Generating SADI Semantic Web Services from Declarative Descriptions

by

Mohammad Sadnan Al Manir

Master of Science (MSc), Vienna University of Technology, 2009
Master in Computer Science, Free University of Bozen-Bolzano, 2009

THIS DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor Of Philosophy

In the Graduate Academic Unit of Computer Science

Supervisor(s): Christopher J.O. Baker, PhD, Dept. of Computer Science
Harold Boley, PhD, Faculty of Computer Science

Examing Board: Josée Tassé, PhD, Dept. of Computer Science
Owen Kaser, PhD, Dept. of Computer Science
Dongmin Kim, PhD, Faculty of Business

External Examiner: Anna-Lena Lamprecht, PhD, Dept. of Information and Computing Sciences, Utrecht University

This dissertation is accepted by the

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

April, 2019

©Mohammad Sadnan Al Manir, 2019
Abstract

Accessing information stored in databases remains a challenge for many types of end users. In contrast, accessing information from knowledge bases allows for more intuitive query formulation techniques. Whereas knowledge bases can be directly instantiated by the materialization of data according to a reference semantic model, a more scalable approach is to rely on queries formulated using ontologies rewritten as database queries at query time. Both of these approaches allow semantic querying, which involves the application of domain knowledge written in the form of axioms and declarative semantic mapping rules. In neither case are users required to interact with the underlying database schemas.

A further approach offering semantic querying relies on SADI Semantic Web services to access relational databases. In this approach, services brokering access to specific data sets can be automatically discovered, orchestrated into workflows, and invoked to execute queries performing data retrieval or data transformation. This can be achieved using specialized query clients built for interfacing with services. Although this approach provides a successful
way of accessing data, creating services requires advanced skills in modeling RDF data and domain ontologies, writing of program code and SQL queries. In this thesis we propose the Valet SADI framework as a solution for automation of SADI Semantic Web service creation. Valet SADI represents a novel architecture comprising four modules which work together to generate and populate services into queryable registries. In the first module declarative semantic mappings are written between source databases and domain ontologies. In a second module, the inputs and outputs of a service are defined in a service ontology with reference to domain ontologies. The third module creates un-instantiated SQL queries automatically based on a semantic query, the target database, domain ontologies and mapping rules. The fourth module produces the source code for a complete and functional SADI service containing the SQL query. The inputs to the first two modules are verified manually while the other modules are fully automated. Valet SADI is demonstrated in two use cases, namely, the creation of a queryable registry of services for surveillance of hospital-acquired infections, and the preservation of interoperability in a malaria surveillance infrastructure.
Acknowledgements

I would like to express my sincere gratitude to my supervisors Dr. Christopher J. O. Baker and Dr. Harold Boley for their continuous support, guidance and supervision throughout the PhD program.

The research reported herein had not been possible without the ideas, foundation work and technical support by Dr. Alexerdre Riazanov.

I am grateful to the members of the examining committee - Dr. Owen Kaser, Dr. Josée Tassé, Dr. Prabhat Mahanti, and Dr. Dongmin Kim for their thorough review of this work that included many insightful observations and helpful hints to improve the thesis.

I am also thankful to the external examiner Dr. Anna-Lena Lamprecht for taking the time of her busy schedule to provide insightful reviews and inspirational comments about the work.

Another source of inspiration for the thesis was from the RuleML community. My gratitude goes to Dr. Tara Athan who was very helpful with her expertise for one of my awards.

The financial support provided by IPSNP Computing Inc., Innovatia Inc.,
RuleML Inc., and Canadian Rivers Institute at UNB, to name a few, has played an important role in funding my work.

Finally, I would like to thank my family and friends for enriching my life beyond my academic endeavors and believing in me.
Table of Contents

Abstract ii

Acknowledgments iv

Table of Contents xiii

List of Tables xiv

List of Figures xvii

Abbreviations xviii

1 Introduction 1

1.1 Overview ......................................................... 1

1.1.1 Problem Statement ........................................... 4

1.1.1.1 Protracted Modeling .................................... 4

1.1.1.2 Fallibility .................................................. 5

1.1.1.3 Adoption Challenges .................................... 5

1.2 Thesis Contribution ............................................... 6

1.3 Thesis Outline .................................................. 9
2 Preliminaries

2.1 The Web ........................................... 11

2.2 Web Service ....................................... 12

2.2.1 Standards ........................................ 13

2.2.1.1 Simple Object Access Protocol (SOAP) .... 14

2.2.1.2 Web Service Description Language (WSDL) . 14

2.2.1.3 Universal Description, Discovery, and Integra-

2.2.2 Representational State Transfer (REST) .......... 15

2.3 Semantic Web ....................................... 16

2.3.1 Resource Description Framework ..................... 17

2.3.2 Propositional Logic .............................. 19

2.3.3 Horn Logic ..................................... 19

2.3.4 First-Order Logic ............................... 20

2.3.5 TPTP First-Order Formula ......................... 21

2.3.6 VampirePrime Reasoner ......................... 22

2.3.7 Description Logic ............................... 22

2.3.8 Web Ontology Language ......................... 24

2.3.8.1 OWL Development Environment ............... 26

2.3.9 PSOA RuleML ..................................... 27

2.3.10 Linked Data .................................... 29

2.3.11 FAIR Data ...................................... 30

2.3.12 SPARQL Protocol and Query Language .......... 31
2.4 Semantic Web Services ............................................ 31
  2.4.1 SADI Semantic Web Services ................................. 34
    2.4.1.1 Motivation ............................................ 34
    2.4.1.2 Operation ............................................. 35
    2.4.1.3 Limitation ............................................ 39
    2.4.1.4 Querying SADI Services ................................. 40
    2.4.1.5 Tools used for Adopting SADI Services ............... 41
    2.4.1.6 Manual and Automated Deployments ................. 43
    2.4.1.7 Required Resources and Competencies ............... 44
  2.4.2 WSDL-based and REST-based Services .................... 47
    2.4.2.1 WSDL-based Approach ................................. 47
    2.4.2.2 REST-based Approach ................................ 52
  2.5 Databases and Ontologies .................................... 53

3 Valet SADI ............................................................ 58
  3.1 Architecture .................................................... 59
    3.1.1 Module-1: Semantic Mapping ............................. 59
      3.1.1.1 Relational Database ................................ 61
      3.1.1.2 Domain Ontology .................................... 61
      3.1.1.3 Mapping Rules ....................................... 61
    3.1.2 Module-2: Service Description .......................... 62
      3.1.2.1 Service Ontology .................................... 62
    3.1.3 Module-3: SQL-template Query Generator .............. 62
3.1.3.1 Service I/O to Semantic Query Converter . . 64
3.1.3.2 Ontology to TPTP Formula Translator . . . . 64
3.1.3.3 PSOA Rules to TPTP Formula Translator . . 64
3.1.3.4 VampirePrime Reasoner + Query Rewriter . . 64
3.1.4 Module-4: Service Generator . . . . . . . . . . . . . . . 64

3.2 Descriptions of the Modules . . . . . . . . . . . . . . . . . . 65
3.2.1 Writing Mapping Rules . . . . . . . . . . . . . . . . . . . 66
3.2.2 Writing Service Descriptions . . . . . . . . . . . . . . . . 66
3.2.3 Automatic Generation of SQL-template Query . . . . . 68
3.2.4 Automatic Generation of Web Service Code . . . . . . 69
3.2.5 Querying SADI Services . . . . . . . . . . . . . . . . . . 71
3.2.6 Target End Users . . . . . . . . . . . . . . . . . . . . . . 73

4 Implementation of Valet SADI 76
4.1 Module-1: Semantic Mapping . . . . . . . . . . . . . . . . . . . 77
4.1.1 Rules for Mapping Database to Ontology . . . . . . . . . 79
4.2 Module-2: Service Description . . . . . . . . . . . . . . . . . . . 82
4.3 Module-3: SQL-template Query Generator . . . . . . . . . . . . 84
4.3.1 Service I/O to Semantic Query Converter . . . . . . . . . 90
4.3.2 Ontology to TPTP Formula Translator . . . . . . . . . . . 91
4.3.3 PSOA Rules to TPTP Formula Translator . . . . . . . . . 93
4.4 Module-4: Service Generator . . . . . . . . . . . . . . . . . . . 95
4.5 Structure of Valet SADI Codebase . . . . . . . . . . . . . . . . 98
5 Service Generation and Target Application Domain

5.1 Hospital-Acquired Infection (HAI) ................................................. 101

5.2 Steps for Generating a SADI Service ........................................... 101

5.2.1 Documenting Semantic Mapping Rules ...................................... 102

5.2.1.1 Relational Database ....................................................... 102

5.2.1.2 Domain Ontology ......................................................... 106

5.2.1.3 Mapping Rules in PSOA RuleML ....................................... 107

5.2.2 Declarative Description of Services .......................................... 110

5.2.2.1 Defining I/O of all Y Services ........................................... 110

5.2.2.2 Defining I/O of get Y By X Services ................................... 111

5.2.3 Generating SQL-template Queries ............................................. 113

5.2.3.1 Converting I/O into Semantic Query .................................... 114

5.2.3.2 Translating Domain Ontology to TPTP-FOF .......................... 116

5.2.3.3 Translating PSOA Rules to TPTP-FOF ................................ 116

5.2.3.4 Rewriting Schematic Answers to SQL .................................. 119

5.2.4 Generation of SADI Service Code ............................................ 119

5.2.5 Additional SADI Services for HAI Surveillance ......................... 121

5.2.5.1 Service Generation Time for Valet SADI ............................. 124

6 Querying HAI Data with SADI Services ........................................ 128

6.1 Query on Hospital-Acquired Infections ....................................... 129

6.1.1 Query Composition and Execution .......................................... 130

6.1.2 Automatic Composition of SPARQL Queries ............................. 133
7 Preservation of Interoperability in Malaria Surveillance

7.1 Malaria Surveillance Today ........................................... 137

7.2 SIEMA Surveillance Platform ...................................... 139

7.2.1 Resources .......................................................... 139

7.2.1.1 Source Data and Standard Terminologies .............. 140

7.2.1.2 Agent-based Analytics Dashboard ....................... 141

7.3 Service-based Querying of Malaria Data ......................... 142

7.3.1 Questions for Malaria Surveillance ............................ 144

7.3.2 Building SADI Services ......................................... 145

7.3.2.1 Source Data and Vocabulary ............................... 145

7.3.2.2 Description of SADI Services ............................. 147

7.3.2.3 Service Registry ............................................. 148

7.3.2.4 Provisioning of SADI Services ............................ 149

7.3.2.5 Building Queries ............................................. 150

7.4 Preservation of Interoperability .................................... 152

7.4.1 Changes in the Service Ontologies ............................. 154

7.4.1.1 Example of Addition ....................................... 155

7.4.2 Status of Services ................................................ 156

7.4.3 Reacting to Changes .............................................. 158

7.4.4 Rebuilding the Services using Valet SADI ................. 160

7.4.4.1 Modified Domain Ontology ............................... 161

7.4.4.2 Modified Malaria-specific Mapping Rules ............. 161

7.5 Discussion .................................................................. 161
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.4.2.1</td>
<td>Resolution Proof Tree</td>
<td>254</td>
</tr>
<tr>
<td>A.4.2.2</td>
<td>Generated Schematic Answers</td>
<td>254</td>
</tr>
<tr>
<td>A.4.2.3</td>
<td>SQL-template Query</td>
<td>256</td>
</tr>
<tr>
<td>A.4.3</td>
<td>S3: getAllergyByPatientId Service</td>
<td>257</td>
</tr>
<tr>
<td>A.4.3.1</td>
<td>Resolution Proof Tree</td>
<td>257</td>
</tr>
<tr>
<td>A.4.3.2</td>
<td>Generated Schematic Answers</td>
<td>257</td>
</tr>
<tr>
<td>A.4.3.3</td>
<td>SQL-template Query</td>
<td>258</td>
</tr>
<tr>
<td>A.5</td>
<td>Source Code of SADI Services</td>
<td>260</td>
</tr>
<tr>
<td>A.5.1</td>
<td>S1: allPatientId Service</td>
<td>260</td>
</tr>
<tr>
<td>A.5.2</td>
<td>S2: getFirstnameLastnameByPatientId Service</td>
<td>261</td>
</tr>
<tr>
<td>A.5.3</td>
<td>S3: getAllergyByPatientId Service</td>
<td>263</td>
</tr>
<tr>
<td>A.6</td>
<td>Composing SPARQL Queries Incrementally on HYDRA</td>
<td>265</td>
</tr>
</tbody>
</table>

Vita
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Symbols and logical meanings of TPTP constructors</td>
<td>22</td>
</tr>
<tr>
<td>2.2</td>
<td>Syntax and semantics of SHOIN</td>
<td>24</td>
</tr>
<tr>
<td>2.3</td>
<td>SADI Service-based system deployments</td>
<td>45</td>
</tr>
<tr>
<td>4.1</td>
<td>Existing approaches for mapping RDBs to ontologies</td>
<td>79</td>
</tr>
<tr>
<td>4.2</td>
<td>Reduction of SHOIN DL to FOL</td>
<td>93</td>
</tr>
<tr>
<td>4.3</td>
<td>Mapping from PSOA constructs to TPTP-FOF constructs</td>
<td>94</td>
</tr>
<tr>
<td>5.1</td>
<td>Auto-generated SQL-template queries for $S_1$, $S_2$ and $S_3$</td>
<td>120</td>
</tr>
<tr>
<td>5.2</td>
<td>Auto-generated SQL-template queries for $S_4$ to $S_{11}$</td>
<td>125</td>
</tr>
<tr>
<td>5.3</td>
<td>Time for generating $S_1$ through $S_{11}$ by Valet SADI</td>
<td>126</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Roles and operations of objects in a Web service architecture</td>
<td>13</td>
</tr>
<tr>
<td>2.2</td>
<td>Semantic Web layer cake diagram</td>
<td>17</td>
</tr>
<tr>
<td>2.3</td>
<td>The four major stages in the development of the Web</td>
<td>32</td>
</tr>
<tr>
<td>2.4</td>
<td>Interaction of client and service endpoint in SADI service</td>
<td>39</td>
</tr>
<tr>
<td>2.5</td>
<td>Mapping approaches from RDB to ontology</td>
<td>56</td>
</tr>
<tr>
<td>3.1</td>
<td>Valet SADI architecture</td>
<td>60</td>
</tr>
<tr>
<td>3.2</td>
<td>Authoring semantic mapping rules</td>
<td>67</td>
</tr>
<tr>
<td>3.3</td>
<td>Defining I/O of SADI services in ontologies</td>
<td>68</td>
</tr>
<tr>
<td>3.4</td>
<td>Steps for generating SQL-template query automatically</td>
<td>70</td>
</tr>
<tr>
<td>3.5</td>
<td>Steps for automatic generation of SADI Web service code</td>
<td>72</td>
</tr>
<tr>
<td>3.6</td>
<td>Querying source data using SADI services</td>
<td>73</td>
</tr>
<tr>
<td>4.1</td>
<td>Implementing service generation at a glance</td>
<td>77</td>
</tr>
<tr>
<td>5.1</td>
<td>The Ottawa Hospital Data Warehouse</td>
<td>103</td>
</tr>
<tr>
<td>5.2</td>
<td>Schema extract from The Ottawa Hospital Data Warehouse</td>
<td>104</td>
</tr>
<tr>
<td>5.3</td>
<td>Hospital-Acquired Infections ontology</td>
<td>107</td>
</tr>
<tr>
<td>5.4</td>
<td>Mapping rules in PSOA RuleML for HAI ontology</td>
<td>109</td>
</tr>
</tbody>
</table>
5.5 Output graph of all PatientIds

5.6 Input graph of get LastnameFirstname By PatientId

5.7 Output graph of get LastnameFirstname By PatientId

5.8 Input graph of get Allergy By PatientId

5.9 Output graph of get Allergy By PatientId

6.1 SQL query for HAI surveillance

6.2 SPARQL query for HAI surveillance

6.3 Screenshot of the HAI surveillance query on HYDRA

7.1 Architecture diagram of SIEMA surveillance framework

7.2 Screenshot of the analytics dashboard

7.3 Screenshots of malaria-specific ontology and RDB schema

7.4 I/O of get InsecticideId By IndoorResidualSprayingId

7.5 Input and output of get Name By InsecticideId

7.6 A fragment of the registry of SADI services

7.7 Malaria-specific mapping rules in PSOA RuleML

7.8 Original query about indoor residual spraying

7.9 A complex query composed of SADI services on malaria

7.10 Modified query about indoor residual spraying

7.11 Difference in output of get Name By InsecticideId

7.12 Detection and identification of changes

7.13 Status of the affected SADI services

7.14 Fragment of the updated malaria-specific PSOA rules
7.15 Status of the rebuilt SADI service ................. 162

A.1 $q_1$: Query to find all patients ....................... 265
A.2 $q_2$: Extention of $q_1$ to list first and last names . . . . . 265
A.3 $q_3$: Extention of $q_2$ with admitting facility .......... 266
A.4 $q_4$: Extention of $q_3$ with diagnosis information . . . . 267
A.5 $q_5$: Extention of $q_4$ with start time of the procedure . . . 268
A.6 $q_6$: Extention of $q_5$ with the antibiotic intake ................ 269
A.7 $q_7$: Extention of $q_6$ with allergic reaction to penicillin . . 270
List of Symbols, Nomenclature or Abbreviations

Symbols:

\( DB \)  
Relational database

\( DB' \)  
Database abstraction

\( Q \)  
Semantic query

\( KB \)  
Knowledge base consisting of mapping rules and domain ontology

get\( Y \) By\( X \)  
Output \( Y \), Input \( X \)

all\( Y \)  
Output \( Y \)

\( cex \)  
OWL Class Expression

\( V \)  
Vertices

\( E \)  
Edges

\( G \)  
Graph

\( V_{cex} \)  
OWL Class Expression as a Vertex

\( G_i \)  
Input graph created from an OWL Class Expression

\( G_O \)  
Output graph created from an OWL Class Expression

\( S_i \)  
Java statement for reading data value

\( S_o \)  
Java statement for writing data value

\( S_{SQL-\text{template}} \)  
SQL statement for a template query

\( S_{Java} \)  
Ordered list of statements in Java

\( \tau_{psoa}(\cdot) \)  
Translation function for mapping

\( \varphi \)  
Antecedent in a mapping rule

\( \psi \)  
Consequent in a mapping rule
Acronyms:

OWL  Web Ontology Language
RDF  Resource Description Framework
RPC  Remote Procedure Calls
PSOA Positional-Slotted Object-Applicative
OID  Object IDentifier
RIF  Rule Interchange Format
TPTP Thousands of Problems for Theorem Provers
FOL  First-Order Logic
DL   Description Logic
SADI Semantic Automatic Discovery & Integration
HAI  Hospital-Acquired Infection
TOHDW The Ottawa Hospital Data Warehouse
SSI  Surgical Site Infection
ICD  International Classification of Diseases
SIEMA Semantics, Interoperability, and Evolution for Malaria Analytics
MIS  Malaria Information System
GTS  Global Technical Strategy
VSMOM Vector Surveillance and Management
MIRO Mosquito Insecticide Resistance Ontology
PHont Public Health Ontology
CDC  Center for Disease Control and Prevention
API  Application Programming Interface
OBDA Ontology-based Data Access
Chapter 1

Introduction

In this chapter, we provide a motivation behind proposing a framework for generating SADI Semantic Web services automatically from their declarative input and output descriptions. We explain why existing approaches are time-consuming, error-prone and they remain unattractive to data providers and end users alike. We list our contributions in order to address these shortcomings and elaborate how this thesis is organized.

1.1 Overview

Data today is primarily stored in relational databases (RDBs) as RDB engines offer scalability, efficiency, reliability, and optimized performance in query execution. Data in RDBs are described by bespoke schemas which essentially represent concepts of a particular domain and relations among them
using constraints, indices etc. In order to retrieve information from databases users need to have knowledge of Structured Query Language (SQL). Knowledge, on the other hand, is stored and represented in ontologies, designed to unambiguously capture the precise meaning of a domain. Ontologies use standardized terms shared by a community of practice. While users of ontologies in a domain can express and formalize queries using these terminologies, querying actual data from a database may remain problematic without good understanding of the data schemas or knowledge of SQL. Semantic querying of relational databases has been used as a solution to address this problem. Semantic querying is based on the automatic application of domain knowledge written in the form of axioms or rules such that users familiar with the domain ontologies can formulate queries using the terminologies of their domain without having any knowledge of the underlying database schema.

A number of existing approaches to semantic querying of RDBs use some kind of semantic mappings to map RDBs to Web Ontology Language (OWL) ontologies or Resource Description Framework (RDF) vocabularies as the target schemas which are aligned to our work in this thesis. These mappings define an interpretation of data from a RDB in terms of the concepts and relations defined in the target schema. The implementations then use these mappings to either completely translate the relational data which is known as the warehouse approach, or apply them in query time which is known as the mediator approach.

In the warehouse approach, data in relational databases are materialized
(i.e. RDFized) to create a RDF triplestore and made available for querying or browsing. An advantage of this approach is that even large triplestores can be processed very efficiently due to the availability of mature highly-optimized query engines. The limitation is that every time new data are added, they have to be RDFized before they can be queried. Triplify \[4\] and Minerva \[5\] are examples of tools supporting the warehouse approach. The mediator approach uses query rewriting where queries in terms of the target semantic schema are converted to equivalent queries in terms of the source relational schema. Live datasets can be queried using this approach, hence it is very popular. Some of the systems that implement query rewriting are reported here in \[2,6–10\].

In contrast, certain tasks such as allowing temporal arithmetics or combining relational data with non-relational data using specialized algorithms are difficult to express and implement with declarative semantic mappings. Riazanov et al. addressed these challenges in a clinical intelligence settings \[11,12\].

To address such challenges the (now decade old) SADI framework was introduced. The SADI framework facilitates the federated querying of multiple heterogeneous, distributed and autonomous data sources, such as online databases and algorithmic resources \[13,16\]. Therefore, the SADI service-based approach copes well with the challenge faced during data integration and remains the main motivation behind the use of semantic querying based on the SADI Semantic Web services. For use by a domain expert lacking sufficient technical knowledge in RDBs, a set of SADI Semantic Web services
can be deployed on top of RDBs in order to retrieve information.

1.1.1 Problem Statement

In this thesis, we investigate the challenges faced during the deployment of semantic querying based on the SADI Semantic Web services. In previous SADI service-based deployments [11–14,16,17] more than 700 SADI services were created and deployed in various domains and these operations were accomplished manually where the following challenges were identified:

1.1.1.1 Protracted Modeling

A SADI service that can be used to query a database must be able to read input values from an RDF input, compose an SQL query to run on the RDB engine, and return data as an RDF output. Therefore, creating a SADI service requires knowledge of data modeling in RDF, ontology modeling in OWL, the SQL query language, a programming language such as Java or Perl, and potentially knowledge of the SPARQL query language to test the services. All these competencies are typically acquired over a long period of time by an expert such as a Semantic Web engineer. Once the modeling of input and output is decided, an experienced engineer still needs a fair amount of time to fully deploy the service depending on the complexity of the modeling and the SQL query. As a result, creating a large number of services is time-consuming when it is done manually even by an experienced engineer. A goal of this thesis is to investigate how to speed up this process.
1.1.1.2 Fallibility

When a SADI service is created manually, each of the steps from modeling input and output to writing SPARQL queries mentioned above is prone to human errors. We investigate how to automate the generation of error-free source code that can read data values from input data and write data values into output data in the uniform RDF format, and to automate the generation of a correct SQL query from the declarative input and output descriptions. However, when declarative semantic mappings are used explicitly, they are also prone to errors because when either the database schema or the ontologies or both evolve, the mappings need to be modified as well. As automatic mapping tools are yet to mature, a goal of this thesis is to investigate devising rules so that mapping rules can be generated at least semi-automatically.

1.1.1.3 Adoption Challenges

Data providers typically expose their data for public use through Application Programming Interfaces (APIs) securely, limiting access to only specific fields of a complete data warehouse. To reach a wider user base APIs are published in many popular languages albeit this is difficult to maintain. The expertise required to access data using the APIs is typical of Software Engineers but remains a challenge to non-technical users. Further to this, due to a lack of semantics in the APIs, software agents cannot discover the data automatically. A goal of this thesis is to investigate if a data provider could benefit from exposing their data using Semantic Web services beside the current
1.2 Thesis Contribution

The contributions of this thesis are listed below:

1. We propose a novel architecture to automate the generation of SADI Semantic Web services. The architecture is called Valet SADI and comprises four modules (numbered as Module-1 to Module-4) which work together as a pipeline. In Module-1, declarative semantic mappings are produced from the source database and the domain ontologies. In Module-2, the input and output of a service are defined in a separate service ontology with reference to the domain ontologies and a semantic query is produced from them. In Module-3, uninstantiated SQL queries are produced automatically with respect to the semantic query, the database, and the domain ontologies. Finally in Module-4, source code for a complete and functional SADI service containing the SQL query is produced and the service is deployed in a registry after compilation and packaging. While the inputs in Module-1 and Module-2 are verified and made available manually, the tasks assigned for Module-3 and Module-4 run without any human intervention. The modular design keeps the door open for further improvements and reuse.

2. We have customized existing resources and developed algorithms in order to execute the tasks assigned for each module. For declarative
semantic mappings we propose rules written in the PSOA RuleML language.

3. In Module-3 we have developed a converter, a translator and extended an existing translator, all of which produce their outputs in TPTP format [18] which are necessary for generating the SQL query automatically.

4. Our algorithm in the converter processes the input and output as graphs and creates a semantic query. One of the translators uses the declarative semantic mapping rules as input and translates them into First-Order axioms in TPTP format. The other translator which translates domain ontologies in OWL into First-Order axioms in TPTP format has been extended from its current form in order to cope with the latest features of ontologies expressed in OWL. We use the incremental query rewriting technique [6] in association with a full First-Order reasoner VampirePrime [19] in order to generate the SQL queries.

5. To show the effective use of Valet SADI framework we have illustrated two deployments for (i) surveillance of Hospital-Acquired Infections and (ii) preservation of interoperability in malaria surveillance infrastructure.

6. We have created 11 SADI services using Valet SADI on top of an extract of The Ottawa Hospital data warehouse. We have demonstrated
how complex queries can be formulated using these services. The target queries illustrated here are similar to the queries used in a previous work [12] where services were created manually. We document the required efforts and time spent for generating these services automatically using the Valet SADI framework, once the semantic mapping rules were specified.

7. In the second deployment we supported yet another surveillance infrastructure called SIEMA [20]. SIEMA is different from the other deployment based on The Ottawa Hospital the way it uses Valet SADI. In this surveillance framework, for malaria intervention, access to distributed data resources in relational databases across various regions are made possible by deploying SADI services. The rapid deployment of SADI services on top of these databases is possible due to Valet SADI’s ability to create services quickly. A feature of this type of deployment is that resources tend to evolve over time and services may fail to return data if they are not updated to cope with the changes. This results in a loss of semantic interoperability among the resources in SIEMA. In order to preserve interoperability, Valet SADI is used a second time to regenerate and re-deploy the corrupted services such that the service-downtime is minimized and the surveillance system remains operational.
1.3 Thesis Outline

The rest of the thesis is organized as follows:

In Chapter 2, we provide the background and resources required for the rest of the thesis. We review the existing Semantic Web services from the viewpoint of SADI Semantic Web services. We briefly describe PSOA RuleML as the choice for semantic mapping rules. Moreover, we describe the relationship between relational databases and ontologies in Semantic Web.

In Chapter 3, we introduce the Valet SADI architecture with a compact diagram. We then expand the diagram and describe in modules in detail, each with their assigned tasks.

In Chapter 4, we document a number of algorithms developed and how existing resources were used to implement the operations of each module.

In Chapter 5, we build 11 SADI services using Valet SADI to demonstrate the effectiveness of the framework. We deployed the services in a registry on top of an extract of The Ottawa Hospital data warehouse. We show that Valet SADI took merely a few seconds on an average to build each of these services once semantic mapping rules were available and the inputs and outputs of the services were defined.

In Chapter 6, we illustrate how services can be used and invoked from a SPARQL query to return answers about infections from the database to the end users. We present the query in raw SPARQL syntax to run on the SHARE query client. We present the same query in graphical form available
on the HYDRA query client, which is translated to a raw SPARQL form automatically, and run the query on the client.

In Chapter 7 we present the SIEMA platform for malaria surveillance and intervention which was built with Valet SADI, a reporting tool presented as a dashboard, and software agents. We demonstrate how SADI services built by Valet SADI are used to access distributed malaria data across various regions. We then demonstrate what happens to SIEMA if any of the resources evolve, effectively rendering the services unable to return data. We show how the dashboard reports loss of interoperability of the surveillance system due to evolving changes in the system. Finally, we show how Valet SADI can be used in order to restore the surveillance operations by regenerating the services and, as a result, preserves the interoperability of SIEMA.

In Chapter 8 we summarize the contributions, defend our claims regarding the advantages Valet SADI provides, look at the open problems of the architecture, and list future work.
Chapter 2

Preliminaries

In this chapter, the resources and technologies underlying the work in this thesis are introduced. Apart from the core concepts, the chapter also addresses some related works and makes comparisons with some of the notions mentioned in this work.

2.1 The Web

The Web was created as a distributed hypertext system. The distributed nature of the Web is the reason for the success that we see today. The Web is based on three main standards: URI, HTML and HTTP. URIs identify resources on the Web, HTML [21] is the cornerstone language used for creating contents on the Web and display on the browser, and HTTP is the protocol which provides methods such as GET, POST, HEAD, PUT, DELETE, for
transmitting the HTML documents to the Web browsers.

2.2 Web Service

Web services are programmable application logic which can be invoked and executed using standard Web protocols such as HTTP. Extensible Markup Language (XML) is considered as the standard format for exchanging messages by the Web services. The Web service architecture is described in Figure 2.1 in terms of a service-oriented architecture. The figure shows the essential roles of a Service Requester and a Service Provider, and an optional role of a Service Registry of a Web service for the operations they perform.

The service provider creates Web services and their definitions which can be published into a service registry. The service requester uses the service definitions to find the binding information of the necessary services from the registry. This information allows the requester to bind and invoke the services directly through the service provider dynamically. The service registry provides a searchable index of service descriptions which are published by the provider. The registry is optional because the requester can obtain the descriptions from other sources such as local files or even FTP sites.
2.2.1 Standards

There are three main Web service standards introduced during the early 2000s. In 1999, W3C working group started the discussion of using XML for distributed applications which resulted in the introduction of Web service standard SOAP in 2000. Another Web service standard WSDL was introduced in the following year in 2001. Almost around the same time, UDDI was proposed as a Web service standard destined to be interrogated by the SOAP messages and to provide access to WSDL document via its registry. These standards are briefly described below:
2.2.1.1 Simple Object Access Protocol (SOAP)

SOAP provides a lightweight protocol which uses XML technologies to exchange messages over a variety of protocols. SOAP offers syntactic service interoperability and the messages are sent to network end points which can be implemented in any platform using Remote Procedure Calls (RPC), Component Object Model (COM) objects, or Perl scripts. The features of SOAP standards were further explained in detail in [25].

2.2.1.2 Web Service Description Language (WSDL)

WSDL [26] provides a grammar in XML to describe the communication and message formats in a structured way. A service is defined as a collection of network endpoints or ports in a WSDL document. The endpoints and messages are defined in an abstract manner which allow them to be reused. A WSDL document comprises data type definitions, message, the action supported by the service, operations supported by one or more network endpoints, binding of data, combination of a binding and a network address, and a collection of endpoints. WSDL 2.0 [27] was introduced later to address the need for RESTful Web services.

2.2.1.3 Universal Description, Discovery, and Integration (UDDI)

UDDI [28] provides a platform-independent XML-based registry where i) Web service providers publish the WSDL document, ii) consumers can search for services and retrieve the WSDL document when the service is found, and
iii) consumers, after finding the document, can invoke the service using a SOAP message and receive output as a SOAP message after the service is executed. Although UDDI was proposed as means of universal service discovery, it did not take off as expected due to lack of public interest. Among the reasons are low degree of voluntary registration of services due to the fact that they lacked quality-control of the registered services and no support for active crawling.

2.2.2 Representational State Transfer (REST)

Representational State Transfer (REST) was proposed by Roy Fielding in [29]. Whereas the Web service standards mentioned above may carry out an arbitrary set of operations, Web services which are REST compliant accomplish tasks using a uniform set of stateless operations. Fielding listed a number of constraints for designing a REST architecture. Some of these are: i) the client-server architectural style which separates concerns of the participants such that user interface concerns are separate from the data storage concerns, ii) the nature of communication must be stateless to ensure that a request of a client must be self-reliant and contains all necessary information to be understood by the server, iii) the data within a response to the request must be declared explicitly or implicitly cacheable so that they can be reused later for similar requests, iv) the components should have uniform interface in order to simplify the overall architecture and improve the visibility of the interactions between these components, v) the components are allowed to be
layered hierarchically in order to restrict each component to its immediate layer during their interactions, and vi) an optional feature of the client’s ability to download and execute code in the form of applets or scripts in order to reduce the number of features to be pre-implemented.

RESTful Web services are easier to create and invoke than SOAP service because they do not have the description layers and interacting is simple due to the existing methods such as GET and POST. Although there are certain criticisms of RESTful Web service due to their lack of Quality-of-Service, security, and reliability, there are ongoing efforts to address them [30].

2.3 Semantic Web

Semantic Web offers a way of incorporating knowledge modeling and automatic deduction on the Web using standardized languages. The innovations of technology standards have been led by the World Wide Web Consortium (W3C) body and facilitated information exchange and interoperability across the boundaries of the Web.

