Bringing the Knowledge on the Web to Software Agents

A Framework for Developing Semantic Wrappers

José Luis Arjona

University of Sevilla

Doctoral Dissertation
Advised by Dr. Rafael Corchuelo

December, 2004

First published in December 2004 by
The Distributed Group
ETSI Informática
Avda. de la Reina Mercedes s/n
Sevilla, 41012. SPAIN

http://www.tdg-seville.info
contact@tdg-seville.info

Classification (ACM 1998): D.2.2 Design Tools and Techniques: Software libraries; D.2.11 Software Architectures; H.3.5 Online Information Services: Web-based services; I.2.4 Knowledge Representation Formalisms and Methods: Representation languages; I.2.6 Learning: Concept learning, Induction, Knowledge acquisition; I.2.11 Distributed Artificial Intelligence: Intelligent agents, Multiagent systems.

Support: Partially supported by the Spanish Ministry of Science and Technology under grants TIC-2000-1106-C02-01, FIT-150100-2001-78, TIC2003-02737-C02-01, and Castilla-La Mancha Local Government under grant PCB-02-001.

UNIVERSITY OF SEVILLA

The committee in charge of evaluating the dissertation presented by José Luis Arjona in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Engineering, hereby recommends _____ of this dissertation and awards the author the grade _____.

_____

Miguel Toro Bonilla
Catedrático de Universidad
Univ. de Sevilla

_____

Mario G. Piattini Velthuis
Catedrático de Universidad
Univ. de Castilla-La Mancha

_____

Carlos Delgado Kloos
Catedrático de Universidad
Univ. Carlos III de Madrid

_____

Rafael Berlanga Llavori
Titular de Universidad
Univ. Jaume I

_____

Emilio Santiago Corchado Rodríguez
Titular de Universidad
Univ. de Burgos

To put record where necessary, we sign minutes in _____, _____.

Information and knowledge by Melanie, aged twelve.

To Sina.
She gives semantics to my life.

Contents

List of Figures

List of Tables

Acknowledgements

Acknowledgement of one another's faults
is the highest duty imposed by our love of truth.

Ambrose G. Bierce, 1842–1914
Newspaper columnist

I am in no doubt that the best moment during the development of your Philosophiæ Doctor Thesis and the writing of your dissertation is when you finish them both. Not only because you reach the end of a hard way, but also because you can have a few minutes to relax, look back, and put your gratitude to many people in black ink.

Many individuals, friends and colleagues have been instrumental in making this dissertation a reality. Pivotal in this role was my research advisor, Dr. Rafael Corchuelo, since I would not have been able to finish my work without his help. Since he was the advisor of my master thesis, he has always trusted me and has always had the suitable, right words in every situation.

The help of a good research team is of uttermost importance. I have been lucky since my thesis work has been developed in the bosom of The Distributed Group, who have given me the encouragement and support I have needed since the beginning. Chiefly Dr. Miguel Toro, for his cheerful willingness to proof-read and criticise this document; without his support, I could not have begun this work. I shall never forget the arguments I had with Dr. David Ruiz, Dr. Antonio Ruiz and Joaquín Peña, or the breakfasts with José A. Pérez, Octavio Martín and David Benavides, which made my long working hours more bearable. Pablo Trinidad has joined to our group recently, and I am sure he shall be a valuable colleague in future.

This research also benefited tremendously from colleagues at other universities. I would like to thank to Dr. Carlos Delgado at Carlos III University of

Madrid, Dr. Rafael Berlanga at Jaume I University of Castellón, and Dr. Asunción Gómez-Pérez at Technical University of Madrid for providing me with valuable insights and feedback for our research work.

Finally, and most important, I thank my parents for bringing me into this world and for supporting me unconditionally in the pursuit of my life; and to my wife, Sina, without whose love, support and tenacity, not a word would have been written.

Abstract

You will have only an opportunity
to make a first impression.

Popular saying

In recent years, the web has consolidated as one of the most important knowledge repositories. A major challenge for software agents has become sifting through an unwieldy amount of data to extract meaningful information. This process is difficult because of the following reasons: first, the information on the web is mostly available in human-readable forms that lack formalised semantics that would help agents understand it; second, the information sources are likely to change their structure, which usually has an impact on their presentation but not on their semantics; and, third, it is a huge repository with about 4 200 Terabytes of information readily available.

