pdf>.
- [LLMW06] Hongkai Liu, Carsten Lutz, Maja Miličić, and Frank Wolter. Reasoning about Actions using Description Logics with general TBoxes. In Michael Fisher, Wiebe van der Hoek, Boris Konev, and Alexei Lisitsa, editors, *JELIA-10*, volume 4160 of *LNAI*, pages 266–279. Springer, 2006.
- [LMJ10] Guoli Li, Vinod Muthusamy, and Hans-Arno Jacobsen. A Distributed Service-oriented Architecture for Business Process Execution. *ACM Transactions Web*, 4:2:1–2:33, January 2010.
- [Loh08] Niels Lohmann. A Feature-Complete Petri Net Semantics for WS-BPEL 2.0. In *WS-FM*, volume 4937 of *LNCS*, pages 77–91. Springer, 2008.

- [LPR05] Holger Lausen, Axel Polleres, and Dumitru Roman, editors. *Web Service Modeling Ontology (WSMO)*. W3C Member Submission, June 2005. Available at <http://www.w3.org/Submission/WSMO>.
- [LR94] Fangzhen Lin and Raymond Reiter. State constraints revisited. *Logic Computing*, 4(5):655–678, 1994.
- [LRD06] Linh Ly, Stefanie Rinderle, and Peter Dadam. Semantic Correctness in Adaptive Process Management Systems. In Schahram Dustdar, José Fiadeiro, and Amit Sheth, editors, *Business Process Management*, volume 4102 of *LNCS*, pages 193–208. Springer, 2006.
- [LRD08] Linh Thao Ly, Stefanie Rinderle, and Peter Dadam. Integration and Verification of Semantic Constraints in Adaptive Process Management Systems. *Data & Knowledge Engineering*, 64(1):3–23, 2008.
- [LS02] Carsten Lutz and Ulrike Sattler. A Proposal for Describing Services with DLs. In *Proceedings of the 2002 International Workshop on Description Logics*, 2002.
- [LS07] Ruopeng Lu and Shazia Sadiq. A Survey of Comparative Business Process Modeling Approaches. In Witold Abramowicz, editor, *Business Information Systems, 10th International Conference*, volume 4439 of *LNCS*, pages 82–94. Springer, 2007.
- [LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4:382–401, July 1982.
- [Lut99] Carsten Lutz. Complexity of Terminological Reasoning Revisited. In Harald Ganzinger, David McAllester, and Andrei Voronkov, editors, *Logic for Programming and Automated Reasoning*, volume 1705 of *LNCS*, pages 181–200. Springer, 1999.
- [Lut08] Carsten Lutz. The Complexity of Conjunctive Query Answering in Expressive Description Logics. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *IJCAR*, volume 5195 of *LNCS*, pages 179–193, Sydney, Australia, August 2008. Springer.
- [LVOS09] Niels Lohmann, Eric Verbeek, Chun Ouyang, and Christian Stahl. Comparing and evaluating Petri net semantics for BPEL. *International Journal of Business Process Integration and Management (IJBPM)*, 4(1):60–73, 2009.
- [MB09] Alistair Miles and Sean Bechhofer, editors. *SKOS Simple Knowledge Organization System*. W3C Recommendation, August 2009. Available at <http://www.w3.org/TR/skos-reference/>.
- [MBE03] Brahim Medjahed, Athman Bouguettaya, and Ahmed K. Elmagarmid. Composing Web services on the Semantic Web. *The VLDB Journal*, 12:333–351, November 2003.
- [MBH<sup>+</sup>04] David Martin, Mark Burstein, Jerry Hobbs, Ora Lassila, Drew McDermott, Sheila McIlraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry

- Payne, Evren Sirin, Naveen Srinivasan, and Katia Sycara. *OWL-S: Semantic Markup for Web Services*. W3C Member Submission, November 2004. Available at <http://www.w3.org/Submission/OWL-S>.
- [MBK<sup>+</sup>09] Nebil Ben Mabrouk, Sandrine Beauche, Elena Kuznetsova, Nikolaos Georgantas, and Valérie Issarny. QoS-Aware Service Composition in Dynamic Service Oriented Environments. In Jean Bacon and Brian Cooper, editors, *Middleware*, volume 5896 of *LNCS*, pages 123–142. Springer, 2009.
- [McC90] John McCarthy. *Formalizing of common sense: papers by John McCarthy / edited by Vladimir Lifschitz*. Ablex Publishing, 1990.
- [McD02] Drew McDermott. Estimated-Regression Planning for Interactions with Web Services. In Malik Ghallab, Joachim Hertzberg, and Paolo Traverso, editors, *Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems (AIPS)*, pages 204–211. AAAI, 2002.
- [MGH<sup>+</sup>98] Drew Mcdermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL - The Planning Domain Definition Language. Technical report, CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.
- [MGH<sup>+</sup>09] Boris Motik, Bernardo Cuenca Grau, Ian Horrocks, Zhe Wu, Achille Fokoue, and Carsten Lutz, editors. *OWL 2 Web Ontology Language Profiles*. W3C Recommendation, October 2009. Available at <http://www.w3.org/TR/owl-profiles/>.
- [MH04] Deborah L. McGuinness and Frank van Harmelen, editors. *OWL 1 Web Ontology Language Overview*. W3C Recommendation, February 2004. Available at <http://www.w3.org/TR/owl-features/>.
- [MH08] Boris Motik and Ian Horrocks. OWL Datatypes: Design and Implementation. In *Proceedings of the 7<sup>th</sup> International Semantic Web Conference (ISWC)*, pages 307–322, 2008.
- [MH09] Yue Ma and Pascal Hitzler. Paraconsistent Reasoning for OWL 2. In Axel Polleres and Terrance Swift, editors, *Web Reasoning and Rule Systems*, volume 5837 of *LNCS*, pages 197–211. Springer, 2009.
- [MHRS06] Boris Motik, Ian Horrocks, Riccardo Rosati, and Ulrike Sattler. Can OWL and Logic Programming Live Together Happily Ever After? In Isabel Cruz, Stefan Decker, Dean Allemang, Chris Preist, Daniel Schwabe, Peter Mika, Mike Uschold, and Lora Aroyo, editors, *The Semantic Web - ISWC 2006*, volume 4273 of *LNCS*, pages 501–514. Springer, 2006.
- [MHS09] Boris Motik, Ian Horrocks, and Ulrike Sattler. Bridging the gap between OWL and relational databases. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(2):74–89, 2009.
- [MIK<sup>+</sup>10] Keyvan Mohebbi, Suhaimi Ibrahim, Mojtaba Khezrian, Kanmani Munusamy, and Sayed Gholam Hassan Tabatabaei. A comparative evaluation of semantic web service discovery approaches. In *Proceedings of the*