Since early 1990s, Tim Berners-Lee has pushed for the Semantic Web with a goal of Web of data instead of the Web of documents. The roadmap for the Semantic Web was coined in an article in [31] and was later widely accepted when Scientific American published “The Semantic Web” in 2001 [32]. Since the, different bodies in W3C have recommended a number of standards which laid the foundation [33] of Semantic (Web) technologies. These standards
include Resource Description Framework (RDF) and its schema RDFS in 2004, Web Ontology Language (OWL), SPARQL Protocol and RDF Query Language in 2008, and rule-based language Rule Interchange Format (RIF) in 2013. Figure 2.2 illustrates how the existing standards XML, URI, Unicode fit in the Semantic Web layer cake diagram [34].

![Figure 2.2: Revised Semantic Web layer cake](image)

### 2.3.1 Resource Description Framework

Resource Description Framework (RDF) [35] is a formal language for describing data and allows applications to exchange them without losing their original meaning. An RDF document represents a directed graph consisting of a set of nodes having directed links among them. RDF uses Uniform Re-
source Identifiers (URI) as names of resources which is a generalization of the Uniform Resource Locators (URL). However, data values are encoded not by URIs but as literals of a certain datatype. RDF also features so-called blank nodes which do not carry any name. Data in RDF is represented by assertions in the form of a subject-predicate-object called a triple. Collectively, the triples form graphs where the subjects and the objects are the nodes and the predicates link them.

Notation 3 (N3) [36], N-Triples, and Turtle are among the widely used syntaxes to represent a RDF document. The N3 syntax, proposed by Tim Berners-Lee in 1998, is considerably more complex with features like path and rules, while the latter two are simpler and essentially part of N3. Although these syntaxes are easy to comprehend, not all programming languages provide support for them while there is abundant support for XML such as the XML-based serialization RDF/XML.

RDF Schema language (RDFS) [37] complements RDF by adding schema knowledge to the data. The vocabularies used by RDFS are merely generic language constructs to semantically enrich the RDF data but are not topic-specific. The vocabularies have semantics containing hierarchical information of classes and properties as well as property restrictions on domain and range, which allow RDFS documents to be machine-processable.

Ontologies are another Semantic Web mechanism for describing vocabularies using one of the Web Ontology Languages (OWLs) which allow more expressive ways to define a domain of interest than that of RDFS. The foundation
of ontologies are based on Description Logics, most of which are decidable subsets of First-Order Logic built on top of Propositional Logic. Some of these logics and the associated tools relevant to this thesis based on these logics are briefly described below.

### 2.3.2 Propositional Logic

Propositional Logic \([38]\) is a subset of Formal Logic that deals with atomic propositions which cannot be decomposed further. Each proposition has a truth value, either *true* or *false*, and depends on the truth values of the proposition symbols in combination with the logical operators of the proposition. The language of propositional logic has the alphabet consisting of

- proposition symbols \(p, q\) etc
- logical connectives \(\land\) (conjunction), \(\lor\) (disjunction), \(\rightarrow\) (implication), \(\leftrightarrow\) (equivalent or bi-implication), \(\neg\) (negation), \(\bot\) (falsity)
- auxiliary symbols (, )

### 2.3.3 Horn Logic

A horn clause \([38]\) is a clause which contains at most one positive literal. The general formula of such a clause can be represented as follows:

\[
\neg p_1 \lor \neg p_2 \lor \ldots \lor \neg p_n \lor q
\]
which can be rewritten as a horn logic implication as follows:

\[(p_1 \land p_2 \land \ldots \land p_n) \implies q\]

Thus a horn clause can be expressed i) as a rule with one positive literal and at least one negative literal, ii) a fact with one positive literal and zero negative literal, and iii) a negated goal with zero positive literal and at least one negative literal.

Horn rules are a tractable subset of FOL and efficient procedures have been developed for reasoning over a knowledge base built from them. The semantics of Horn logic is based on the minimal Herbrand model as well as procedural semantics based on the SLD-resolution \[39\].

### 2.3.4 First-Order Logic

First-Order Logic (FOL) \[38\] which is also called Predicate Logic, is built on top of Propositional Logic. Atomic propositions in propositional logic are interpreted as predicates in FOL. FOL consists of terms, predicates, logical quantifiers \(\exists\) and universal quantifiers \(\forall\), where a term can be a variable, a constant or a function. These are the building blocks of a FOL formula and from where the truth value of a FOL formula is deduced. FOL is more expressive than propositional logic and it allows us to describe properties of objects and relations between objects in interpreting constants as objects, predicates as relations and functions as functional relations.
A FOL Horn clause is a disjunction of literals with one positive literal, with all variables universally quantified, which can be rewritten into a rule-like form as shown above. The function-free First-Order Horn logic is known as (monotonic) Datalog [40].

2.3.5 TPTP First-Order Formula

Thousands of Problems for Theorem Provers (TPTP) [18] is a collection of test problems for automated theorem provers which has a syntax for its intuitive readability and strength to encode all kinds of problems. The TPTP language is based on a core language which is built upon Church’s simple theory of types called THFO [41] and it supports four different languages: First-Order Form (FOF) and Clause Normal Form (CNF) for untyped First-Order Logic, typed First-Order Form (TFF) and Typed Higher-Order (THF). A TPTP-FOF problem is a list of annotated formulas of the form:

```plaintext
fof(name, role, formula, source, useful info)
```

where name is a given name to the formula, role describes the role of the formula such as ‘axiom’ or ‘theorem’ or ‘conjecture’, formula is the encoding of the formula in the FOL language, and the rest are optional. Table 2.1 lists typical symbols and logical meanings of the constructors of TPTP-FOF.
2.3.6 VampirePrime Reasoner

VampirePrime is an open source reasoner originating from Vampire [42], a mature high-performance reasoner for full FOL. It supports standard tasks such as checking consistency and entailment.

2.3.7 Description Logic

Description Logics (DLs) were originally designed to provide a precise semantic characterization in a formal and structured way for network-based systems [43]. DLs are expressed in predicate level with no variables present in the formalism which is more understandable to human users than FOL. Most of the DLs are decidable subsets of FOL.

The $\textit{SHOIN}$ language is an expressive DL which with inverse roles $\mathcal{I}$ and is restricted to unqualified number restriction $\mathcal{N}$ [44].

Let $\mathbf{A}$, $\mathbf{R}_A$ and $\mathbf{I}$ be pairwise disjoint sets of concept names, abstract role names, and individual names. The set of roles are defined by $\mathbf{R}_A \cup \{R^- | R \}$.
A knowledge base $\mathcal{K}$ is a finite set of axioms in $\text{SHOIN}$. For two roles $S$, $R$ in $\mathcal{K}$, the transitive reflexive closure of $\sqsubseteq$ on roles is defined as $S \sqsubseteq^* R$ in $\mathcal{K}$ if $S = R$, $S \sqsubseteq R \in \mathcal{K}$, $\text{Inv}(S) \sqsubseteq \text{Inv}(R) \in \mathcal{K}$, or there exists some role $Q$ such that $S \sqsubseteq^* Q$ in $\mathcal{K}$ and $Q \sqsubseteq^* R$ in $\mathcal{K}$. A role $R$ is called simple in $\mathcal{K}$ if for each role $S$ such that $S \sqsubseteq^* R$ in $\mathcal{K}$, $\text{Trans}(S) \notin \mathcal{K}$ and $\text{Trans}(\text{Inv}(S)) \notin \mathcal{K}$, where $\text{Trans}(S)$ refers to object role transitivity for $S$ (see Table 2.2).

The semantics of $\text{SHOIN}$ is defined using an interpretation $\mathcal{I} = (\Delta^\mathcal{I}, \cdot^\mathcal{I})$ consisting of a non-empty domain $\Delta^\mathcal{I}$ and a mapping $\cdot^\mathcal{I}$, which interprets atomic and complex concepts, roles and nominals (i.e. individual constants).

The symbol $\#I$ means set cardinality.

The interpretation $\mathcal{I}$ satisfies $\mathcal{K}$ if and only if it satisfies each axiom in $\mathcal{K}$. A $\text{SHOIN}$ concept $C$ is satisfiable with respect to $\mathcal{K}$ if and only if there is an interpretation $\mathcal{I}$ with $C^\mathcal{I} \neq \emptyset$ that satisfies $\mathcal{K}$. A concept $C$ is subsumed by a concept $D$ with respect to $\mathcal{K}$ if and only if $C^\mathcal{I} \sqsubseteq D^\mathcal{I}$ for each $\mathcal{I}$ satisfying $\mathcal{K}$.

Two concepts are equivalent with respect to $\mathcal{K}$ if and only if they subsume each other with respect to $\mathcal{K}$. A knowledge base $\mathcal{K}_1$ entails a knowledge base $\mathcal{K}_2$ if and only if every interpretation of $\mathcal{K}_1$ is also an interpretation of $\mathcal{K}_2$.

The syntax and semantics of $\text{SHOIN}$ are shown in Table 2.2.
<table>
<thead>
<tr>
<th>Constructor Name</th>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic concept</td>
<td>$A$</td>
<td>$A^I \subseteq \Delta^I$</td>
</tr>
<tr>
<td>abstract role</td>
<td>$R$</td>
<td>$R^I \subseteq \Delta^I \times \Delta^I$</td>
</tr>
<tr>
<td>individuals</td>
<td>$o$</td>
<td>$o^I \subseteq \Delta^I$</td>
</tr>
<tr>
<td>inverse role</td>
<td>$R^-$</td>
<td>$(R^-)^I = (R^I)^-$</td>
</tr>
<tr>
<td>conjunction</td>
<td>$C_1 \cap C_2$</td>
<td>$(C_1 \cap C_2)^I = C_1^I \cap C_2^I$</td>
</tr>
<tr>
<td>disjunction</td>
<td>$C_1 \cup C_2$</td>
<td>$(C_1 \cup C_2)^I = C_1^I \cup C_2^I$</td>
</tr>
<tr>
<td>negation</td>
<td>$\neg C_1$</td>
<td>$(-C_1)^I = \Delta^I \setminus C_1^I$</td>
</tr>
<tr>
<td>oneOf</td>
<td>${o_1, \ldots}$</td>
<td>${o_1, \ldots}^I = {o_1^I, \ldots}$</td>
</tr>
<tr>
<td>exists restriction</td>
<td>$\exists R.C$</td>
<td>$\exists R.C)^I = {x \mid \exists y. \langle x,y \rangle \in R^I \text{ and } y \in C^I}$</td>
</tr>
<tr>
<td>value restriction</td>
<td>$\forall R.C$</td>
<td>$(\forall R.C)^I = {x \mid \forall y. \langle x,y \rangle \in R^I \rightarrow y \in C^I}$</td>
</tr>
<tr>
<td>atleast restriction</td>
<td>$\geq n R$</td>
<td>$(\geq n R)^I = {x \mid #({y. \langle x,y \rangle \in R^I}) \geq n}$</td>
</tr>
<tr>
<td>atmost restriction</td>
<td>$\leq n R$</td>
<td>$(\leq n R)^I = {x \mid #({y. \langle x,y \rangle \in R^I}) \leq n}$</td>
</tr>
<tr>
<td>Axiom Name</td>
<td>Syntax</td>
<td>Semantics</td>
</tr>
<tr>
<td>concept inclusion</td>
<td>$C_1 \sqsubseteq C_2$</td>
<td>$C_1^I \subseteq C_2^I$</td>
</tr>
<tr>
<td>object role inclusion</td>
<td>$R_1 \sqsubseteq R_2$</td>
<td>$R_1^I \subseteq R_2^I$</td>
</tr>
<tr>
<td>object role transitivity</td>
<td>$\text{Trans}(R)$</td>
<td>$R^I = (R^I)^+$</td>
</tr>
<tr>
<td>individual inclusion</td>
<td>$a : C$</td>
<td>$a^I \in C^I$</td>
</tr>
<tr>
<td>individual inclusion</td>
<td>$\langle a, b \rangle : R$</td>
<td>$\langle a^I, b^I \rangle \in R^I$</td>
</tr>
<tr>
<td>individual equality</td>
<td>$a = b$</td>
<td>$a^I = b^I$</td>
</tr>
<tr>
<td>individual inequality</td>
<td>$a \neq b$</td>
<td>$a^I \neq b^I$</td>
</tr>
</tbody>
</table>

Table 2.2: Syntax and semantics of \(SHOIN\)

### 2.3.8 Web Ontology Language

While an RDFS document allows for the derivation of implicit knowledge, it has a limited expressiveness such as not being able to model cardinality restrictions. The Web Ontology Language (OWL) is the recommended standard for modeling of ontologies on the Web. OWL is based on Description Logics (DLs) \[45\] which are the formal knowledge representation language.

An OWL ontology is expressed using classes and properties (or roles) as in RDFS and individuals as RDF to define instances of classes, which are the
basic building blocks for modeling an ontology. A property has a domain and range. Properties can be either object properties (link to other individuals) or data properties (have literal values). Properties in OWL can be functional, inverse, transitive and symmetric. OWL enables complex class expressions using restrictions. Classes can be defined using set operators, constraints on properties (cardinality, range, value), and universal and existential restrictions.

There are two main specifications of OWL and both are W3C recommendations: OWL 1.0 in 2004 [46] and OWL 2 in 2009 [47]. OWL 1.0 has three sublanguages which have various degrees of expressiveness. They are OWL Full, OWL DL, and OWL Lite. While OWL DL contains only OWL Lite, OWL Full contains both OWL DL and OWL Lite.

1. **OWL Lite**: This is the least expressive language of the three, is decidable, and its worst-case computational complexity is ExpTime. It supports named classes, named properties, individuals, property values, intersection, union, negation, existential and universal value restriction, and unqualified number restrictions.

2. **OWL DL**: Contains OWL Lite, hence more expressive. It is also decidable but its worst-case computational complexity is NExpTime. Including all OWL Lite features, it supports arbitrary number restrictions, property value and enumeration.

3. **OWL Full**: Contains the above two languages, hence it is the most
expressive of all. However OWL Full does not restrict the way they are used, as a result there is no computational guarantee.

OWL 2 is more recent and is fully backward compatible with OWL 1.0, but is more expressive. There are three OWL 2 profiles which are sublanguages of OWL 2 DL. They are:

1. **OWL EL**: It supports ontologies with a large number of classes and properties and reasoning can be performed in a polynomial time with respect to the size of the ontology.

2. **OWL QL**: It supports efficient query answering in applications that have large volumes of instance data.

3. **OWL RL**: Restricts modeling constructs, so this lightweight ontology language resembles an OWL-based rule language, which can operate directly on the RDF serialisation of OWL. This allows the implementation of OWL RL reasoning over existing RDF databases that support some form of production rules for inferencing.

### 2.3.8.1 OWL Development Environment

As ontologies are gaining much attention across various domains, researchers have been working on a wide range of ontology development tools. Among them OntoStudio [48], protégé [49], Swoop [50], TopBraid [51] are worth mentioning. However, protégé has been supported by a strong community
of academic, government, and corporate users over the years because of the active community, W3C support and extensible open source environment. Among the syntaxes protégé uses the Manchester OWL Syntax [52] for all expressions. The expression editor allows a user to create class expressions very easily, especially because of an autocomplete feature. The input and output of each SADI service are defined as OWL class expressions in a service ontology. Throughout the rest of the document, whenever the input and output of a SADI service are defined, they are assumed to be authored in protégé in Manchester OWL Syntax.

2.3.9 PSOA RuleML

Positional-Slotted Object-Applicative (PSOA) RuleML [53, 54] is an object-relational rule language generalizing Positional-Slotted Language (POSL) [55], Object-Oriented (OO) RuleML [56], Frame logic (F-logic) [57], and Rule Interchange Format-Basic Logic Dialect RIF-BLD [58]. PSOA RuleML provides positional-slotted object-applicative (psoa) atoms, integrating object identifiers with applications of predicates to positional and/or slotted arguments.

The characterization of PSOA RuleML given here has been specialized for our purpose. The language introduces psoa atoms including the following ones, where memberships are a special case of the form o#f(), abbreviated

\footnote{For PSOA’s complete evolution see $\text{http://wiki.ruleml.org/index.php/PSOA_RuleML#References}$}

27
as $o^#f$:

$$o^#f([t_1, \ldots t_{n_1}] + [t_{n_1}, \ldots t_{n_m}] \ p_1 -> v_1 \ \ldots \ p_k -> v_k)$$

In a psoa atom, $o$ is the Object IDentifier (OID) which uniquely identifies the object described by the tuples and/or slots.

A psoa atom integrates three types of information: 1) The class membership $o^#f$ makes $f$ the type of instance $o$; 2) every tupled argument $[t_{i_1}, \ldots t_{i_{n_1}}]$ associates $o$ with a sequence of terms; 3) every slotted argument $p_i -> v_i$ associates $o$ with an attribute $p_i$ and its value $v_i$.

PSOA RuleML uses such generalized (psoa) atoms as facts, as/in conclusions of rules, and as/in conditions (premises of rules or top-level queries).

The conditions are formulas, which are used as queries or premises in rules. Conjunctions and disjunctions of formulas are indicated by $\text{And}$, and $\text{Or}$, respectively. Existentially quantified variables are denoted by $\text{Exists}$ while the list of free variables is preceded by $\text{Var}^+$. An atomic formula is a (psoa) atom, an equality formula, or a subclass formula. A (psoa) term is a constant, variable, expression, or $\text{External}$ expression.

A clause consisting of an atomic formula is a fact. A clause that is an implication constitutes a rule with an atomic formula or a conjunction of atomic formulas for its conclusion and an atomic formula or a conjunction or disjunction of formulas for its condition. Conclusions can have existentially ($\text{Exists}$-)quantified OID variables. An $\text{Assert}$ wraps a knowledge base con-
taining any number of clauses (rules and/or facts).

We would like to recall the example *Rule-defined anonymous family frame* (updated from Example 1 in [53]) to discuss a typical rule and two facts. The **Assert** contains a universally quantified implication formula, followed by two ground facts. The **Forall** quantifier around the implication declares universal argument variables ?Hu, ?Wi, ?Ch (given in the left-hand side source version) as well as universal OID variables ?2, ?3, ?4 (generated for the right-hand side objectified version). The **Exists** quantifier (generated for the objectified version) around the conclusion declares an existential OID variable ?1. The infix :- separates the conclusion from the condition, deriving an OID-existential family frame from a *married* relation and from a *kid* of the *husb* or *wife*.

```
Assert ( 
  Forall ?Hu ?Wi ?Ch ( 
    family(husb->?Hu wife->?Wi child->?Ch) :- 
      And(married(?Hu ?Wi) 
        Or(kid(?Hu ?Ch) kid(?Wi ?Ch))) ) 
  married(Joe Sue) 
  kid(Sue Pete) )
)
```

```
Assert ( 
    Exists ?1 ( 
      ?1#family(husb->?Hu wife->?Wi child->?Ch)) :- 
      And(?2#married(?Hu ?Wi) 
        Or(?3#kid(?Hu ?Ch) ?4#kid(?Wi ?Ch))) ) 
    _1#married(Joe Sue) 
    _2#kid(Sue Pete) )
)
```

### 2.3.10 Linked Data

The term *Linked Data* was coined in Tim Berners-Lee’s design note in 2006 [59] which refers to a set of best practices for publishing structured data on the Web. These are: using URIs as names of *things*, using HTTP URIs
for looking up those names, returning useful information when the names are looked up, including links to other URIs so that more things can be discovered.

The note also suggests rating of publishing Linked Data to determine the quality. In recent years the amounts of Linked Data that are available on the Web have grown considerably and data sources have become increasingly inter-linked to form a Linked Data Cloud [60].

2.3.11 FAIR Data

FAIR data principles [61] are a set of guidelines and best practices for making objects and the associated metadata open and reusable. The acronym FAIR is the abbreviation of Findable, Accessible, Interoperable, and Reusable, respectively. The principles for data to be FAIR compatible are briefly described below:

- Data should have rich metadata with unique and persistent identifier, and stored in a searchable registry so that they are findable.

- Data should be retrievable by their identifier with proper authentication using an open protocol. Even if data are no longer available, the metadata should be accessible.

- In order to attain interoperability, metadata should use a formal, accessible, shared and broadly applicable language for knowledge representation.
• In order to be reusable, data should have a clear license for use and should provide accurate information on provenance.

2.3.12 SPARQL Protocol and Query Language

SPARQL is a query language alike SQL but for querying data that can be mapped to RDF. The SPARQL specification \textsuperscript{62} is a widely adopted W3C recommendation. Queries in SPARQL contain a graph pattern (a set of triples containing variables) and when processed, matching RDF graphs are returned from the RDF dataset. It provides a number of advanced functions for constructing advanced query patterns, for stating additional filtering conditions, and for formatting the final output. The named graph support of SPARQL allows an input dataset to be separated into multiple parts where each can be named and accessed by URIs. SPARQL offers a number of options for working with these graphs and allows one to use the URIs of such graphs within queries.

2.4 Semantic Web Services

The journey towards implementing Semantic Web services is realized best by Figure 2.3. One way is along the vertical path via Web Services, by complementing e.g. WSDL with Semantics. The other way is along the horizontal path via Semantic Web, using only standards of the Semantic technologies such as OWL, RDF(S).
The current Web service standards and their uses are limited because the functionality they provide are expressed using syntactic descriptions. The syntactic descriptions are not suitable for software agents to perform automatic discovery, orchestration, and execution which are done mostly manually. As the contents and services are not marked up semantically, they lack support for the vision towards Semantic Web.

The vision for Semantic Web Services (SWS) is to utilize semantic technologies to achieve automatic discovery, invocation, composition and execution of Web services [63][64]. In order to achieve this goal Web services need to be augmented or marked up with semantic descriptions so that they can be interpreted and reasoned about by semantic-aware software clients.

Cabral et al. list three key criteria in [65] for Semantic Web services: i) activities which include publishing, discovery, selection, composition, invoca-
tion, deployment and ontology management, ii) architecture which includes a register, a reasoner, a matchmaker, a decomposer and an invoker, and iii) service ontology which may include input, output, precondition, post-condition, cost, category, atomic service and composite service.

For Semantic Web services the activities and the service ontologies are the two most important factors because the service ontology describes the elements of the services’ semantic descriptions and the activities are among the features a service needs to implement regardless of the architecture.

The following list briefly defines the activities:

- Publishing refers to the advertising of description of a service’s capability in a registry.

- Discovery refers to the event of matching a given query to the services which depends on their semantic descriptions. The descriptions typically involve name, input, output, preconditions and postconditions.

- Selection refers to the event of choosing between two or more matching services, based on other criteria, such as cost or category.

- Composition or choreography refers to the automatic or semi-automatic composition of larger services from other services, and the control of how that composition is executed.

- Invocation refers to the actual invocation of the service, like preparing inputs and dealing with exceptions.
• Deployment refers to mechanism for deploying an instance of a service in a registry which may be independent of the publishing of its semantic description.

• Ontology management refers to the management of the domain and service ontologies in terms of accessibility, updating and maintenance.

In this thesis, the Semantic Automated Discovery and Integration (SADI) framework is chosen for creating Semantic Web services. A brief description of SADI services comes next.

2.4.1 SADI Semantic Web Services

The RESTful Semantic Automated Discovery and Integration (SADI) Web services exchange RDF data and OWL expressions are used to define their functionality. SADI services and SADI-related tools use the myGrid/Moby service ontology to extend the service descriptions with service name, description, input, output etc. and encode the service metadata graph with them so that software clients can discover and invoke the services.

2.4.1.1 Motivation

The original motivation for the development of SADI was the complexity of discovering, accessing, and integrating public data and software in the domain of bioinformatics. RDF enables automated merging of data sets and OWL enables automated logical reasoning over data. The SADI framework
provides a set of best practices for publishing data and software resources that would simultaneously offer the benefits of Semantic Web standards and technologies, such as the ability to automatically merge data sets and to automatically compute logical inferences from data. This way, the problem of incompatible data models, schemas, and software interfaces which hinders the combined use of numerous bioinformatics databases and software tools available online, will be solved to greater degree. Although SADI has been motivated by bioinformatics, it is applicable to other domains.

2.4.1.2 Operation

A SADI service operates by attaching new properties to an input URI described with an input RDF graph ensuring related input and output instances from a service invocation are explicitly linked. This explicit relationship between the service’s input and output saves a client from the task of tracking input-output relationships on its own, and ensures that the RDF produced by service invocations forms a connected graph that is queryable in a meaningful manner. In the input and output descriptions, the predicates in the property assertions are fixed for a SADI service. These specifications, known as the semantic description of the service, specify the service functionality. For instance, a SADI service having an output predicate has_facility_description in an ontology may be invoked by the query clients to retrieve the description of the facility to which the patient is admitted.

Metadata about a service is retrievable as an RDF document by issuing
an HTTP GET to the service URL, while service invocation is realized by issuing an HTTP POST to the service URL with an input RDF document as the request body. The response to a service invocation is likewise an RDF document. SADI also imposes minimal constraints on data modeling as it does not provide any rules about how service input and output data should be modeled in RDF. Thus, service providers are free to encode the data using any OWL or RDFS ontologies deemed suitable. Generally, the DL family of OWL has been recommended.

Within the RDF metadata document for the service, all metadata items must be represented as part of a single, connected RDF graph whose root URI is the URL of the service. The metadata graph must include the following items:

- An input OWL class describing the expected structure of the input RDF graphs consumed by the service
- An output OWL class describing the expected structure of the output RDF graphs generated by the service

Among others, the metadata contains service name as a human readable label for the service and a human readable description of the service functionality. SADI services may use the myGrid/Moby\(^2\) service ontology to encode the service metadata graph.

\(^{2}\text{http://www.mygrid.org.uk/mygrid-moby-service/}\)
The ability to describe service interfaces using OWL classes provides a machine-readable representation of the service interface which allows automation of various data and service matchmaking tasks such as service identification, input extraction, and forming execution chains of services including reasoning. The primary purpose of the input OWL class is to identify and extract valid input instances for a service from a given RDF data set. Each SADI service has exactly one input OWL class which must either be referenced by or directly included in the metadata graph for the service. The most important consideration when authoring an input OWL class is that the conditions for class membership should be defined using necessary and sufficient (‘if and only if’) conditions. In OWL, necessary conditions (‘if’) are defined using the rdfs:subClassOf property whereas necessary and sufficient conditions (‘if and only if’) are defined using the owl:equivalentClass property. The output OWL class provides a machine readable description of the output instances produced by a service, which facilitates the automated identification of services that produce data of interest to either human or software clients. Each SADI service has exactly one output OWL class which must be either referenced by or directly included in the metadata graph for the service. Each input instance that is sent to a service produces exactly one output instance, and each pair of corresponding input and output instances always have the same URI.

Figure 2.4 shows the transaction steps for a simple SADI service, which retrieves the name of the disease when a patient name is given. A series of
transactions occurred between the client (on left) and the service endpoint (on right) using GET and POST. In A, the client calls HTTP GET on the service endpoint at \texttt{http://cbakerlab.unbsj.ca:8080/getDiagnosisByPatientId}. In B, the service description is returned with a reference to the input and output of the service. In C, the client sends the input data value ‘1’ as the ID of the patient to the service endpoint using HTTP POST. The root node \texttt{http://cbakerlab.unbsj.ca:8080/pat_by_ID?ID=1} which is of type (\texttt{rdf:type}) \texttt{Patient} is shown as the input graph. In D, the service strips the data value, executes the services to find the string-valued \texttt{Arthritis} as the diagnosis for the patient, attaches it using the property \texttt{has_diagnosis}, and returns to the client as the output.
2.4.1.3 Limitation

According to the SADI design-pattern, the unique constraint is that the URI of the input individual and the URI of the output individual must be identical. Due to this reason, a typical SADI service is in the form of get YByX, where the output Y is retrieved based on the input X and is then
connected to the input, decorating $X$ using an explicit relationship.

Although services in the form of $\text{get} Y \text{By} X$ meet most demands, there are cases where no input $X$ is required. In a prior work, Al Manir et al. demonstrated in [68] that a set of SADI services were deployed where all $Y$ services were used to retrieve all instances of an attribute $Y$ from a table of a relational database. The work identified the need for SADI services in the form of all $Y$ where no inputs are required to be present. While these all $Y$ services were created manually, they still suffer from similar problems in generating $\text{get} Y \text{By} X$ services which are error-prone and time-consuming.

2.4.1.4 Querying SADI Services

A service registry allows users to query metadata about a collection of services, in order to discover services that accomplish a task. SHARE [69] is a specialized open-source query engine which allows end users to discover, orchestrate, and invoke SADI services from SPARQL queries. SHARE does not search an existing RDF dataset for a subgraph with a specified triple pattern structure. Instead, SHARE utilizes the SADI framework by matching individual triple patterns in the SPARQL query against SADI service interface descriptions in OWL to discover services capable of generating those triple patterns. Thus the RDF data required to answer any given SPARQL query is dynamically generated in response to the query. Note that while distributed SPARQL endpoints may provide some data relevant to the query, other data may be dynamically generated through execution of an application or algo-
rithm since all resources are exposed as Web services, thus making the nature of the underlying data resource opaque to the client.

The services are discovered by finding a match to the predicates attached to the input URIs after comparing both the semantic descriptions of the service input and the output OWL class. Each triple pattern in the SPARQL query is resolved by checking the predicates in the service registry. The discovered services are then orchestrated and invoked to execute the desired tasks. Even if the inverse of a predicate is used, SHARE is still able to discover the service by looking at the ontology. The output triples are produced through the binding of subject/object values to the corresponding variables in the SPARQL query. To generate the complete output every triple in the query is resolved.

HYDRA is a commercial query engine for SADI services, developed by IP-SNP Computing Inc. HYDRA can be used as a Java API, a simple command line application or through an intelligent GUI supporting ad hoc query composition by non-technical users by combining Google-style keyword-based querying with query graph editing.

2.4.1.5 Tools used for Adopting SADI Services

Since its inception, SADI has been employed in a number of Semantic Knowledge management tools and there are others emerging because of its simplicity.

http://ipsnp.com/hydra/
The IO Informatics’s Sentient Knowledge Explorer[4] is a retrieval, integration, visualization, query, and exploration environment for semantically rich data. The SADI plug-in allows the tool to discover SADI services with the help of Semanticscience Integrated Ontology (SIO) [70]. By converting the input OWL class definition into an equivalent SPARQL query and with the help of SIO-compliant metadata the services are discovered.

Another well known adoption of SADI is the semantically guided workflow management system Taverna [71]. Areas where Taverna has been include astronomy, bioinformatics, biodiversity, and digital preservation. The SADI plugin allows SADI services to be accessed from Taverna. The plugin allows users to build workflow and interactive service discovery. From the SADI services in the registry, Taverna can search for the service that consume the datatype as an output of the existing workflow node using output ports.

Another workflow management system which is becoming increasingly popular among biologists is Galaxy [72]. Galaxy is a Web server and it allows a user to store and analyse data as well as send the output of one tool as input to the next. This process facilitates creating workflows which then can be extracted. With the SADI-Galaxy tool [73], a SADI service becomes a Galaxy tool that has features such as data storage, history, workflow creation, and publication. The semi-automated packing and unpacking of data into RDF keeps provision for other Galaxy tools to be combined with the

An attempt to automatically configure a large number of SADI services was demonstrated in [74]. More than 22,000 SADI services have been made available through the utilization of descriptive metadata on top of OpenLifeData—a well-curated Linked dataset based on enriched Bio2RDF’s RDFS semantics to OWL expressivity. Each triple pattern for the dataset becomes a SADI service which requires some pre-processing with the help of configuration files. This technique enables the generation of SADI services automatically and the application is limited to Linked Data.

Except for OpenLifeData, most tools require that the services are to be created and deployed in the service registry. In most cases the data sources are Linked Data in RDF format and service creation is a tedious process which is error-prone and time consuming. As most of the data on the Web are still in legacy form e.g. Relational Databases, adoption of SADI could get a wider audience other than the biology community if service creation can be automated in large scale and RDBs could be a point of focus as much as Linked Data.

2.4.1.6 Manual and Automated Deployments

Over the years, a number of systems have been deployed based on SADI services. The resources used for these deployments include but are not limited to scientific literature, statistical and financial data in agriculture, biomedical images, data on Hospital-Acquired Infections, data on malaria intervention...
and control, toxicological data on fish and ecology, data on mutation impact, data on drug-drug interactions in personal health, lipidomics data, chemoinformatics data, and Linked Data in life sciences.

To deploy SADI service-based systems with these resources, a number of SADI services were either created manually or automatically. In some of these systems where services were generated automatically, the source data were in the form of Linked Data and not the relational databases. With Valet SADI, we have deployed a few systems where services were generated on top of RDBs, not on Linked Data. Example of such deployments were done for agriculture, the new version of clinical intelligence and malaria analytics. Any other deployments where services were generated automatically were done on top of Linked Data. Table 2.3 shows the domain on which SADI services were deployed, their references, and whether services were created manually or automatically or both.