The members of The Distributed Group have been working on distributed systems since 1997. They have focused on multiparty interaction models which provide the programmer with adequate mechanisms to describe complex interactions from a conceptual point of view. Results obtained have been materialised in major journal publications and theses. The research work in this dissertation opened a new research path in the group. It focuses on enabling the design and implementation of clean, reusable, understandable agents. Currently, this research path is being reinforced in Joaquín Peña's thesis, in which mechanisms to abstractedly describe complex interactions in multi-agent societies are been developed.

In this dissertation, we present a new framework to extract semantically-meaningful information from today's non-semantic web. Its main advantages are that it associates semantics with the information extracted, which improves agent interoperability; it can also deal with web changes, which improves adaptability; furthermore, it achieves a complete separation of issues

in the task of knowledge extraction, automating the development of distributed knowledge extractors. Last, the detailed study of other authors' work proves that our proposal constitutes a novel, original contribution.

Resumen

Sólo tendrás una oportunidad
de dar una primera impresión.

Dicho popular

En los últimos años la web se ha consolidado como uno de los repositorios de información más importantes. Un gran reto para los agentes software ha sido tratar con esa cantidad, poco manejable de datos, para extraer información con significado. Este proceso es difícil por las siguientes razones: en primer lugar, la información en la web tiene como objeto su consumo por seres humanos y no contiene una descripción de su semántica, lo que ayudaría a los agentes entenderla; en segundo lugar, la web cambia continuamente, lo que tiene generalmente un impacto en la presentación de la información pero no en su semántica; por último, es un enorme repositorio con 4 200 Terabytes de información lista para ser consumida.

Los miembros de The Distributed Group han estado trabajando en sistemas distribuidos desde 1997. Concretamente, han trabajado en modelos de interacción multipartitos que proporcionan al programador los mecanismos adecuados para describir interacciones complejas desde un punto de vista conceptual. Los resultados obtenidos se han materializado en publicaciones en revistas importantes y tesis doctorales. El trabajo de investigación en esta memoria abrió una nueva línea de investigación en el grupo. Su objetivo es facilitar el diseño e implementación de agentes software. Actualmente, esta línea de investigación se refuerza con la tesis de Joaquín Peña, en la que se están desarrollando mecanismos para describir abstractamente las interacciones complejas en sociedades multi-agentes.

En esta memoria presentamos un nuevo marco de trabajo para la extracción de información con significado de la web sintáctica actual. Sus principales ventajas son: asocia semántica a la información extraída, mejorando la interoperabilidad del agente; trata los cambios en la web, potenciando la

adaptabilidad; además, establece una separación de responsabilidades en la tarea de extracción, automatizando el desarrollo de extractores de conocimiento distribuidos. Por último, el detallado estudio del trabajo relacionado demuestra que nuestra propuesta constituye una contribución original.

Part I

Preface

Chapter 1

Introduction

There is nothing more difficult to take in hand,
more perilous to conduct, or more uncertain in its success,
than to take the lead in the introduction
of a new order of things.

Niccolo Machiavelli, 1469–1527
Italian dramatist, historian, and philosopher

In this dissertation, we report on our work to design a new framework that helps software agent developers to construct software agents able to unveil the semantically-meaningful information residing on today's non-semantic web. In this chapter, we first introduce the elements that constitute the context of our research work in Section §1.1; we then summarise our main contributions in Section §1.2; finally, we describe the structure of the dissertation in Section §1.3.

1.1 Research context

In this section, we present a bird's eye view of the main concepts we use through the rest of the dissertation. First, we introduce the software agents paradigm in Section §1.1.1; our view of ontologies in the web context is presented in Section §1.1.2; finally, the difference between information and knowledge is shown in Section §1.1.3.

1.1.1 Software agents

A software agent is a piece of software that exhibits the characteristics firstly described by Wooldridge and Jennings in Ref. [122], namely: autonomy, reactivity, pro-activity and social ability. Autonomy means that an agent operates without direct intervention of other agents or humans and has control over its actions and its internal state. Reactivity means that an agent perceives its environment and responds in a timely fashion to changes that occur in it. Pro-activity means that an agent does not simply react to changes in the environment, but exhibits goal-directed behaviour and takes the initiative when it considers it appropriate. Social ability means that an agent interacts with other agents (if it is needed) to complete its tasks and helps or contends with others to achieve their goals. Many researchers agree in that this vision is just as a characterisation; depending on their community, they also attach new attributes to software agents, e.g., mobility in distributed systems, adaptability in machine learning, or intelligence in artificial intelligence.