- 12th International Conference on Information Integration and Web-based Applications & Services, iiWAS '10*, pages 33–39, New York, NY, USA, 2010. ACM.
- [Mil99] Robin Milner. *Communicating and Mobile Systems: The  $\pi$ -calculus*. Cambridge University Press, Cambridge, UK, 1999.
- [Mil08] Maja Miličić. *Action, Time and Space in Description Logics*. PhD thesis, University of Dresden, Dresden, Germany, 2008.
- [Min74] Marvin Minsky. A Framework for Representing Knowledge. Technical Report Artificial Intelligence Memo 306, Massachusetts Institute of Technology, Cambridge, MA, USA, 1974.
- [Min81] Marvin Minski. *Mind Design*, chapter A Framework for Representing Knowledge. MIT Press, 1981.
- [MKP09] Maria Maleshkova, Jacek Kopecký, and Carlos Pedrinaci. Adapting SAWSDL for Semantic Annotations of RESTful Services. In Robert Meersman, Pilar Herrero, and Tharam Dillon, editors, *On the Move to Meaningful Internet Systems: OTM 2009 Workshops*, volume 5872 of LNCS, pages 917–926. Springer, 2009.
- [MLM<sup>+</sup>06] C. Matthew MacKenzie, Ken Laskey, Francis McCabe, Peter F. Brown, and Rebekah Metz, editors. *Reference Model for Service Oriented Architecture 1.0*. OASIS Standard, 2006. Available at <http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.html>.
- [MMR11] Andrea Marrella, Massimo Mecella, and Alessandro Russo. Featuring Automatic Adaptivity through Workflow Enactment and Planning. In *Proceedings of the 7th International Conference on Collaborative Computing (CollaborateCom)*, Orlando, Florida, USA, 2011.
- [Mos85] Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. The MIT Press, 1985.
- [Mot06] Boris Motik. *Reasoning in Description Logics using Resolution and Deductive Databases*. PhD thesis, Universität Karlsruhe (TH), Karlsruhe, Germany, January 2006.
- [MP09] Annapaola Marconi and Marco Pistore. Synthesis and Composition of Web Services. In Marco Bernardo, Luca Padovani, and Gianluigi Zavattaro, editors, *Formal Methods for Web Services*, volume 5569 of LNCS, pages 89–157. Springer, 2009.
- [MPG09] Boris Motik, Peter F. Patel-Schneider, and Bernardo Cuenca Grau, editors. *OWL 2 Web Ontology Language Direct Semantics*. W3C Recommendation, October 2009. Available at <http://www.w3.org/TR/owl-direct-semantics/>.
- [MPPS09] Boris Motik, Bijan Parsia, and Peter F. Patel-Schneider, editors. *OWL 2 Web Ontology Language XML Serialization*. W3C Recommendation, October 2009. Available at <http://www.w3.org/TR/owl2-xml-serialization/>.

- [MPSP09] Boris Motik, Peter F. Patel-Schneider, and Bijan Parsia, editors. *OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax*. W3C Recommendation, October 2009. Available at <http://www.w3.org/TR/owl-syntax/>.
- [MRKS92] Sharad Mehrotra, Rajeev Rastogi, Henry F. Korth, and Abraham Silberschatz. A Transaction Model for Multidatabase Systems. In *Proceedings of the 12<sup>th</sup> International Conference on Distributed Computing Systems (ICDCS)*, pages 56–63, Yokohama, Japan, June 1992.
- [MS02] Sheila A. McIlraith and Tran Cao Son. Adapting Golog for Composition of Semantic Web Services. In *8th International Conference on Principles of Knowledge Representation and Reasoning*, pages 482–496, 2002.
- [MS07] Thorsten Möller and Heiko Schuldt. A Platform to Support Decentralized and Dynamically Distributed P2P Composite OWL-S Service Execution. In *Middleware for SoC Workshop, 8<sup>th</sup> Middleware Conf.*, Newport Beach, CA, USA, 2007. ACM.
- [MS08] Thorsten Möller and Heiko Schuldt. Control flow intervention for semantic failure handling during composite service execution. In *ICWS '08: Proceedings of the 2008 IEEE International Conference on Web Services*, pages 834–835, Washington, DC, USA, 2008. IEEE Computer Society.
- [MS10a] Thorsten Möller and Heiko Schuldt. Optimized data access for efficient execution of semantic services. In *ICDE Workshops*, pages 257–262. IEEE, 2010.
- [MS10b] Thorsten Möller and Heiko Schuldt. OSIRIS Next: Flexible Semantic Failure Handling for Composite Web Service Execution. In *ICSC '10: Proceedings of the IEEE Fourth International Conference on Semantic Computing*, pages 212–217, September 2010.
- [MSGK06] Thorsten Möller, Heiko Schuldt, Andreas Gerber, and Matthias Klusch. Next-generation applications in healthcare digital libraries using semantic service composition and coordination. *Health Informatics Journal*, 12(2):107–119, 2006.
- [MSH09] Boris Motik, Rob Shearer, and Ian Horrocks. Hypertableau Reasoning for Description Logics. *Journal of Artificial Intelligence Research*, 36:165–228, 2009.
- [Mur89] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr 1989.
- [NCS04] Mangala Gowri Nanda, Satish Chandra, and Vivek Sarkar. Decentralizing Execution of Composite Web Services. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*, pages 170–187, New York, NY, USA, 2004. ACM.
- [Nie94] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann, San Francisco, Calif., 1994.