2.4.1.7 Required Resources and Competencies

Certain resources and competencies are required to generate a SADI service. Typically, SADI services can be used as data services for accessing structured and unstructured data and algorithmic services for performing specific computations over a set of data. As our work involves source data that reside in relational databases, our focus remains on the list of resources and competencies for generating data-specific SADI services.
Table 2.3: SADI Service-based system deployments

<table>
<thead>
<tr>
<th>Domain of Implementation</th>
<th>References</th>
<th>Service Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agriculture</td>
<td>68</td>
<td>Manual + Automatic</td>
</tr>
<tr>
<td>Clinical intelligence (older version)</td>
<td>11, 12</td>
<td>Manual</td>
</tr>
<tr>
<td>Clinical intelligence (new version)</td>
<td>75, 76</td>
<td>Automatic</td>
</tr>
<tr>
<td>Malaria analytics</td>
<td>20</td>
<td>Automatic</td>
</tr>
<tr>
<td>Mutation impact text-mining</td>
<td>13</td>
<td>Manual</td>
</tr>
<tr>
<td>Lipidomics</td>
<td>14</td>
<td>Manual</td>
</tr>
<tr>
<td>Ecotoxicology</td>
<td>17</td>
<td>Manual</td>
</tr>
<tr>
<td>Fish toxicology</td>
<td>16</td>
<td>Manual</td>
</tr>
<tr>
<td>BIO2RDF Linked Data</td>
<td>74</td>
<td>Automatic</td>
</tr>
<tr>
<td>Biomedical image</td>
<td>77</td>
<td>Manual</td>
</tr>
<tr>
<td>Chemioinformatics</td>
<td>15</td>
<td>Manual</td>
</tr>
<tr>
<td>Personal health</td>
<td>78</td>
<td>Manual</td>
</tr>
</tbody>
</table>

Knowledge of Ontology Modeling: The input and output definitions of a SADI service are expressed in OWL in a service ontology. The vocabularies used in these definitions are referenced in terms of the domain ontologies. If certain vocabularies are missing, they need to be created in the domain ontology so that they can be reused in the definition in the service ontologies. Moreover, logical connectives in axioms can be used to impose restrictions on these vocabularies. Therefore, having expertise in ontology modeling is critical for defining a SADI service.

Knowledge of Data Modeling: While a SADI service only consumes RDF as input and produces RDF as the output, the source data is stored in relational tables in the database. Hence, it is necessary to know the modeling of both the relational schema as well as the RDF data model.
**Knowledge of a Database Query Language:** All source data are stored in relational tables and accessing these tables requires expertise in SQL query language. Typically, database programmers are familiar with database schema and their expertise is needed to write SQL queries to retrieve data from the database.

**Knowledge of a Programming Language:** The program code required to implement a SADI service can be divided in three blocks: i) input code retrieving the input values from the RDF input, ii) code with an instantiated SQL query statement with the values retrieved in the previous block, and iii) output code creating the RDF output instance using the results after executing the SQL query in the previous block. Writing these code blocks requires expertise in programming languages such as Java, Perl, and Python.

**Knowledge of SPARQL Query Language:** Once the services are generated and deployed, they can be invoked and executed from SADI-specific query clients such as SHARE or HYDRA. The clients execute W3C standard SPARQL queries submitted by the end users. Statements in a SPARQL query are resolved and subsequently the SADI services are discovered, invoked, orchestrated, and executed in order. This is why knowledge of SPARQL queries are necessary in order to return the results to the end users.
2.4.2 WSDL-based and REST-based Services

We will focus our attention to service-oriented implementations of SWS, namely WSDL-based and REST-based services. Although SADI is a REST-based service-oriented approach, comparison with WSDL-based approaches will be addressed wherever possible.

2.4.2.1 WSDL-based Approach

Semantic Annotations for WSDL (SAWSDL) [79], which was developed from WSDL-S [80], is a lightweight bottom-up specification for annotating WSDL service descriptions. The specification is called Semantic Annotations for WSDL and XML Schema. SAWSDL defines an extension layer over WSDL that allows the semantics to be attributed to various WSDL components. The attributes can be applied to elements both in WSDL and in XML Schema in order to annotate WSDL interfaces, operations and the input and output messages.

SAWSDL discards the precondition and effect attributes that were parts of WSDL-S, and it aims to be compatible with existing specifications and improve the automation of discovery and composition. In SAWSDL, the attribute sawsdl:modelReference is added to the elements of the inputs and outputs whose values would be URIs that point to concepts in an ontology. The attributes loweringSchemaMapping and liftingSchemaMapping are used to provide mappings of XML datatypes to and from a semantic data model such as RDF, respectively. The required mappings might be accom-
plished with XSLT for the lifting transformation and SPARQL followed by XSLT for the lowering transformation. SAWSDL can be used as an adaptor layer that maps a WSDL service to the expected behavior of a SADI service. A composition platform named SemanticSCo [81] can be cited as a recent attempt to support the semi-automated composition of RESTful web services semantically annotated using SAWSDL.

Web Ontology Language for Service OWL-S [82], in contrast to SAWSDL, follows a top-down approach for modeling of Semantic Web services, where the semantics of the service is described independently of the actual realization of the service implementation and then the semantics is grounded in the underlying description such as WSDL. The separation ensures that implementation details are not influencing the semantics.

OWL-S is inherently an OWL ontology comprising three interrelated sub-ontologies:

- **Profile:** Refers to the purpose of the service with functional and non-functional properties. The functional properties are used to enable advertising, construction of service requests, and matchmaking. The functional properties are represented by Inputs, Outputs, Preconditions and Effects (IOPE) properties as well as structural descriptions.

- **Process model:** Refers to the mechanism of the service execution. It specifies how a client may interact with the service for invocation, enactment, composition, monitoring and recovering from failures. The
root class \textit{Process} in the model has three subclasses: i) \textit{atomic} process which can be invoked by one message, completes execution in one step and returns one message; ii) \textit{composite} process which can be decomposed into simpler processes via control structures for different paths of execution, and iii) \textit{simple} process which are merely abstractions that can be realized by either an atomic process or a composite process but cannot be invoked.

- Grounding: Refers to the way constructs of the process model are mapped to concrete message formats and protocols. This is basically mapping from the ontological description to a concrete specification of a service, typically to WSDL.

The OWL language together with the OWL-S vocabulary makes up the OWL-S Web service specification language. The combination of OWL-S with UDDI registries facilitate service discovery [83]. OWL-S also offers automatic service composition as shown in [84, 85]. OWL-S does not contain variables which are required to specify all necessary states and descriptions in the precondition and effects of IOPE. A way to resolve this problem is to allow rule languages such as Semantic Web Rule Language (SWRL) [86] for defining rules for preconditions and effects.

Among recent contributions, a framework [87] for auto-processing information was described by extending OWL-S. In Internet of Things (IoT), Web Services are commonly used to describe the entity functions in the trans-
action process and conventional Web services cannot fully meet the needs of execution and control of the transactions. The goal of the framework is to manage, query, and combine entities and observation data using Semantic technologies so that IoTs can be directed towards the Semantic Web of Things (SWoT) \[88\]. Another OWL-S extension called OWLS-LO was proposed recently in the e-Learning domain \[89\].

Web Service Modeling Ontology (WSMO) \[90\] also follows a top-down approach for modeling Semantic Web services. The conceptual background of WSMO lies in the Web Service Modeling Framework (WSMF) which has specified a family of languages called Web Service Modeling Language (WSML) \[91\].

The four main components of WSMF are the ontologies that define the terminologies, goals as the request types, Web services, and the mediators that help in interacting among the components.

WSML-Core is defined by the intersection of Description Logic and Horn logic and has the least expressive power. WSML-DL extends WSML-Core and captures the expressiveness of the \(\mathcal{SHOIN}(D)\). WSML-Flight extends WSML-Core and uses features from OWL Full. WSML-Rule also extends WSML-Core with Horn logic based on minimal model semantics. WSML-Full combines all WSML variants under a First-Order umbrella with extensions to support specific features of WSML-Rule, such as minimal model semantics and default negation.

The Web service descriptions in WSMO consist of precondition (input in
OWL-S), postcondition (output in OWL-S), assumption (precondition in OWL-S), and effect (effect in OWL-S). Although lack of interest in recent years were noticed in WSMO, its lighter-weight variants such as MicroWSMO \cite{92} and WSMO-Lite \cite{93} are currently getting more attention.

Semantic Web Services Framework (SWSF) \cite{94} focuses on workflows and has its own language for defining the Semantic Web Services Ontology (SWSO) called Semantic Web Services Language (SWSL). Whereas OWL-S uses Description Logics, SWSL uses First-Order Logic. As a result, the SWSL is more expressive than the languages used in OWL-S and there is support for URIs, XML built-in types, XML namespaces, and XML import. SWSF provides a more comprehensive and rich behavioral process model for web services that introduces concepts for control, ordering, states and exceptions.

There are two sublanguages of SWSL having similar syntax but different semantics. One of them is SWSL-FOL which is used for the service ontologies. The other is SWSL-Rules which is used for reasoning of the use of services.

In comparison to OWL-S, WSMO, and SWSF, SADI is a simpler standard that is limited to the description of stateless services. SADI uses OWL ontologies only for defining the schema of input data, output data, and service execution parameters, and not to define the effects or choreography of a service. The choreography of a SADI service is fixed to one of two possibilities, corresponding to synchronous services and asynchronous services respectively. Both WSMO and SWSF are more expressive than SADI and OWL-S because of their use of logic programming via WSML and SWSL.
2.4.2.2 REST-based Approach

In REST-based approaches, existing RESTful Web services are semantically described.

hRESTS (HTML for RESTful Services) [92] is an HTML microformat for describing RESTful Web Services which aims to provide machine-readable representation of common Web service and API descriptions. In hRESTS, the attribute values are: service, operation, method, input and output. The microformat in hRESTS serves as the basis for extensions for both SA-REST and MicroWSMO.

Semantic Annotations for REST (SA-REST) [95] uses RDFa [96] or GRDDL [97] to embed semantic annotations into HTML/XHTML. SA-REST services provide support for interoperability of dynamic mashups. The SA-REST extension defines classes for describing the data format and programming language binding properties of Web APIs which allow developers to search in homogeneous groups and create mashups.

MicroWSMO extends hRESTS to add semantics annotations through adopting the SAWSDL extensions. WSMO-Lite [93] is a light-weight version of the WSMO service ontology, that can be used to describe services on top of MicroWSMO and also SAWSDL. Its aim is to reduce the overhead in describing services and to be able to annotate RESTful Web services. MicroWSMO is a SAWSDL-like layer on top of hRESTS.

RESTfulGrounding [98] uses a new grounding ontology in OWL-S to semantically describe RESTful Web services. In Semantic Bridge for Web Services
(SBWS) [99], a service is described using the Web Application Description Language (WADL) by creating a monolithic XML file containing all the information about the service interface.

With the increasing amount of Linked Data being added everyday several types of Semantic Web services have emerged. Linked Data Services (LIDS) [100] focuses on describing data services using RDF and SPARQL where by augmenting Linked Datasets dynamically with data extracted from Web APIs.

Linked Open Services (LOS) [101] provide semantic wrappers for WSDL and Web APIs to function as RDF producers and consumers. It describes the functionality of services, using RDF and graph patterns, and then describes their composition using SPARQL queries. A LOS specifies not only what graph patterns a service consumes and produces but also how the input RDF is consumed and produced using SPARQL constructs.

A detailed comparison of the Semantic Web services framework listed above is out of scope of this thesis. Interested readers are encouraged to look at [102] for details.

2.5 Databases and Ontologies

Ontology is defined as “a formal, explicit specification of a shared conceptualization” [103]. Ontologies contain a vocabulary of an abstract model in the form of concepts and the relationships between the concepts. Ontologies cap-
ture general knowledge and they can be accepted by a group and be shared. A relational database represents the structure and integrity of the data. The schema is developed for a specific application and it is not intended to be shared by others.

The main differences of an ontology and a database are given below:

- A database is a collection of information that has been organized in a specific manner while an ontology is a “specification of a conceptualization”.

- Database contents are typed while everything in the ontology is a concept.

- Concepts have either datatype property or object property while a database table has attributes and relationship with other tables.

- Concepts and properties in the ontology can be organized by hierarchies.

- Databases encode application semantics implicitly, ontologies explicitly.

- By design, ontologies are trivially extensible. Database schemas are brittle.

In general, ontologies are more expressive and have richer semantics than relational schemas, but that does not mean that a relational schema does
not encode domain semantics. These semantics are implicit in the relational schema.

Generally, extracting information from relational database and rendering them to be used in an ontology involves one of the following strategies:

- creating from scratch a new ontology based on information extracted from a relational database
- generating RDF statements that conform to one or more predefined ontologies and reflect the contents of a relational database
- answering semantic queries directly against a relational database
- discovering correspondences between a relational database and a given ontology

A comprehensive survey of these strategies is reported in [104] in detail. A simplified high-level diagram of these strategies is seen in Figure 2.5. For an existing ontology, the domain should be compatible with the database schema and contents so that meaningful (i.e. semantic) mappings can be defined. On the other hand, the existence of a domain ontology is not a prerequisite in the creation of a new ontology, which is expanded throughout the left branch.

The methods that create a new ontology are further subdivided in two subcategories, depending on the domain of the generated ontology. On one hand, there are approaches that develop an ontology with the domain of the
Figure 2.5: Classification of approaches

relational model. The generated ontology, called a database schema ontology, consists of concepts and relationships that reflect the constructs of the relational model, mirroring the structure of the input relational database. Since the generated ontology does not implicate any other knowledge domain, these approaches are mainly automatic relying on the input database schema alone. On the other hand, there are methods that generate domain-specific ontologies, depending on the domain described by the contents of the input database.

The last classification category is the application of database reverse engineering techniques in the process of creating a new domain-specific ontology.
Approaches that apply reverse engineering try to recover the initial conceptual schema from the relational schema and translate it to an ontology expressed in a target language. On the other hand, there are methods that apply translation rules instead of reverse engineering techniques, from the relational to the RDF Model and/or rely on the human expert for the definition of complex mappings and the enrichment of the generated ontology model.

In this thesis, we use the approach where one or more existing ontologies are used to access databases which is popularly known as the mediator approach. In this approach, a technique called query rewriting is used where a user formulates queries using the terminologies of the existing ontologies, typically in SPARQL. The queries are then converted to equivalent queries in terms of the source database schema, typically as SQL queries.
Chapter 3

Valet SADI

In this chapter, an architecture of Valet SADI framework is presented. A compact diagram of the architecture is introduced which gives an overview of its components and how they interact with other other. There are four separate modules in Valet SADI among which two expect inputs entered manually by a human user and the two others use them in a series to automatically generate a complete package for a SADI Semantic Web service. The compact overview diagram is later expanded with each module described separately to allows the reader become familiar with their functionality in detail.
3.1 Architecture

Valet SADI works as a personal assistant for creating SADI Semantic Web services automatically for retrieving information from the RDBs. Just like a valet in a car parking lot, the implementation of this architecture takes care of the critical task of creating the service’s source code from a mere declarative description of the input and the output when suitable mapping rules are available. Hence the architecture is named Valet SADI.

The workflow for the complete service generation process is illustrated by the compact architecture diagram in Figure 3.1. The process relies on four modules: a Semantic Mapping Module, a Service Description Module, an SQL-template Query Generator Module, and a Service Generator Module, forming a pipeline.

3.1.1 Module-1: Semantic Mapping

A core resource required for implementing semantic querying over a relational database requires mapping of the database schema to relevant domain ontologies. The RDB represents information in tables using rows and columns. Data in these tables are related according to common keys. Relevant parts of the database schema may be manually mapped to the terminologies in the domain ontology which may be represented using an expressive rule language. PSOA rules have been demonstrated to be useful for such mappings.\footnote{http://wiki.ruleml.org/index.php/Rule-Based_Data_Access}
hence PSOA RuleML is chosen as the rule language for mapping specification in Valet SADI. Once the mapping rules are defined, they do not need to be revisited anymore. However, any changes occurring in the database in future which are used in the rules must be reflected in the mappings. This will require the rules to be revisited and perhaps rewritten to ensure the correct operations of the generated services. Further discussion and context for the regeneration of SADI services with a set of revised mapping rules are provided in Chapter 7 in detail.
The three components comprising the *Semantic Mapping Module* are described here:

### 3.1.1.1 Relational Database

A relational database presents information in tables with rows and columns. A table is referred to as a *relation* as it is a collection of objects of the same type (row). Data in a table can be related according to common keys and the ability to retrieve related data from a table is the basis for the term relational database. It is expected that data can be retrieved from multiple tables through table-join based on the primary key and foreign key attributes.

### 3.1.1.2 Domain Ontology

The domain ontology in this module must correspond to the same domain of the relational database, using ontology concepts, individuals, object properties, data properties, and possibly axioms. The higher the degree of the correspondence, the easier it is to represent mappings between the two.

### 3.1.1.3 Mapping Rules

To access data from the relational database, queries are formulated using the terminologies from the domain ontologies. Mapping between the domain ontologies and the database schema is required to access the data. These mappings are manually written using PSOA rules whose expressiveness extends the function-free (Datalog) expressiveness level of deductive databases.
and realizes the expressiveness level of Horn logic [54,105].

3.1.2 Module-2: Service Description

This module displays the tasks related to the service ontology. The input and output of a SADI service are defined as OWL class expressions as previously mentioned in Section 2.3.8.1. Each service would be represented by a service ontology.

3.1.2.1 Service Ontology

In a service ontology, the input and output descriptions are defined as classes by using concepts from the domain ontology and various OWL class constructs, such as property restrictions, and class intersections. These I/O classes define the shape of RDF graphs that are eligible as service input and those that can be expected as service output. In general terms, a URI with an RDF graph describing it must be classified as an instance of a service input class to be deemed legitimate input to the service. Likewise, an output of a service must be classifiable as an instance of the output class.

3.1.3 Module-3: SQL-template Query Generator

In Valet SADI, the I/O OWL class definitions are used to generate an SQL query to be placed in the code of the generated service. Three additional inputs are also required for this purpose: i) the RDB abstraction which is a
representation (i.e. a generalization) of facts from a table (e.g. \(\text{patient}(X)\) for all \(\text{patient}(p_1), \ldots, \text{patient}(p_{100})\)), ii) the domain ontology which is implicitly available for import (i.e. for reference) into the service ontology, and iii) the semantic mapping rules. These inputs are submitted, in an appropriate format – TPTP, to the VampirePrime reasoner \[19\], which implements a technique called \textit{Incremental Query Rewriting} (IQR) \[6\], and together with some auxiliary SQL-specific code, produces an SQL query that can be used to select the data from the source database. The input is represented with formal parameters in the SQL query, corresponding to the relevant parts of an input RDF graph, and for this reason the query is considered a template. It is instantiated by substituting the formal parameters with specific input values each time the service is executed.

The module, without needing any human intervention, thus creates an SQL-template query with the help of five components (see Module-3 in Figure 3.1). The reasoner produces schematic answers to a semantic query which are then rewritten into an SQL-template query mentioned above \[6\]. In order for the reasoner to resolve the query, it requires four separate inputs indicated by the four incoming arrows all of which deliver the inputs in TPTP format: one from the RDB, one from a converter, and two from the two translators. The converter and the two translators are among our contributions in this module which are described below. The reasoner and the query rewriter are reused and readers interested in them can refer to \[6\] for details. The abstraction of RDB is trivial, hence no converter or translators are necessary.
3.1.3.1 Service I/O to Semantic Query Converter

This converter uses the input and output description expressed in OWL syntax in the service ontology as its input and produces a *semantic query* as its output in TPTP-FOF.

3.1.3.2 Ontology to TPTP Formula Translator

The translator takes the domain ontology as its input and translates it into the equivalent TPTP-FOF as its output.

3.1.3.3 PSOA Rules to TPTP Formula Translator

This translator takes the mapping rules specified in PSOA RuleML syntax as its input and translates them into the equivalent TPTP-FOF as its output.

3.1.3.4 VampirePrime Reasoner + Query Rewriter

The reasoner uses all 4 inputs and finds schematic answers to the semantic query. The logical representation of these answers are converted into SQL.

3.1.4 Module-4: Service Generator

The service generator module generates Java source code for the SADI Web service. The code is composed of three separate code-blocks: i) input code block which reads input data values from the input RDF graph and stores the reference in an object variable, ii) executable query code block which
contains the instantiated SQL query statement within the WHERE clause, and iii) output code block which creates the output RDF graph from the result sets after the query is executed. In addition, this module also compiles the code and deploys the Web service in the service registry by the Web Service Deployer. Once the services are deployed in the service registry, the end users are able to execute the services from the SPARQL query.

Overall, once the input and output descriptions are defined, the rest of the service generation process becomes equivalent to clicking a Generate and Deploy button. With the mapping rules in place, it becomes significantly easier to create and deploy services without having any knowledge of the underlying database schema than to generate them manually. While the SHARE query engine requires the end users to have knowledge of SPARQL, the commercial HYDRA query engine eliminates this shortcoming by providing a graphical query canvas. The users can compose complex query graphs by merely typing keywords on the query canvas, which is then automatically converted to a query in SPARQL. The availability of such GUI-based query engine in combination with the Valet SADI makes the creation, deployment, and querying of services an efficient and less error-prone operation.

3.2 Descriptions of the Modules

The operations of each module are described below with supporting diagrams. These diagrams depict the roles played by all components and the users
3.2.1 Writing Mapping Rules

Figure 3.2 shows how mapping rules are written for a specific domain of interest. The source data is represented by relational tables and typically stored using one of the modern RDB systems. One or more domain ontologies can be used to represent the source data. Terminologies in these ontologies may have already been defined by experts in the domain and released as part of ontologies on the Web. Thus, some terminologies may readily be imported and reused from existing ontologies while others can be created at the convenience of the users to represent the RDB schema. Given that they share the same domain of interest, correspondence can be established between these two formalisms. These correspondences may be expressed in the form of expressive rules which facilitate access to source data.

3.2.2 Writing Service Descriptions

Once the domain ontology is ready with all necessary terminologies, either by importing existing ones or creating them from scratch, the input and output of a service can be defined. With the availability of modern and user-friendly ontology development environments previously mentioned in Section 2.3.8.1, it has become increasingly easier to define the I/O as OWL class expressions. In the query engine, a SPARQL query can be used to
execute one or more SADI services and each action of a data retrieval can be identified and performed by a SADI service. Conventionally, a service ontology contains the input and the output of a distinct service. Figure 3.3 shows how SADI services are defined in service ontologies in relation to the domain ontology.
3.2.3 Automatic Generation of SQL-template Query

One of the most important technical contributions of the architecture lies in the generation of a template query in SQL automatically. Once the input and the output of a service is defined and the mapping rules are ready, the generation process can begin. The reasoner takes four different inputs and runs by generating schematic answers as the solutions to a semantic query. One input is $DB'$, which is an abstraction of the logical representation $DB$ of the relational database. The declarative input and output described in OWL syntax are converted into a specialized semantic query $Q$ in TPTP format consisting of unary (concepts) and binary predicates (object and data properties). The predicates converted from the output and the input descrip-
tions correspond to the generation of SELECT and WHERE clause of the template-SQL query, respectively. The knowledge base $KB$ is a finite set of FOL axioms corresponding to both the domain ontologies and the semantic mapping rules. There are two translators: the $Ontology$ $to$ $TPTP$ $Formula$ $Translator$ translates the domain ontology into First-Order formulas in TPTP syntax while the $PSOA$ $Rules$ $to$ $TPTP$ $Formula$ $Translator$ translates the mapping rules into the similar TPTP syntax. The creation of the FROM clause depends on the $DB'$ and both elements in the $KB$. VampirePrime reasoner uses resolution steps to generate schematic answers where answers with similar proofs are obtained in bulk rather than wasting reasoning steps leading to similar results or no results at all. An SQL-specific Incremental Query Rewriter (IQR) is then used on the answers to generate the SQL-template query. The generated query is termed as a template because the WHERE clause is only instantiated when a service is invoked and executed. Figure $3.4$ shows various inputs and processing steps for generating the final template query. The execution steps for generating a few queries related to one of the application domain in Hospital-Acquired Infections are illustrated in Appendix $A.4$.

3.2.4 Automatic Generation of Web Service Code

The source code for the Web service is automatically generated once the SQL-template query is successfully created. The source code contains all necessary components of a typical Web application written in the Java programming
Figure 3.4: Steps for generating SQL-template query automatically

language with parameters specified to act as a SADI Semantic Web service. A processor called *Generator* traverses the input and the output from the service ontology as graphs and creates the source code for reading the input values from the RDF input graph and the code to generate the RDF output
graph after the execution of the instantiated SQL query. The Generator also places the SQL-template query in the source code so that it can be instantiated properly with input values, if necessary. The service code blocks are written in a `ServiceName.java` file, where `ServiceName` represents a unique name. Several other source files are also dynamically generated such as the `DBConnections.java` for connecting to the database with the access credentials specified in `database.properties` file, `web.xml` file containing the configuration and deployment information, and `index.jsp` file for indexing the Web application on the browser. Figure 3.5 shows all inputs for generating the source code of a typical layout of a SADI service presented as a Maven Web application.

### 3.2.5 Querying SADI Services

Once the source code for a SADI Web service is generated, it is then compiled and packaged for indexing in a SADI registry. End users execute queries to services on a SADI-specific query engine. The queries are composed as W3C standard SPARQL queries. When a query is submitted to the query engine, it looks up services in the registry. Once they are discovered, the query engine plans the execution steps of each query statement by orchestrating them in order based on the OWL object/data properties and the types of classes and individuals they are attached to. The services are then invoked for execution in order. Each SADI service reads input values from an RDF input

Figure 3.5: Steps for automatic generation of SADI Web service code graph, instantiates the SQL-template query using those values, executes the instantiated query on the source database, pulls results from the database, and returns results as an RDF output graph. The engine integrates all results from the services and returns answers to the query to the end users. Figure 3.6 shows how information stored in RDBs are returned as RDF via the SADI services when end users submit SPARQL queries on a SADI query engine.
3.2.6 Target End Users

The implementation of SADI services was initiated by targeting the integration of national and international bioinformatics data and resources which are disparate yet inter-related. Contrary to the standard Web services, SADI provided rich descriptions of functional requirements which allow service interoperability, autonomous discovery as well as access to the resources in real-time for bioinformatic researchers.

The adoption of SADI-specific deployment based on RDBs depends on building services manually as the first step and the ability to compose SPARQL
queries for executing these services. Hence, expertise in certain areas are required as previously mentioned in Section 2.4.1.7. Such background knowledge involve knowledge in ontology modeling, data modeling, RDB query language such as SQL, programming language such as Java or Perl, and SPARQL query language. Typically, software engineers in the area of Semantic Technologies, say Semantic Web engineers, are known to acquire such expertise and they would be able to deploy SADI services and query them. Bioinformaticians are domain experts who could benefit from SADI service deployments. Assuming that services have been manually created and deployed by the Semantic Web engineers, they still need to be well-versed in composing SPARQL query which requires considerable amount of time, effort and training. The intelligent GUI on HYDRA query engine allows these group of users without expertise in writing SPARQL queries to automatically create them from a graphical query canvas by merely typing keywords from the domain. Users can create query graphs on HYDRA which are translated into SPARQL queries automatically. This feature allows not only bioinformaticians, but also experts in any domain, to compose SPARQL queries using the terminologies they are familiar with.

However, building SADI services still remain a challenge for the Semantic Web engineers because writing program code and SQL queries manually are error-prone and time-consuming. The availability of the Valet SADI tool reduces these problems to a great extent because program code for a SADI service is written automatically without error in a few seconds and SQL
queries are also generated automatically. With Valet SADI, the role of a Semantic Web engineer can thus be limited to writing mapping rules instead of a broad range of duties because once the mapping is available, SADI services can be generated automatically from their I/O definitions. With our Valet SADI tool, service generation becomes less error-prone and more services can be created and deployed faster than ever before. Software engineers with lower technical skills can be hired compared to those with standard skills had the services been created manually. Moreover, domain experts who are typically knowledgeable in ontology modeling can consult with the engineers and participate in checking the mapping rules. The final goal is to enable domain experts with sufficient training to create SADI service automatically using Valet SADI.
Chapter 4

Implementation of Valet SADI

In this chapter, we present the formal algorithms for prototyping the generation of a SADI service. The inputs consumed and the outputs produced in each module are accompanied by the corresponding algorithms where applicable.

Figure 4.1 illustrates the roles of the algorithms used to generate a SADI service. Algorithm 1 uses the input and output descriptions of a SADI service and creates a graph for each of them. Algorithm 2 then uses these two graphs as input and creates a semantic query $Q$. The domain ontology is translated into First-Order formulas in TPTP syntax called TPTP-FOF according the reduction formula listed in Table 4.2. The semantic mapping rules in PSOA RuleML are also translated into TPTP-FOF according the mapping formulas listed in Table 4.3. Once the SQL-template queries are created from the schematic answers by the reasoner and subsequently a query rewriter,
Algorithm 3 is used to create the service code for a SADI service with the help of the input and the output graph created earlier. The generated code reads input values based on the structure of the input graph, instantiates the SQL-template query using these values, and creates the output based on the structure of the output graph, when the service is executed.

4.1 Module-1: Semantic Mapping

There are several approaches to map a database to an ontology and a number of them can be classified broadly into two categories depending on whether
there is an existing ontology to map to or not. For an existing ontology, mapping becomes challenging due to the fact that either creating a new ontology at will or altering an existing ontology may not be allowed. This is why mapping of an RDB to an existing ontology is difficult to automate and they are created either completely manually as in [106, 107] or semi-automatically as in [108], where user-inputs may be required in different stages of the mapping process, or even automatically to an extent as in [109, 110] which relies heavily on the data volume, lexical matching etc. for the best possible mapping. Even with the automated approach, validation of the mapping from an expert is necessary in order to retrieve the correct answers.

Mapping of RDBs to the terminologies of the domain ontologies at the schema level using axioms and/or rules are preferable to RDB materialization as these rules are easy to write and modify whenever RDB schemas and/or terminologies evolve over time. The goal is to find the best possible mapping between these two formalisms. Several approaches described in [111–132] aim to map the RDB schemas to the ontologies which allow interoperability between the two formalisms. Together, these approaches cover a wide range of challenges in mapping due to the fact that features that are common in a complex RDB schema may not be so common in an ontology. The principle examples of these discourses are one-to-many relations (foreign-key attributes), simple inheritance, multiple inheritance, many-to-many relations (with attributes), n-ary relations, among others. Table 4.1 shows how these
problems have been addressed by existing mapping approaches over the years. The fundamental difference between these approaches to ours is that the answers are retrieved using a layer of SADI services, and we only need to map at the schema level and not at the instance level.

4.1.1 Rules for Mapping Database to Ontology

To write the mapping rules, the positional psoa terms previously described in Section 2.3.9 are used. Inspired by the existing approaches described above, the following rules are used in Valet SADI to map the RDB schema and the domain ontologies:

**Rule 1:** A table containing attributes without any foreign keys is mapped to a class in the domain ontology. The types of services generated in this scenario are all Y, where all instances of Y are retrieved without requiring any input and Y maps to an attribute in the table (see Section 5.2).
Rule 2: A table containing at least one foreign key is mapped to a class in the domain ontology. For each foreign key, two object properties may be used for mapping where one is mutually inverse of the other, if available. The domain of the property refers to the instance of the current class and the range refers to the instance of the referenced class.

Rule 3: A table containing a primary key which is also a foreign key referencing a table represents a simple inheritance as all tables of this type are considered sub-tables in a hierarchy. Each of the sub-tables would be mapped to class in the domain ontology which would be a subclass of the class represented by the table referenced through the foreign key. In [133][134] it was noted that to preserve simple inheritance in a hierarchy an entity can be a member of at most one of the subclasses and every entity of a superclass must belong to at least one subclass, also known as disjointment and totalness, respectively.

Rule 4: A table containing a composite primary key (consisting of two or more columns) whose components are also foreign keys whose fields are referencing exactly two tables, is mapped to object properties where one is mutually inverse of the other. If the table of this category contains simple columns which may have resulted from many-to-many relation with attributes, then the combination of this rule and the following Rule 5 is applied.
**Rule 5:** A table containing a composite primary key whose components are also foreign keys referencing more than two tables which do not contain any duplicate of the simple attributes represents an \( n \)-ary relation. As OWL only support unary and binary relations, a couple of solutions have been proposed in [132] to address \( n \)-ary relations. One way to resolve is to create a new class and \( n \) new properties where an instance of the relation linking the \( n \) individuals is an instance of this class. A second way is applicable when the \( n \)-ary relationship links individuals playing different roles in a structure without any single individual acting as a subject of the relation. Unlike the previous case, there is no subject and an individual is created to represent the relation instance which links all participants. A class which corresponds to the bridge table which relates to the classes representing the associated tables having \( is-a \) relations is used for the mapping. If all records of the participating tables are used then the representation of \( allValuesFrom \) restriction (i.e. value restriction in Table 2.2) is used, otherwise \( someValuesFrom \) restriction (i.e. exist restriction in Table 2.2) is used.

**Rule 6:** The rule addresses multiple inheritance in the RDB schema where the tables are subclasses in more than one class or subclass relationship. These tables in the RDB schema may look like bridge-tables as part of \( n \)-ary relations in [Rule 5] due to the fact that they are in fact weak entities whose primary keys consist of two or more foreign keys referencing two or more tables. However, the distinguishing feature is to assume that in addition
to the primary key the table must contain inherited attributes belonging to super-tables.

Each sub-table is mapped to a class which is a subclass of the classes corresponding to the tables referenced by the foreign key of the sub-tables. Such mapping essentially requires that a similar hierarchy is present in the domain ontology.

**Rule 7:** For a primary key (or a column with UNIQUE constraint) in a table, inverse functional equations (or inverse functional object property) are used.

**Rule 8:** For a simple column i.e. without a foreign key constraint and a column with NOT NULL constraint, data properties are used where the domain is the class representing the table and the range is the datatype expressed with XML schema. Similarly, mapping rules in the absence of NOT NULL use data properties with the respective domain. However, the range in this case may be specified as the type of available data or simply without specifying any type information.

### 4.2 Module-2: Service Description

The input and output of a SADI service are described as OWL class expressions in a service ontology. The input and output of each service are essentially graphs (see Section 2.4.1.2) because their instantiations represent
RDF graphs. In this service description module, a graph is generated from each of the input and output OWL class expression. Each of these graphs will later be traversed for further processing in order to i) create the semantic query $Q$, and ii) create the source code to read input values from the input RDF instance and the source code to write output RDF instance after the execution of the instantiated SQL query with the input values. Algorithm 1 creates a directed graph from a valid OWL class expression $cex$ as an input. For an input class expression $cex_I$ and an output class expression $cex_O$, the algorithm produces a graph $G_I = (V_I, E_I)$ and an output graph $G_O = (V_O, E_O)$, respectively.