Autonomy, reactivity, pro-activity and social ability are not Boolean properties. We need to think in terms of dimension or measure of degree of them [83] in such a way that a software agent is a piece of software that exceeds a certain predefined threshold value for these attributes. Therefore, software applications that deserve to be called agents do not necessarily need to exhibit a maximum degree of autonomy, reactivity, pro-activity and social ability. Properties are not quantifiable and the adjustment of the threshold is totally subjective. Thus, software agents range from a simple procedure with prescriptive directions to new generation software that truly exhibits learning and artificial intelligence capabilities.

Nwana presents in Ref. [98] a classification of existing agent-based software. Seven different types of agents are defined: collaborative, interface, mobile, information, reactive, hybrid and smart agents. He also argues that some of these types could be seen as characteristics in a multi-dimensional space, therefore enabling the development of heterogeneous agent systems.

He presents the motivation and benefits of each type, as well as some application examples. For the purposes of this dissertation, we focus on information agents.

Information agents are software agents that typically have access to multiple, heterogeneous and geographically distributed information web sources, and manage relevant information on behalf of their users or other agents. This includes retrieving, extracting, analysing, manipulating, and integrating information. These agents are used frequently to analyse competitors' offers or to forecast what their future directions shall be based on the information residing on some web pages [23, 48, 69], e.g., BargainFinder which compares CDs prices amongst Internet stores for CDs, or Jasper which works on behalf of a user or community of users to store, retrieve and inform other agents of useful information on the web. In this dissertation, we focus on this kind of agents, that is, information agents that need pieces of information residing on web to accomplish their goals, what ever they are.

1.1.2 Ontologies

The term ontology has been used for long time ago in philosophy, and refers to a philosophical theory about the nature of existence. This term was redefined later in artificial intelligence. Currently, there exist many definitions of the term ontology, providing different and complementary points of views of the same idea: ontologies are models of the world. Guarino [56] established a comprehensive survey of the ontology definition from the artificial intelligence community.

Nowadays, in the artificial intelligence community, the most cited ontology definition is the one by Gruber [54, 55]:

> *An ontology is an explicit specification of a conceptualisation.*

This definition was refined by Borst in Ref. [17]:

> *Ontologies are defined as a formal specification of a shared conceptualisation.*

Studer and his colleagues analysed previous definitions and explained some terms [117]:

> *Conceptualization refers to an abstract model of phenomena in the world by having identified the relevant concepts of those phenomena. Explicit means*

> *that the type of concepts used, and the constraints on their use are explicitly defined. Formal refers to the fact that the ontology should be machine readable. Shared reflects that the ontology should capture consensual knowledge accepted by the communities.*

Another important definition was given by Berners-Lee and his colleagues in the context of the semantic web [15]:

> *An ontology is a document or file that formally defines the relations among terms. A typical ontology has a taxonomy defining the classes and their relations and a set of inference rules powering reasoning functions.*

Two main ideas follow from this definition: the former is that, in the semantic web, an ontology is more than a simple taxonomy of concepts; the latter is that an ontology should enable automatic reasoning on the domain being described.

For the purposes of this dissertation, an ontology is a formal explicit description of concepts and relationships amongst them in a domain of discourse that allows us to specify knowledge about the information on web. It is composed of the following elements: concepts that represent entities in the domain, properties describing relationships amongst them or their attributes, restriction on properties, and axioms to model axioms.

1.1.3 Information and knowledge

Information and knowledge have been defined in several contexts, e.g., philosophy, information technology, social sciences or economy. The definition that we use in this dissertation is established in the context of the semantic web, this means that the model used for the representation of the information or knowledge is the web, and that software agents are the consumers of information or knowledge.

Information are facts, statements about a particular subject, without a formal, explicit description of their semantics. The web is a suitable model for the representation of information. In the web model, we use HTML [104, 105] for publishing information on the web. HTML documents are text files that consist of tags that are used to specify how a web browser renders the information residing in them for human consumption.