- [NM02] Srini Narayanan and Sheila A. McIlraith. Simulation, verification and automated composition of web services. In *Proceedings of the 11th international conference on World Wide Web*, pages 77–88, New York, NY, USA, 2002. ACM.
- [NØ10] Ragnar Normann and Lene T. Østby. A theoretical study of ‘Snapshot Isolation’. In *Proceedings of the 13th International Conference on Database Theory, ICDT*, pages 44–49, New York, NY, USA, 2010. ACM.
- [Noy04] Natalya F. Noy. Semantic Integration: A Survey of Ontology-based Approaches. *SIGMOD Record*, 33:65–70, December 2004.
- [NS10] Linh Nguyen and Andrzej Szalas. Three-Valued Paraconsistent Reasoning for Semantic Web Agents. In Piotr Jędrzejowicz, Ngoc Nguyen, Robert Howlet, and Lakhmi Jain, editors, *Agent and Multi-Agent Systems: Technologies and Applications*, volume 6070 of LNCS, pages 152–162. Springer, 2010.
- [NW10] Thomas Neumann and Gerhard Weikum. x-RDF-3X: Fast Querying, High Update Rates, and Consistency for RDF Databases. *PVLDB*, 3:256–263, September 2010.
- [Ope] OpenGALEN Foundation. OpenGALEN: open source medical terminology and tools. <http://www.opengalen.org>. Visited on July 27, 2012.
- [ORS11] Magdalena Ortiz, Sebastian Rudolph, and Mantas Simkus. Query Answering in the Horn Fragments of the Description Logics SHOIQ and SROIQ. In Toby Walsh, editor, *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, pages 1039–1044. AAAI, Juli 2011.
- [OVA<sup>+</sup>07] Chun Ouyang, Eric Verbeek, Wil M.P. van der Aalst, Stephan Breutel, Marlon Dumas, and Arthur H.M. ter Hofstede. Formal semantics and analysis of control flow in WS-BPEL. *Science of Computer Programming*, 67:162–198, July 2007.
- [PA06] Maja Pesic and Wil M.P. van der Aalst. A declarative approach for flexible business processes management. In Johann Eder and Schahram Dustdar, editors, *Business Process Management Workshops*, volume 4103 of LNCS, pages 169–180. Springer, 2006.
- [Pap03] Mike P. Papazoglou. Service-Oriented Computing: Concepts, Characteristics and Directions. *International Conference on Web Information Systems Engineering*, 0:3, 2003.
- [Pat05] Patrick Stickler. *CBD - Concise Bounded Description*. W3C Member Submission, June 2005. Available at <http://www.w3.org/Submission/CBD>.
- [Pee05] Joachim Peer. Web Service Composition as AI Planning – a Survey. Second revised version, 2005.
- [Pel03] Chris Peltz. Web services orchestration and choreography. *Computer*, 36:46–52, 2003.

- [Pep08] Pavlos Peppas. *Handbook of Knowledge Representation*, chapter 8 - Belief Revision, pages 317–359. Elsevier, 2008.
- [Per82] Alan J. Perlis. Epigrams on programming. *SIGPLAN Notices*, 17:7–13, September 1982.
- [PH03] Jeff Z. Pan and Ian Horrocks. Web Ontology Reasoning with Datatype Groups. In *Proceedings of the 2nd International Semantic Web Conference (ISWC)*, pages 47–63. Springer, 2003.
- [PKPS02] Massimo Paolucci, Takahiro Kawamura, Terry Payne, and Katia Sycara. Semantic matching of web services capabilities. In *The Semantic Web – ISWC 2002*, pages 333–347, 2002.
- [Pre04] Chris Preist. A conceptual architecture for semantic web services. In Sheila A. McIlraith, Dimitris Plexousakis, and Frank van Harmelen, editors, *The Semantic Web – ISWC 2004*, volume 3298 of *LNCS*, pages 395–409. Springer, 2004.
- [PSM09] Peter F. Patel-Schneider and Boris Motik, editors. *OWL 2 Web Ontology Language Mapping to RDF Graphs*. W3C Proposed Recommendation, September 2009. Available at <http://www.w3.org/TR/owl2-mapping-to-rdf/>.
- [PSSA07] Maja Pesic, Helen Schonenberg, Natalja Sidorova, and Wil M.P. van der Aalst. Constraint-based Workflow Models: Change Made Easy. In Robert Meersman and Zahir Tari, editors, *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS*, volume 4803 of *LNCS*, pages 77–94. Springer, 2007.
- [PT09] Graham Priest and Koji Tanaka. Paraconsistent Logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, CSLI, Stanford University, summer 2009 edition, 2009. Available at <http://plato.stanford.edu/archives/sum2009/entries/logic-paraconsistent/>.
- [Put94] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1994.
- [Qui67] Ross M. Quillian. Word concepts: A theory and simulation of some basic semantic capabilities. *Behavioral Science*, 12:410–430, 1967.
- [QZCY07] Zongyan Qiu, Xiangpeng Zhao, Chao Cai, and Hongli Yang. Towards the theoretical foundation of choreography. In *Proceedings of the 16th international conference on World Wide Web, WWW '07*, pages 973–982, New York, NY, USA, 2007. ACM.
- [Rah11] Erhard Rahm. Towards Large-Scale Schema and Ontology Matching. In Zohra Bellahsene, Angela Bonifati, and Erhard Rahm, editors, *Schema Matching and Mapping, Data-Centric Systems and Applications*, pages 3–27. Springer, Berlin, Heidelberg, 2011.