In lines 2-4, the class expression and the set of vertices and edges are initialized. Due to the depth-first traversal of the entities in the OWL class expressions, a stack is used as the data structure to keep track of the vertices. In lines 5-19, an OWL class expression is traversed if it is an instance of intersectionOf OWL expression (conjunction in Table 2.2). Line 6 adds the instance of the expression to the set of vertices. If the expression starts with an object or data property, lines 7-12 creates an edge and adds to the set of edges. In lines 13-15, each operand is pushed to the stack. In lines 16-18, each operand is recursively explored for continuing with the graph creation.

In lines 20-31, vertices and edges are created for an instance of the object- SomeValuesFrom OWL class expression (exists restriction in Table 2.2). Line 30 is used to generate graph for the filler of the object property which may be a simple or a complex class expression. This kind of class expression consists
of a property and a class expression called a filler. In lines 32-43, vertices
and edges are added to the set of vertices and edges for an instance of the
objectExactCardinality OWL class expression (exists restriction with exact
cardinality in Table 2.2) in a similar way using a filler. In lines 44-66, vertices
and edges are added to the sets of vertices and edges for the instance of OWL
objectHasValue class expression (value restriction in Table 2.2). The edges
are object properties for all these instances. The range of the object prop-
erty is an individual, hence it is directly added to the set of vertices. In lines
67-86, data properties are added as the edges and datatypes are added as ver-
tices for the instance of dataSomeValuesFrom expression (exists restriction
in Table 2.2). In lines 87-106, vertices and edges are added for the instance
of dataExactCardinality expression (exists restriction with exact cardinality
in Table 2.2). Finally, in line 107 the graph is returned.

4.3 Module-3: SQL-template Query Genera-
tor

In order to generate an SQL-template query automatically, the method im-
plemented in [6], based on semantic querying, is used. The setting in that
work, similar to Valet SADI, involves an RDB containing the source data,
one or more expressive knowledge bases which are related to the schema of
the database, and a logical description of the relations in the RDB to link
them to the concepts and relations defined by the KB. The method is based
Algorithm 1 generateGraph(OWLClassExpression cex)

Require: OWL Class Expression cex, an empty stack of vertices vStack, initialize property label from parent node propLabel_{frmPar} to null

Ensure: Return a rooted directed graph G = (V, E)

1: procedure generateGraph(OWLClassExpression cex)
2: \[ cex \leftarrow \emptyset \]
3: \[ V \leftarrow \emptyset \]
4: \[ E \leftarrow \emptyset \]
5: if cex is an instance of intersectionOf then
6: \quad Add vertex \( V_{cex} \) to \( V \)
7: \quad if propLabel_{frmPar} has an object/data property label then
8: \quad \quad if vStack is not empty then
9: \quad \quad \quad Add edge (pop(vStack), \( V_{cex} \)) to \( E \) with label \( propLabel_{frmPar} \)
10: \quad \quad propLabel_{frmPar} \leftarrow \text{null}
11: \quad end if
12: end if
13: for each operand ce in cex do
14: \quad push ce into vStack
15: end for
16: for each operand ce in cex do
17: \quad generateGraph(ce)
18: end for
19: end if
20: if cex is an instance of objectSomeValuesFrom then
21: \quad if propLabel_{frmPar} has an object/data property label then
22: \quad \quad if vStack is not empty then
23: \quad \quad \quad Add vertex \( V_{cex} \) to \( V \)
24: \quad \quad \quad Add edge (pop(vStack), \( V_{cex} \)) to \( E \) with label \( propLabel_{frmPar} \)
25: \quad \quad propLabel_{frmPar} \leftarrow \text{null}
26: \quad \quad push cex into vStack
27: \quad end if
28: end if
29: propLabel_{frmPar} \leftarrow \text{object property attached to } cex
30: generateGraph(cex_{filler})
31: end if
32: if cex is an instance of objectExactCardinality then
33:   if propLabel_{frmPar} has an object/data property label then
34:     if vStack is not empty then
35:       Add vertex V_{cex} to V
36:       Add edge (pop(vStack), V_{cex}) to E with label propLabel_{frmPar}
37:       propLabel_{frmPar} ← null
38:       push cex into vStack
39:     end if
40:   end if
41:   propLabel_{frmPar} ← object property attached to cex
42:   generateGraph(cex_{filler})
43: end if
44: if cex is an instance of objectHasValue then
45:   if propLabel_{frmPar} has no label then
46:     if vStack is not empty then
47:       Add vertex with named individual V_{ind} of cex to V
48:       Add edge (pop(vStack), V_{ind}) to E labeled with object property
49:     else
50:       Add vertex V_{cex} to V
51:       Add vertex with named individual V_{ind} of cex to V
52:       Add edge (V_{cex}, V_{ind}) to E labeled with object property
53:       propLabel_{frmPar} ← null
54:       push cex into vStack
55:     end if
56:   else
57:     Add vertex V_{cex} to V
58:     if vStack is not empty then
59:       Add edge (pop(vStack), V_{cex}) to E with label propLabel_{frmPar}
60:     end if
61:     propLabel_{frmPar} ← null
62:     push cex into vStack
63:     Add vertex with named individual V_{ind} of cex to V
64:     Add edge (V_{cex}, V_{ind}) to E labeled with object property
65: end if
66: end if
if \( cex \) is an instance of dataSomeValuesFrom then

if \( propLabel_{frmPar} \) has no label then

if \( vStack \) is not empty then

Add vertex with datatype \( V_{\text{datatype}} \) of \( cex \) to \( V \)

Add edge (pop(\( vStack \)), \( V_{\text{datatype}} \)) to \( E \) labelled with data property

else

Add vertex \( V_{cex} \) to \( V \)

Add vertex with datatype \( V_{\text{datatype}} \) of \( cex \) to \( V \)

Add edge (\( V_{cex}, V_{\text{datatype}} \)) to \( E \) labelled with data property

end if

else

if \( vStack \) is not empty then

Add vertex \( V_{cex} \) to \( V \)

Add edge (pop(\( vStack \)), \( V_{cex} \)) to \( E \) with label propLabel_{frmPar}

\( propLabel_{frmPar} \leftarrow \) null

Add vertex with named datatype \( V_{\text{datatype}} \) of \( cex \) to \( V \)

Add edge (\( V_{cex}, V_{\text{datatype}} \)) to \( E \) labelled with data property

end if

end if

end if

if \( cex \) is an instance of dataExactCardinality then

if \( propLabel_{frmPar} \) has no label then

if \( vStack \) is not empty then

Add vertex with datatype \( V_{\text{datatype}} \) of \( cex \) to \( V \)

Add edge (pop(\( vStack \)), \( V_{\text{ datatype}} \)) to \( E \) labelled with data property

else

Add vertex \( V_{cex} \) to \( V \)

Add vertex with datatype \( V_{\text{ datatype}} \) of \( cex \) to \( V \)

Add edge (\( V_{cex}, V_{\text{ datatype}} \)) to \( E \) labelled with data property of \( cex \)

end if

end if
on the use of resolution for incremental transformation of semantic queries into sequences of SQL queries which can then be directly executed on the RDB Management System (RDBMS).

The implementation of this method uses a FOL reasoner which does the smart part of the job as a query preprocessor, in combination with a conventional RDBMS which efficiently processes large volumes of data to answer the generated SQL queries. The reasoner accepts a semantic query, the relevant knowledge bases which combine the domain ontologies and the semantic mappings together, and an abstraction of the database to generate a (possibly infinite) number of expressions that are called schematic answers which are then converted into SQL-template queries.

In order to answer a query over a RDB deductively modulo some KB, where the DB can be represented as a collection of ground atomic facts, a resolution-based FOL reasoner can be used. This approach is quite ineffi-
cient because if the RDB is very large and the selectivity of the query is not very high, many answers will be obtained with structurally identical proofs. For example, a database containing facts $\text{patient}(s_1), \ldots, \text{patient}(s_{100})$ from a table of $\text{patient}$ will provide 100 answers to the query $\text{patient}(X)$ with a refutational proof tree (see [6]). The reasoning becomes wasteful because the number of proofs grows with the answers, in addition to the unsuccessful attempts represented by derivations that do not lead to any answers.

Therefore, the main idea is that answers with similar proofs should be obtained in bulk by using reasoning to find schematic answers to queries to avoid wasteful repetitive resolution steps. Finding these answers is performed by reasoning on an abstraction of the RDB in some resolution-based and paramodulation-based calculus [135, 136] such that the schematic answers can then be turned into regular RDBMS queries such as SQL.

The three inputs required for the resolution steps are: the semantic query $Q$, the abstraction of the $DB$, and the $KB$ representing both the domain ontology and the semantic mapping.

The logical representation $DB$ of the relational database is a set of ground atomic non-equality formulas, each representing a row in a table in the database. For example, a table containing facts $\text{patient}(p_1), \ldots, \text{patient}(p_{100})$ representing a table $tbPatient$ can be generalized as $\text{patient}(X)$, and thus an abstraction of the table. As the abstraction is straightforward, no further transformation tool is necessary in order to be used by the reasoner for generating schematic answers. On the other hand, the generation of a semantic
query requires the use of a converter, while an ontology translator is used on the domain ontology and a translator is used to translate the semantic mapping rules. The conversion and translation processes are described next.

4.3.1 Service I/O to Semantic Query Converter

It was previously described in Section 2.4.1.2 that both the input and the output of a SADI service are modeled such that they are rooted at the same node. The converter described here creates a semantic query according to this modeling. The creation of the query results from the combination of the two graphs generated in Section 4.2.

The semantic query is \( Q = \langle \phi \rangle \) where the goal \( \phi \) is a construct of the form \( \langle X_1, \ldots, X_k \rangle \langle Y_1, \ldots, Y_m \rangle C \), where \( C \) is a nonempty clause, \( k, m \geq 0 \), \( \{X_1, \ldots, X_k, Y_1, \ldots, Y_m\} = vars(C) \), all \( X_i, Y_j \) are pairwise distinct, all \( X_i \)'s are distinguished variables, and all \( Y_j \)'s are undistinguished variables. With the query \( Q \) the reasoner tries to find all distinguished variables \( X_i \) such that there exist some \( Y_j \), such that \( \phi(X, Y) \) is inconsistent with \( DB \cup KB \) using the refutational theorem proving. The set \( V_{dist} \) contains the distinguished variables in Algorithm 2. Lines 2-10 loops through each edge in a depth-first traversal order to create the predicates of the semantic query. Each directed edge \((v_1, v_2)\) in \( E_I \cup E_O \) is checked for a label. If a label is found, it is then checked against all object or data properties in the domain ontologies. In line 11, the root node of \( G_I \) is assumed as the root of both \( G_I \) and \( G_O \) graphs in order to make sure that one of the predicates matches
the explicit relationship between the input and output graph. A set for containing the distinguished variables is initialized in line 12. In lines 13-19, the distinguished variables are collected as data values from the data properties by traversing only the output graph $G_O$. The reason is generalization of the requirement of having the output defined in both all $Y$ and get $Y$By$X$ types of services, while all $Y$ services requires no input. Lines 20-24 appear to return the semantic query $\phi$.

4.3.2 Ontology to TPTP Formula Translator

Most of the Description Logics (DLs) are specific subsets of First-Order Logic (FOL). Thus DL interpretations have the same structures as FOL interpretations based on the assumption that individuals are constants, concepts are unary predicates and roles are binary predicates. As a result, reasoning problems in DLs are transferable to equivalent reasoning problems in FOL. Thus, DLs can be reduced to FOL formulas, such as in a TPTP-FOF syntax which can be processed in the FOL reasoner VampirePrime.

Table 4.2 shows the reduction from $\textit{SHOIN DL}$ to FOL where $C$ and $C_1 \ldots C_n$ are concept expressions, and $R$ is a role. The domain ontology in Valet SADI is translated to TPTP-FOF sentences. A translated domain ontology is added to Appendix [A.1].
Algorithm 2 createSemanticQuery($G_I$, $G_O$)

Require: $G_I = (V_I, E_I), G_O = (V_O, E_O)$
Ensure: A semantic query $Q$

1: procedure createSemanticQuery($G_I, G_O$)
2: for each $(v_1, v_2) \in E_I \cup E_O$ in a depth-first traversal order do
3:   if $\text{label}((v_1, v_2)) \neq \emptyset$ then
4:     if $v_2$ is an individual or a class then
5:       create a unary predicate $\text{label}((v_1, v_2))(v_2)$ and add to $\phi$
6:     else
7:       create a binary predicate $\text{label}((v_1, v_2))(v_1, v_2)$ and add to $\phi$
8:   end if
9: end if
10: end for
11: assume root of $G_I$ as the root of $G_O$ on $\phi$
12: $V_{\text{dist}} \leftarrow \emptyset$
13: for each $(v_{O1}, v_{O2}) \in E_O$ in a depth-first traversal order do
14:   if $\text{label}((v_{O1}, v_{O2})) \neq \emptyset$ then
15:     if $\text{label}((v_{O1}, v_{O2}))$ is a data property then
16:       add $v_{O2}$ to $V_{\text{dist}}$
17:     end if
18:   end if
19: end for
20: if $V_{\text{dist}} = \emptyset$ then
21:   return null
22: else
23:   return $\phi$
24: end if
25: end procedure
Table 4.2: Reduction of \textit{SHOIN} Description Logics to First-Order Logic

<table>
<thead>
<tr>
<th>Description Logics</th>
<th>First-Order Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_1 \sqcap \ldots \sqcap C_n$</td>
<td>$C_1(x) \land \ldots \land C_n(x)$</td>
</tr>
<tr>
<td>$C_1 \sqcup \ldots \sqcup C_n$</td>
<td>$C_1(x) \lor \ldots \lor C_n(x)$</td>
</tr>
<tr>
<td>$\neg C$</td>
<td>$\neg C(x)$</td>
</tr>
<tr>
<td>{o₁, \ldots, oₙ}</td>
<td>$x = o₁ \lor \ldots \lor x = oₙ$</td>
</tr>
<tr>
<td>$\exists R. C$</td>
<td>$\exists y. (R(x, y) \land C(y))$</td>
</tr>
<tr>
<td>$\forall R. C$</td>
<td>$\forall y. (R(x, y) \rightarrow C(y))$</td>
</tr>
<tr>
<td>$\geq nR$</td>
<td>$\exists^n y. R(x, y)$</td>
</tr>
<tr>
<td>$\leq nR$</td>
<td>$\exists^\leq^n y. R(x, y)$</td>
</tr>
<tr>
<td>$C_1 \sqsubseteq C_2$</td>
<td>$\forall x. (C_1(x) \rightarrow C_2(x))$</td>
</tr>
<tr>
<td>$R_1 \sqsubseteq R_2$</td>
<td>$\forall x, y. (R_1(x, y) \rightarrow R_2(x, y))$</td>
</tr>
<tr>
<td>Trans($R$), $R$ is transitive</td>
<td>$\forall x, y, z. (R(x, y) \land R(y, z) \rightarrow R(x, z))$</td>
</tr>
<tr>
<td>a : C</td>
<td>$C(a)$</td>
</tr>
<tr>
<td>{a, b} : R</td>
<td>$R(a, b)$</td>
</tr>
<tr>
<td>a = b</td>
<td>$a = b$</td>
</tr>
<tr>
<td>a $\neq$ b</td>
<td>$\neg (a = b)$</td>
</tr>
<tr>
<td>$R^\sim$, $R$ is symmetric</td>
<td>$\forall x, y. (R(x, y) \Leftrightarrow R(y, x))$</td>
</tr>
</tbody>
</table>

4.3.3 PSOA Rules to TPTP Formula Translator

The PSOA2TPTP translator \cite{137} maps a KB and a query in the PSOA RuleML presentation syntax exemplified in Section \cite{2.3.9} to the TPTP interchange language. The translated documents are then executed to answer the query by the VampirePrime reasoner. The Valet SADI framework also requires translation of the semantic mapping rules written in PSOA RuleML to be used by VampirePrime. Since PSOA2TPTP was designed to answer object-relational queries, one of its normalization steps ("objectification") introduces OIDs, which are not needed for our translation purposes, hence prevent its reuse. This motivated us to implement a syntactic translator which
translates the mapping rules containing positional terms into their equivalent TPTP-FOF with positional arguments. The slotted terms in PSOA RuleML are used to represent object-centered knowledge bases. Hence, translation of slots has not been considered as a requirement for implementing the Valet SADI framework.

Inspired by PSOA2TPTP, we reuse the translation function $\tau_{psoa}(\cdot)$ for mapping PSOA constructs typically used in the mapping rules to a TPTP construct. Table 4.3 shows the mapping of PSOA to TPTP-FOF constructs used in our mapping rules. During the translation, the variables are converted to uppercase as TPTP variables start with uppercase letters.

<table>
<thead>
<tr>
<th>PSOA Constructs</th>
<th>TPTP-FOF Constructs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{?v}$</td>
<td>$V$</td>
</tr>
<tr>
<td>$a = b$</td>
<td>$\tau_{psoa}(a) = \tau_{psoa}(b)$</td>
</tr>
<tr>
<td>$p(x_1 x_2 \ldots x_n)$</td>
<td>$p(\tau_{psoa}(x_1), \tau_{psoa}(x_2), \ldots, \tau_{psoa}(x_n))$</td>
</tr>
<tr>
<td>$\text{And}(f_1 \ldots f_n)$</td>
<td>$(\tau_{psoa}(f_1) &amp; \ldots &amp; \tau_{psoa}(f_n))$</td>
</tr>
<tr>
<td>$\text{Or}(f_1 \ldots f_n)$</td>
<td>$(\tau_{psoa}(f_1) \mid \ldots \mid \tau_{psoa}(f_n))$</td>
</tr>
<tr>
<td>$\text{Exists}(\text{?v}_1 \ldots \text{?v}_n f)$</td>
<td>$[\tau_{psoa}(\text{?v}<em>1) \ldots \tau</em>{psoa}(\text{?v}<em>n)] : \tau</em>{psoa}(f)$</td>
</tr>
<tr>
<td>$\text{Forall}(\text{?v}_1 \ldots \text{?v}_n f)$</td>
<td>$![\tau_{psoa}(\text{?v}<em>1) \ldots \tau</em>{psoa}(\text{?v}<em>n)] : \tau</em>{psoa}(f)$</td>
</tr>
<tr>
<td>$\varphi : \neg \psi$</td>
<td>$\tau_{psoa}(\psi) \Rightarrow \tau_{psoa}(\varphi)$</td>
</tr>
</tbody>
</table>

Table 4.3: Mapping from PSOA constructs to TPTP-FOF constructs

For example, the following rule in the KB has two relations $\text{has\_last\_name}$ and $\text{db\_N\_patient}$ containing positional arguments stating that each patient has a last name. The uninterpreted function $\text{identityForPatient}$ with the help of function $\text{identityForPatientToPatWID}$, which are inverse functional to each other, is used to identify a unique patient (see lines 3-4 in Figure 5.4).
and their translations in Appendix A.3.

\[
\text{forall } \text{?patWID } \text{?patFacWID } \text{?patLastName } \text{?patFirstName} ( \\
\text{has_last_name(identityForPatient(?patWID) ?patLastName)} :- \\
\text{db_Npatient(?patWID ?patFacWID ?patLastName ?patFirstName))}
\]

The translated rule is expressed as a TPTP-FOF axiom as follows:

\[
\text{fof(attribute\_patLastName\_represents\_last\_name\_of\_patient,axiom,} \\
\text{! [PatWID, PatFacWID, PatLastName,PatFirstName] :} \\
\text{(db_Npatient(PatWID, PatFacWID, PatLastName,PatFirstName)} \\
\text{ =>} \\
\text{has_last_name(identityForPatient(PatWID), PatLastName))})
\]

In Section 5.4, the mapping rules in PSOA RuleML are illustrated and their translations are added in Appendix A.3.

### 4.4 Module-4: Service Generator

In Figure 3.5, we have shown what a typical SADI service consists of. Here, we present an algorithm that only generates the source code for executing the tasks of the service: extracting input values, instantiating SQL queries with the values in place of formal parameters within the condition (i.e. WHERE clause), and creating the output after the query is executed. The rest of the auxiliary source files are dynamically generated from templates and the
process is out of the scope of this chapter. In order to generate the code, Algorithm 3 uses the graphs representing the input and output and the uninstantiated SQL statement $S_{\text{SQL-template}}$ generated in Module-3. Lines 3-14 shows how the input graph $G_I$ is traversed to create statements $s_i$ to read data values and instances of classes from the RDF input. In line 15, the instantiated SQL statement is appended to the ordered list of statements in $S_{\text{Java}}$. Lines 16-25 shows how the graph $G_O$ representing the output description is traversed to generate the code for creating an RDF output graph by adding the results from the execution, as data values with data properties and as individuals with object properties. Line 26 appears to return an ordered list of statements.

In order for the service code to be generated automatically, the roles of Algorithm 1, Algorithm 2, Table 4.2 and Table 4.3 have been thoroughly tested. Although the subsequent chapters illustrate a number of services to show the usefulness of these methodologies, not all of them are suitable during the service generation. For example, a domain ontology may have disjunctive class expressions (see disjunction in Table 2.2) which are translated by the reduction formulas in Table 4.2. In the current Valet SADI implementation, the service ontology does not support input or output of a service to be expressed as disjunctive class expressions. The reason is that for a single disjunctive expression two or more services have to be generated from a single definition, one for each operand. Although this seems like a restriction, disallowing disjunctive expressions either in the input or the output can be easily
complemented by defining two or more distinct services for each operand.

**Algorithm 3 Generate SADI Service code**

**Require:** $G_I = (V_I, E_I), G_O = (V_O, E_O), S_{SQL-template}$

**Ensure:** Ordered list of statements $S_{Java}$

1: procedure GENERATESADICODE($G_I, G_O, S_{SQL-template}$)
2:     $S_{Java} \leftarrow \emptyset$
3:     for each $(v_{I_1}, v_{I_2}) \in E_I$ do
4:         if $\text{label}((v_{I_1}, v_{I_2})) \neq \emptyset$ then
5:             if $\text{label}((v_{I_1}, v_{I_2}))$ is a data property then
6:                 create $s_i$ to extract data value $I_{lit}$ of $v_{I_2}$
7:                 instantiate $S_{SQL-template}$ with $I_{lit}$ in place of condition
8:             else if $v_{I_2}$ is an individual or a class then
9:                 create $s_i$ to extract instance $I_{ind}$ of $v_{I_2}$
10:                instantiate $S_{SQL-template}$ with $I_{ind}$ in place of condition
11:            end if
12:         append $s_i$ to $S_{Java}$
13:     end if
14:     end for
15:     append $S_{SQL-template}$ to $S_{Java}$
16:     for each $(v_{O_1}, v_{O_2}) \in E_O$ do
17:         if $\text{label}((v_{O_1}, v_{O_2})) \neq \emptyset$ then
18:             if $\text{label}((v_{O_1}, v_{O_2}))$ is a data property then
19:                 create $s_o$ to add data value $O_{lit}$ as $v_{O_2}$
20:             else if $v_{O_2}$ is an individual or a class then
21:                 create $s_o$ to add instance $O_{ind}$ as $v_{O_2}$
22:             end if
23:         append $s_o$ to $S_{Java}$
24:     end if
25:     end for
26: return $S_{Java}$
27: end procedure
4.5 Structure of Valet SADI Codebase

The Java source of the Valet SADI framework is hosted on bitbucket\(^2\). The implementation of the framework is organized into following 7 components:

- **src**: It contains the source code for all the algorithms listed in this chapter and a list of resources. Among the resources, a validator is used to ensure the I/O validity of the services, a JAXB library-based schema definition for translating the domain ontology, the VampirePrime reasoner, and the skeletons and stub for creating the codebase of a SADI service in Java.

- **knowledgebase**: It contains both the domain ontology and semantic mapping rules. Both of the input rules in PSOA RuleML syntax and its translation into TPTP-FOF are maintained.

- **rdb**: It contains a View of the relational database and its abstraction.

- **semanticquery**: It contains the semantic query created from the declarative input and output definitions of the service in the service ontology. Both Algorithm\(^1\) and Algorithm\(^2\) are used before the query is created.

- **schematicanswers**: The answers are generated once the inputs in Module-3 are fed to the VampirePrime reasoner. The answers are essentially a set of results to the semantic query.

\(^2\)https://bitbucket.org/sadnanalmanir/valetsadi
- **sql:** The SQL-specific query rewriter creates the SQL-template query using the schematic answers.

Once the SQL-template query is created successfully, the source code for a SADI service is created as a standalone Web application source.
Chapter 5

Service Generation and Target Application Domain

In this chapter, we demonstrate how SADI Semantic Web services are generated automatically using the Valet SADI framework. For this we use a schema extracted from The Ottawa Hospital Data Warehouse as the source. We create an ontology as part of our knowledge base which describes the domain corresponding to the schema. We show how to write declarative semantic mappings between the schema and the ontology using PSOA RuleML and how to define the input and the output of a service in OWL. We then show the generation of SQL-template queries and code for a complete SADI service. Although a total of 11 SADI services are generated, we only provide the steps for 3 services in detail.
5.1 Hospital-Acquired Infection (HAI)

The target application domain in this chapter is dedicated to the surveillance for Hospital-Acquired Infections (HAI) which is a common scenario in clinical intelligence. Infections acquired in hospitals are a serious problem costing millions of dollars every year. To address the problem access to infections-related information should be available to experts such as surveillance practitioners. In previous work [11, 12] a surveillance infrastructure was set up to answer target queries using SADI services so that infections related data could be quickly accessed by the practitioners who lack technical expertise in querying relational databases. Although this approach showed promising results, creating SADI services manually was found to be a tedious process. With Valet SADI framework, service generation can be automated. The services are used to retrieve data from a single table or multiple tables through joins and return data as RDF. In the following sections, we list all steps for generating SADI services automatically using Valet SADI.

5.2 Steps for Generating a SADI Service

A typical SADI service is uniquely named in the form of get $Y$By$X$, where the output $Y$ is retrieved based on the input $X$. A special case of a SADI service is all $Y$, where all instances of $Y$ are retrieved without requiring any input. The steps for generating a SADI service by Valet SADI, either as get $Y$By$X$ or as all $Y$, are demonstrated next. The service get AllergyByPatientId is an
example of \( Y \)By\( X \) while the service allPatientIds is an example of all\( Y \).

5.2.1 Documenting Semantic Mapping Rules

In order to generate a SADI service using our framework, suitable declarative semantic mappings must be established between the schemata of the source data and the domain ontology. In this module, the mappings are expressed using PSOA RuleML language.

5.2.1.1 Relational Database

Figure 5.1 shows the normalized data model of The Ottawa Hospital Data Warehouse (TOHDW). In this model, the table Npatient contains 37 attributes and the table Nservice contains 28 attributes. Similarly, NhrProcedure, NhrDiagnosis, NhrAllergy, Nfacility, and NphmService tables contain large numbers of attributes. Due to space constraints and in order to simplify the illustration, an extract of this model is used containing select tables and attributes. The extract is sufficient to illustrate all the steps necessary for generating SADI services based on the declarative semantic mappings between the schema and the domain ontologies. The illustration can easily be extended to replicate service generation for the complete TOHDW model.

The extract of the model is presented by the schema in Figure 5.2. The schema contains tables representing patients, their allergies, the diagnoses and the procedures performed on them, the facilities they are admitted in,
Figure 5.1: The Ottawa Hospital Data Warehouse

and the services provided by the pharmacy for their medication.

In the schema extract, table `Npatient` contains basic information about all patients; table `NhrDiagnosis` contains information about diagnoses, table `NhrAbstract` contains general medical abstract information regarding the diagnoses and procedures, table `Nfacility` contains description of the hospital facilities, table `Nallergy` contains information about allergies, table `Nencounter` contains records during the encounter of patients for various services such as pharmacy, diagnosis etc., table `Nservice` contains generic
information regarding services received by patients, and table \texttt{NphmService} contains pharmacy-related records of the administered drugs.

The integer-valued attribute \textit{patWID} is used to identify patients, \textit{hdgWID} identifies each diagnosis record, \textit{hraWID} is the primary key for \texttt{NhrAbstract}, \textit{facWID} identifies a hospital facility, and \textit{algWID} identifies an allergy. Diagnosis information in \texttt{NhrDiagnosis} for a patient in \texttt{Npatient} is accessed via
the foreign key attribute *hraPatWID* in *NhrAbstract*. Attribute *patFacWID* in *Npatient* refers to the facility the patient is admitted in while the attribute *algPatWID* in *Nallergy* refers to the allergy status of a patient. The attribute *encWID* is a unique identifier for each encounter entry, *svcWID* is the unique identifier for each service entry such as pharmacy, radiology, laboratory etc., *svcEncWID* is a pointer to a unique identifier for each encounter, *svcTableCd* refers to the type of service, *svcOrderedBy* refers to the person who ordered the service, *svcPerformedBy* refers to the person who performed the service. The attribute *phmWID* is a unique identifier for each pharmacy service entry, *phmSvcWID* is the foreign key for each service, *phmOrderDesc* refers to the brand name/ingredient of the medication, *phmStartDtm* refers to the start date/time, *phmEndDtm* refers to end date/time, *phmStrengthDesc* refers to strength description, and *phmDosageForm* refers to dosage form (e.g. tablet).

Mapping may typically be guided case-by-case and by requirements of the types of services to be generated. While in some cases domain ontologies may be sufficiently rich in classes, properties, axioms to be mapped easily to the RDB schema, there may be other cases where the ontologies lack corresponding entities to map to the schema. This problem may be noticed in case of RDB schemas as well. However, our approach is flexible to accommodate changes or having multiple relations between tables, such as having *has* and *is_afraid_of* relations between the two tables *Npatient* and *Nallergy*. For modeling and generation of an additional service to work when a new prop-
erty such as is_afraid_of is added, it is sufficient in Valet SADI to add a new mapping rule using is_afraid_of, similar to an existing mapping rule which uses the has property. Section 7.4 illustrates a scenario in the context of change management for malaria surveillance where a mapping rule is needed to be modified due to an unexpected change in the ontology.

5.2.1.2 Domain Ontology

In the domain ontology in Figure 5.3, the hierarchy of classes includes Person, Patient (subclassed to Person), Diagnosis, Allergy, Facility, Procedure, Service, Abstract, Encounter and Pharmacy.

The object properties represent the relations between entities in the domain ontology. Among these properties has_abstract_record links diagnoses information to medical abstract records, abstractRecordForPatient links abstract records to patients, has_diagnosis links patients to diagnoses, has_allergy links patients to allergies, has_facility_reference links patients to hospital facilities, undergo_procedure links patients with procedures performed on them, cancel_at links patients with their pharmacy, encountersPatient links encounter information to a patient, service_for_encounter links service information to encounter information, and prescription_service links pharmacy information to service information.

The data properties representing the literal values of the entities include has_diagnosis_code specifying the codes of diagnoses, has_adverse_reaction_to specifying the name of the allergens, has_facility_description specifying the
hospital facility, description_of_order specifying the medication, has_start_time specifying the time of start, has_first_name and has_last_name specifying the first and last name, respectively.

```xml
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix terminology: <http://localhost:8080/healthcare/domain-ontologies/terminology.owl#> .
@prefix : <http://localhost:8080/healthcare/domain-ontologies/terminology.owl#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@base <http://localhost:8080/healthcare/domain-ontologies/terminology.owl> .
<http://localhost:8080/healthcare/domain-ontologies/terminology.owl> rdf:type owl:Ontology ;
# Object Properties
terminology:abstractRecordForPatient rdf:type owl:ObjectProperty .
terminology:has_abstract_record rdf:type owl:ObjectProperty .
terminology:has_facility_reference rdf:type owl:ObjectProperty .
terminology:has_allergy rdf:type owl:ObjectProperty .
terminology:undergo_procedure rdf:type owl:ObjectProperty .
terminology:counsel_at rdf:type owl:ObjectProperty .
terminology:encountersPatient rdf:type owl:ObjectProperty .
terminology:service_for_encounter rdf:type owl:ObjectProperty .
terminology:prescription_service rdf:type owl:ObjectProperty .

# Data Properties
terminology:has_adverse_reaction_to rdf:type owl:DatatypeProperty .
terminology:has_diagnosis_code rdf:type owl:DatatypeProperty .
terminology:has_first_name rdf:type owl:DatatypeProperty .
terminology:has_last_name rdf:type owl:DatatypeProperty .
terminology:has_start_time rdf:type owl:DatatypeProperty .

# Classes
terminology:Allergy rdf:type owl:Class .
terminology:Diagnosis rdf:type owl:Class .
terminology:Facility rdf:type owl:Class .
terminology:Patient rdf:type owl:Class ;
  rdfs:subClassOf terminology:Person .
terminology:Person rdf:type owl:Class .
terminology:Pharmacy rdf:type owl:Class .
terminology:Procedure rdf:type owl:Class .
terminology:Service rdf:type owl:Class .
terminology:Abstract rdf:type owl:Class .
terminology:Encounter rdf:type owl:Class .
```

Figure 5.3: Terminologies in the Hospital-Acquired Infections domain ontology terminology.owl

5.2.1.3 Mapping Rules in PSOA RuleML

Semantic mapping rules are specified based on the rules listed in Section 4.1.1. The set of PSOA rules in Figure 5.4 illustrates the semantic mappings of...
the terminologies from the domain ontology to four relational tables, namely
\texttt{Npatient}, \texttt{NhrDiagnosis}, \texttt{NhrAbstract} and \texttt{Nfacility}. These rules define
a \textit{virtual RDF graph} containing all triples that can be inferred from the table
records. Equations in lines 3-4, 13-16, 22-23, and 41-42 introduce and axiomatise auxiliary uninterpreted functions like \texttt{identityForPatient}, mapping patient identifiers from the database to entities representing the corresponding patients in the virtual semantic model, and the inverse functions like \texttt{identityForPatientTopatWID}, mapping such patient entities back to the identifiers from the database. Rules in lines 5-7, 24-26, and 43-45 classify the entities as instances of the corresponding classes \texttt{Patient}, \texttt{Diagnosis}, and \texttt{Facility}, respectively, in the virtual semantic model. Rules in lines 17-19, 27-32, 46-48, and 51-53 populate the properties \texttt{abstractRecordForPatient}, \texttt{has_abstract_record}, \texttt{has_diagnosis_code}, \texttt{has_facility_description}, and \texttt{has_facility_reference} from single records in the tables.