Knowledge is information that a software agent can understand. In order to understand information on the web, a software agent needs an ontology

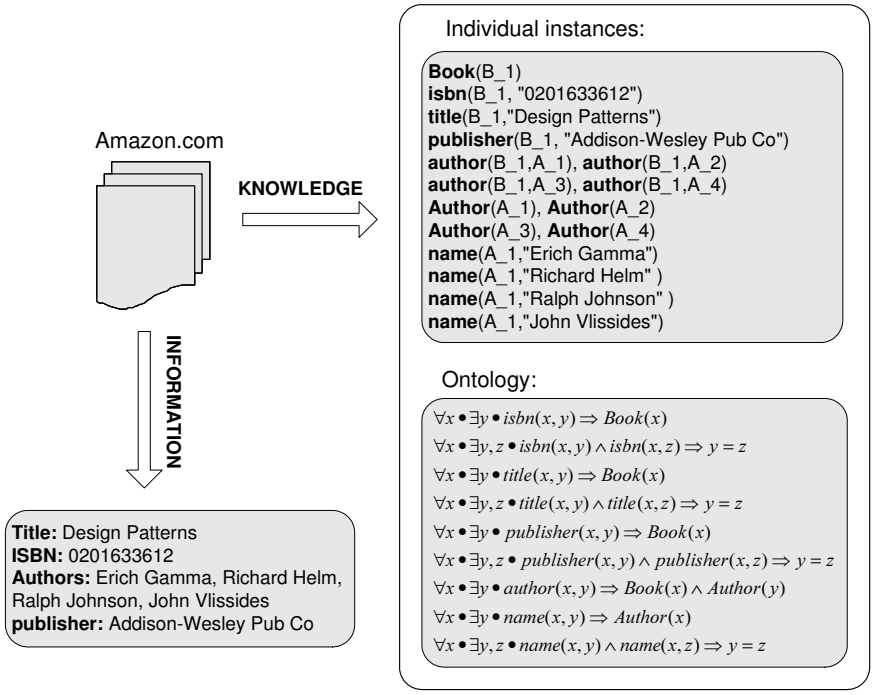


Figure 1.1: *Information vs. knowledge.*

that specifies the semantic information about that information, and a set of individual instances that relate the information on the web with the concepts specified in the ontology (an ontology with a set of individual instances of concepts is a knowledge base). The term "understand" refers to the ability of automatically reasoning on the knowledge, and there exist several degrees of reasoning, from the classification of concepts and individuals (taxonomy), to the discovery of new knowledge from axioms specified in the ontology.

Figure §1.1 illustrates the difference between information and knowledge. It offers us two views: the former is a piece of information; it is represented as simple text which is not a suitable representation format for software agents to formal automatic reasoning, since there is not an external specification of the semantics of information on text; reasoning on this information requires to embed the knowledge about the text into the functional logic of software agents. The latter is a piece of knowledge composed of an ontology formulated in first-order logic and a set of individual instances that relates the information on web with the ontology; software agents are able to reason automatically on this view, because there exists an explicit description of the semantics.

1.2 Summary of contributions

Our dissertation focuses on providing engineering support for software developers when information agents that require a piece knowledge residing on web pages to satisfy their goals need to be build. We have analysed the most important proposals in knowledge extraction carefully, and concluded that they are not usually well-suited to be applied to the development of software agents able to "understand" the information on the web at a sensible cost because of three problems, namely: the web is user-friendly, that is, the implied meaning of the terms that appear in web pages can be easily interpreted by humans, but there is not a reference to the ontology that describes them precisely, which complicates communication and interoperability amongst agents [15]; the layout and the appearance of web pages may change unexpectedly, which does not change the meaning of the information they provide, but may invalidate unexpectedly the automatic extraction methods used so far [19, 46, 82, 116]; last, the web is huge and distributed, so handcrafted solutions are infeasible and automatisation and distribution are necessary features to take advantage of this potential.

Our main result is a new framework to which we refer to as WebMeaning. Its main advantages are that it associates semantics with the information extracted, which improves agent interoperability; it can also deal with changes to the structure of a web page, which improves adaptability, and it achieves a complete separation of issues in the task of knowledge extraction, automating the development of distributed knowledge extractors.

Many results have materialised in publications in conferences and journals. Below is a list of publications in which we point out the main cornerstones during the development of our results:

2001: We analysed the information extraction problem in the context of a multi-agent society, and proposed a reference architecture for software agents to access to the information in the web. The results were presented in the ZOCO meeting [6].

2002: We refined our results and devised an abstraction mechanism to bring the knowledge on the web to software agents. This mechanism was defined as core agents called knowledge channels in a multi-agent society. The results were published as a short paper in the 14th Conference on Advanced Information Systems Engineering (CAiSE '02) [9], and as regular paper in UPGRADE/Novatica [30].