- [RBR06] Stefanie Rinderle, Sarita Bassil, and Manfred Reichert. A Framework for Semantic Recovery Strategies in Case of Process Activity Failures. In *Proceedings of the 8th Int'l conference on Enterprise Information Systems (ICEIS)*, pages 136–143, 2006.
- [RD98] Manfred Reichert and Peter Dadam. ADEPTflex—Supporting Dynamic Changes of Workflows Without Losing Control. *Journal of Intelligent Information Systems*, 10:93–129, 1998. 10.1023/A:1008604709862.
- [Rei88] Raymond Reiter. On integrity constraints. In *Proceedings of the 2nd conference on Theoretical aspects of reasoning about knowledge, TARK '88*, pages 97–111, San Francisco, CA, USA, 1988. Morgan Kaufmann Publishers Inc.
- [Rei01] Raymond Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, Massachusetts, MA, illustrated edition edition, 2001.
- [RH95] Fernando de Ferreira Rezende and Theo Härder. Concurrency Control in Nested Transactions with Enhanced Lock Models for KBMSs. In *DEXA'95*, pages 604–613, London, UK, 1995. Springer.
- [Rin04] Jussi Rintanen. Complexity of Planning with Partial Observability. In Shlomo Zilberstein, Jana Koehler, and Sven Koenig, editors, *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS)*, pages 345–354. AAAI, 2004.
- [RPZ10] Yuan Ren, Jeff Z. Pan, and Yuting Zhao. Closed World Reasoning for OWL2 with NBox. *Tsinghua Science & Technology*, 15(6):692–701, 2010.
- [RRD09] Manfred Reichert, Stefanie Rinderle-Ma, and Peter Dadam. Flexibility in Process-Aware Information Systems. In Kurt Jensen and Wil van der Aalst, editors, *Transactions on Petri Nets and Other Models of Concurrency II*, volume 5460 of *LNCS*, pages 115–135. Springer, 2009.
- [RRW08] Stefanie Rinderle-Ma, Manfred Reichert, and Barbara Weber. Relaxed Compliance Notions in Adaptive Process Management Systems. In Qing Li, Stefano Spaccapietra, Eric Yu, and Antoni Olivé, editors, *Conceptual Modeling*, volume 5231 of *LNCS*, pages 232–247. Springer, 2008.
- [RS04] Jinghai Rao and Xiaomeng Su. A Survey of Automated Web Service Composition Methods. In *Proceedings of the 1st Workshop on Semantic Web Services and Web Process Composition (SWSWPC)*, pages 43–54, San Diego, California, USA, 2004.
- [SABS02] Heiko Schuldt, Gustavo Alonso, Catriel Beerli, and Hans-Jörg Schek. Atomicity and Isolation for Transactional Processes. *ACM Transactions on Database Systems (TODS)*, 27(1):63–116, 2002.
- [SC03] Stefan Schlobach and Ronald Cornet. Non-standard reasoning services for the debugging of description logic terminologies. In *Proceedings of the 18th international joint conference on Artificial intelligence*, pages 355–360, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers Inc.

- [Sch01] Heiko Schuldt. Process Locking: A Protocol based on Ordered Shared Locks for the Execution of Transactional Processes. In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 289–300, New York, NY, USA, 2001. ACM.
- [Sch09] Michael Schneider, editor. *OWL 2 Web Ontology Language RDF-Based Semantics*. W3C Recommendation, October 2009. Available at <http://www.w3.org/TR/owl2-rdf-based-semantics/>.
- [SCMF06] Michael Stollberg, Emilia Cimpian, Adrian Mocan, and Dieter Fensel. A Semantic Web Mediation Architecture. In Mamadou Tadiou Koné and Daniel Lemire, editors, *Canadian Semantic Web*, volume 2 of *Semantic Web and Beyond*, pages 3–22. Springer US, 2006.
- [SGA07] Rudi Studer, Stephan Grimm, and Andreas Abecker, editors. *Semantic Web Services: Concepts, Technologies, and Applications*. Springer, Berlin, 2007.
- [She98] Amit P. Sheth. Changing Focus on Interoperability in Information Systems: From System, Syntax, Structure to Semantics. In M.F. Goodchild, M.J. Egenhofer, R. Fegeas, and C.A. Kottman, editors, *Interoperating Geographic Information Systems*, pages 5–30. Kluwer Academic, 1998.
- [SHF11] Michael Stollberg, Jörg Hoffmann, and Dieter Fensel. A caching technique for optimizing automated service discovery. *International Journal of Semantic Computing*, 5(1):1–31, 2011.
- [SHS08] Michael Schumacher, Heikki Helin, and Heiko Schuldt, editors. *CAS-COM: Intelligent Service Coordination in the Semantic Web*. Whitestein Series in Software Agent Technologies and Autonomic Computing. Birkhäuser, 2008.
- [Sir06] Evren Sirin. *Combining Description Logic reasoning with AI planning for composition of Web Services*. PhD thesis, University of Maryland at College Park, College Park, MD, USA, 2006. AAI3241437.
- [SM10] Shirin Sohrabi and Sheila McIlraith. Preference-Based Web Service Composition: A Middle Ground between Execution and Search. In Peter Patel-Schneider, Yue Pan, Pascal Hitzler, Peter Mika, Lei Zhang, Jeff Pan, Ian Horrocks, and Birte Glimm, editors, *Proceedings of the 9th International Semantic Web Conference (ISWC)*, volume 6496 of *LNCS*, pages 713–729. Springer, 2010.
- [Smi04] David E. Smith. Choosing objectives in over-subscription planning. In Shlomo Zilberstein, Jana Koehler, and Sven Koenig, editors, *ICAPS*, pages 393–401. AAAI, 2004.
- [SMM10] Marco Luca Sbodio, David Martin, and Claude Moulin. Discovering Semantic Web services using SPARQL and intelligent agents. *Web Semantics: Science, Services and Agents on the World Wide Web*, 8(4):310–328, 2010.
- [SMR<sup>+</sup>08] Helen Schonenberg, Ronny Mans, Nick Russell, Nataliya Mulyar, and Wil M.P. van der Aalst. Towards a Taxonomy of Process Flexibility. In