Due to space constraints, mapping rules for the \texttt{Nallergy} table are shown in Appendix\texttt{[A.2]}. Lines 56-57 introduce and axiomatise auxiliary uninterpreted function \texttt{identityForAllergy} mapping allergy identifiers in the table to the virtual model while the inverse function \texttt{identityForAllergytoalgWID} maps such allergy entities back to the identifiers from the table. Rules in lines 61-66 populate the properties \texttt{is_allergic_to} and \texttt{has_adverse_reaction_to} from records in the table.
RuleML (
  Assert (
      Forall ?patWID (identityForPatientTopatWID(identityForPatient(?patWID)) = ?patWID)
      forall ?P (identityForPatient(identityForPatientTopatWID(?P)) = ?P)
      forall ?patWID ?patFacWID ?patLastName ?patFirstName (Patient(identityForPatient(?patWID)) :-
        db_Npatient(?patWID ?patFacWID ?patLastName ?patFirstName))
      forall ?P (identityForPatientTopatWID(identityForPatient(?P)) = ?P)
      forall ?patWID ?patLast Name ?patFirstName (has_last_name(identityForPatient(?patWID) ?patLastName) :-
        db_Npatient(?patWID ?patFacWID ?patLastName ?patFirstName))
  )
  Assert (
      forall ?hraWID (identityForAbstractTohraWID(identityForAbstract(?hraWID)) = ?hraWID)
      forall ?P (identityForAbstract(identityForAbstractTohraWID(?P)) = ?P)
      forall ?hraWID ?hraPatWID (abstractRecordForPatient(identityForAbstract(?hraWID) identityForPatient(?hraWID)) :-
        db_Nabstract(?hraWID ?hraPatWID))
  )
  Assert (
      forall ?hdgWID (identityForDiagnosisTohdgWID(diagnosisEntity(?hdgWID)) = ?hdgWID)
      forall ?P (identityForDiagnosis(identityForDiagnosisTohdgWID(?P)) = ?P)
      forall ?hdgWID ?hdgHraWID ?hdgCd (Diagnosis(identityForDiagnosis(?hdgWID)) :-
        db_NhraDiagnosis(?hdgWID ?hdgHraWID ?hdgCd)
        has抽象_record(identityForDiagnosis(?hdgWID) identityForAbstract(?hdgHraWID)) :-
        has_diagnosis_code(identityForDiagnosis(?hdgWID) ?hdgCd)
      )
  )
  Assert (
      forall ?Diag ?Abs ?Pat (has_diagnosis(identityForPatient(?Pat) identityForDiagnosis(?Diag)) :-
        And(has抽象_record(identityForAbstract(?Abs) identityForPatient(?Pat))
        diagnosis_encountered(identityForDiagnosis(?Diag) identityForAbstract(?Abs))))
  )
  Assert (
      forall ?facWID (identityForFacilityTofacWID(identityForFacility(?facWID)) = ?facWID)
      forall ?P (identityForFacility(identityForFacilityTofacWID(?P)) = ?P)
      forall ?facWID ?facFacilityDesc (Facility(identityForFacility(?facWID)) :-
        db_Nfacility(?facWID ?facFacilityDesc)
      )
  )
  Assert (
      forall ?patWID ?patFacWID ?patLastName ?patFirstName (has_facility_reference(identityForPatient(?patWID) identityForFacility(?patFacWID)) :-
        db_Npatient(?patWID ?patFacWID ?patLastName ?patFirstName))
  )
)

Figure 5.4: Mapping rules in PSOA RuleML between the schema extract and terminology.owl ontology
5.2.2 Declarative Description of Services

In this module, the input and output of a SADI service are defined in a service ontology as OWL class expressions, ideally using an ontology editing tool such as protégé (see 2.3.8.1). For convenience, the services are identified as $S_1$ through $S_{11}$.

5.2.2.1 Defining I/O of all $Y$ Services

Only one instance of service-generation of this form is described here. The service $S_1$ is used to retrieve all instances of an attribute from a single table. This is a special case of a SADI service because it generates output without expecting any input (see 2.4.1.3).

$S_1$: The service retrieves all patient ids from the Npatient table. The output is an instance of the class Patient represented by an id and the input is expressed as an enumerated individual of Patient which is distinguished by the curly braces ($\{, \}$). It does not expect any value other than the root node at the input level. Hence, the input is treated as a dummy input.

Name: allPatientIds

Input: \{Patient\}

Output: type value Patient

Source Table: Npatient

As an instance of all $Y$ services, the input definition is not processed because no input values are expected for this type of services. The output defini-
tion is processed according to the graph generation algorithm presented in Section 4.2 and converted into the directed graph shown in Figure 5.5.

Figure 5.5: Output graph of all PatientIds

5.2.2.2 Defining I/O of get YByX Services

Whereas all Y services are used to draw data from a single table without expecting any input, get YbyX services draw data Y from one or more tables based on the input X. The service get LastnameFirstnameByPatientId is used to retrieve data from a single table while the service get AllergyByPatientId retrieves data from two tables using a table-join. More services drawing data from multiple tables are listed in the subsequent sections.

$S_2$: It retrieves the last name and the first name of a patient based on the patient id from the Npatient table. The input is an instance of a Patient represented by an id and the output decorates the input instance by adding the last name and the first name as string literals using the has_last_name and has_first_name properties, respectively.

Name: get LastnameFirstnameByPatientId
Input: type value Patient

Output: \((\text{has\_last\_name}\ \text{some\ string})\) and \((\text{has\_first\_name}\ \text{some\ string})\)

Source table: Npatient

The graphs generated from the input and the output definitions are represented in Figure 5.6 and Figure 5.7, respectively.

![Input graph of getLastnameFirstnameByPatientId](image1)

![Output graph of getLastNameFirstnameByPatientId](image2)

\(S_3:\) It retrieves all allergens based on a patient id from Npatient and Nallergy tables. The input is an instance of a Patient represented by an id and the output decorates the input instance by adding an instance of Allergy using the \text{has\_allergy} object property which is caused by the string-valued allergens using the \text{has\_adverse\_reaction\_to} data property.

Name: getAllergyByPatientId

Input: type value Patient

Output: \text{has\_allergy} some ((type value Allergy) and (\text{has\_adverse\_reaction\_to} some string))

Source table: Npatient and Nallergy
The generated directed graphs for the input and the output are shown in Figure 5.8 and Figure 5.9 respectively.

Figure 5.8: Input graph of getAllergyByPatientId
Figure 5.9: Output graph of getAllergyByPatientId

5.2.3 Generating SQL-template Queries

The following inputs are required to generate the SQL-template query: database, a semantic query, the domain ontology as part of the knowledge base, and the declarative semantic mapping rules as part of the knowledge base.

The tables in our database schema can be represented logically as follows:

```
db_Nfacility(facWID, facFacilityDesc)
db_Npatient(patWID, patFacWID, patLastName, patFirstName)
db_Nallergy(algWID, algPatWID, algAdverseReaction)
db_Nencounter(encWID, encPatWID)
db_Nservice(svcWID, svcEncWID, svcTableCd, svcOrderedBy, svcPerformedBy)
db_NphmIngredient(phmWID, phmSvcWID, phmOrderDesc, phmStartDtm, phmEndDtm, phmStrengthDesc, phmDosageForm)
db_NhrAbstract(hraWID, hraEncWID, hraPatWID)
db_NhrDiagnosis(hdgWID, hdgHraWID, hdgHraEncWID, hdgCd)
db_NhrProcedure(bprcWID, bprcHraWID, bprcHraEncWID, bprcStartTime)
```

As predicate names in TPTP syntax (see Section 2.3.5) are constants which start with lowercase letters, the table names are presented with a prefix
Throughout the rest of the chapter, the rest of the predicates starting with a capital letter are represented with a prefix $p_-$ except for the $answer$ predicate which is a special predicate as part of the semantic query containing distinguished variables only.

5.2.3.1 Converting I/O into Semantic Query

A converter is used to transform the I/O descriptions of a service into a semantic query based on Algorithm 2 described in Section 4.3.1. The graphs generated from the input and output service descriptions are traversed in order to list the distinguished and undistinguished variables. The leaf nodes matching the classes and individuals are converted into unary predicates while the directed edges matching object properties and data properties are converted into binary predicates. Concrete representations of the converted I/O descriptions of services are illustrated below:

$S_1$: The following semantic query is created by traversing the graph in Figure 5.5 where all distinguished variables are added to arguments of an $answer$ predicate. Variable $Z_0$ is the distinguished variable here to be substituted by a unique patient identifier.

$$ p_{-Patient} (identityForPatient(Z_0)), \ answer(Z_0) $$ (5.1)

$S_2$: The semantic query is generated after traversing both graphs in Figure 5.6 and Figure 5.7. The distinguished variables $Z_1$ and $Z_2$ are added to
the answer predicate as arguments which would be substituted for the last and first name. The double-quote symbols around the undistinguished variable $Z0$ indicate that it is to be expected as an input. Ultimately, $Z0$ will be placed inside the WHERE clause of the SQL-template query as shown in [6].

\[
p\_Patient(identityForPatient("Z0")), \\
p\_has\_last\_name(identityForPatient("Z0"), Z1), \\
p\_has\_first\_name(identityForPatient("Z0"), Z2), \\
\text{answer}(Z1, Z2)
\]

(5.2)

$S_3$: The semantic query generated from Figure [5.8] and Figure [5.9] consists of \textit{has\_allergy} and \textit{has\_adverse\_reaction\_to} predicates. The distinguished variable $Z2$ is added to the answer predicate and is expected to be substituted by the name of the allergen. Among the undistinguished variables, $Z0$ will be substituted during the instantiation of the SQL query.

\[
p\_Patient(identityForPatient("Z0")), \\
p\_Allergy(identityForAllergy(Z1)), \\
p\_has\_allergy(entityForPatient("Z0"), entityForAllergy(Z1)), \\
p\_has\_adverse\_reaction\_to(entityForAllergy(Z1), Z2), \text{answer}(Z2)
\]

(5.3)
5.2.3.2 Translating Domain Ontology to TPTP-FOF

The domain ontology is translated to TPTP-FOF axioms based on the translation schemes shown in Table 4.2. The translator uses an XML Schema Definition (XSD) document for an OWL ontology and creates Java classes for each OWL constructs by traversing the schema. A parser based on the JAXB library [138] is used for parsing the schema. The generated Java classes are then used to create Abstract Syntax Objects (ASOs) for OWL constructs. Based on the formulas to reduce $\text{SHOIN}(D)$ DL to FOL in Table 4.2, the ASOs are translated to TPTP-FOF. Due to space constraints the complete translation of the domain ontology terminology.owl is listed in Appendix A.1.

5.2.3.3 Translating PSOA Rules to TPTP-FOF

We illustrate only those translations of PSOA mapping rules which are relevant to the services $S_1$, $S_2$, and $S_3$. More rules and equations translated from PSOA RuleML to TPTP-FOF are added to Appendix A.3.

Service $S_1$: The following two equations and one rule are needed from the semantic mapping rules corresponding to the lines 3-7 of the mapping rules illustrated in Figure 5.4.

Each axiom has a unique identifier which can be randomly generated. For clarity, an identifier such as $\text{inverse_for_entityForPatient}_1$ is used here to show a translated TPTP-FOF axiom. Similar identifiers are used for the rest of the translated rules.
fof(inverse_for_entityForPatient_1, axiom,
    ! [PatWID] : (entityForPatientToPatWID(entityForPatient(PatWID)) = PatWID).
).
fof(inverse_for_entityForPatient_2, axiom,
    ! [P] : (entityForPatient(entityForPatientToPatWID(P)) = P).
).
fof(table_Npatient_represents_instances_of_concept_Person, axiom,
    ! [PatWID, PatFacWID, PatLastName, PatFirstName] :
        (db_Npatient(PatWID, PatFacWID, PatLastName, PatFirstName) =>
         p_Patient(entityForPatient(PatWID))).
).

S2: The translation of two rules map the last name and the first name to an instance of a patient. The rule to map last name corresponds to the lines 8-10 of the mapping rule in Figure 5.4. The rule to map the first name can be translated in a similar way.

fof(attribute_PatLastName_represents_last_name_of_a_Patient, axiom,
    ! [PatWID, PatFacWID, PatLastName, PatFirstName] :
        (db_Npatient(PatWID, PatFacWID, PatLastName, PatFirstName) =>
         p_has_last_name(entityForPatient(PatWID), PatLastName)).
).
fof(attribute_PatFirstName_represents_first_name_of_a_Patient, axiom,
    ! [PatWID, PatFacWID, PatLastName, PatFirstName] :
        (db_Npatient(PatWID, PatFacWID, PatLastName, PatFirstName) =>
         p_has_first_name(entityForPatient(PatWID), PatFirstName)).
).
$S_3$: The translation includes two identity equations and three rules. The equations are used for the unique primary key of the $N$allergy table. The rest of the rules map the table $N$allergy to the concept $A$llergy, two tables $N$patient and $N$allergy using a foreign key, and an attribute for the allergen called $A$lgAdverseReaction to a data value.

```plaintext
fof(inverse_for_identityForAllergy_1, axiom,
    [AlgWID] : (identityForAllergyToAlgWID(identityForAllergy(AlgWID)) = AlgWID).
).
fof(inverse_for_identityForAllergy_2, axiom,
    [P] : (identityForAllergy(identityForAllergyToAlgWID(P)) = P).
).
fof(table_Nallergy_represents_instances_of_concept_Allergy, axiom,
    [AlgWID, AlgPatWID, AlgAdverseReaction] : 
    (db_Nallergy(AlgWID, AlgPatWID, AlgAdverseReaction) => p_Allergy(identityForAllergy(AlgWID))).
).
fof(attribute_AlgPatWID_used_as_foreign_key_to_patient, axiom,
    [AlgWID, AlgPatWID, AlgAdverseReaction] : 
    (db_Nallergy(AlgWID, AlgPatWID, AlgAdverseReaction) => p_has_allergy(identityForPatient(AlgPatWID), identityForAllergy(AlgWID))).
).
fof(attribute_AlgAdverseReaction_represents_data_value_of_allergy_name, axiom,
    [AlgWID, AlgPatWID, AlgAdverseReaction] : 
    (db_Nallergy(AlgWID, AlgPatWID, AlgAdverseReaction) => p_has_adverse_reaction_to(identityForAllergy(AlgWID), AlgAdverseReaction)).
).
```
5.2.3.4 Rewriting Schematic Answers to SQL

Now that all 4 inputs are ready in TPTP-FOF format, the VampirePrime reasoner is used to generate schematic answers which are then converted into the SQL-template query by an SQL-specific rewriter. The resolution over RDB abstractions in the form of constrained clauses may produce (infinitely) many schematic answers by the reasoner and consequently SQL queries by the rewriter. The union of their answers covers the whole set of concrete answers to the query as they are produced one by one.

The automatically generated SQL-template queries for $S_1$, $S_2$ and $S_3$ are shown in Table 5.1. For each of these 3 services, a proof tree produced during the resolution step, the schematic answers to the semantic query, and the steps involved during the conversion of schematic answers into SQL-template queries are shown in Appendix A.4.

5.2.4 Generation of SADI Service Code

The *Generator* component uses the I/O graphs in Module-1 to create code implementing the tasks assigned to a service. The tasks include reading the input values in RDF, instantiating the SQL-template query with the input values, and writing the results returned by service in RDF based on the algorithm described in Section 4.4. The source code is generated based on the methods provided by the JENA API [139]. The source code for all three services are added to Appendix A.5.
<table>
<thead>
<tr>
<th>Service</th>
<th>Form</th>
<th>Table Name</th>
<th>SQL-template Query</th>
</tr>
</thead>
</table>
| $S_1$   | allY   | Npatient   | SELECT TABLE0.patWID AS X0  
FROM Npatient AS TABLE0 |
| $S_2$   | getYByX| Npatient   | SELECT TABLE0.patLastName AS X1,  
TABLE1.patFirstName AS X5  
FROM Npatient AS TABLE0,  
Npatient AS TABLE1  
WHERE TABLE0.patWID = "Z0"  
AND TABLE1.patWID = "Z0" |
| $S_3$   | getYByX| Npatient, Nallergy  | SELECT TABLE2.algAdverseReaction AS X6  
FROM Npatient AS TABLE0,  
Nallergy AS TABLE1,  
Nallergy AS TABLE2  
WHERE TABLE0.patWID = "Z0"  
AND TABLE1.algPatWID = "Z0"  
AND TABLE1.algWID = TABLE2.algWID |

Table 5.1: Automatically generated SQL-template queries for $S_1$, $S_2$ and $S_3$. The variable $Z0$ in the condition will be instantiated by data value during the execution of the service.
Steps for generating the rest of the services from $S_4$ to $S_{11}$ are not explained in detail here due to space constraints. However, the descriptions of the services and the automatically generated SQL-template queries are described next.

### 5.2.5 Additional SADI Services for HAI Surveillance

**Service $S_4$:** The service retrieves the diagnosis id of a patient based on a patient id. The input is an instance of a Patient represented by an id and the output decorates the input instance by adding the instance of Diagnosis using the `has_diagnosis` property.

**Name:** `getDiagnosisIdByPatientId`

**Input:** `type` value Patient

**Output:** `has_diagnosis` some (`type` value Diagnosis)

**Source tables:** `Npatient`, `NhrAbstract`, `NhrDiagnosis`

**Service $S_5$:** The service retrieves the diagnosis code of a patient based on a diagnosis id. The input is an instance of a Diagnosis represented by an id and the output decorates the input instance by adding the ICD10 code as a string using the `has_diagnosis_code` property.

**Name:** `getDiagnosisCodeByDiagnosisId`

**Input:** `type` value Diagnosis

**Output:** `has_diagnosis_code` some string

**Source table:** `NhrDiagnosis`
Service $S_6$: The service retrieves the admitting facility of a patient based on a patient id. The input is an instance of a Patient represented by an id and the output decorates the input instance by adding the description of the admitting facility as a string using the $has\_facility\_reference$ and $has\_facility\_description$ properties.

Name: $getFacilityDescriptionByPatientId$

Input: type value Patient

Output: $has\_facility\_reference$ some ($type$ value Facility) and ($has\_facility\_description$ some string))

Source tables: Npatient, Nfacility

Service $S_7$: The service retrieves the procedure id of a patient based on a patient id. The input is an instance of a Patient represented by an id and the output decorates the input instance by adding the instance of Procedure using the $undergo\_procedure$ property.

Name: $getProcedureIdByPatientId$

Input: type value Patient

Output: $undergo\_procedure$ some ($type$ value Procedure)

Source tables: NhrProcedure, NhrAbstract, Npatient

Service $S_8$: The service retrieves the start time of the procedure of a patient based on a procedure id. The input is an instance of a Procedure represented by an id and the output decorates the input instance by adding the
time the procedure is scheduled to start as a string using the has_start_time property.

Name: getStartTimeByProcedureId

Input: type value Procedure

Output: has_start_time some dateTime

Source table: NhrProcedure

**Service S_9:** The service retrieves the pharmacy id of a patient based on a patient id. The input is an instance of a Patient represented by an id and the output decorates the input instance by adding the instance of Pharmacy using the councilAt property.

Name: getPharmacyIdByPatientId

Input: type value Patient

Output: councilAt some (type value Pharmacy)

Source tables: Npatient, Nencounter, Nservice, NhpmIngredient

**Service S_{10}:** The service retrieves the start time of the medication prescribed for a patient based on a pharmacy id. The input is an instance of a Pharmacy represented by an id and the output decorates the input instance by adding the time the medication is scheduled to start as a string using the has_start_time property.

Name: getDateTimeByPharmacyId

Input: type value Pharmacy

Output: has_start_date some dateTime
*Source table:* NhpmIngredient

**Service** $S_{11} :$ The service retrieves the administered medication of a patient based on a pharmacy id. The input is an instance of a Pharmacy represented by an id and the output decorates the input instance by adding the description of the medication as a string using the `description_of_order` property.

*Name:* `getOrderDescriptionByPharmacyId`

*Input:* `type` value Pharmacy

*Output:* `description_of_order` some string

*Source table:* NhpmIngredient

The SQL-template queries for services $S_4$ to $S_{11}$ are listed in Table 5.2.

### 5.2.5.1 Service Generation Time for Valet SADI

The services listed in this chapter were generated using Valet SADI once the semantic mapping rules and the definitions of the services were manually verified by an expert. Table 5.3 shows how much time it took for Valet SADI to generate the source code which contains the automatically generated SQL-template query. The tests were done on a machine with a 4-core Intel Core i5-M520 CPU at 2.4GHz speed for each core and 4 Gigabytes of memory running on an 32-bit Ubuntu 12.04 LTS operating system. On an average it took a little more than 2 seconds for generating each service. Approximately, 25 seconds were needed to generate all 11 services and they were deployed immediately in a service registry. During our tests we found that the time
<table>
<thead>
<tr>
<th>Service</th>
<th>Type</th>
<th>Description</th>
<th>SQL-template Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>S4</td>
<td>getYByX</td>
<td>Retrieve diagnosis id for a patient id</td>
<td>SELECT TABLE1.hdgWID AS X3 FROM Npatient AS TABLE0, NhrDiagnosis AS TABLE1, NhrAbstract AS TABLE2 WHERE TABLE0.patWID = &quot;Z0&quot; AND TABLE2.hraPatWID = &quot;Z0&quot; AND TABLE1.hdgHraWID = TABLE2.hraWID</td>
</tr>
<tr>
<td>S5</td>
<td>getYByX</td>
<td>Retrieve diagnosis code for a diagnosis id</td>
<td>SELECT TABLE0.HdgCd AS X2 FROM NhrDiagnosis AS TABLE0, NhrDiagnosis AS TABLE1 WHERE TABLE0.hdgWID = &quot;X0&quot; AND TABLE1.hdgWID = &quot;X0&quot;</td>
</tr>
<tr>
<td>S6</td>
<td>getYByX</td>
<td>Retrieve description of the admitting facility for a patient id</td>
<td>SELECT TABLE1.facFacilityDesc AS X3 FROM Npatient AS TABLE0, Nfacility AS TABLE1 WHERE TABLE0.patFacWID = TABLE1.facWID</td>
</tr>
<tr>
<td>S7</td>
<td>getYByX</td>
<td>Retrieve procedure id for a patient id</td>
<td>SELECT TABLE0.hprcWID AS X3 FROM NhrProcedure AS TABLE0, NhrProcedure AS TABLE1 WHERE TABLE0.hprcWID = &quot;Z0&quot; AND TABLE1.hprcWID = &quot;Z0&quot;</td>
</tr>
<tr>
<td>S8</td>
<td>getYByX</td>
<td>Retrieve start time of the procedure for a procedure id</td>
<td>SELECT TABLE0.phmStartDtm AS X2 FROM NphmIngredient AS TABLE0, Nservice AS TABLE1, Npatient AS TABLE2, Nencounter AS TABLE3 WHERE TABLE2.encPatWID = TABLE3.encWID</td>
</tr>
<tr>
<td>S9</td>
<td>getYByX</td>
<td>Retrieve pharmacy id for a patient id</td>
<td>SELECT TABLE0.phmWID AS X0 FROM NphmIngredient AS TABLE0, NhrProcedure AS TABLE1 WHERE TABLE0.phmSvcWID = TABLE1.svcWID AND TABLE1.svcEncWID = TABLE3.encWID</td>
</tr>
<tr>
<td>S10</td>
<td>getYByX</td>
<td>Retrieve start time of medication for a pharmacy id</td>
<td>SELECT TABLE0.phmStartDtm AS X2 FROM NphmIngredient AS TABLE0, NphmIngredient AS TABLE1 WHERE TABLE0.phmWID = &quot;Z0&quot; AND TABLE1.phmWID = &quot;Z0&quot;</td>
</tr>
<tr>
<td>S11</td>
<td>getYByX</td>
<td>Retrieve description of the prescribed medication for a pharmacy id</td>
<td>SELECT TABLE0.phmOrderDesc AS X1 FROM NphmIngredient AS TABLE0 WHERE TABLE0.phmWID = &quot;Z0&quot;</td>
</tr>
</tbody>
</table>

Table 5.2: Automatically generated SQL-template queries for S4 to S11. The variable Z0 in the condition will be instantiated by data value during the execution of the service.
shown here is essentially spent on generating the SQL-template queries because the code generation is almost instantaneous. For cases with multiple table-joins we found almost no difference in SQL generation.

<table>
<thead>
<tr>
<th>Service No.</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>2.5055</td>
</tr>
<tr>
<td>S2</td>
<td>2.3774</td>
</tr>
<tr>
<td>S3</td>
<td>2.3006</td>
</tr>
<tr>
<td>S4</td>
<td>2.2075</td>
</tr>
<tr>
<td>S5</td>
<td>2.2692</td>
</tr>
<tr>
<td>S6</td>
<td>2.3133</td>
</tr>
<tr>
<td>S7</td>
<td>2.1128</td>
</tr>
<tr>
<td>S8</td>
<td>2.0887</td>
</tr>
<tr>
<td>S9</td>
<td>2.0923</td>
</tr>
<tr>
<td>S10</td>
<td>2.0921</td>
</tr>
<tr>
<td>S11</td>
<td>2.0921</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>24.4515</strong></td>
</tr>
</tbody>
</table>

Table 5.3: Time (sec) spent in generating $S_1$ through $S_{11}$ using the Valet SADI framework

The overall time for creating new services should also include the time for establishing the necessary auxiliary resources, namely ontologies and schema mappings in PSOA RuleML. If these resources are preexisting the creation of services can be achieved with relative ease and timing described above. Increasingly ontologies are made available to the community through subject-specific registries or portals such as Bioportal [140]. The existence of these portals can accelerate the discovery of ontologies necessary for the mappings and the I/O definitions of the services. In cases where new mappings must be established the human effort required to create declarative semantic map-
pings must be considered in the overall speed of deployment.
While modeling of the service definitions may take a few minutes for a domain expert, semantic mappings generally take longer to write manually. Although simple schemas are not difficult to map if the domain ontologies have sufficient correspondences, the more complex the database schema is, the more time it may take to produce mappings. Unless the database schema and the terminologies change frequently, once the mapping rules are written at the beginning, the automation of service generation still saves lot of time and labor. As tools become more mature to produce mappings, the total time can only be reduced, thereby allowing service creation even faster.
Chapter 6

Querying HAI Data with SADI Services

In this chapter, we demonstrate how SPARQL queries are used to compose complex queries on the Hospital-Acquired Infections data by invoking the services generated automatically using the Valet SADI framework. We introduce an authentic query on HAI and present an SQL statement for its execution. The SQL statement shows the complexity of formulating such a query in terms of multiple table-joins which is a difficult task for non-technical users such as surveillance practitioners. We also show how a SPARQL query can do the same job by invoking all the HAI services listed as $S_1$ to $S_{11}$. Finally, we show how the graphical query interface on the HYDRA query engine provides an end user to compose query easily.
6.1 Query on Hospital-Acquired Infections

A Surgical Site Infection (SSI) sometimes occurs at the site of a surgical incision when germs get into the incision. Generally SSIs develop within 30 days of an operation. Infections may develop later as well when implants are used e.g. joint replacement operation. Although SSIs may be minor they are the most common type of infections which lead to longer stay or readmission to the hospital. One of the most preventative steps towards developing SSIs is the timing of administering antibiotics just before a hip or a knee joint replacement surgery. The recommended time is generally 30 minutes prior to incision \[141\] and the antibiotics should not be administered more than 2 hours prior to surgery. Therefore, administering antibiotics for the patients at risk of SSIs is of utmost importance. Hospitals generally store patients’ information in RDBs. By using SADI Web services, necessary information from RDBs can be extracted and the antibiotics can be administered in time. Let us assume that a surveillance practitioner is interested in the following query

“Find the first and last name, start time of the procedure, description of the order of the antibiotic, and the start time for administering the medication for all patients who have been admitted into the Orthopedic Surgery facility, diagnosed with Osteoarthritis of knee (ICD-10-CM code M17) \[142\], are scheduled to undergo a procedure for knee replacement but allergic to Penicillin as an
antibiotic to avoid Surgical Site infections.”

6.1.1 Query Composition and Execution

In order to answer the query the surveillance practitioner needs to access all 9 tables in the schema in Figure 5.2 and integrate the data.

```
SELECT TABLE0.phmOrderDesc AS X1, TABLE1.phmStartDtm AS X2, TABLE5.patLastName AS X3, TABLE6.patFirstName AS X4, TABLE9.hprcStartTime AS X5
FROM NphmIngredient AS TABLE0, Nservice AS TABLE1, Npatient AS TABLE2, NhrDiagnosis AS TABLE3, NhrProcedure AS TABLE4, NhrAbstract AS TABLE5, Nallergy AS TABLE6, Nfacility AS TABLE7, Nencounter AS TABLE8
WHERE TABLE3.HdgCd = "M17" AND TABLE6.algAdverseReaction = "Penicillin" AND TABLE7.facFacilityDesc = "Orthopedic Surgery" AND TABLE0.phmSvcWID = TABLE1.svcWID AND TABLE1.svcEncWID = TABLE8.encWID AND TABLE2.patWID = TABLE4.hprcWID AND TABLE2.patWID = TABLE5.hraPatWID AND TABLE2.patWID = TABLE6.algPatWID AND TABLE2.patWID = TABLE8.encPatWID AND TABLE2.patWID = TABLE8.encPatWID AND TABLE2.patWID = TABLE8.encPatWID AND TABLE2.patWID = TABLE8.encPatWID AND TABLE2.patWID = TABLE8.encPatWID
```

Figure 6.1: An SQL query asking “Find the first and last name, start time of the procedure, description of the order of the antibiotic, and the start time for administering the medication for all patients who have been admitted into the Orthopedic Surgery facility, diagnosed with Osteoarthritis of knee (ICD-10-CM code M17), are scheduled to undergo a procedure for knee replacement but allergic to Penicillin as an antibiotic to avoid Surgical Site infections.”

130
An SQL statement which may answer the query is shown in Figure 6.1 which uses a number of joins to retrieve the answers. A database programmer possesses such skills but a practitioner or a clinician is not expected to have such expertise. However, as a domain expert they are familiar with the domain terminologies and can formalize queries in SPARQL. As query clients such as SHARE and HYDRA allow users to execute SPARQL queries on SADI services, the surveillance practitioners can integrate the information by invoking them from the SPARQL query.

The following SPARQL query invokes all services $S_1$ through $S_{11}$ on either SHARE or HYDRA in order to answer the query. Line 1 retrieves the unique identifiers of all patients by invoking $S_1$, lines 2-4 are used to retrieve the unique identifiers based on the identifiers of the patients while lines 5-6 are used to find out which diagnosis codes match the ICD-10 code M17, lines 7-8 are used to retrieve both the last names and first names of these patients, lines 9-10 retrieve those who are undergoing a procedure, lines 11-12 are used to retrieve the start time of the procedure, lines 13-14 are used to find the services for patients’ medication administration, line 15 is used to find the start time for administering the medication, line 16 is used to identify the medication ordered for the patients, lines 17-19 are used to find out which patients are allergic to Penicillin, and lines 20-22 are used to identify those patients who are admitted to the Orthopedic Surgery facility of the hospital.
Figure 6.2: A SPARQL query asking “Find the first and last name, start time of the procedure, description of the order of the antibiotic, and the start time for administering the medication for all patients who have been admitted into the Orthopedic Surgery facility, diagnosed with Osteoarthritis of knee (ICD-10-CM code M17), are scheduled to undergo a procedure for knee replacement but allergic to Penicillin as an antibiotic to avoid Surgical Site infections.”

The SPARQL query can be executed either on the SHARE or on the HYDRA query client to retrieve the answers. All 11 services were automatically discovered, orchestrated, invoked and executed by the query clients.
6.1.2 Automatic Composition of SPARQL Queries

Although SHARE provides an auto-complete feature as part of its interface to a SPARQL query composer, the user needs to know how to formalize them. The large SPARQL query shown in Figure 6.2 is quite complex and requires proficiency in SPARQL language. Since average users are not expected to have such skills, visual interface could be an alternative approach which would allow them to compose queries graphically hiding the syntax behind the scene.

Recent developments in query platforms such as OptiqueVQS [143], FedViz [144], QueryVOWL [145], faceted search [146], and yearsSparqlFilterFlow [147] show increasing interests in providing visual SPARQL query creation capabilities. In a similar fashion, the HYDRA query engine provides a GUI which facilitates a relatively easier way of creating and executing SPARQL queries with SADI services.

Users can enter a series of keywords on the HYDRA query canvas and editable query graphs can be generated and modified. The variables in the queries can be added and modified until the end user is satisfied with the query. These features allow users to compose queries without having to know any SPARQL syntax, and therefore queries can be composed relatively quickly.