We reported on two case studies in which our work was valuable: the former was the personalisation of web sites based on users' profiles (the

information from the web that is interesting to a user can be extracted automatically and displayed in a suitable form); this work was presented at The Adaptive Hypermedia and Adaptive Web-Based Systems (AH2002) Conference [8]. The latter was to develop a web agent that helps a Spanish political party decide if it should organise an outdoors meeting to attract voters; data mining techniques establish the agent decision rules and our framework feeds them with information from the web; this work was published in the International Journal of Computers, Systems and Signals [29].

2003: We polished our ideas and devised a formalisation of the kernel of our framework and the results were published as short paper in the 15th Conference on Advanced Information Systems Engineering (CAiSE '03), and as regular paper in the 2003 IEEE/WIC International Conference on Web Intelligence (WI2003) [11].

2004: Recently, we have submitted a paper describing the main results in this dissertation to the Software: Practice and Experience journal.

Other important results are the citations, because they transmit us that our research work is on the good way. Below is a complete list of references to our work:

Conference and journal citations. Our work has been cited in the context of knowledge management and software agents by the following papers: "Enhancing the adaptivity of an existing website with an epiphyte recommender system" [109] in the journal of New Review of Hypermedia and Multimedia; "An Ontology-Based Knowledge Management Platform" [3] presented at the Workshop on Information Integration on the Web (IIWEB'03) at IJCAI'03; "Agent Systems Today: Methodological Considerations" [100] presented at the International Conference on Management of e-Commerce and e-Government; "A Generic Model for Distributed Real-Time Scheduling Based on Dynamic Heterogeneous Data" [16] presented at the Pacific Rim International Workshop on Multi-Agent Systems; "A Multi-agent System for Semantic Information Retrieval" [121] presented at the 17th Conference of the Canadian Society for Computational Studies of Intelligence; we would like to mention a reference from a paper[†1] we cannot understand because it is in Japanese.

Recommended bibliography. Our work appears as recommended bibliography in the doctoral course "Retrieval and extraction of information from

[†1] `http://www.im.fju.edu.tw/conference/proceeding.htm`

the Web" offered by the Department of Computer Science at the Rosario National University (Argentina)[†2]. Also, in an undergraduate course offered by the Department of Computer Engineering & Informatics at the University of Patras (Greece)[†3]. Last, the Multimedia Web Search Agents (DIOGENES) research project[†4], at the Department of Computer Science and Engineering in The University of Texas at Arlington recommends one of our papers for reading.

Thesis. The Master Thesis "Using Software Agents to Index Data for an E-Travel System" [95] from the Computer Science Department at Oklahoma State University at Tulsa, references our work in the context of information extraction.

Technical reports. The technical report "The World of Travel: A Comparative Analysis of Classification Methods" [51], from the Computer Science Department at Oklahoma State University at Tulsa, references our work in the context of a categorisation of travel-related web resources.

1.3 Structure of this dissertation

This dissertation is organised as follows:

Part I: Preface. It comprises this introduction only.

Part II: Background information. Here, our goal is to provide the reader with a deep understanding of the research context in which our results have been developed. In Chapter §2, we present the current web, then the new semantic web is presented; we also report on the formalisms and languages that enable the transition to this new web era. In Chapter §3 and Chapter §4, we focus other proposals, which are classified and presented proposals along two different dimensions: information extraction and knowledge extraction. In Chapter §5, we report on the web services since they are the key to the proof-of-concept implementation of our framework.

Part III: Our contribution. It is the core of our dissertation, and it is organised into five chapters. In Chapter §6, we motivate our research and prove that current solutions are not practical enough. In Chapter §7,

[†2] `http://www.unr.edu.ar`
[†3] `http://ceid.upatras.gr/proptyxiaka/diplomatikes/tsakali0405.pdf`
[†4] `http://ranger.uta.edu/~alp/ix`

we rigorously formalise the WebMeaning framework; we also define the knowledge extraction process as a compound of four activities, namely: information extraction, information verification, translation of information into knowledge, and semantic verification. In Chapter §8, we focus on presenting our proposal for the translation of information into knowledge from a very abstract point of view. In Chapter §9, we materialise it by defining the implementation of three algorithms. Finally, in Chapter §10, we devise a proof-of-concept implementation of our framework; it defines a comprehensive architecture based on SOA in which the core elements of WebMeaning are mapped onto web services.

Part IV: Final Remarks. It consists of one chapter in which we report on our main conclusions and future research directions.