- Zohra Bellahsène, Carson Woo, Ela Hunt, Xavier Franch, and Remi Colletta, editors, *CAiSE Forum*, volume 344 of *CEUR Workshop Proceedings*, pages 81–84. CEUR-WS.org, 2008.
- [SP04] Evren Sirin and Bijan Parsia. The OWL-S Java API. In *Proceedings of the Third International Semantic Web Conference (ISWC)*, 2004.
- [SP07] Evren Sirin and Bijan Parsia. SPARQL-DL: SPARQL Query for OWL-DL. In Christine Golbreich, Aditya Kalyanpur, and Bijan Parsia, editors, *OWLED*, volume 258 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.
- [SPG<sup>+</sup>07] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(2):51–53, 2007.
- [SPL<sup>+</sup>04] Jeff Shneidman, Peter Pietzuch, Jonathan Ledlie, Mema Roussopoulos, Margo Seltzer, and Matt Welsh. Hourglass: An Infrastructure for Connecting Sensor Networks and Applications. Technical report, Harvard University, 2004.
- [SS77] John Miles Smith and Diane C. P. Smith. Database Abstractions: Aggregation and Generalization. *ACM Transactions on Database Systems*, 2(2):105–133, 1977.
- [SS83] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1:222–238, August 1983.
- [SS04] Karsten Schmidt and Christian Stahl. A Petri net semantic for BPEL4WS – validation and application. In Ekkart Kindler, editor, *Proceedings of the 11<sup>th</sup> Workshop on Algorithms and Tools for Petri Nets (AWPN)*, pages 1–6, Paderborn, Germany, 2004.
- [SST<sup>+</sup>05] Christoph Schuler, Heiko Schuldt, Can Türker, Roger Weber, and Hans-Jörg Schek. Peer-to-Peer Execution of (Transactional) Processes. *International Journal of Cooperative Information Systems (IJCIS)*, 14(4):377–405, 2005.
- [SVV11] Thanos Stavropoulos, Dimitris Vrakas, and Ioannis Vlahavas. A Survey of Service Composition in Ambient Intelligence Environments. *Artificial Intelligence Review*, pages 1–24, 2011.
- [SW01] Khodakaram Salimifard and Mike Wright. Petri net-based modelling of workflow systems: An overview. *European Journal of Operational Research*, 134(3):664–676, 2001.
- [SWKL02] Katia Sycara, Seth Widoff, Matthias Klusch, and Jianguo Lu. Larks: Dynamic Matchmaking Among Heterogeneous Software Agents in Cyberspace. *Autonomous Agents and Multi-Agent Systems*, 5:173–203, 2002.
- [SWSS03] Christoph Schuler, Roger Weber, Heiko Schuldt, and Hans-J. Schek. Peer-to-Peer Process Execution with OSIRIS. In Maria Orłowska, Sanjiva Weerawarana, Michael Papazoglou, and Jian Yang, editors, *Service-Oriented Computing - ICSOC*, volume 2910 of *LNCS*, pages 483–498. Springer, 2003.

- [SWSS04] Christoph Schuler, Roger Weber, Heiko Schuldt, and Hans-Jörg Schek. Scalable Peer-to-Peer Process Management – The OSIRIS Approach. In *Proceedings of the IEEE International Conference on Web Services*, pages 26–34, July 2004.
- [SYS09] SYSTAP. Bigdata Architecture Whitepaper. Technical report, SYSTAP, LLC, 2009. Available at [http://www.bigdata.com/whitepapers/bigdata\\_whitepaper\\_10-13-2009\\_public.pdf](http://www.bigdata.com/whitepapers/bigdata_whitepaper_10-13-2009_public.pdf).
- [Tar56] Alfred Tarski. *Logic, Semantics, Metamathematics: Papers from 1923 to 1938*. Oxford University Press, 1956.
- [TH06] Dmitry Tsarkov and Ian Horrocks. FaCT++ Description Logic Reasoner: System Description. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning*, volume 4130 of *LNCS*, pages 292–297. Springer, 2006.
- [Thi05] Michael Thielscher. *Reasoning Robots: The Art and Science of Programming Robotic Agents*. Number 33 in Applied Logic. Springer, 2005.
- [Thi11] Michael Thielscher. A unifying action calculus. *Artificial Intelligence Journal*, 175(1):120–141, 2011.
- [Tob01] Stephan Tobies. Complexity Results and Practical Algorithms for Logics in Knowledge Representation. *CoRR*, cs.LO/0106031, 2001.
- [TPR10] Edward Thomas, Jeff Pan, and Yuan Ren. TrOWL: Tractable OWL 2 Reasoning Infrastructure. In Lora Aroyo, Grigoris Antoniou, Eero Hyvönen, Annette ten Teije, Heiner Stuckenschmidt, Liliana Cabral, and Tania Tudorache, editors, *The Semantic Web: Research and Applications*, volume 6089 of *LNCS*, pages 431–435. Springer, 2010.
- [TRBD11] Eran Toch, Iris Reinhartz-Berger, and Dov Dori. Humans, semantic services and similarity: A user study of semantic Web services matching and composition. *Web Semantics*, 9:16–28, March 2011.
- [TSBM10] Jiao Tao, Evren Sirin, Jie Bao, and Deborah L. McGuinness. Integrity Constraints in OWL. In Maria Fox and David Poole, editors, *AAAI*, Atlanta, Georgia, USA, July 2010. AAAI Press.
- [TTM09] Vuong Xuan Tran, Hidekazu Tsuji, and Ryosuke Masuda. A new QoS ontology and its QoS-based ranking algorithm for Web services. *Simulation Modelling Practice and Theory*, 17(8):1378–1398, 2009.
- [Tur02] Hudson Turner. Polynomial-Length Planning Spans the Polynomial Hierarchy. In *Proceedings of the European Conference on Logics in Artificial Intelligence*, JELIA '02, pages 111–124, London, UK, UK, 2002. Springer.
- [vdA97] Wil M.P. van der Aalst. Verification of Workflow Nets. In Pierre Azéma and Gianfranco Balbo, editors, *Application and Theory of Petri Nets 1997*, volume 1248 of *LNCS*, pages 407–426. Springer, 1997.
- [VHBS98] Radek Vingralek, Haiyan Hasse-Ye, Yuri Breitbart, and Hans-Jörg Schek. Unifying Concurrency Control and Recovery of Transactions with Semantically Rich Operations. *Theoretical Computer Science*, 190(2):363–396, 1998.