The manually created SPARQL query shown in Figure 6.2 can thus be created on the HYDRA GUI relatively easily using the keyword-based approach graphically. The query graph composed on the GUI is shown in Figure 6.3 which is equivalent to the raw SPARQL query in Figure 6.2.
A series of SPARQL queries build on HYDRA GUI for HAI surveillance are shown in Appendix A.6. The complexity of the queries grows progressively as new nodes and edges are added to the query graph. One or more of these edges represent a service from $S_1$ to $S_{11}$ created by Valet SADI.
Figure 6.3: Screenshot of the query created on HYDRA GUI using all 11 SADI services asking “Find the first and last name, start time of the procedure, description of the order of the antibiotic, and the start time for administering the medication for all patients who have been admitted into the Orthopedic Surgery facility, diagnosed with *Osteoarthritis of knee* (ICD-10-CM code M17), are scheduled to undergo a procedure for knee replacement but allergic to Penicillin as an antibiotic to avoid Surgical Site infections.”
Chapter 7

Preservation of Interoperability in Malaria Surveillance

According to the World Health Organization (WHO), malaria surveillance is presently weakest in countries and regions with the highest malaria burden. A core obstacle is that the data required to perform malaria surveillance is fragmented in multiple data silos distributed across geographical regions. Furthermore, consistent integrated malaria data sources are few and a low degree of interoperability exists between them. As a result, it is difficult to identify disease trends and plan for effective interventions.

The Semantics, Interoperability, and Evolution for Malaria Analytics (SIEMA) platform [20] for malaria surveillance is based on semantic data federation. Using SIEMA approach, it is possible to access distributed data, extend and preserve interoperability between multiple dynamic distributed malaria
resources and facilitate detection of system changes that can interrupt the mission-critical global surveillance activities.

7.1 Malaria Surveillance Today

Malaria is an infectious disease with significant impact on developing countries. In 2016 alone, there were 445,000 deaths worldwide [148] and globally more than 200 million cases of malaria have been reported in 91 countries. Populations in sub-Saharan African countries are most susceptible, with 80% of observed cases and recorded deaths worldwide [149, 150]. Whereas a decrease in case incidence has been observed since 2010, the rate of decline appears to have stalled, in part due to lack of adequate surveillance and intervention programs. An essential prerequisite for accelerating the decline of the disease and the optimal targeting of resources is an efficient surveillance infrastructure that can reliably deliver robust data sets. Poor data quality and sparseness and the coordination between different surveillance systems are ongoing challenges for the malaria surveillance community [151]. The increasing number of stakeholders including international organizations, governments, NGOs, and private sectors [152] that contribute in gathering the data can lead to siloed heterogeneous information systems and data sources which need to be integrated [153]. Overall, infrastructures for malaria surveillance are brittle [154] and stakeholders do not have sufficient confidence in the aggregated data sets to adopt any conclusions that could be meaningfully
A comprehensive study has shown that Malaria Information Systems (MIS) that can collect, store, analyze, and provide feedback to developers based on real-time information are few. They are generally limited by the absence of real-time data aggregation, inconsistent decision support, and low levels of resolution e.g. no mapping to households, and no extrapolation across geographical borders. In specific cases where an MIS has been upgraded with improved visualization and reporting tools, other challenges remain unresolved. Data entered from field stations are managed centrally and updates occurring at the central data source are not reflected in the field level immediately. Also, the options to generate reports are generally predetermined, hence ad-hoc queries cannot be answered without introducing significant overhaul of the systems. Consequently, the types of surveillance queries that can be run to derive actionable knowledge in a timely manner are relatively few.

Furthermore, eleven widely used Web platforms were studied in order to assess how internet and Web technologies are used in the fight against malaria. The elements of this study were focused on data, metadata, Web services and categories of users. The results revealed that although heterogeneous spatiotemporal malaria data came from multiple disciplines, they were rarely updated dynamically; no metadata was used to standardize them, Web services were inflexible for reuse and non-standardized, and the platforms primarily served the scientific communities. The authors opined that in order
to improve these systems, interoperability through standardization is necessary. The WHO’s Global Technical Strategy (GTS) \[161\] also identifies that surveillance challenges in sub-Saharan African countries exist partly because mechanisms for facilitating data integration from distributed data silos and ensuring system interoperability are lacking.

### 7.2 SIEMA Surveillance Platform

We introduce a prototype surveillance and change management platform, known as SIEMA, built from a combination of third party tools, community developed terminologies and custom algorithms. The methodology and core infrastructure used to facilitate interoperable access to distributed data sources using SADI Semantic Web services are illustrated using an architecture diagram in Figure 7.1. A dashboard that reports on terminology changes that can render SADI services inactive, jeopardizing system interoperability, is briefly mentioned which allows end users to control and reactively rebuild services to preserve interoperability and minimize service downtime.

#### 7.2.1 Resources

The SIEMA surveillance platform relies on the coordination and customization of a number of existing frameworks, and software and custom developed algorithms. The architecture diagram in Figure 7.1 describes these resources in an abstract representation of the SIEMA platform first introduced in \[162\].
Resources such as *SADI Semantic Web service*, *Query Clients for SADI Services*, and *Valet SADI* have already been introduced before. Therefore, we only focus on the rest of the resources below:

### 7.2.1.1 Source Data and Standard Terminologies

The current research focuses on middleware for enabling discovery of data sets and tools for agile query composition, but deployments of these method-
ologies beyond prototypes will require full access to malaria data. Existing
data repositories such as the Scalable Data Integration for Disease Surveillance [163] or Global Malaria Mapper [164] would be target resources for building service descriptions and including them in a generic registry. However, a recent study [160] suggests that in order to facilitate change management and semantic interoperability these data sources must be represented by standard terminologies related to their domain. Rather than defining new terminologies, it is a best practice to use community adopted domain terminologies, if available, for defining the services. For this purpose, several ontologies are leveraged including Ontology for Vector Surveillance and Management (VSMO) [165], the Malaria Ontology (IDOMAL) [166], the Mosquito Insecticide Resistance Ontology (MIRO) [167] and the Public Health Ontology (PHont) [168]. CDISC [169] also offers relevant ontologies.

7.2.1.2 Agent-based Analytics Dashboard

Detecting changes in the source data schema as well as the domain and service ontologies is a prerequisite step for change management. Studies on the evolution of large domain ontologies [170] have shown that the vocabularies are subject to change [171] because they have shared authorship, and they evolve to represent new knowledge. Because of the high degree of dependency between system components in a relational data schema, even a small change in a data schema could result in cascading impacts across an application stack. In a previous work, we presented features of a web-based
analytics dashboard for detecting and reporting changes [172]. Underpinning
the dashboard, software agents enable two key actions; i) detecting changes
and identifying their types, and ii) restoring the modified component to an
operational state through repair and rebuilding. The types of changes we
accommodate are; addition (i.e. extension), deletion (i.e. obsoleting), re-
naming (i.e. refining) of components in domain ontologies and service on-
tologies [173, 174], such as concepts, property restrictions, individuals, and
axioms. Meurice et al. [175] illustrates how changes in the source data such
as tables and their attributes, datatypes and indices can be identified, which
can then be reported on the dashboard. Likewise, changes to saved queries
and the corresponding service dependencies can be detected. Further details
of the agents designed to capture changes in various components of the in-
frastructure and their interaction with other software agents in the system
are reported in [162]. Figure 7.2 shows a snapshot of the landing page of the
dashboard.

7.3 Service-based Querying of Malaria Data

The target surveillance queries are inspired by the objectives stated in the
National Malaria Control Program by the Ugandan Ministry of Health [176].
These objectives resemble authentic queries for a surveillance practitioner in
the malaria domain. These questions have been converted into formal queries
by semantically presenting them as one or more SADI services. This section
Figure 7.2: Landing page of Agent-based analytics dashboard. The dashboard detects changes in the SIEMA components and identifies their types. The changes are currently reported for domain ontologies, service ontologies, relational databases, status of the services and the affected queries.
illustrates how the queries can be represented using a combination of SADI services on a graphical query canvas of the HYDRA query engine. They are listed below:

- **Q1.** Which indoor residual sprayings used permethrin as an insecticide?

- **Q2.** Which districts of Uganda that used permethrin-based long-lasting insecticide-treated nets in 2015 saw a decrease in anopholes gambiae s.s. population but no decrease of new malaria cases between 2015 and 2016?

- **Q3.** What are the future high-risk areas and at-risk time periods in Uganda?

### 7.3.1 Questions for Malaria Surveillance

Primarily a series of widely used malaria control interventions have been selected. The importance of Indoor Residual Spraying (IRS) has been well established by numerous studies throughout the world, especially in Africa [176][177], as an effective way of killing the mosquitoes that transmit malaria. Hence, the first question in Q1 is about IRS with a specific insecticide. This question is used as a running example in the rest of the paper. The use of insecticide-treated bed nets is the second most effective intervention against malaria. The question in Q2 is more complex to formalize and answer because it requires aggregating information about bed nets from geographic locations,
insecticides used, and a comparison of the status of mosquito populations within a certain time frame in a specific location. In Q3, it is anticipated that a malaria surveillance practitioner will be interested in predicting future trends and outbreaks to improve the allocation of resources where they are most needed. These three queries are considered as authentic target queries and adopted as a basis for illustrating and evaluating the capabilities and appropriateness of the proposed technical approach.

7.3.2 Building SADI Services

To answer the questions a list of SADI services has been created and deployed in a registry. We will focus henceforth on how to create those services.

7.3.2.1 Source Data and Vocabulary

Vocabularies from one or more domain ontologies are used to define the input and the output of a service. The data schema of the source data is also necessary. The vocabularies and the data schema in Figure 7.3 are used to define the SADI services. In the domain ontology, the hierarchy of classes includes VSMO:0001957 and MIRO:10000239 which represent the concepts of spraying and insecticide, respectively. The object properties has_insecticide and located_in are used to link the records of spraying to the records of insecticides and geographic regions. The data property has_name represents the literal values of names of the specific spraying activity, the geographic region, and the insecticide used. In the data schema, table spraying contains
information about all spraying activities; table geographicregion contains information about the locations where the insecticides were sprayed, and table insecticide contains insecticide information. In table spraying, the integer-valued attribute id is used to identify each spraying activity, attribute name represents the name of the activity, location.id refers to the location where the activity was performed, year represents the time it was performed, and insecticide.id refers to the insecticide used in the spray. In table geographicregion, the integer-valued attribute id represents each region and the attribute name represents the name of that region. Finally, in table insecticide, the integer-valued attribute id represents a pesticide, the attribute name identifies the name of the pesticide, and the attribute mode.of.action decides if the pesticide is effective on mosquitoes which come in contact or both contact & airborne. The attributes location.id and insecticide.id act as foreign keys to the region and insecticide, respectively.
7.3.2.2 Description of SADI Services

The input and the output of every service are defined in a service ontology using the terminologies from the domain ontologies. One such service is getInsecticideIdByIndoorResidualSprayingId, which takes an instance of spraying as input, that is any element whose type is indoor residual spraying. The service returns an id representing an insecticide. The input is decorated explicitly by the relation has_insecticide. The descriptions of this service are defined in Figure 7.4.
Another service is `get Name By InsecticideId`, which takes an instance of an insecticide as input. The service returns a string as output, representing the name of the insecticide in the data, decorating the input by the relation `has_name`. Figure 7.5 shows the I/O descriptions of this service.

7.3.2.3 Service Registry

Once the services have been generated, they are stored in a service registry. Figure 7.6 shows a fragment of the service registry that we use for malaria surveillance. HYDRA uses the registry to discover the services that will be called during the execution of the queries.
7.3.2.4 Provisioning of SADI Services

Building SADI services are cumbersome, error-prone and tedious for non-technical end users [11]. The steps for generating services automatically by Valet SADI were illustrated in Chapter 5 in detail. Thus Valet SADI allows a provision for accelerating the generation for these users once source data can be accessed via a suitable mapping from the domain ontologies using rules which are used to explain how to interpret the data. Figure 7.7 shows an example of such mapping rules. Once these rules are available, Valet SADI uses them to access the data by rewriting the description of the services into correct queries for the mapped data schema.
1. Forall ?insecticideID (identityForInsecticideToInsecticideID(
   identityForInsecticide(?insecticideID)) = ?insecticideID)
2. Forall ?P (identityForInsecticide(identityForInsecticideToInsecticideID(?P)) = ?P)
3. Forall ?id ?name ?mode.of.action (Insecticide(identityForInsecticide(?id)) :-
   db_insecticide(?id ?name))
4. Forall ?id ?name ?mode.of.action (has_name(identityForInsecticide(?id) ?name) :-
   db_insecticide(?id ?name))

Figure 7.7: Example of malaria-specific mapping rules in PSOA RuleML

7.3.2.5 Building Queries

To illustrate query building, consider the question Q1 introduced in Section 7.3 which is easily translated to a query. Its graph representation is shown in Figure 7.8. The graphical user interface of HYDRA makes it easier to build queries because relationships are easier to understand and process in graph form than in SPARQL syntax.

The query in Figure 7.8 calls on four distinct SADI services:

i) allPublicHealthActivities

ii) getNameByPublicHealthActivityId

iii) getInsecticideIdByIndoorResidualSprayingId, and

iv) getNameByInsecticideId

Services in iii) and iv) are described above while the service in i) retrieves all identifiers of public health activities in Uganda and the service in ii) retrieves the names of these activities. The branch on the right in Figure 7.8 uses the services i), iii) and iv) while the left branch requires
Figure 7.8: The graph representation of a query for the question “Which indoor residual sprayings used permethrin as an insecticide?” prepared on the HYDRA GUI.
services i) and ii). The construction of queries is performed in an incremental fashion. The root node starts with the service all `PublicHealthActivities`, which can then be extended either to the left or to the right. The service `get NameByPublicHealthActivityId` is used on the left to name the intervention represented by the variable `inter_{name}`. The root node is then decorated by the property `has_insecticide` with the description of the service `getInsecticideIdByIndoorResidualSprayingId` to represent an identifier of an insecticide. Finally, the service `get NameByInsecticideId` is used to represent the name of the insecticide with a value Permethrin. As queries become more complex, the advantage of graph form over the raw SPARQL syntax becomes evident. The graph-form of the question in Q2 is shown in Figure 7.9. Although the query graph is significantly more complex and larger than the query in Q1, it is much easier to understand for any user than the raw SPARQL representation. The descriptions of each service used in the query graph of Q2, as well as in Q3 are avoided here due to space constraints.

### 7.4 Preservation of Interoperability

The previous section has outlined how, using Semantic Web services, it is possible to answer complex questions relevant to malaria surveillance. Special attention is needed before considering the introduction of a new methodology in a dynamic context where data and middleware are not static. A review of several possible changes that could occur in a malaria surveil-
Figure 7.9: Which districts of Uganda that used permethrin-based long-lasting insecticide-treated nets in 2015 saw a decrease in anopheles gambiae s.s. population but no decrease of new malaria cases between 2015 and 2016?
lance framework has been described and classified according to the degree to which they impact data access and their likelihood to impact interoperability of the system. In the following, an approach to address the notion of change management through the introduction of change detection tools and triggers for the reactive rebuilding of service to ensure service uptime will be illustrated. Specifically, we discuss renaming and addition of terms that are used in service ontologies and the use of a reporting tool for detecting and displaying these types of changes.

7.4.1 Changes in the Service Ontologies

Whenever a definition of an existing service is modified, the associated service ontology is changed but the code implementing the service remains unchanged. As a result, when a service is invoked during the execution of a query it fails to return the anticipated output because the terms used in the code are incompatible with the new definition in the service ontology. In the SIEMA framework the change capture agent implemented within the dashboard detects the changes in the terms used in the service ontology by comparing the modified version of the ontology to the one it was modified from. The role of the change capture agent can be illustrated in the case of a term addition.
7.4.1.1 Example of Addition

To illustrate a scenario involving the addition of new terms, consider the query in Figure 7.8 again. The mode of action, mentioned in the WHO recommendation [178], plays an important role in choosing insecticides for indoor residual spraying against malaria vectors.

Figure 7.10: Which indoor residual spraying used permethrin as an insecticide and which kind of mosquitoes will be affected by it?

Let us assume that the end user is interested in the query “Which indoor residual spraying used permethrin as an insecticide and which kind of mosquitoes will be affected by it?”. Figure 7.10 shows the graph representation of such a query. To be able to answer this query, the existing service getNameByInsecticideId can be modified by adding the term
has_mode_of_action and the datatype xsd:string that it returns. The new output description in Figure 7.11 is defined by adding the two terms to the old description.

![Figure 7.11: Old (left) and new (right) output description of the service get\_NameBy\_InsecticideId](image)

A change capture agent detects the changes of two terms, identifies them as addition, and displays them on the dashboard in a tabular form. Figure 7.12 shows the consolidated information about the changes on the dashboard which displays the time the changes were detected under the Timestamp column, the textual description of the change under the Description of Change column, the types of changes identified as either addition under the Entity Added column, deletion under the Entity Deleted column, or rename under the Entity Renamed column, the list of services impacted under the Affected Service column, and the list of queries impacted under the Affected Query column.

### 7.4.2 Status of Services

At any time, the status of all deployed services in the registry is displayed on the dashboard. Services can be either active which can be used in queries,
## Service Ontologies

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Description of Change</th>
<th>Entity Added</th>
<th>Entity Deleted</th>
<th>Entity Renamed</th>
<th>Affected Service</th>
<th>Affected Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>2018-01-21T14:33:08</td>
<td>An entity is added to the output definition</td>
<td>has_mode_of_action</td>
<td>-</td>
<td>-</td>
<td>getNamelyInsecticideId</td>
<td>Which indoor residual sprayings used permethrin as an insecticide?</td>
</tr>
<tr>
<td>2018-01-21T14:33:08</td>
<td>An entity is added to the output definition</td>
<td>xsd:string</td>
<td>-</td>
<td>-</td>
<td>getNamelyInsecticideId</td>
<td>Which indoor residual sprayings used permethrin as an insecticide?</td>
</tr>
</tbody>
</table>

Figure 7.12: Detection and identification of changes in the service ontology
or inactive which need to be repaired before using them again. Figure 7.13 displays a list of active and inactive services, which display in color as green and red, respectively. The URI of the service is tabulated under the Service URI column, the description is under Description, the time the service was created is under Time of Creation, and the time the service was rebuilt is under Time of Rebuild. The final column is Request Rebuild that allows for placing an inactive service in a queue to be rebuilt and subsequently redeployed in the service registry. The get
\textit{NameByInsecticideId} service is shown in red because it became inactive due to the addition of terms in its output definition.

### 7.4.3 Reacting to Changes

The addition of terms to the definition of an active service renders the service description incompatible with the target functionality and existing service code and renders associated queries dysfunctional. To resolve the inconsistency, it is necessary to repair and rebuild the services in line with the new requirements. Specifically, the end-user now wants to access the data that was not previously available from a service, namely, in the current example the mode of action of an insecticide. It is thus necessary to re-implement the service corresponding to the altered service description ensuring the domain ontologies, the data schemata, and the PSOA semantic mapping rules underpinning the service are accurate and will support the new target functionality.
**Status of Services**

<table>
<thead>
<tr>
<th>Service URI</th>
<th>Description</th>
<th>Time of Creation</th>
<th>Time of Rebuild</th>
<th>Request Rebuild</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://localhost:9999/sadi.services/getNameByPublicHealthActivityId">http://localhost:9999/sadi.services/getNameByPublicHealthActivityId</a></td>
<td>Retrieves the name of a population health activity</td>
<td>2018-01-21T14:33:08</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td><a href="http://localhost:9999/sadi.services/getNameByInsecticideId">http://localhost:9999/sadi.services/getNameByInsecticideId</a></td>
<td>INACTIVE</td>
<td>2018-01-21T14:33:08</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td><a href="http://localhost:9999/sadi.services/getInsecticideIdByIndoorResidualSprayingId">http://localhost:9999/sadi.services/getInsecticideIdByIndoorResidualSprayingId</a></td>
<td>Retrieves the insecticide of an IRS</td>
<td>2018-01-21T14:33:08</td>
<td>N/A</td>
<td></td>
</tr>
</tbody>
</table>

Figure 7.13: Status of SADI services. Active services are shown in green and inactive services in red.
Given that a data resource contains the information about the mode of action of insecticides, the key question is whether the semantic mapping rules already map that data to an existing concept or relation of the domain ontologies. If it is the case, all components required to rebuild the services exist and it is possible to proceed to the next step in the Valet SADI rebuild. Otherwise, it is necessary to identify missing rules and add them, or extend a local domain ontology with missing concept or relation that exists in the service ontology. Once this is done a rule must be created to define a new mapping and making possible to rebuild the service.

Figure 7.14: Fragment of the updated malaria-specific PSOA rules

7.4.4 Rebuilding the Services using Valet SADI

By leveraging Valet SADI’s auto-generation capability, the damaged service can be quickly rebuilt and deployed once changes are detected, identified, and a rebuild is requested. To illustrate this, we refer to the query shown in Figure 7.10 that calls services retrieving data from the source database schema shown in Figure 7.3 and list the changes below.
7.4.4.1 Modified Domain Ontology

In the domain ontology, the data property \textit{has\_mode\_of\_action} is added. As a result, the service ontology becomes compatible and supports the extended query as it only uses existing concepts and properties.

7.4.4.2 Modified Malaria-specific Mapping Rules

The PSOA rules are also modified to populate the newly added data property \textit{has\_mode\_of\_action} as shown in Figure 7.14. The lines 10’-12’ are added in contrast to the original version in Figure 7.8. Valet SADI can rewrite the modified input and output descriptions with the help of the mapping rules, generate the SQL query and the complete program code automatically to fetch answers from the source data. The services are thus rebuilt and redeployed in the service registry so that the modified query returns the correct answers. Figure 7.15 shows the updated status of the services. The \texttt{get\_NameByInsecticideId} service is now shown in green because the service was rebuilt by Valet SADI and redeployed in the service registry. The exact time the service was rebuilt is tabulated along the Time of Rebuild column.

7.5 Discussion

Surveillance remains a challenge for the malaria community and many factors play a role in limiting access to relevant data resources for analysis and reuse \cite{160}. In this study, the suitability of a solution to the challenges of sys-
Status of Services

<table>
<thead>
<tr>
<th>Service URI</th>
<th>Description</th>
<th>Time of Creation</th>
<th>Time of Rebuild</th>
<th>Rebuild</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://localhost:9999/sadi-services/getNameByPublicHealthActivityId">http://localhost:9999/sadi-services/getNameByPublicHealthActivityId</a></td>
<td>Retrieves the name of a population health activity</td>
<td>2018-01-21T14:33:08</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td><a href="http://localhost:9999/sadi-services/getNameByInsecticideId">http://localhost:9999/sadi-services/getNameByInsecticideId</a></td>
<td>Retrieves the name of an insecticide</td>
<td>2018-01-21T14:33:08</td>
<td>2018-01-23T09:03:15</td>
<td></td>
</tr>
<tr>
<td><a href="http://localhost:9999/sadi-services/getInsecticideIdByIndoorResidualSprayingId">http://localhost:9999/sadi-services/getInsecticideIdByIndoorResidualSprayingId</a></td>
<td>Retrieves the insecticide of an IRS</td>
<td>2018-01-21T14:33:08</td>
<td>N/A</td>
<td></td>
</tr>
</tbody>
</table>

Figure 7.15: Status of services after being rebuilt by Valet SADI
tem interoperability, distributed data access, semantic integration of data and semantic support for query composition has been investigated. The SIEMA framework comprises a number technologies and standards and is further customized to address the proposed targets needs and interests of surveillance practitioners. The contribution takes three main directions. First, using SADI web services allows for the easy access to distributed data. This task is simplified further using Valet SADI, which enables a programmer to create services in an efficient and straightforward way. Second, due to the user interface features of the HYDRA query engine, SIEMA offers end-users a more appealing way to build surveillance queries. HYDRA’s ability to discover and call the services that are needed for a query permits the user to simply use the data as an abstract construct without having to look at its actual structure. Third, to make the system more robust and flexible a dashboard has been introduced. The dashboard informs users when changes have occurred that render the services and or queries inactive. This enables users to know which queries may no longer be reliable and identify which parts of the service infrastructure must be rebuilt to restore it to its fully interoperable state. Deployed together this combination of technologies offered by the SIEMA framework, is exhibiting key functionalities that are of great value to the community.

The initial studies in malaria surveillance [162, 179] and other domains [11, 14, 180] showed us that this approach is a viable solution for enhancing interoperability. The successful extension of this methodology to the malaria use
case appears promising based on the results from the initial implementation. The addition of a dashboard and Valet SADI extend the capabilities of the SADI framework to make it suitable for change detection, service restoration and preservation of interoperability.

Whereas a systematic evaluation of the many components of this framework, individually and together, is beyond the scope of this initial study, we are aware of other malaria surveillance systems in sub-Saharan African countries have been reported and evaluated in part \cite{159,181} according to attributes recommended by Center for Disease Control and Prevention (CDC) \cite{182}. Indeed, it would be valuable to make some direct system comparisons. Regrettably, few technical details of these systems are reported and their key features such as the architecture, supported datatypes and data quality, the use of data representation standards, capacity for change management, and degree of interoperability supported have not been disclosed. Without such information it is not possible to compare them with SIEMA. It is apparent, however, that the systems were designed for centralized data warehousing rather than seamless access to distributed data, and this is a significant distinction.

A brief assessment of SIEMA according to attributes recommended by CDCP, namely simplicity, data quality, flexibility, stability, timeliness, can be made based on the initial experiments reported in the presented study to date. To address the attribute of simplicity, the activity of creating surveillance queries can be assessed. Whereas writing syntactic queries (SPARQL) for
submission to a query client manually requires that a user has expertise with
the query language, the HYDRA GUI can save users time and allow more
complex surveillance queries to be composed by persons with less technical
skill [183]. Figure 7.8 shows a graphical representation of a query built with
the GUI which in turn is translated to SPARQL automatically. To address
the attribute of data quality SIEMA’s adoption of SADI ensures that W3C
standard OWL, RDF[S] is used to describe and format every piece of data
accessed through the framework. The attribute of flexibility is also met since
access to distributed data can be provisioned by using SADI services that are
easily deployed in a registry, making them discoverable and readily usable
for a variety of ad-hoc queries. Likewise, the attributes of stability and time-
liness of the system are met as the implemented dashboard tracks service
uptime, reporting failures immediately after detecting the faults. System
rebuilds with Valet SADI ensure stability so data can continue to be pro-
vided and the system stays operational if changes occur. In this way there
is a preservation of interoperability and any service downtime is kept to a
minimum.

Overall, it is anticipated that the ongoing trials with the SIEMA framework
will give the research and development team further insight into the real-
world requirements for interoperability and change management in malaria
surveillance leading to further improvements in adaptability and performance.
Given the critical need for timely integration of distributed data from multi-
ple heterogeneous sources in an efficient way, we hope to build co-operative
partnerships between multiple disciplines, organizations, and sectors. In addition, insights gained from this research are likely transferable to a range of global surveillance projects.

7.6 Conclusion

In this chapter, it has been demonstrated that authentic questions asked in malaria surveillance can be formalized as queries and mapped to a combination of Semantic Web services designed to deliver target data from distributed data sources. It has been shown that using SIEMA and leveraging terminologies from community developed ontologies offers flexibility both for integrating data and easy query composition. The developed infrastructure also offers a solution to the problem of change management, an important process for maintaining interoperability and integrity of an integrated surveillance system. Given that changes in the form of addition/renaming/deletion of terminologies can frequently occur in the face of evolving system requirements, a change management dashboard was introduced. This makes it possible to identify important changes, report on the status of services as a consequence of changes and offer users the option to rebuild inactive services. The dashboard and service re-authoring routines serve as an important vehicle to maintain system interoperability of mission-critical global surveillance programs. The infrastructure has been implemented and its relevance has been demonstrated with an authentic use case, with the goal of soliciting
further requirements from the Malaria Analytics community. In future work SIEMA will be deployed on live dynamic data sources.
Chapter 8

Conclusion and Future Work

In this chapter, we will summarize our main contributions which have been published in various venues. We will also list some of the key benefits of our work and provide arguments on their behalf. Finally, we will conclude this chapter by describing some of the shortcomings of our work and list potential research directions for future in order to improve the Valet SADI framework.

8.1 Contributions

In this work, we addressed the problem encountered in one aspect of semantic querying paradigm where a large number of Semantic Web services need to be deployed on top of one or more relational databases manually in order to draw data by end users who are familiar with the domain of interest but lack technical skills in querying databases. An infrastructure which can support
such semantic querying capability could be assembled by provisioning two technology stacks, a registry of Semantic Web services and a query client which can discover, orchestrate, and invoke these services to execute them. For SADI Semantic Web services query clients such as SHARE and HYDRA do exist. However, a number of research studies and subsequent real-world deployments of SADI services in [11–14, 16, 17] have proved that creating SADI services manually is a time-consuming and error-prone endeavor, and it requires understanding of a number of resources and competencies in certain technical areas. Acquiring the skills for a user requires efforts and perseverance which are not expected of domain experts in non-technical fields. In order to take full advantage of query clients we realized that SADI service needs to be generated and deployed rapidly. Creating SQL queries to run on the databases still remain a hurdle which renders the generation process slow. Therefore, automation of generating both the service code and SQL query has become a critical target of our work.

We have implemented the Valet SADI framework which is used to automate the task of generating SADI Semantic Web services and SQL queries from their declarative input and output descriptions. Valet SADI is inspired by the paradigm of semantic querying where one or more relational databases store the source data, and one or more expressive knowledge bases for a domain exist, to which the database schema is related with such a degree that suitable explicit mappings can be specified in the form of axioms or rules. These mapping are logical description of the relations in the database
to link them to the concepts and relations defined by the knowledge bases. The goal of semantic querying is to permit end users to be able to formulate queries logically using the terminologies in their domain (such as concepts, relations) and answer them with respect to the knowledge bases and the databases. The services generated by Valet SADI consume input values in RDF format, execute SQL queries to retrieve answers from the source data in one or more relational databases, and create output in RDF format. The services are deployed in a service registry and are automatically discovered, orchestrated, and invoked from a SPARQL query client when an end user executes a SPARQL query.

In Chapter 1 we have provided an overview of existing approaches to accessing information stored in relational databases from standardized knowledge bases of domains of interests. We have listed the statements of problems when Semantic Web service-based approaches were previously deployed. We have concluded the chapter by providing our solutions to these problems in the form of a framework.

In Chapter 2 we have briefly described the resources and technologies in order to explain our contributions. The topics include the transition from traditional Web towards the Semantic Web and traditional Web services to Semantic Web services. We have explained the operations of SADI Semantic Web services in particular, and how their operations compare to the operations of other Semantic Web services. We have concluded the chapter by briefly mentioning the relationships between relational databases and the
Semantic Web.

In Chapter 3, we have introduced our main contribution as a framework named Valet SADI with a compact diagram. The modular architecture of Valet SADI offers a concise description of each component. There are four modules which work in tandem to produce the final service and the components in these modules integrate a number of essential inputs. In Module-1 the product is a set of semantic mapping rules which are defined manually from the correspondence of one or more relational databases as one form of input and domain ontologies as the knowledge base as the other input. In Module-2, declarative input and output OWL class expressions are the inputs described manually in a service ontology and the outputs are the graph representations of these expressions. The tools in Module-3 process the outputs both these modules and produce an (uninstantiated) SQL-template query automatically. Finally, the components in Module-4 use the template query and the graphs from Module-3 as their inputs to produce the source code of a SADI service in Java along with the supporting source files, which are then compiled, packaged, and deployed as a Web application service into a SADI registry. During this whole process only Module-1 and Module-2 require human intervention. In order to assist explaining the tasks carried out by the modules of the compact diagram, an expanded diagram for each module has been included.

In Chapter 4, having introduced the architecture, we have developed a number of algorithms and reused existing resources to implement the operations
of each module.

The mapping rules in Module-1 are written in a compact rule language in PSOA RuleML and verified manually, possibly by a Semantic Web engineer. We have provided a set of rules in Section 4.1.1 which are sufficient to create simple mapping. Although rule languages such as RIF-BLD, RIF-PRD, Prolog have been in existence and they can be used for semantic mapping, PSOA has become a viable alternative since its inception. Rules expressed in PSOA RuleML allow a uniform representation of both positional and slotted arguments in a single psoa term compared to RIF rules. For mapping, we have used positional arguments because they are suitable for expressing relational knowledge bases. We have chosen PSOA rules because without losing their expressiveness they can be translated to First-Order formulas in TPTP syntax which are compatible with the VampirePrime reasoner, although other rule syntaxes could have been used if their translation to TPTP had been available.

In order to generate graphs in Module-2 from OWL class expressions we have developed an algorithm in Section 4.2 which uses some basic features of an existing library for creating and traversing graphs [184].

The components in Module-4 integrate four inputs to generate an SQL-template query. In Section 4.3.1 we have developed an algorithm which is used by the converter to produce a semantic query in TPTP format from the generated graphs in Module-2. For translating domain ontologies into TPTP-FOF axioms, an existing translator has been extended by the latest OWL
constructs. The translator uses the latest features from OWL-API [185] tool for OWL ontologies. We have implemented another translator to perform a direct translation of the semantic mapping rules written in PSOA RuleML into TPTP-FOF axioms. The abstractions of the databases are direct representations of the relations from the database schemas, hence no translation is required. The VampirePrime reasoner uses the abstractions, the semantic query, the translated domain ontology, and the translated mapping rules, all in TPTP format, in order to generate schematic answers which are then converted to SQL queries using an SQL generator. All these components in Module-4 are used to transform the service descriptions into SQL queries based on the Incremental Query Rewriting technique.