Part V: Appendices. We summarise the mathematical notation we use in Appendix §A; and we prove the equivalence between two data structures for representing knowledge in Appendix §B. Appendix §C spells out most acronyms used in this dissertation.

Part II

Background information

Chapter 2

From information to knowledge

Internet is so big, so powerful and pointless
that for some people it is a complete substitute for life.

George Carlin, 1937–
American comedian and actor

The nowadays web was designed as an information source for human consumption. An important goal for the future web is that software applications should be able to participate. This amounts to a transition from a syntactic web to a semantic web in which metadata expresses information residing on the web in a machine-readable form, that is, programs can read and "understand" it, c.f. Section §1.1.3 for a definition of what we mean by "understand". This chapter is organised as follows: in Section §2.2, we introduce several concepts from the knowledge representation field to understand the previous transition; in Section §2.3, we discuss on the characteristics of the nowadays web; then the future web, which is being currently settled, is presented in Section §2.4; Section §2.5 briefly presents current strategies for reasoning in the future web; finally, Section §2.6 summarises the ideas in this chapter.

2.1 Introduction

The current World Wide Web is a repository of information. The main consumers of this information are humans, who are aware of its value. According to the UCLA Center for Communication Policy, the web is viewed as an important source of information by the vast majority of people who go online; in 2002, 60.5 percent of all users considered the Internet to be a very important or extremely important source of information [45]. The idea of having a fully accessible repository of information propitiated that researchers began working on proposals whose goal was to manage the information on the web [12, 44]. Nowadays, managing this information is a big challenge because the web is a massive, distributed, dynamic, unbeknownst, and syntactic repository of structured, semi-structured and unstructured web documents.

The current web is changing to a new semantic environment in which agents are able to understand the web content, to run around it, and to perform complex actions for their users with less human intervention [15]. This change requires a lot of research work, and the integration of technologies from fields such as software engineering or artificial intelligence in order to develop the infrastructure that shall make it a reality.

Ontologies [28] play an important role to fulfill the vision of a web for agents consumption. They provide a representation of a shared conceptual model of a particular domain that can be communicated across people and agents. Thus, ontologies allow us to specify semantic information about the information on the web. In addition, ontological web languages shall represent these conceptual models in machine-understandable forms, while reasoners shall allow agents to reason and compute over those models.

2.2 Knowledge representation

In this preliminary section, we provide a brief description of the formalisms and languages from a central subfield of artificial intelligence called knowledge representation. Research in knowledge representation focuses on highly expressive formalisms and languages to represent knowledge bases and on powerful reasoning services over them.

The difference between a formalism and a language is in the abstraction level at which they represent knowledge. Formalisms, also known as paradigms, are at the bottom level; they are the foundation of knowledge representation languages; that is, languages build on one or more formalisms; thus,

languages aim to represent knowledge bases at higher abstraction levels. For instance, frame systems, and first-order logic are two formalisms that are combined in the Ontolingua [38] knowledge representation language to provide a distributed collaborative environment to manage ontologies.

Note that there is an important difference between the notion of ontology and the formalism or language for expressing it. Many different formalisms or languages from different communities are used to express the same conceptual model. For instance, research in databases is related to the representation of the structure of information, and mainly deals with efficient storage and retrieval with powerful query languages. In the databases field, models are used to express the overall logical structure of a database graphically; e.g., the entity-relationship model. Also when constructing software applications, the software engineer uses models to represent the domain of the problem, for instance, class diagrams. However, in the database and software engineering fields reasoning over the knowledge plays a minor role.

2.2.1 Formalisms

Formalisms for knowledge representation are classified into non-logical and logical. Non-logical formalisms were developed motivated by the idea that logic could not be the right formalism for knowledge representation; they are based on cognitive experiments about how knowledge is stored in human brains. On the other side, logical formalisms are logic-oriented approaches to knowledge representation.

Semantic networks, frame systems, and conceptual graphs are the most important non-logical formalisms in current use, namely:

Semantic networks. This formalism was introduced in 1967 by Quillian [107]. It is a graphical notation for representing knowledge in patterns of interconnected vertices, and directed and labelled edges (directed graph). A great variety of semantic network formalisms were proposed [18], since then. In all these formalisms, vertices represent concepts or individuals, whereas the edges represent relations amongst concepts or properties attached to a concept. This formalism was criticised for lack of a precise semantics. Some researches aimed to specify the semantics of semantics networks and this led to the definition of the first description logic.