- [VKVF08] Tomas Vitvar, Jacek Kopecký, Jana Viskova, and Dieter Fensel. WSMO-Lite Annotations for Web Services. In Sean Bechhofer, Manfred Hauswirth, Jörg Hoffmann, and Manolis Koubarakis, editors, *The Semantic Web: Research and Applications*, volume 5021 of LNCS, pages 674–689. Springer, 2008.
- [Vog09] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52:40–44, January 2009.
- [VWS08] Roman Vaculín, Kevin Wiesner, and Katia Sycara. Exception Handling and Recovery of Semantic Web Services. In *Fourth International Conference on Networking and Services*, pages 217–222, march 2008.
- [W3C] W3C Wiki Page. Literals as Subjects. [http://www.w3.org/2001/sw/wiki/Literals\\_as\\_Subjects](http://www.w3.org/2001/sw/wiki/Literals_as_Subjects). Visited on July 27, 2012.
- [W3C09] W3C OWL Working Group. *OWL 2 Web Ontology Language Document Overview*. W3C Recommendation, October 2009. Available at <http://www.w3.org/TR/owl2-overview/>.
- [WCB01] Matt Welsh, David Culler, and Eric Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the eighteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 230–243, New York, NY, USA, 2001. ACM.
- [Wel94] Daniel S. Weld. An Introduction to Least Commitment Planning. *AI Magazine*, 15(4):27–61, 1994.
- [Wes07] Mathias Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer, 1 edition, November 2007.
- [WHM10] Ingo Weber, Jörg Hoffmann, and Jan Mendling. Beyond Soundness: On the Verification of Semantic Business Process Models. *Distributed and Parallel Databases*, 27:271–343, 2010.
- [Wie92] Gio Wiederhold. Mediators in the Architecture of Future Information Systems. *IEEE Computer*, 25(3):38–49, March 1992.
- [Win88a] Marianne Winslett. A framework for comparison of update semantics. In *Proceedings of the seventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, PODS '88*, pages 315–324, New York, NY, USA, 1988. ACM.
- [Win88b] Marianne Winslett. Reasoning about Action Using a Possible Models Approach. In *AAAI*, pages 89–93, 1988.
- [Win90] Marianne Winslett. *Updating logical databases*. Cambridge University Press, New York, NY, USA, 1990.
- [WS92] Gerhard Weikum and Hans-Jörg Schek. *Concepts and Applications of Multilevel Transactions and Open Nested Transactions*, chapter 13. In: *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, 1992.

- [WV02] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, San Francisco, CA, 2002.
- [WVKS08] Kevin Wiesner, Roman Vaculín, Martin Kollingbaum, and Katia Sycara. Recovery Mechanisms for Semantic Web Services. In René Meier and Sotirios Terzis, editors, *Distributed Applications and Interoperable Systems*, volume 5053 of *LNCS*, pages 100–105. Springer, 2008.
- [WVV<sup>+</sup>01] H. Wache, T. Vögele, U. Visser, H. Stuckenschmidt, G. Schuster, H. Neumann, and S. Hübner. Ontology-Based Integration of Information – A Survey of Existing Approaches. In *Proceedings of the IJCAI-01 Workshop on Ontologies and Information Sharing*, Seattle, USA, 2001.
- [WWWD96] Dirk Wodtke, Jeanine Weissenfels, Gerhard Weikum, and Angelika Kotz Dittrich. The Mentor Project: Steps Towards Enterprise-wide Workflow Management. In *Proceedings of the Twelfth International Conference on Data Engineering (ICDE)*, pages 556–565, 1996.
- [ZBN<sup>+</sup>04] Liangzhao Zeng, Boualem Benatallah, Anne H.H. Ngu, Marlon Dumas, Jayant Kalagnanam, and Henry Chang. QoS-aware Middleware for Web Services Composition. *IEEE Transactions on Software Engineering*, 30(5):311–327, May 2004.
- [ZLW10] Xiaowang Zhang, Zuoquan Lin, and Kewen Wang. Towards a Paradoxical Description Logic for the Semantic Web. In Sebastian Link and Henri Prade, editors, *Foundations of Information and Knowledge Systems*, volume 5956 of *LNCS*, pages 306–325. Springer, 2010.
- [ZML<sup>+</sup>06] Jian Zhou, Li Ma, Qiaoling Liu, Lei Zhang, Yong Yu, and Yue Pan. Minerva: A Scalable OWL Ontology Storage and Inference System. In Riichiro Mizoguchi, Zhongzhi Shi, and Fausto Giunchiglia, editors, *The Semantic Web – ASWC 2006*, volume 4185 of *LNCS*, pages 429–443. Springer, 2006.
- [ZNB01] Aidong Zhang, Marian Nodine, and Bharat Bhargava. Global Scheduling for Flexible Transactions in Heterogeneous Distributed Database Systems. *IEEE Transactions on Knowledge and Data Engineering*, 13:439–450, May 2001.
- [ZNBB94] Aidong Zhang, Marian Nodine, Bharat Bhargava, and Omran Bukhres. Ensuring Relaxed Atomicity for Flexible Transactions in Multidatabase Systems. *SIGMOD Record*, 23:67–78, May 1994.
- [Zol] Evgeny Zolin. Description logic complexity navigator. <http://www.cs.man.ac.uk/~ezolin/dl/>. Visited on July 27, 2012.
- [ZW97] Amy Moormann Zaremski and Jeannette M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6:333–369, October 1997.