The Generator component in Module-4 integrates the SQL query capable of joining multiple tables into the source code of a SADI service. The source code for reading and writing RDF data are produced by traversing the graphs generated in Module-2. We have developed an algorithm in Section 4.4 which creates the source code for a SADI service automatically in such an order that the statements i) read input values from RDF input data if available, ii) instantiate the SQL query with the input values, and iii) create the output data in RDF format with the results after executing the query and retrieving the results. All other auxiliary source files are also automatically created so that together they can be used as a Web application. The Web Service Deployer is then used to compile, package, and index services in a registry. After the services are deployed, SHARE and HYDRA can look up service
descriptions matching an end user’s SPAQL query and invoke the service to return results. The availability and consideration of the various features of these clients also played a role in the design of Valet SADI and the realization of the automation of service generation.

In Chapter 5, we have created 11 SADI services using Valet SADI. We have demonstrated the operational steps implemented in each module for creating a variety of SADI services. We have been able to generate the services within a total of almost 25 seconds once the mapping and the services descriptions were ready. We have tabulated the time taken by Valet SADI for each service.

In Chapter 6, we have demonstrated how SPARQL queries can be used to discover, invoke and execute SADI services. The SPARQL query is shown in a textual form which has been used in the SHARE query client. We have built an equivalent graphical query on HYDRA, which can also import queries in text form submitted to SHARE.

In Chapter 7, we have presented the SIEMA surveillance and change management platform as one of the deployments. SIEMA supports the creation of flexible surveillance queries across distributed data resources. It uses SADI Semantic Web services to enable data access and improve interoperability and the GUI-enabled semantic query engine HYDRA to implement the target queries typical of malaria surveillance. We have implemented a custom algorithm to detect changes to community-developed terminologies, data sources, and services that are core to SIEMA. A dashboard reports these changes to the user and detects loss of interoperability. Valet SADI is used in two stages,
to create new services to enable distributed data access, and then again to mitigate the impact of changes by rebuilding the affected services.

8.2 Benefits of Valet SADI

Several infrastructures were deployed using SADI services in order to implement target queries from heterogeneous data sources. Although the deployments provided improved ability for querying compared to the existing systems, several drawbacks were identified because of the manual creation of SADI services which are i) time consuming, ii) prone to errors, and iii) unattractive to both data providers and query composers alike because i) and ii) are the main barriers for adopting a service-based approach.

The automation provided by our Valet SADI framework addresses the above shortcomings to a great extent. These benefits have been illustrated by the real-world deployments for surveillance of Hospital-Acquired Infections and malaria prevention and control, as part of global health surveillance infrastructure.

A prototype SADI service-based surveillance infrastructure HAIKU [11] was deployed where semantic querying was implemented on top of an extract of The Ottawa Hospital Data Warehouse. In this prototype services were created and deployed to answer sets of authentic queries. These services were built manually by an experienced group of Semantic Web engineers and researchers over a period of two years. Most of the time was spent on writing
program codes and correct SQL queries. Our tests reveal that with a similar infrastructure Valet SADI can create similar program code and SQL queries in only 2 seconds, which speeds up the deployment of SADI-based systems significantly.

One way HAIKU differs from our Valet SADI framework is the presence of semantic mapping rules. Although the code generated by Valet SADI is not error-prone, which the HAIKU prototype suffers from, the mapping rules still have to be verified by an expert once they are created. Moreover, if there are changes either in the ontologies or in the database schemas, the rules have to be re-verified and rewritten to address the correct correspondences, which is still manually done in our framework. There are several reports of mapping tools that offer semi-automatic and sometimes automatic mapping depending on the complexity of the schema and the ontologies. Some of them are two-stage systems such as BootOX [186], Automap4OBDA [187] which produce a target-agnostic ontology followed by an alignment to the target ontology. IncMap [188] is an example of one-stage system which directly produces the mapping. Among the other mapping tools ontop [189], MIRROR [190], COMA++ [191], MASTRO [192] are notable. However, each of these tools have shortcomings because they fail to detect $n : m$ (i.e. many-to-many relations), due to junction table, and $n : 1$ class-table relationships, due to column splitting. Many of them do not utilize annotation properties and miss out on many properties matching for tables having $1 : 1$ relationship with other tables. For these reasons a combination of mapping tools are sometimes
used depending on the features of the schemas and the ontologies, and their correspondence. Most of the tools cannot automate a mapping if there is a low degree of lexical correspondence. This is primarily why successful automation of mapping is difficult to achieve, and needs human intervention. There is very likely an incorrect correspondence as the schema or the ontology or both evolve over time. Similarly, semantic mapping in the Valet SADI architecture needs to go through manual verification to avoid errors, and the absence of an automated mapping is not considered a shortcoming. Given the modular architecture it is conceivable that future 3rd party mapping tools with high accuracy could be incorporated.

Valet SADI offers ease of use for data providers who are interested in sharing their data and for query composers who are end users able to formalize SPARQL queries to retrieve answers from these data.

Data providers generally permit end users to access their data by providing an Application Programming Interface (API) in various languages so that they can be securely accessed. The parameters described in these APIs are syntactic in nature which application developers with technical expertise understand and can build custom applications on top. If the data providers were to expose these data using Semantic Web services such as SADI, machines (query clients) could find the data based on the semantics embedded in them. Instead of providing multiple APIs to attract a wide range of application developers which is difficult to maintain, Valet SADI can be used to build registries full of SADI services very quickly. While it is one thing
for a data provider to offer an API, it is more attractive to offer many services which are easily findable and accessible, as described by the recent introduction of the FAIR data principle [61]. End users also benefit when there are sufficient number of services exposing source data. One of the key advantages of the SADI query client HYDRA is that end users with less technical skills can compose SPARQL queries graphically by typing key phrases on the HYDRA GUI. The query graph is automatically converted into the equivalent SPARQL query which the user does not required any expertise in. Due to the existence of this type of query client, the pool of end users that could adopt SADI is potentially quite large to build and share queries.

8.3 SADI Service-based System Deployment

A number of SADI service-based systems have been deployed where mission-critical tasks on more than one data resources have been performed. With Valet SADI the services were built quickly and with SHARE and HYDRA the services were automatically discovered, orchestrated, invoked and executed from an SPARQL query in an orderly fashion based on an intelligent query planning. Some of the key deployments are briefly summarized below:
8.3.1 Surveillance for Hospital-Acquired Infections

The availability of Valet SADI in combination with HYDRA allows domain experts/end users to pose authentic queries with a high degree of complexity which would previously have been achieved by a software developer. A number of SADI service-based systems have been deployed where mission-critical tasks on more than one data resources have been performed. In most of these cases the services were built manually. One of these is an infrastructure for surveillance of infections [11], where a list of authentic target queries were executed by using SADI services. The services were built manually which was time-consuming. In Chapter 5-6 it was illustrated how surveillance tasks for Hospital-Acquired Infectious (HAI) diseases can be performed by using a set of 11 SADI services created by our Valet SADI framework. Once semantic mappings were written and the input and the outputs of a SADI services were created, Valet SADI took approximately 25 seconds to create all of the 11 SADI services. In Chapter 6 we have demonstrated a complex SPARQL query which requires invoking all services to execute a complex query involving all tables in the schema shown in Section 5.2.1.1. The SPARQL query was executed successfully on both SHARE and HYDRA and the results returned by the services were integrated and shown to the end users which were verified.

The queries were evaluated on SHARE and HYDRA query clients which returned correct results.
8.3.2 Surveillance of Malaria Control and Intervention

The speed at which Valet SADI can be used to build services has proved to be very useful in the context of surveillance and in particular, surveillance in global health. The ongoing challenges in this sector arise from the fact that the environment they are operated in requires access to distributed resources across different regions. Few surveillance systems deployed in sub-Saharan African countries which have been reported in [156] can cope with the challenge because they lack real-time aggregation of sparse resources. The majority of these systems are operated on centralized databases and among them a few [157-159] have been upgraded with improved visualization and reporting tools, while the main challenge to integrate distributed resources remains unresolved.

In a surveillance system overlooking distributed resources, when resources are independently managed and poorly coordinated, changes occurring in resources in one location are not reflected globally, resulting in a loss of interoperability of the system. Our recent work in [162] identified that when resources evolve over time, the changes occurring in them need to be managed properly to avoid the loss of interoperability. Next, SADI web services were deployed on top of distributed resources in [179] to demonstrate the effectiveness of SPARQL queries on distributed resources. A dashboard (see 7.2.1.2) implemented for detecting and identifying the changes and their types is reported in [172]. The types of changes detected in domain ontologies and service ontologies are addition (i.e. extension), deletion (i.e. obsoleting), re-
naming (i.e. refining) of terminologies. In the source database, changes are
detected if they occur in tables and their attributes, data types and indices.
Likewise, changes to saved queries and the corresponding service dependen-
cies can be detected. The integration of Valet SADI with the dashboard led
to a viable solution for the interoperability problem.
The integration is seamless across various regions because the schemas in the
backend are accessed using meaningful terminologies which are standardized
during the ontology development. The availability of HYDRA gives end
users access to the services from anywhere across the region. The dashboard
detects any changes in resources and reports them to the authority so that a
service which is affected by them can be noticed immediately. This reports
loss of interoperability of the surveillance system which needs to be restored
to minimize the downtime of the affected service. This can be done simply
by regenerating the service which is done by bringing Valet SADI back to
regenerating the same service with the new changes. The SIEMA architecture
illustrated in Figure 7.1 shows interactions among Valet SADI, the dashboard
and the invocation of Valet SADI again by automated software agents when
services are down due to changes. The deployment of SIEMA surveillance
system for malaria intervention has been described in detail in Chapter 7.

8.3.3 Decision Support System in Agriculture

Another deployment of SADI service using Valet SADI is reported in [68],
but not in detail in this thesis, which describes how agricultural consultants
accessing distributed data silos can integrate relevant information to offer useful information such as finding optimal crop varieties and computing profit margins of each variety using a complex cost model. A total of 9 services were created and deployed in a service registry. The range of queries include helping a farmer decide what eggplant variety to plant, finding how much the farmer hopes to harvest based on the potential profit margin, choosing pesticide and its effect on bees. Among the services there are 4 services in which SQL queries are used to retrieve data from the tables. Therefore, these services were generated by Valet SADI. The rest of the services perform various computations only, and were beyond the scope of the current implementation.

### 8.4 Open Problems of Valet SADI

The independent deployments of SADI services in three different areas prove the usefulness of the Valet SADI framework in setting up a registry of services for querying relational data semantically, which is a speedy and less error-prone operation and offers benefits to data providers as well as end users. The framework also offers an avenue towards preserving system interoperability in support of change management when minimizing downtime is critically important. This feature opens a door to many domains where a service-based semantic querying infrastructure can be deployed on top of single or distributed relational databases.
However, it is observed that Valet SADI suffers from a few limitations. The services created by Valet SADI are essentially query services, hence they are read only. Currently there are no support for create, update and delete operations to be performed by services created by the tool.

The mapping rules in Valet SADI need to be created and verified manually until either the ontologies or the database schemas evolve. Most of the mapping tools are based on the Ontology-based Data Access (OBDA) principle, and to our knowledge there are no mapping tools which automate the creation of mapping rules in an expressive language such as PSOA RuleML. Although the OBDA mapping tools have considerably matured over time, the mappings they produce still need to be verified by experts manually. Until such tools are available the mapping rules will be prone to error, especially where the database and/or the ontologies evolve.

The deployment of SADI service-based decision support system in agriculture shows a trivial limitation of generating services from declarative input and output descriptions. Specifically a declarative expression denotes what an input or an output should be, but does not say how to get there. Therefore, even simple computations cannot be expressed in OWL expressions which, in turn, cannot be implemented in a SADI service automatically. Thus, a system requiring some computing services cannot be created completely automatically by Valet SADI.
8.5 Future Work

In the short-term future, one of the goals is to develop a tool which produces mapping rules in PSOA RuleML automatically. We believe that a rudimentary mapping tool can be developed fairly quickly based on the mapping rules described in Section 4.1.1 if the structure of the relational schema and the domain ontologies are simple and there is a certain degree of lexical correspondence. There are tools such as AutoMap4OBDA [187], Map-on [193], Karma [194], OptiqueVQS [195] which offer users a visual mapping experience and are currently getting much attention. Therefore, we would like to provide a comparable GUI to make the mapping experience attractive for the adopters of Valet SADI.

During our work on the deployment of SIEMA (see Chapter 7) we have learned that field workers in many parts of the world collect data manually and store them in simple spreadsheets (e.g. comma separated values (*.csv), Excel (*.xlsx)). We have had similar experience during the deployment of SADI service-based decision support system [68] in the agriculture domain. This made us aware of the potential of reaching a wider range of users and the importance of Valet SADI if it can be used to generate services automatically and deploy them quickly on top of spreadsheets. Therefore, another area for future extensions lies in the inclusion of simple spreadsheets as the source data in Valet SADI.

Another potential area of future work is in ability to determine the number
of potential services that can be generated by the Valet SADI framework automatically by looking at the database schemas and the domain ontologies. Our attempts, although not documented in this thesis, have shown that this can be accomplished because simple input and output OWL class expressions can be automatically created by randomly combining classes and properties from the domain ontologies. However, these expressions might not be semantically meaningful to the end users and may need to be verified manually by domain experts. There is also no guarantee that any SQL queries can be generated from these exercises by Valet SADI. However, this is an interesting problem area where more research and experiments are needed.

The usefulness of Valet SADI framework has been proven in SIEMA for preserving semantic interoperability while accessing distributed resources across different regions. Simply by using Valet SADI in two stages with a supporting dashboard, we were able to resolve the loss of interoperability which occurs during the evolution of resources. In future, we would like to run more tests in this regard by replicating the SIEMA model in other domains.
Bibliography


[4] Sören Auer, Sebastian Dietzold, Jens Lehmann, Sebastian Hellmann, and David Aumüller. Triplify: light-weight linked data publication from relational databases. In Juan Quemada, Gonzalo León, Yoëlle S. Maarek, and Wolfgang Nejdl, editors, Proceedings of the 18th Interna-


[10] Dejing Dou, Paea LePendu, Shiwoong Kim, and Peishen Qi. Integrating databases into the semantic web through an ontology-based


http://www.w3.org/TR/2007/REC-grddl-20070911/


[107] Nikolaos Konstantinou, Dimitrios-Emmanuel Spanos, Michael Chalas, Emmanuel Solidakis, and Nikolas Mitrou. VisAVis: An Approach to an Intermediate Layer between Ontologies and Relational Database Con-


[140] Patricia L. Whetzel, Natalya Fridman Noy, Nigam H. Shah, Paul R. Alexander, Csongor Nyulas, Tania Tudorache, and Mark A. Musen. BioPortal: enhanced functionality via new Web services from the National Center for Biomedical Ontology to access and use ontolo-


[179] Jon Haël Brenas, Mohammad Sadnan Al Manir, Kate Zinszer, Christopher J.O. Baker, and Arash Shaban-Nejad. Exploring Semantic Data Federation to Enable Malaria Surveillance Queries. In *Build-


[189] Diego Calvanese, Benjamin Cogrel, Elem Guzel Kalayci, Sarah Komla Ebri, Roman Kontchakov, Davide Lanti, Martin Rezk, Mariano


Appendix A

Sources for HAI SADI Services

A.1 Translated Domain Ontology

The domain ontology translated into TPTP-FOF is hosted at https://bitbucket.org/sadnanalmanir/valetsadi/src/master/knowledgebase/domain-ontology/tptpfof/. The complete translation is listed here:

% Common axioms:

fof('owl:Thing U dataDomain cover all model elements',axiom,( ! [X] :
        ( pThing(X)
         | dataDomain(X) ) ))).

fof('There is at least one individual',axiom,( ? [X] : pThing(X) )).
fof('There is at least one data value', axiom,(
    ? [X] : dataDomain(X) )).

fof('owl: Thing and dataDomain are disjoint', axiom,(
    ! [X] : ~ ( p_Thing(X) & dataDomain(X) ) )).

fof('owl: Nothing is empty', axiom,(
    ! [X] : ~ p_Nothing(X) )).

fof('integer literals are distinct from string−without−language literals', axiom,(
    ! [X,Y] : intLit(X) != stringLitNoLang(Y) )).

fof('integer literals are distinct from string−with−language literals', axiom,(
    ! [X,Y] : intLit(X) != stringLitWithLang(Y) )).

fof('integer literals are distinct from general typed literals', axiom,(
    ! [X,Y] : intLit(X) != typedLit(Y) )).
fof('string-without-language literals are distinct from string-with-language literals ', axiom, 
  ! [X,Y] : stringLitNoLang(X) != stringLitWithLang(Y) ).

fof('string-without-language literals are distinct from general typed literals ', axiom, 
  ! [X,Y] : stringLitNoLang(X) != typedLit(Y) ).

fof('string-with-language literals are distinct from general typed literals ', axiom, 
  ! [X] : stringLitWithLang(X) != typedLit(Y) ).

fof('intLit is a constructor ', axiom, 
  ! [X,Y] :
    ( intLit(X) = intLit(Y) => X = Y ) ).

fof('stringLitNoLang is a constructor ', axiom, 
  ! [X,Y] :
    ( stringLitNoLang(X) = stringLitNoLang(Y)
      => X = Y ) ).

fof('stringLitWithLang is a constructor ', axiom, 
  ! [X,Y] :
(stringLitWithLang(X) = stringLitWithLang(Y) => X = Y).

fof('typedLit is a constructor', axiom, (
    ! [X,Y] : (typedLit(X) = typedLit(Y) => X = Y))).

% End of common axioms.

fof('Subject of http://localhost:8080/healthcare/domain-ontologies/terminology.owl#has_diagnosis must be an individual', axiom, (
    ! [X,Y] : (p_has_diagnosis(X,Y) => p_Thing(X)))).

fof('Object of http://localhost:8080/healthcare/domain-ontologies/terminology.owl#has_diagnosis must be an individual', axiom, (
    ! [X,Y] : (p_has_diagnosis(X,Y) => p_Thing(Y)))).

fof('http://localhost:8080/healthcare/domain-ontologies/terminology.owl#Facility is a subclass of owl:Thing', axiom, (
    ! [X] : (p_Facility(X)
f( 'http://localhost:8080/healthcare/domain-ontologies/terminology.owl#' Procedure is a subclass of owl:Thing', axiom,
  ! [X] :  
    ( p.Procedure(X) 
    => p.Thing(X) ) )).

f( 'Subject of http://localhost:8080/healthcare/domain-ontologies/
  terminology.owl#has_last_name must be an individual', axiom,
  ! [X,Y] :  
    ( p.has_last_name(X,Y) 
    => p.Thing(X) ) )).

f( 'Object of http://localhost:8080/healthcare/domain-ontologies/
  terminology.owl#has_last_name must be a data value', axiom,
  ! [X,Y] :  
    ( p.has_last_name(X,Y) 
    => dataDomain(Y) ) )).

f( 'Subject of http://localhost:8080/healthcare/domain-ontologies/
  terminology.owl#has_facility_reference must be an individual', axiom,
  ! [X,Y] :  
    ( p.has_facility_reference(X,Y) 
    => p.Thing(X) ) )).
fof('Object of http://localhost:8080/healthcare/domain-ontologies/
terminology.owl#has_facility_reference must be an individual',axiom,(  ! [X,Y] :    ( p_has_facility_reference(X,Y)  => p_Thing(Y) ) )).

fof('Subject of http://localhost:8080/healthcare/domain-ontologies/
terminology.owl#has_adverse_reaction_to must be an individual',axiom,(  ! [X,Y] :    ( p_has_adverse_reaction_to(X,Y)  => p_Thing(X) ) )).

fof('Object of http://localhost:8080/healthcare/domain-ontologies/
terminology.owl#has_adverse_reaction_to must be a data value',axiom,(  ! [X,Y] :    ( p_has_adverse_reaction_to(X,Y)  => dataDomain(Y) ) )).

fof('Subject of http://localhost:8080/healthcare/domain-ontologies/
terminology.owl#has_facility_description must be an individual',axiom,(  ! [X,Y] :    ( p_has_facility_description(X,Y)  => p_Thing(X) ) )).
fof('Object of http://localhost:8080/healthcare/domain-ontologies/terminology.owl#has_facility_description must be a data value’, axiom, ( ! [X,Y] : ( p_has_facility_description(X,Y) => dataDomain(Y) ) )).

fof('Subject of http://localhost:8080/healthcare/domain-ontologies/terminology.owl#diagnosis_encountered must be an individual’, axiom, ( ! [X,Y] : ( p_diagnosis_encountered(X,Y) => p_Thing(X) ) )).

fof('Object of http://localhost:8080/healthcare/domain-ontologies/terminology.owl#diagnosis_encountered must be an individual’, axiom, ( ! [X,Y] : ( p_diagnosis_encountered(X,Y) => p_Thing(Y) ) )).

fof('http://localhost:8080/healthcare/domain-ontologies/terminology.owl#Service is a subclass of owl:Thing’, axiom, ( ! [X] : ( p_Service(X) => p_Thing(X) ) )).

fof('http://localhost:8080/healthcare/domain-ontologies/terminology.owl#Pharmacy is a subclass of owl:Thing’, axiom, ( ! [X] :...
( p:\text{Pharmacy}(X)
\Rightarrow p:\text{Thing}(X) )
).

\text{fof}(\text{Subject of http://localhost:8080/healthcare/domain-ontologies/}
\终结ology.owl#\text{has abstract record must be an individual'},\text{axiom},(
! [X,Y] :
 ( p:\text{has abstract record}(X,Y)
\Rightarrow p:\text{Thing}(X) )
).

\text{fof}(\text{Object of http://localhost:8080/healthcare/domain-ontologies/}
\终结ology.owl#\text{has abstract record must be an individual'},\text{axiom},(
! [X,Y] :
 ( p:\text{has abstract record}(X,Y)
\Rightarrow p:\text{Thing}(Y) )
).

\text{fof}(\text{Subject of http://localhost:8080/healthcare/domain-ontologies/}
\终结ology.owl#\text{is allergic_to must be an individual'},\text{axiom},(
! [X,Y] :
 ( p:\text{is allergic_to}(X,Y)
\Rightarrow p:\text{Thing}(X) )
).

\text{fof}(\text{Object of http://localhost:8080/healthcare/domain-ontologies/}
\终结ology.owl#\text{is allergic_to must be an individual'},\text{axiom},(
! [X,Y] :
 ( p:\text{is allergic_to}(X,Y)
\Rightarrow p:\text{Thing}(Y) )
).

230
fof('http://localhost:8080/healthcare/domain-ontologies/terminology.owl#
   Abstract is a subclass of owl:Thing',axiom,
   ! [X] :
   ( p_Abstract(X)
     => p_Thing(X) )).

fof('http://localhost:8080/healthcare/domain-ontologies/terminology.owl#
   Person is a subclass of owl:Thing',axiom,
   ! [X] :
   ( p_Person(X)
     => p_Thing(X) )).

fof('Subject of http://localhost:8080/healthcare/domain-ontologies/
   terminology.owl#encountersPatient must be an individual',axiom,
   ! [X,Y] :
   ( p_encountersPatient(X,Y)
     => p_Thing(X) )).

fof('Object of http://localhost:8080/healthcare/domain-ontologies/
   terminology.owl#encountersPatient must be an individual',axiom,
   ! [X,Y] :
   ( p_encountersPatient(X,Y)
     => p_Thing(Y) ).
fof('Subject of http://localhost:8080/healthcare/domain-ontologies/terminology.owl#has_first_name must be an individual', axiom,(
  ! [X,Y] :
  ( p_has_first_name(X,Y)
  => p_Thing(X) ) ))).

fof('Object of http://localhost:8080/healthcare/domain-ontologies/terminology.owl#has_first_name must be a data value', axiom,(
  ! [X,Y] :
  ( p_has_first_name(X,Y)
  => dataDomain(Y) ) )).

fof('Subject of http://www.w3.org/1999/02/22-rdf-syntax-ns#type must be an individual', axiom,(
  ! [X,Y] :
  ( p_type(X,Y)
  => p_Thing(X) ) )).

fof('Object of http://www.w3.org/1999/02/22-rdf-syntax-ns#type must be an individual', axiom,(
  ! [X,Y] :
  ( p_type(X,Y)
  => p_Thing(Y) ) )).

fof('http://localhost:8080/healthcare/domain-ontologies/terminology.owl#Patient is a subclass of owl:Thing', axiom,(
  ! [X] :
( p_Patient(X)
   => p_Thing(X) )).

fof('Subject of http://localhost:8080/healthcare/domain-ontologies/
   terminology.owl#has_diagnosis_code must be an individual',axiom,(  
! [X,Y] :  
 ( p_has_diagnosis_code(X,Y)
 => p_Thing(X) )).

fof('Object of http://localhost:8080/healthcare/domain-ontologies/
   terminology.owl#has_diagnosis_code must be a data value',axiom,(  
! [X,Y] :  
 ( p_has_diagnosis_code(X,Y)
 => dataDomain(Y) )).

fof('Subject of http://localhost:8080/healthcare/domain-ontologies/
   terminology.owl#description_of_order must be an individual',axiom,(  
! [X,Y] :   
 ( p_description_of_order(X,Y)
 => p_Thing(X) )).

fof('Object of http://localhost:8080/healthcare/domain-ontologies/
   terminology.owl#description_of_order must be a data value',axiom,(  
! [X,Y] :   
 ( p_description_of_order(X,Y)
 => dataDomain(Y) )).

fof('Subject of http://localhost:8080/healthcare/domain-ontologies/
   terminology.owl#abstractRecordForPatient must be an individual',axiom,

233
fom('Object of http://localhost:8080/healthcare/domain-ontologies/
terminology.owl#abstractRecordForPatient must be an individual', axiom,
! [X,Y] :
( p_abstractRecordForPatient(X,Y)

⇒ p Thing(X) ) )).

fom('Subject of http://localhost:8080/healthcare/domain-ontologies/
terminology.owl#has_allergy_to_patient_reference must be an individual
', axiom,
! [X,Y] :
( p_has_allergy_to_patient_reference(X,Y)

⇒ p Thing(X) ) )).

fom('Object of http://localhost:8080/healthcare/domain-ontologies/
terminology.owl#has_allergy_to_patient_reference must be an individual
', axiom,
! [X,Y] :
( p_has_allergy_to_patient_reference(X,Y)

⇒ p Thing(Y) ) )).
fof('http://localhost:8080/healthcare/domain-ontologies/terminology.owl#
    Allergy is a subclass of owl:Thing',axiom,(  
    ! [X] :  
        ( p.Allergy(X)  
        => p.Thing(X) ) )).

fof('http://localhost:8080/healthcare/domain-ontologies/terminology.owl#
    Diagnosis is a subclass of owl:Thing',axiom,(  
    ! [X] :  
        ( p.Diagnosis(X)  
        => p.Thing(X) ) )).

fof('http://localhost:8080/healthcare/domain-ontologies/terminology.owl#
    Encounter is a subclass of owl:Thing',axiom,(  
    ! [X] :  
        ( p.Encounter(X)  
        => p.Thing(X) ) )).

fof('Subject of http://localhost:8080/healthcare/domain-ontologies/
    terminology.owl#procedure_reference must be an individual',axiom,(  
    ! [X,Y] :  
        ( p.procedure_reference(X,Y)  
        => p.Thing(X) ) )).

fof('Object of http://localhost:8080/healthcare/domain-ontologies/
    terminology.owl#procedure_reference must be an individual',axiom,(  
    ! [X,Y] :  
        ( p.procedure_reference(X,Y)  
        => p.Thing(Y) ) )).
fof('Subject of http://localhost:8080/healthcare/domain-ontologies/
terminology.owl#service_for_encounter must be an individual',axiom,(  ! [X,Y] :  ( p.service_for_encounter(X,Y)  => p.Thing(X) ) )).

fof('Object of http://localhost:8080/healthcare/domain-ontologies/
terminology.owl#service_for_encounter must be an individual',axiom,(  ! [X,Y] :  ( p.service_for_encounter(X,Y)  => p.Thing(Y) ) )).

fof('Subject of http://localhost:8080/healthcare/domain-ontologies/
terminology.owl#prescription_service must be an individual',axiom,(  ! [X,Y] :  ( p.prescription_service(X,Y)  => p.Thing(X) ) )).

fof('Object of http://localhost:8080/healthcare/domain-ontologies/
terminology.owl#prescription_service must be an individual',axiom,(  ! [X,Y] :  ( p.prescription_service(X,Y)  => p.Thing(Y) ) )).

fof('Subject of http://localhost:8080/healthcare/domain-ontologies/
terminology.owl#counsel_at must be an individual',axiom,(  ! [X,Y] :  ( p.counsel_at(X,Y)  => p.Thing(X) ) )).
A.2 Mapping Rules for Nallergy Table

The mapping rules expressed in PSOA RuleML is hosted at https://bitbucket.org/sadnanalmanir/valetsadi/src/master/knowledgebase/semanticmapping/psoaruleml/ The rules to map the Nallergy table are listed here:

```plaintext
1 RuleML {
55  Assert (
56   Forall ?algWID (identityForAllergyToalgWID(identityForAllergy(?algWID)) = ?algWID)
57   Forall ?P (identityForAllergy(identityForAllergyToalgWID(?P)) = ?P)
58   Forall ?algWID ?algPatWID ?algAdverseReaction {
59     Allergy(identityForAllergy(?algWID)) :-
60     db_Nallergy(?algWID ?algPatWID ?algAdverseReaction)
61     Forall ?algWID ?algPatWID ?algAdverseReaction (is_allergic_to(identityForPatient(?algPatWID) identityForAllergy(?algWID)) :-
62     db_Nallergy(?algWID ?algPatWID ?algAdverseReaction))
63     Forall ?algWID ?algPatWID ?algAdverseReaction {
64       has_adverse_reaction_to(identityForAllergy(?algWID) ?algAdverseReaction) :-
65       db_Nallergy(?algWID ?algPatWID ?algAdverseReaction))
66   } )
67 }
68 )
```
A.3 Translated Mapping Rules

The translated PSOA RuleML mapping rules expressed as TPTP-FOF are hosted at https://bitbucket.org/sadnanalmanir/valentsadi/src/master/knowledgebase/semanticmapping/tptpfof/. These translated rules are listed here:

```
% TABLE: Npatient
% TABLE PREDICATE: dbNpatient
% PRIMARY-KEY-TO-ENTITY FUNCTION: identityForPatient
% INVERSE FOR PRIMARY-KEY-TO-ENTITY FUNCTION: identityForPatientToPatWID

% Setting up primary key PatWID
% identityForPatientToPatWID is an inverse function for identityForPatient.

fof(inverse_for_entityForPatient_1, axiom,
    ! [PatWID] : (identityForPatientToPatWID(identityForPatient(PatWID)) = PatWID).
).

fof(inverse_for_entityForPatient_2, axiom,
    ! [P] : (identityForPatient(identityForPatientToPatWID(P)) = P).
).

% Mapping the attribute Npatient.patWID to entities of the concept HAI: Patient.

fof(table_Npatient_represents_instances_of_concept_Person, axiom,
    ! [PatWID, PatFacWID, PatLastName, PatFirstName] : 
    (db_Npatient(PatWID, PatFacWID, PatLastName, PatFirstName) =>
```
\[ p \_Patient(\text{identityForPatient}(\text{PatWID})) \]

\[ \text{fof(}\text{attribute_PatLastName_represents_last_name_of_a_Patient}, \text{axiom}, \]

\[ \text{! [PatWID, PatFacWID, PatLastName, PatFirstName]} : \]

\[ (\text{db_Npatient}(\text{PatWID}, \text{PatFacWID}, \text{PatLastName}, \text{PatFirstName}) \]

\[ => \]

\[ p \_\text{has_last_name}(\text{identityForPatient}(\text{PatWID}), \text{PatLastName}) \]

\[ ) \].

\[ \text{fof(}\text{attribute_PatFirstName_represents_first_name_of_a_Patient}, \text{axiom}, \]

\[ \text{! [PatWID, PatFacWID, PatLastName, PatFirstName]} : \]

\[ (\text{db_Npatient}(\text{PatWID}, \text{PatFacWID}, \text{PatLastName}, \text{PatFirstName}) \]

\[ => \]

\[ p \_\text{has_first_name}(\text{identityForPatient}(\text{PatWID}), \text{PatFirstName}) \]

\[ ) \].

\[ \text{fof(}\text{attribute_PatFacWID_assigned_fk_to_table_Nfacility}, \text{axiom}, \]

\[ \text{! [PatWID, PatFacWID, PatLastName, PatFirstName]} : \]

\[ (\text{db_Npatient}(\text{PatWID}, \text{PatFacWID}, \text{PatLastName}, \text{PatFirstName}) \]

\[ => \]

\[ p \_\text{has_facility_reference}(\text{identityForPatient}(\text{PatWID}), \text{identityForFacility}(\text{PatFacWID})) \]

\[ ) \].

%%%%%%%%%%%%%%%%%%%%% % TABLE: Nfacility % TABLE_PREDICATE: db_Nfacility % PRIMARY-KEY-TO-ENTITY FUNCTION: identityForFacility % INVERSE FOR PRIMARY-KEY-TO-ENTITY FUNCTION: identityForFacilityToFacWID %%%%%%%%%%%%%%%%%%%%%% % Setting up primary key FacWID
% identityForFacilityToFacWID is an inverse function for identityForFacility.

fof(inverse_for_entityForFacility_1, axiom, 
    ! [FacWID] : (identityForFacilityToFacWID(identityForFacility(FacWID)) = FacWID).
).

fof(inverse_for_entityForFacility_2, axiom, 
    ! [P] : (identityForFacility(identityForFacilityToFacWID(P)) = P).
).

% Mapping the attribute Nfacility.FacWID to entities of the concept HAI: Facility.

fof(table_Nfacility_represents_instances_of_concept_Facility, axiom, 
    ! [FacWID, FacFacilityDesc] : 
        (db_Nfacility(FacWID, FacFacilityDesc) => p_Facility(identityForFacility(FacWID))).
).