Frame systems. Minsky introduced frame systems in 1975 [92]. A frame is a data structure for representing a concept or situation in an object-oriented way that typically consists of slots (or attributes), where each slot can be

attached a description or a procedure; a collection of frames is organised and interconnected in frame systems. The declarative or monotonic part of this formalism is described semantically in first-order logic, but there is not a precise semantic description for the non-declarative or non-monotonic part of frame systems.

Conceptual graphs. They were introduced by Sowa in 1984 [114]. This formalism is the most popular to represent knowledge in a graphical way, and can be viewed as a descendant of frame systems and semantic networks. Conceptual graphs are labelled graphs where "concept" nodes are connected by "relation" nodes. This formalism is given a formal semantics by translating them into first-order formulae. This means that not only can we represent knowledge using it, but also reason about it.

First-order logic, description logics, and non-monotonic logics are the most popular logical formalisms, namely:

First-order logic. It is the most important and expressive knowledge representation formalism. It allows to represent complex facts about the world in a domain of discourse and to infer conclusions. Furthermore, it guarantees that, if the initial facts were true then conclusions are true, too. It is a well understood formal language, with well-defined syntax, semantics and inference rules. However, first-order logic is semi-decidable, i.e., the inference problem is computationally not tractable.

In order to solve this problem, it is possible to define decidable fragments of first-order logic. For instance, if $\mathcal{L}^k$ denotes a first order logic over unary and binary predicates with at most k variables, and $\mathcal{C}^k$ denotes a first-order logic over unary and binary predicates with at most k variables and counting quantifiers $\exists^{\geq n}$, $\exists^{\leq n}$; then both $\mathcal{L}^2$ and $\mathcal{C}^2$ are *NEXPTIME*-complete.

Description logics. This is a family of knowledge representation formalisms with a formal, logic-based semantics [13]. It is considered a unifying formalism for structured representation, since, for instance, frames, object-oriented representations, or data base entity-relation diagrams can be formulated with description logics. The formal semantics of a description logics language is specified using a subset of first-order logic; thus, description logics can be seen as fragments of first-order logic.

The knowledge base represented in a description logics-based system is divided into two parts: the Tbox (terminological box), which defines the structure of the domain, and the Abox (assertional box), which describes a concrete example of the domain. The Tbox consists of a set of axioms

that can be either concept definition axioms or concept inclusion or subsumption axioms. Concept definition axioms allow to define a new concept in terms of other previously defined concepts; the concept inclusion or subsumptions axiom allow to state that a concept is considered more general than other. An Abox consists of a set of assertional axioms that can be either concept assertions or property assertions. A concept assertion states that an individual is an instance of a concept, and a property assertion states one individual is related to another by a given role.

Non-monotonic logics. The term non-monotonic logic denotes a family of formalisms devised to capture and represent defeasible inference [79]. This kind of inference allows reasoners to draw conclusions tentatively, reserving the right to retract them in light of further information. Such inferences are called non-monotonic because the set of conclusions does not increase with the size of the knowledge base. This is in contrast with classical first-order logic, in which inferences can never be undone by new information. There exist three major approaches to formalising non-monotonic reasoning, namely: circumscription, default logic and modal logic. In circumscription [88], theories are written in classical first-order logic, however the entailment relation is not classical; the idea of default logic [71] is to reason in first-order logic but to have available a set of default rules which are used only if an inference cannot be obtained within the first order formulation; modal logic was designed to express possibility, necessity, belief, knowledge, temporal progression, obligations and other modalities.

2.2.2 Traditional languages

Languages for knowledge representation are classified into traditional languages [27] and ontological web languages. This classification is motivated by the semantic web, that is, the development of new web-centric languages that focus on specifying the semantics of information contained on web pages and the exchange of ontologies across the web. Here, we present a sketch of most popular traditional languages only since the ontological web languages shall be presented latter in the semantic web context.

CARIN. It is a family of languages, each of which combines a description logic $\mathcal{L}$ with non-recursive Horn rules without functions [81]. For instance, CARIN-$\mathcal{L}$ is one such language whose knowledge base is formed by three components: a description logic terminology, a set of Horn

rules, and a set of ground facts. There exist sound and complete algorithms which provide reasoning services over CARIN-$\mathcal{L}$ knowledge bases.

Frame logic. It integrates frames and first-order logic for specifying object-oriented databases, frame systems, and logical programs [76]. Its main achievement is to integrate conceptual modelling constructs as classes, attributes, inheritance, or axioms, into a coherent logical framework. Frame logic has a model-theoretic semantics and a sound and complete resolution-based proof theory.