# Index

## A

ABox, 20  
abstract operation, 46  
abstract role, 18  
action, 45, 111  
    information-providing, 116  
    parametrized, 113  
acyclic, *see* Petri net  
acyclic TBox, 20  
add operation, *see* update operation  
*ALC*, 18  
annotation, *see* OWL annotation  
annotation property, 36  
answer set variable, *see* distinguished variable  
assertion, 16, 20  
assertional knowledge, 20  
assumption  
    determinism (A1), 43  
    discrete data (A2), 44  
    effect consistency (A6), 69  
    fail-safe operation (A8), 102  
    failure detector, reliable messaging (A7), 99  
    service description (A4), 51  
    stateless operation (A3), 45  
    TBox protected (A5), 57  
atomic service, 75  
atomicity, 153  
axiom, 16  
axiom pinpointing, 73

## B

backward recovery, 4  
belief revision, 32  
belief update, 32, 59  
binding, 46  
blank node closure, 165  
business process, 3

## C

changeset, 154

choreography, 50  
class, *see* concept  
classification task, 29  
closed world assumption, 23, 58  
commit pipe, 161  
commit-projection, 155  
commutativity, 152  
compensatable, 136  
complex service, 75  
composite service, 75  
concept, 16, 18  
    atomic, 18  
    complex, 18  
    defined, 19  
    primitive, 19  
concept assertion, 20  
concept constructor, 18  
concomitant update, 63  
concrete domain, 24  
concrete parameter, *see* profile parameter  
concrete role, 26  
concurrency control, 147  
conditional effect, 60, 68  
conflict  
    update operation, 154  
conjunctive ABox query, 29, 190  
conjunctive normal form, 159  
consistency  
    knowledge base, 21  
continuous operation mode, *see* operation mode  
control flow, 49  
    patterns, 83  
control flow graph, 75, 79  
    interface, *see* Petri net interface  
    sound, 84  
    subflow, 80  
    unfolding, 79  
control state, 81

cut-and-replace process modification,  
103

cyclic TBox, 20

## D

data flow, 49

divide, fork, merge, 86

data flow graph, 75, 89

data item, 150

OWL, 150

data property, *see* concrete role

data range, 26

datatype, 25

datatype map, 25

default graph, *see* RDF dataset

defined concept/role, 19

degree of match, 107, 108

delete operation, *see* update operation

description logic, 15

direct KB update, 33

discrete operation mode, *see* operation  
mode

disjoint match, 108

disjunctive normal form, 159

distinguished variable, 30

distributed execution, 178

divide operator, 86

DL-Lite, 68

domain constraint, 57

dry-run, 82

## E

effect, 48, 59

conditional, *see* conditional effect

forward, *see* forward effect

information-providing, *see* knowl-  
edge effect

effect system, 59

empty profile, 73

entity, *see* OWL entity

equivalent execution, 121

structure-aware, 122

structure-nescient, 124

exact match, 108

execution engine, 3, 98

execution failure, 101

execution state, 81, 128

initial, 85

expression conflict, 159

external nondeterminism, 44

## F

facet expression, 26

failure recovery path, 137

filler, *see* role filler

final marking, 82

final place, 77, 78

finite model property, 23

first committer wins, 155

first updater wins, 157

flow relation, *see* Petri net

forward effect, 127, 130

forward recovery, 4

frame, 193

frame caching, 189

functional unit, 43

compensatable, retrieable, pivot, 136

## G

general concept inclusion axiom, 19

general parameter, *see* profile parameter

general TBox, *see* cyclic TBox

goal, 112, 124, 127

state, 114

temporally extended, 114

grounding, 46

## H

hard goal, 115, 134

heartbeat, 181

history, 154

## I

implementation of an operation, 46

implication, 19, 38

individual, 16

individual equality, 20

individual inequality, 20

initial marking, 78

initial place, 77, 78

input, 47

input node, 77

integrity constraint, 22, 162, 171

- Integrity Isolation, 171
  - internal nondeterminism, 44, 83
  - intersection match, 108
  - inverse role, 17
  - invocation failure, 101
  - isolation, 153
- J**
- join node, 79, 86
- K**
- knowledge base, 21
    - direct update, 33
    - update problem, *see* belief update
  - knowledge effect, 126
- L**
- lexical space, 25
  - literal, *see* RDF literal
  - logical update, 62
- M**
- marked Petri net, 78
  - marking, *see* Petri net
  - matchmaking, 106
    - domain, 106
    - preference-based, 107
    - problem, 106
  - maximum degree of parallelism, 83, 179
  - merge operator, 86
  - model-theoretic semantics, 21, 50, 53
  - monolithic matchmaking, 108
  - monotonicity
    - DL, 28
- N**
- named graph, *see* RDF dataset
  - negation normal form, 159
  - no-op operation, 75
  - node
    - Petri net, 77
    - RDF graph, 34
  - notational conventions, 18
- O**
- object, *see* RDF triple
  - object property, *see* abstract role
  - occlusion, 61
  - $\mathcal{W}$ , 36
  - ontology, *see* OWL ontology
  - open world assumption, 23, 58
  - operation, 45, 74
  - operation mode, 44
  - operator, *see* parametrized action
  - orchestration, 50
  - ordinary node, 79
  - output, 47
  - output node, 77
  - OWL, 35
    - annotation, 36
    - DL, 37
    - EL, 37
    - entity, 36
    - Full, 37
    - import closure, 36
    - knowledge base, *see*  $\mathcal{W}$
    - mapping to RDF, 38
    - ontology, 35
    - profiles, 37
    - QL, 38, 68
    - RL, 38
    - syntactic instance closure, 36
  - OWL data item, *see* data item
- P**
- parameter, *see* profile parameter
  - parameter names assumption, 22
  - parametrized action, 113
  - path, *see* Petri net
    - elementary, 77
  - PDDL, 114
  - Petri net, 77
    - acyclic, 77
    - flow relation, 77
    - free choice, 77
    - interface, 77
    - marking, 78
    - path, 77
    - place, 77
    - strongly connected, 77
    - token, 78
    - transition, 77
  - pivot, 136