% Mapping the attribute Nfacility.FacFacilityDesc to the property HAI: has_facility_description.

fof(attribute_PatLastName_assigns_has_last_name, axiom, 
    ! [FacWID, FacFacilityDesc] : 
        (db_Nfacility(FacWID, FacFacilityDesc) => p_has_facility_description(identityForFacility(FacWID), FacFacilityDesc)).
).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% % TABLE: NhrAbstract
% % TABLE PREDICATE: db_NhrAbstract

240
%% PRIMARY-KEY-TO-ENTITY FUNCTION: identityForAbstract
%% INVERSE FOR PRIMARY-KEY-TO-ENTITY FUNCTION: identityForAbstractTohraWID

% Setting up primary key hraWID
% identityForAbstractTohraWID is an inverse function for identityForAbstract.

fof(inverse_for_entityForAbstract_1, axiom,
    ! [HraWID] : (identityForAbstractTohraWID(identityForAbstract(HraWID)) = HraWID).
).

fof(inverse_for_entityForAbstract_2, axiom,
    ! [P] : (identityForAbstract(identityForAbstractTohraWID(P)) = P).
).

% Mapping the attribute NhrAbstract.hraWID to entities of the concept HAI: Abstract.

fof(table_NhrAbstract_represents_instances_of_concept_Abstract, axiom,
    ! [HraWID, HraEncWID, HraPatWID] :
    (db_NhrAbstract(HraWID, HraEncWID, HraPatWID) => p_Abstract(identityForAbstract(HraWID))).
).

% Mapping the attribute NhrAbstract.hraWID to entities of the concept HAI: Abstract.

% TABLE: NhrDiagnosis
% TABLE PREDICATE: db_NhrDiagnosis
% PRIMARY-KEY-TO-ENTITY FUNCTION: identityForDiagnosis
% INVERSE FOR PRIMARY-KEY-TO-ENTITY FUNCTION: identityForDiagnosisTohdgWID

241
% Setting up primary key hdgWID
% identityForDiagnosisTohdgWID is an inverse function for
termsForDiagnosis.

fof(inverse_for_entityForDiagnosis_1, axiom, 
   ! [hdgWID] : (identityForDiagnosisTohdgWID(diagnosisEntity(hdgWID)) = hdgWID).
).

fof(inverse_for_entityForDiagnosis_2, axiom, 
   ! [P] : (identityForDiagnosis(identityForDiagnosisTohdgWID(P)) = P).
).

% Mapping the attribute NhrDiagnosis.hdgWID to entities of the concept HAI: Diagnosis.

fof(table_NhrDiagnosis_represents_instances_of_concept_Diagnosis, axiom, 
   ! [hdgWID, hdgHraWID, hdgHraEncWID, hdgCd] : 
   (db_NhrDiagnosis(hdgWID, hdgHraWID, hdgHraEncWID, hdgCd) => 
    p_Diagnosis(identityForDiagnosis(hdgWID))). 
).

% Mapping the attribute NhrDiagnosis.hdgCd to the property HAI:
% has_diagnosis_code.

fof(table_NhrDiagnosis_represents_has_diagnosis_code, axiom, 
   ! [hdgWID, hdgHraWID, hdgHraEncWID, hdgCd] : 
   (db_NhrDiagnosis(hdgWID, hdgHraWID, hdgHraEncWID, hdgCd) => 
    p_has_diagnosis_code(identityForDiagnosis(hdgWID), hdgCd)). 
).
% Mapping the attribute NhrDiagnosis.hdgHraEncWID to the property HAI:
  diagnosisEncountered.

fof(table_NhrDiagnosis_represents_diagnosis_encountered,axiom,
    ! [HdgWID, HdgHraWID, HdgHraEncWID, HdgCd] : 
      (db_NhrDiagnosis(HdgWID, HdgHraWID, HdgHraEncWID, HdgCd) 
        => 
          p_diagnosis_encountered(identityForDiagnosis(HdgWID), identityForEncounter(HdgHraEncWID)))).

% Mapping the attribute NhrDiagnosis.hdgHraWID to the property HAI:
  hasAbstractRecord.

fof(table_NhrAbstract_represents_instances_of_concept_Abstract,axiom,
    ! [HraWID, HraEncWID, HraPatWID] : 
      (db_NhrAbstract(HraWID, HraEncWID, HraPatWID) 
        => 
          p_abstractRecordForPatient(identityForAbstract(HraWID), identityForPatient(HraPatWID))) 
          p_abstractRecordForPatient(identityForPatient(HraPatWID), identityForAbstract(HraWID)))).

fof(table_NhrDiagnosis_represents_has_diagnosis_code,axiom,
    ! [HdgWID, HdgHraWID, HdgHraEncWID, HdgCd] : 
      (db_NhrDiagnosis(HdgWID, HdgHraWID, HdgHraEncWID, HdgCd) 
        => 
          p_diagnosis_encountered(identityForDiagnosis(HdgWID), identityForAbstract(HdgHraWID)))).

fof(table_NhrAbstract_represents_instances_of_concept_Abstract,axiom,
% both directions work
% p_has_abstract_record(identityForPatient(HraPatWID), identityForAbstract(HraWID))
% p_has_abstract_record(identityForAbstract(HraWID), identityForPatient(HraPatWID))
).

fof(arbitrary_rule,axiom,

  ! [PatWID, HraWID, HdgWID] :
    ( db_NhrAbstract(HraWID, HraEncWID, HraPatWID)
    =>
      % both directions work
      % p_has_abstract_record(identityForPatient(HraPatWID), identityForAbstract(HraWID))
      p_has_abstract_record(identityForAbstract(HraWID), identityForPatient(HraPatWID))
   ).

% Setting up primary key AlgWID
% identityForAllergyToAlgWID is an inverse function for identityForAllergy.

% TABLE: Nallergy
% TABLE PREDICATE: db_Nallergy
% PRIMARY-KEY-TO-ENTITY FUNCTION: identityForAllergy
% INVERSE FOR PRIMARY-KEY-TO-ENTITY FUNCTION: identityForAllergyToAlgWID

% Setting up primary key AlgWID
% identityForAllergyToAlgWID is an inverse function for identityForAllergy.
fof(inverse_for_entityForAllergy_1, axiom, 
   ! [AlgWID] : (identityForAllergyToAlgWID(identityForAllergy(AlgWID)) = AlgWID).
).

fof(inverse_for_entityForAllergy_2, axiom, 
   ! [P] : (identityForAllergy(identityForAllergyToAlgWID(P)) = P).
).

% Mapping the attribute Nallergy.AlgWID to entities of the concept HAI:
% Allergy.
fof(table_Nallergy_represents_instances_of_concept_Allergy, axiom, 
   ! [AlgWID, AlgPatWID, AlgAdverseReaction] : 
   (db_Nallergy(AlgWID, AlgPatWID, AlgAdverseReaction) => 
   p_Allergy(identityForAllergy(AlgWID))).
).

% Mapping the attribute Nallergy.AlgAllergyDesc to the property HAI:
% has_facility_description.
fof(attribute_AlgPatWID_used_as_foreign_key_to_patient, axiom, 
   ! [AlgWID, AlgPatWID, AlgAdverseReaction] : 
   (db_Nallergy(AlgWID, AlgPatWID, AlgAdverseReaction) => 
   p_has_allergy(identityForPatient(AlgPatWID), identityForAllergy(AlgWID))).
).

fof(attribute_AlgAdverseReaction_represents_data_value_of_allergy_name, axiom, 
   ! [AlgWID, AlgPatWID, AlgAdverseReaction] : 
   (db_Nallergy(AlgWID, AlgPatWID, AlgAdverseReaction) => 
   p_has_adverse_reaction_to(identityForAllergy(AlgWID), AlgAdverseReaction)).

245
% Setting up primary key hprcWID
% identityForProcedureTohprcWID is an inverse function for
    identityForProcedure.

fof(inverse_for_entityForProcedure_1, axiom, 
    ! [HprcWID] : (identityForProcedureTohprcWID(identityForProcedure(HprcWID)) = HprcWID)
).

fof(inverse_for_entityForProcedure_2, axiom, 
    ! [P] : (identityForProcedure(identityForProcedureTohprcWID(P)) = P)
).

% Mapping the attribute NhrProcedure.hprcWID to entities of the concept HAI: Procedure.

fof(table_NhrProcedure_represents_instances_of_concept_Procedure , axiom, 
    ! [HprcWID, HprcHraWID, HprcHraEncWID, HprcStartTime] : 
    (db_NhrProcedure(HprcWID, HprcHraWID, HprcHraEncWID, HprcStartTime) 
    => 
    p_Procedure(identityForProcedure(HprcWID)))
).

% Mapping the attribute NhrProcedure.hprcStartTime to the property HAI: has_start_time.

246
fof(table_NhrProcedure_represents_has_start_time, axiom, ! [HprcWID, HprcHraWID, HprcHraEncWID, HprcStartTime] : 
 (db_NhrProcedure(HprcWID, HprcHraWID, HprcHraEncWID, HprcStartTime) =>
  p_has_start_time(identityForProcedure(HprcWID), HprcStartTime)) ).

fof(table_NhrProcedure_represents_procedure_encounterd, axiom, ! [HprcWID, HprcHraWID, HprcHraEncWID, HprcStartTime] : 
 (db_NhrProcedure(HprcWID, HprcHraWID, HprcHraEncWID, HprcStartTime) =>
  p_procedure_reference(identityForProcedure(HprcWID), identityForAbstract(HprcHraWID))) ).

fof(arbitrary_rule_for_procedure_for_patient, axiom, ! [PatWID, HprcWID, HraWID] :
 ( 
  (p_has_abstract_record(identityForAbstract(HraWID), identityForPatient(PatWID)) &
   p_procedure_reference(identityForProcedure(HprcWID), identityForAbstract(HraWID))
  )
 ) =>
  p_undergo_procedure(identityForPatient(PatWID), identityForProcedure(HprcWID))) ).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% TABLE: Nencounter
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
fof(inverse_for_entityForEncounter_1, axiom, 
  ! [EncWID] : (identityForEncounterToEncWID( identityForEncounter(EncWID) ) = EncWID).
).

fof(inverse_for_entityForEncounter_2, axiom, 
  ! [P] : (identityForEncounter( identityForEncounterToEncWID(P) ) = P).
).

fof(table_Nencounter_represents_instances_of_concept_Encounter, axiom, 
  ! [EncWID, EncPatWID] : 
    (db_Nencounter(EncWID, EncPatWID) => p_Encounter( identityForEncounter(EncWID) )).
).

fof(attribute_EncPatWID_assigned_fk_to_table_Npatient, axiom, 
  ! [EncWID, EncPatWID] : 
    (db_Nencounter(EncWID, EncPatWID) => p_encountersPatient( identityForEncounter(EncWID), identityForPatient(EncPatWID) )).
).

fof(inverse_for_entityForService_1, axiom, 
  ! [SvcWID] : (identityForServiceToSvcWID( identityForService(SvcWID) ) = SvcWID).
).

fof(inverse_for_entityForService_2, axiom, 

248
! [P] : (identityForService(identityForServiceToSvcWID(P)) = P)
).

fof(table_Nservice_represents_instances_of_concept_Service, axiom,
  ! [SvcWID, SvcEncWID, SvcTableCd, SvcOrderedBy, SvcPerformedBy] :
  (db_Nservice(SvcWID, SvcEncWID, SvcTableCd, SvcOrderedBy, SvcPerformedBy)
  =>
  p_Service(identityForService(SvcWID)))
).

fof(attribute_SvcEncWID_assigned_fk_to_table_Nencounter, axiom,
  ! [SvcWID, SvcEncWID, SvcTableCd, SvcOrderedBy, SvcPerformedBy] :
  (db_Nservice(SvcWID, SvcEncWID, SvcTableCd, SvcOrderedBy, SvcPerformedBy)
  =>
  p_service_for_encounter(identityForService(SvcWID), identityForEncounter(SvcEncWID)))
).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% TABLE: NphmIngredient
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

fof(inverse_for_entityForPharmacy_1, axiom,
  ! [PhmWID] : (identityForPharmacyToPhmWID(identityForPharmacy(PhmWID)) = PhmWID)
).

fof(inverse_for_entityForPharmacy_2, axiom,
  ! [P] : (identityForPharmacy(identityForPharmacyToPhmWID(P)) = P)
).

fof(table_NphmIngredient_represents_instances_of_concept_Pharmacy, axiom,
  ! [PhmWID, PhmSvcWID, PhmOrderDesc, PhmStartDtm, PhmEndDtm, PhmStrengthDesc, PhmDosageForm] :
  ...
(db_NphmIngredient(PhmWID, PhmSvcWID, PhmOrderDesc, PhmStartDtm, PhmEndDtm,
                 PhmStrengthDesc, PhmDosageForm))
=>
p_Pharmacy(identityForPharmacy(PhmWID)))
).

fof(attribute_NphmIngredient_assigned_fk_to_table_Nservice,axiom,
    ! [PhmWID, PhmSvcWID, PhmOrderDesc, PhmStartDtm, PhmEndDtm,
           PhmStrengthDesc, PhmDosageForm] :
    (db_NphmIngredient(PhmWID, PhmSvcWID, PhmOrderDesc, PhmStartDtm, PhmEndDtm,
                      PhmStrengthDesc, PhmDosageForm)
=>
p_prescription_service(identityForPharmacy(PhmWID), identityForService(
                      PhmSvcWID))))
).

fof(arbitrary_rule_for_pharmacy_for_patient,axiom,
    ! [PatWID, EncWID, SvcWID, PhmWID] :
      (p_encountersPatient(identityForEncounter(EncWID), identityForPatient(PatWID))
&
      p_service_for_encounter(identityForService(SvcWID), identityForEncounter(
                      EncWID))
&
      p_prescription_service(identityForPharmacy(PhmWID), identityForService(
                      SvcWID))
)
=>
p_counsel_at(identityForPatient(PatWID), identityForPharmacy(PhmWID))
))).

fof(attribute_PhmsStartDtm_assigned_to_has_start_time,axiom,
A.4 Generating SQL-template Query

A.4.1 S1: allPatientId Service

The proof tree generated during the resolutions in VampirePrime reasoner is an important step towards finding the schematic answers. For each service, the proof tree is listed below for documentation purposes only and is not saved in Valet SADI. However, the schematic answers to a semantic query is hosted at [https://bitbucket.org/sadnanalmanir/valetsadi/src/master/schematicanswers/](https://bitbucket.org/sadnanalmanir/valetsadi/src/master/schematicanswers/) while the SQL-template queries are hosted at [https://bitbucket.org/sadnanalmanir/valetsadi/src/master/sql/template/](https://bitbucket.org/sadnanalmanir/valetsadi/src/master/sql/template/)

For each service, the schematic answers and the corresponding SQL-template

251
queries are listed below.

A.4.1.1 Resolution Proof Tree

1  %  114.  \neg \text{Patient}(\text{entityForPatient}(X0)) | <answer>(X0)
2  %  224. \text{Patient}(\text{entityForPatient}(X0)) | \{\neg \text{db}_{\text{Npatient}}\}(X0, X1, X2, X3)
3  %  227. \neg \text{Patient}(\text{entityForPatient}(X0)) | <answer>(X0)
4  %  337. \text{Patient}(\text{entityForPatient}(X0)) | \{\neg \text{db}_{\text{Npatient}}\}(X0, X1, X2, X3)
5  %  373. \{\neg \text{db}_{\text{Npatient}}\}(X0, X1, X2, X3) \lor <answer>(X0)

A.4.1.2 Generated Schematic Answers

<generic_answer>
  <condition>
    <literal polarity="neg">
      <atom predicate="\text{db}_{\text{Npatient}}">
        <arg>
          <var sym="X0"/>
        </arg>
        <arg>
          <var sym="X1"/>
        </arg>
        <arg>
          <var sym="X2"/>
        </arg>
        <arg>
          <var sym="X3"/>
        </arg>
      </atom>
    </literal>
  </condition>
  <conclusion>
    <literal polarity="pos">
    </literal>
  </conclusion>
</generic_answer>
A.4.1.3 SQL-template Query

Separation results:
Generating: \( \neg \text{db}_\text{Npatient}(X0,X1,X2,X3) \)
Testing view literals: []
Test literals: []

Flattening results:
\( Dx = \neg \text{db}_\text{Npatient}(Y0,Y1,Y2,Y3) \)
Generating = \( \neg \text{db}_\text{Npatient}(Y0,Y1,Y2,Y3) \)
Testing = []
Ea = \( \neg \text{equal}(Y0,X0) \)
Ec = []
Ed = []

Table aliases:
\( \neg \text{db}_\text{Npatient}(Y0,Y1,Y2,Y3) \rightarrow \text{TABLE0} \)

SELECT \text{TABLE0}.\text{patWID} AS X0
FROM \text{Npatient} AS \text{TABLE0}

<sail_generated_sql_query>
<query_text>
SELECT \text{TABLE0}.\text{patWID} AS X0
FROM \text{Npatient} AS \text{TABLE0}
</query_text>
</sql_views>
A.4.2  S2: getLastnameFirstnameByPatientId Service

A.4.2.1 Resolution Proof Tree

1. % 114. \( \neg \text{p.has_last_name(entityForPatient(”Z0”), }X0 \) \ | \( \neg \text{p.has_first_name(entityForPatient(”Z0”), }X1 \) \ / \( \langle \langle \text{answer}\rangle \rangle (X0,X1) \)

2. % 222. p.has_first_name(entityForPatient(X0),X1) | \{ {\neg \text{db.Npatient} } \}(X0,X2,X3,X1)

3. % 223. p.has_last_name(entityForPatient(X0),X1) | \{ {\neg \text{db.Npatient} } \}(X0,X2,X1,X3)

4. % * 227. \( \neg \text{p.has_last_name(entityForPatient(”Z0”), }X0 \) \ | \( \neg \text{p.has_first_name(entityForPatient(”Z0”), }X1 \) \ / \( \langle \langle \text{answer}\rangle \rangle (X0,X1) \)

5. % * 335. p.has_first_name(entityForPatient(X0),X1) | \{ {\neg \text{db.Npatient} } \}(X0,X2,X3,X1)

6. % * 336. p.has_last_name(entityForPatient(X0),X1) | \{ {\neg \text{db.Npatient} } \}(X0,X2,X1,X3)

7. % * 372. \( \neg \text{p.has_first_name(entityForPatient(”Z0”), }X0 \) \ | \{ {\neg \text{db.Npatient} } \}(”Z0”,X1,X2,X3)

A.4.2.2 Generated Schematic Answers

<generic_answer>
<condition>
<literal polarity="neg">
<atom predicate="db.Npatient">
<arg>
<const sym="&quot; Z0&quot;"/>
</arg>
<arg>
<var sym="X0"/>
</generic_answer>
\[
\text{\texttt{\textbackslash arg}}
\]
\[
\text{\texttt{\textbackslash arg}}
\]
\[
\text{\texttt{\textbackslash var \ sym="X1"/}}
\]
\[
\text{\texttt{\textbackslash arg}}
\]
\[
\text{\texttt{\textbackslash arg}}
\]
\[
\text{\texttt{\textbackslash var \ sym="X2"/}}
\]
\[
\text{\texttt{\textbackslash arg}}
\]
\[
\text{\texttt{\textbackslash atom}}
\]
\[
\text{\texttt{\textbackslash literal}}
\]
\[
\text{\texttt{\textbackslash literal \ polarity="neg"}}
\]
\[
\text{\texttt{\textbackslash atom \ predicate="db\_Npatient"}}
\]
\[
\text{\texttt{\textbackslash arg}}
\]
\[
\text{\texttt{\textbackslash arg}}
\]
\[
\text{\texttt{\textbackslash const \ sym="\textquoteright Z0\textquoteright\"/}}
\]
\[
\text{\texttt{\textbackslash arg}}
\]
\[
\text{\texttt{\textbackslash arg}}
\]
\[
\text{\texttt{\textbackslash arg}}
\]
\[
\text{\texttt{\textbackslash arg}}
\]
\[
\text{\texttt{\textbackslash arg}}
\]
\[
\text{\texttt{\textbackslash atom}}
\]
\[
\text{\texttt{\textbackslash literal}}
\]
\[
\text{\texttt{\textbackslash condition}}
\]
\[
\text{\texttt{\textbackslash conclusion}}
\]
\[
\text{\texttt{\textbackslash literal \ polarity="pos"}}
\]
\[
\text{\texttt{\textbackslash atom \ predicate="answer"}}
\]
\[
\text{\texttt{\textbackslash arg}}
\]
\[
\text{\texttt{\textbackslash arg}}
\]
\[
\text{\texttt{\textbackslash var \ sym="X1"/}}
\]
\[
\text{\texttt{\textbackslash arg}}
\]
\[
\text{\texttt{\textbackslash arg}}
\]
\[
\text{\texttt{\textbackslash var \ sym="X5"/}}
\]
\[
\text{\texttt{\textbackslash arg}}
\]
A.4.2.3 SQL-template Query

Separation results:
- Generating: \[ \sim \text{db}_{\text{Npatient}} ("Z0", X0, X1, X2), \sim \text{db}_{\text{Npatient}} ("Z0", X3, X4, X5) \]
- Testing view literals: []
- Test literals: []

Flattening results:
- Dx = \[ \sim \text{db}_{\text{Npatient}} (Y0, Y1, Y2, Y3), \sim \text{db}_{\text{Npatient}} (Y4, Y5, Y6, Y7) \]
- Generating = \[ \sim \text{db}_{\text{Npatient}} (Y0, Y1, Y2, Y3), \sim \text{db}_{\text{Npatient}} (Y4, Y5, Y6, Y7) \]
- Testing = []
- Ea = \[ \sim \text{equal} (Y2, X1), \sim \text{equal} (Y7, X5) \]
- Ec = \[ \sim \text{equal} (Y0, "Z0"), \sim \text{equal} (Y4, "Z0") \]
- Ed = []

Table aliases:
- \sim \text{db}_{\text{Npatient}} (Y0, Y1, Y2, Y3) \rightarrow \text{TABLE0}
- \sim \text{db}_{\text{Npatient}} (Y4, Y5, Y6, Y7) \rightarrow \text{TABLE1}

<sail_generated_sql_query>
<query_text>

SELECT \text{TABLE0}.\text{patLastName} AS X1,
       \text{TABLE1}.\text{patFirstName} AS X5
FROM \text{Npatient} AS \text{TABLE0},
     \text{Npatient} AS \text{TABLE1}
WHERE \text{TABLE0}.\text{patWID} = "Z0"
     AND \text{TABLE1}.\text{patWID} = "Z0"
</query_text>
A.4.3 S3: get AllergyByPatientId Service

A.4.3.1 Resolution Proof Tree

\[ % 114. \neg p \_ \text{has}_{\text{allergy}}(\text{entityForPatient}("Z0"), \text{entityForAllergy}(X0)) | \neg p \_ \text{has}_{\text{adverse}_{\text{reaction}}\_\text{to}}(\text{entityForAllergy}(X0), X1) ~/<<\text{answer>>}(X1) \]

\[ % 200. p \_ \text{has}_{\text{adverse}_{\text{reaction}}\_\text{to}}(\text{entityForAllergy}(X0), X1) | \{\{\neg db\_\text{Nallergy}\}\}(X0, X2, X1) \]

\[ % 201. p \_ \text{has}_{\text{allergy}}(\text{entityForPatient}(X0), \text{entityForAllergy}(X1)) | \{\{\neg db\_\text{Nallergy}\}\}(X1, X0, X2) \]

\[ % 227. \neg p \_ \text{has}_{\text{adverse}_{\text{reaction}}\_\text{to}}(\text{entityForAllergy}(X0), X1) ~/<<\text{answer>>}(X1) \]

\[ % * 313. p \_ \text{has}_{\text{adverse}_{\text{reaction}}\_\text{to}}(\text{entityForAllergy}(X0), X1) | \{\{\neg db\_\text{Nallergy}\}\}(X0, X2, X1) \]

\[ % * 314. p \_ \text{has}_{\text{allergy}}(\text{entityForPatient}(X0), \text{entityForAllergy}(X1)) | \{\{\neg db\_\text{Nallergy}\}\}(X1, X0, X2) \]

\[ % * 354. \neg p \_ \text{has}_{\text{adverse}_{\text{reaction}}\_\text{to}}(\text{entityForAllergy}(X0), X1) | \{\{\neg db\_\text{Nallergy}\}\}(X0, "Z0", X2) ~/<<\text{answer>>}(X1) \]

A.4.3.2 Generated Schematic Answers

<generic_answer>
  <condition>
    <literal polarity="neg">
      <atom predicate="db\_Nallergy"/>
    </literal>
    <arg>
      <var sym="X0"/>
    </arg>
  </condition>
</generic_answer>
A.4.3.3 SQL-template Query
Separation results:
Generating: [¬ db_Nallergy(X0,"Z0",X1), ¬ db_Nallergy(X0,X2,X3)]
Testing view literals: []
Test literals: []

Flattening results:
Dx = [¬ db_Nallergy(Y0,Y1,Y2), ¬ db_Nallergy(Y3,Y4,Y5)]
Generating = [¬ db_Nallergy(Y0,Y1,Y2), ¬ db_Nallergy(Y3,Y4,Y5)]
Testing = []
Ea = [¬ equal(Y5,X3)]
Ec = [¬ equal(Y1,"Z0")]
Ed = [¬ equal(Y0,Y3)]

Table aliases:
¬ db_Nallergy(Y0,Y1,Y2) --> TABLE0
¬ db_Nallergy(Y3,Y4,Y5) --> TABLE1

<select_generated_sql_query>
<query_text>
SELECT TABLE1.algAdverseReaction AS X3
FROM Nallergy AS TABLE0,
     Nallergy AS TABLE1
WHERE TABLE0.algPatWID = "Z0"
     AND TABLE0.algWID = TABLE1.algWID
</query_text>
</sql_views>
</var_map>
{queryVar0=X3}
</var_map>
</select_generated_sql_query>
A.5 Source Code of SADI Services

A.5.1 S1: allPatientId Service

```java
public class AllPatientIds extends SimpleSynchronousServiceServlet {

    public void processInput(Resource input, Resource output) {

        // Assign input values from the input RDF
        // No input is required

        // Instantiate SQL Query
        String sql_query =
            " SELECT TABLE0.patWID AS X0"
            +
            " FROM Npatient AS TABLE0";

        // Create output RDF
        // Initialize the empty output model
        Model outputRDF = output.getModel();
        try {
            // Connect to the MySQL source database
            java.sql.Statement stmt = MySqlDatabase.connection.createStatement();
            // Execute the SQL query
            ResultSet rs = stmt.executeQuery(sql_query);
            // Iterate through each item from the resultset
            while (rs.next()) {
                // Create a resource for the patient identifier
                Resource patient_id = outputModel
                    .createResource("http://.../Patient_by_ID?ID=" + rs.getString(1));
                // Set the type of the resource
                patient_id.addProperty(Vocab.type, Vocab.Patient);
                output.addProperty(Vocab.type, Vocab.Patient);
            }
        }
```
catch (Exception e) {
    ...
}

/* Vocabularies for Resources and types */
private static final class Vocab {
    private static Model m_model = ModelFactory.createDefaultModel();
    public static final Resource Patient = m_model .createResource("http://.../terminologies.owl#Patient");
    public static final Resource Input = m_model .createResource("http://.../allPatientIds.owl#Input");
    public static final Resource Output = m_model .createResource("http://.../allPatientIds.owl#Output");
    public static final Property type = m_model .createProperty("http://www.w3.org/1999/02/22-rdf-syntax-ns#type");
}

A.5.2 S2: get_LastnameFirstnameByPatientId Service

public class get_LastnameFirstnameByPatientId extends
    SimpleSynchronousServiceServlet {
    ...
    public void processInput(Resource input, Resource output) {
        /* Get input value from the input RDF */
        String Z0 = input.getURI().substring("http://.../Patient_by_ID?ID=").length();

        /* Instantiate SQL Query */
        String sql_query =
" SELECT TABLE0.patLastName AS X1,"+
" TABLE1.patFirstName AS X5"+
" FROM Npatient AS TABLE0"+
" Npatient AS TABLE1"+
" WHERE TABLE0.patWID = "+Z0+
" AND TABLE1.patWID = "+Z0;

/* Create output RDF */
Model outputRDF = output.getModel();
try {
    java.sql.Statement stmt = MySqlDatabase.connection.createStatement();
    ResultSet rs = stmt.executeQuery(sqlQuery);
    while(rs.next()) {
        output.addProperty(Vocab.type, Vocab.Patient);
        output.addLiteral(Vocab.hasLastName, rs.getString(1));
        output.addLiteral(Vocab.hasFirstName, rs.getString(2));
    }
} catch(Exception e){
    ...
}

/* Vocabularies for Resources and types */
private static final class Vocab {
    private static Model m_model = ModelFactory.createDefaultModel();
    public static final Resource Patient = m_model
        .createResource("http://.../terminologies.owl#Patient");
    public static final Resource Input = m_model
        .createResource("http://.../getLastNameFirstnameByPatientId.owl#Input");
    public static final Resource Output = m_model
        .createResource("http://.../getLastNameFirstnameByPatientId.owl#Output");
    public static final Property type = m_model
}
public static final Property has_last_name = m_model
.createProperty("http://.../terminologies.owl#has_last_name");
public static final Property has_first_name = m_model
.createProperty("http://.../terminologies.owl#has_first_name");
}

A.5.3 S3: getAllergyByPatientId Service

public class getAllergyByPatientId extends SimpleSynchronousServiceServlet
{
...
public void processInput(Resource input, Resource output)
{
    /* Get input value from the input RDF */
    String Z0 = input.getURI().substring("http://.../Patient by ID?ID=").length();

    /* Instantiate SQL Query */
    String sql_query =
    " SELECT TABLE2.algAdverseReaction AS X6"
    " FROM Npatient AS TABLE0,"
    " Nallergy AS TABLE1,"
    " Nallergy AS TABLE2,"
    " WHERE TABLE0.patWID = " + Z0 +
    " AND TABLE1.algWID = TABLE2.algWID";

    /* Create output RDF */
    Model outputRDF = output.getModel();
    try {
        java.sql.Statement stmt = MySqlDatabase.connection.createStatement();
        ResultSet rs = stmt.executeQuery(sql_query);
        while (rs.next()) {
            output.addProperty(Vocab.type, Vocab.Patient);
            Resource Allergy_instance = outputRDF.createResource();
            ...
Allergy_instance.addProperty(Vocab.type, Vocab.Allergy);
output.addProperty(Vocab.has_allergy, Allergy_instance);
Allergy_instance.addLiteral(Vocab.has_allergic_reaction_to, rs.getString(1));
}
}
catch(Exception e){
...
}
}

/* Vocabularies for Resources and types */
private static final class Vocab
{
    private static Model m_model = ModelFactory.createDefaultModel();
    public static final Resource Allergy = m_model
        .createResource("http://.../terminologies.owl#Allergy");
    public static final Resource Patient = m_model
        .createResource("http://.../terminologies.owl#Patient");
    public static final Resource Input = m_model
        .createResource("http://.../getAllergyByPatientId.owl#Input");
    public static final Resource Output = m_model
        .createResource("http://.../getAllergyByPatientId.owl#Output");
    public static final Property type = m_model
        .createProperty("http://www.w3.org/1999/02/rdf-syntax-ns#type");
    public static final Property has_allergy = m_model
        .createProperty("http://.../terminologies.owl#has_allergy");
    public static final Property has_allergic_reaction_to = m_model
        .createProperty("http://.../terminologies.owl#has_allergic_reaction_to");
}
A.6 Composing SPARQL Queries Incrementally on HYDRA

Figure A.1: Find all patients.

Figure A.2: List the first and last names of all patients.
Figure A.3: List first and last names of all patients who are admitted to the Orthopedic Surgery facility.
Figure A.4: List first and last names of all patients who are admitted to the Orthopedic Surgery facility and diagnosed with Osteoarthritis of knee (ICD-10-CM code M17).
Figure A.5: List first and last names of all patients who are admitted to the Orthopedic Surgery facility, diagnosed with Osteoarthritis of knee (ICD-10-CM code M17), and the start time of scheduled procedure.
Figure A.6: List first and last names of all patients who are admitted to the *Orthopedic Surgery* facility, diagnosed with Osteoarthritis of knee (ICD-10-CM code *M17*), the start time of scheduled procedure, description of the order of the antibiotic, and the start time for administering the medication.
Figure A.7: List first and last names of all patients who are admitted to the Orthopedic Surgery facility, diagnosed with Osteoarthritis of knee (ICD-10-CM code M17), the start time of scheduled procedure, description of the order of the antibiotic, and the start time for administering the medication for those who are allergic to Penicillin as an antibiotic.
Vita

Candidate’s full name: Mohammad Sadnan Al Manir

University attended:
Vienna University of Technology, Master of Science (MSc), 2009
Free University of Bozen-Bolzano, Master in Computer Science, 2009
Khulna University of Engineering & Technology, BSc Engineering in Computer Science & Engineering, 2005

Publications:
4. Jon Haël Brenas, Mohammad Sadnan Al Manir, Kate Zinszer, Christopher
J. O. Baker, Arash Shaban-Nejad: Exploring Semantic Data Federation to Enable Malaria Surveillance Queries. MIE 2018: 6-10
5. Alec Gordon, Mohammad Sadnan Al Manir, Brandon Smith, Amir Hossein Rezaie, Christopher J. O. Baker: Design of a framework to support reuse of Open Data about agriculture. SWAT4LS 2018
9. Mohammad Sadnan Al Manir, Alexandre Riazanov, Harold Boley, Artjom Klein, Christopher J. O. Baker: Automated generation of SADI semantic web services for clinical intelligence. SBD@SIGMOD 2016: 6