place  
 final, *see* final place  
 initial, *see* initial place  
 place node, *see* Petri net  
 plan, 112  
 optimal, 112  
 plan checking, 112  
 plan existence problem, 112  
 planning  
 classic, 113  
 complete, sound, 112  
 contingency, 116  
 dynamic, 116  
 static, 116  
 planning domain, 112  
 planning problem, 112  
 plug-in match, 87, 108  
 strict, *see* strict plug-in/subsume match  
 weak, *see* weak plug-in/subsume match  
 possible models approach, 60  
 possible worlds approach, 60  
 post-set, 77  
 pre-set, 77  
 precondition, 47, 58  
 precondition system, 58  
 predicate, *see* RDF triple  
 predicate lock, 166  
 prenex normal form, 159  
 prepared query, 189  
 primal update, 63  
 primitive concept/role, 19  
 process, 49  
 deterministic, 83  
 transactional, *see* transactional process  
 profile, 47, 73  
 profile parameter, 51, 52  
 assignment function, 54, 181  
 link to precondition, effect variable, 57, 66  
 property, *see* role  
 protected part, 57  
 prudence principle, 58, 66

## Q

qualification problem, 102  
 query answering problem, 30  
 query entailment problem, 30  
 query subsumption, 30

## R

RBox, 19  
 RDF, 34  
 dataset, 35  
 graph, 34  
 literal, 34  
 merge, 35  
 triple, 34  
 read operation, 152  
 read/write model, 150  
 realization task, 29  
 replacement, 103  
 resource, 34  
 Resource Description Framework, *see* RDF  
 retrievable, 136  
 rigid relation, 61  
 role, 16, 17  
 defined, 19  
 filler, 20  
 primitive, 19  
 role assertion, 20  
 role constructor, 18  
 role inclusion axiom, 17  
 rule, 38

## S

satisfiability, 28  
 satisfiability conflict, 160  
 schedule, 154  
 serializability, 154  
 server, 98  
 service, 47, 75  
 application, 178  
 description, 50  
 instance, 47  
 native, 178  
 system, 178  
 volatile, 49

SHOIN

- semantics, 21
- syntax, 17
- simple role, 17
- simple service, 75
- sink, 86
- Snapshot Isolation, 155
- soft goal, 115, 134
- solution set variable, *see* distinguished variable
- source, 86
- spare input, 109
- spare output, 108
- split node, 79, 86
- SRIOQ*, 23
- standard names assumption, 22, 113
- state trajectory, 114, 135
- stateless operation, 45
- strict plug-in/subsume match, 110
- STRIPS, 113
- strongly connected, *see* Petri net
- structural substitution, 103
- structured matchmaking, 107
- sub process, 98
- sub service, 75
- subflow, *see* control flow graph
- subject, *see* RDF triple
- sublanguage, 18
- subsume match, 87, 108
- syntactic construct, 21
- syntactic instance, 21, 36

**T**

- task, 45
- TBox, 19
  - acyclic, *see* acyclic TBox
  - cyclic, *see* cyclic TBox
- terminological knowledge, 19
- token, *see* Petri net
- transaction, 153
  - active, 155
- transactional process, 135
- transition node, 77
- transitive role, 17
- triple, *see* RDF triple

**U**

- undistinguished variable, 30
- unfolding, *see* control flow graph
- unique name assumption, 20, 22, 68
- update operation, 32, 152
  - add, 152
  - commutativity, 152
  - conflict, 154
  - delete, 152
- update transaction, *see* transaction

**V**

- value space, 25
- vocabulary, 16, 36
- volatile service/operation, 49

**W**

- weak plug-in/subsume match, 110
- Web Ontology Language, *see* OWL
- well-handled Petri net, 78
- well-structured Workflow net, 78
- workflow, 3
- Workflow net, 78
  - bounded, 84
  - live, 84
- world state, 56, 81
- write skew, 163





# Curriculum Vitae

## Thorsten Möller

- 27.05.1972** Born in Saalfeld/Saale, Deutschland  
Son of Christine and Siegfried Möller  
German citizenship
- 1978–1988** Primary/Secondary School  
Saalfeld/Saale and Probstzella, Germany
- 1988–1991** Apprenticeship at Deutsche Bahn, Germany  
**1991** Vocational qualification as Kommunikationselektroniker,  
Fachrichtung Informationstechnik
- 1991–1998** IT Technican  
Rosenbauer IT Systems, Neustadt, Germany
- 1998** Multimedia Developer and Trainer  
Connect Telezentrum GmbH & Co. KG , Neustadt, Germany
- 1994–1997** Evening College Coburg, Germany
- 1998** Specific higher education entrance qualification  
Dresden University of Technology, Germany
- 1998–2004** Study of Computer Science  
Dresden University of Technology, Germany  
**2004** Diplom-Informatiker (equiv. to M.Sc. in Computer Science)  
Dresden University of Technology, Germany
- 2004** Software Engineer at  
Fink & Partner Media Services GmbH, Dresden, Germany
- 2004–2005** Research Assistant in the group of Prof. Heiko Schuldt  
Information and Software Engineering  
UMIT Hall in Tyrol, Austria
- 2006–2011** Research Assistant in the group of Prof. Heiko Schuldt  
Database and Information Systems  
University of Basel, Switzerland