LOOM. It is a language and an environment for constructing intelligent applications [85]. The core of LOOM is a knowledge representation system that is based on description logics and production rules. Knowledge in LOOM consists of definitions, rules, facts, and default rules. A deductive engine uses forward-chaining, semantic unification and object-oriented truth maintenance technologies in order to compile the knowledge into a network designed to efficiently support on-line deductive query processing.

OCML. It supports the construction of knowledge models by means of several types of constructs [93]. It allows the specification of functions, relations, classes, instances and rules. To make the execution of the language more efficient, it also adds some extra logical mechanisms for efficient reasoning, e.g., procedural attachments. It offers mechanisms to query knowledge represented using an OCML model.

Ontolingua. It provides a distributed collaborative environment in order to browse, create, edit, modify, and use ontologies [38]. The language to represent knowledge is based on frame systems and first-order predicate calculus in KIF [52] notation. It allows to build ontologies by either using a frame ontology vocabulary, KIF expressions, or both languages simultaneously (including KIF expressions in definitions based on the frame ontology vocabulary). There exists an inference engine for Ontolingua.

2.3 The nowadays web

The current web was described the W3C as follows [14]:

It is a wide-area hypermedia information retrieval initiative aiming to give universal access to a large universe of documents.

Since then, it has overcome all the initial expectations. Not only is it the biggest universally accessible information repository, but also has important repercussions on society. A good example is Europe's Information Society thematic portal[†1], which defines a set of activities that aim at developing modern public services and a dynamic environment for e-business. Amongst these activities are eGovernment (for the exchange of governmental practices), eHealth (a set of tools and services for keeping citizens informed, administrative support, and home care/tele-medicine), or eLearning (new multimedia technologies to improve the quality of learning and services as remote exchanges and collaboration).

The idea of having a fully accessible repository of information propitiates that researchers from database [44] or artificial intelligence [12] fields began working on proposals whose goal was to manage information on the web. However, the web has some characteristics that makes it hard to manage the information it provides, namely [61]:

The web is syntactic. HTML documents are text files that consist of tags that are used to specify how to render information, but there is not a reference to the semantic information that describes the meaning of this information. HTML was never meant for computer consumption, but for human presentations.

The web is dynamic. The web is changing all the time [19, 82, 116]. Francisco-Revilla et al. [46] presented a categorisation of web change based on the nature of the changes, namely: content or semantic, presentation, structural and behavioral. The former refers to modifications of the page contents from the point of view of the reader; the second are changes related to the document representation that do not reflect changes in the topic presented in the document; the third refers to the underlying connection of the document to other documents (hyperlinks); the latter refers to modifications to the active components of a document (changes in scripts, plug-ins or applets).

The web is massive. According to `Cyveillance.com`, estimations for the year 2000 indicate that the surface web, i.e., static publicly available pages, is composed of about 2.5 billion web pages, with a rate of growth of 7.3 million pages per day. With this information, it is possible to estimate the total amount of information on the surface ranges from 10 to 20 Terabytes. The size of the deep web, i.e., pages that exist only on demand, is estimated to be 550 billion web pages, which amounts to approximately 4 200 Terabytes of information.

[†1] `http://europa.eu.int/information_society/eeurope/2005`

Furthermore, this information is not centralised. From an abstract point of view, the web is a directed graph whose nodes are web pages and whose edges are links between them. Each web page or web resource can reside on a different machine. Therefore the information on the web is spread out over thousands of computers, making it a distributed system that provides the content available to any user from any location in the world with an Internet connection.

2.4 The semantic web

The semantic web was defined in Ref. [15] as follows:

The semantic web is an extension to the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation.

The semantic web is not a separate web but an extension to the current one. This extension is achieved by annotating web pages with metadata that describes the concepts that define the semantics associated with the information in which we are interested. The semantic web shall simplify and improve the accuracy of current information extraction techniques tremendously. Nevertheless, this extension requires a great deal of effort to annotate current web pages with semantics.

Ontological web languages enable the specification of ontologies and the annotation of information in web pages with semantics. A semantic web page has two different views: the former is for human consumption, a document in HTML format that browsers render to a friendly multimedia document (current web pages); the latter is for software agents consumption, a structured document with semantic metadata annotations (references to concepts and properties defined in some external ontologies) of the information of interest.

Ontological web languages are influenced by the knowledge representation field. However, they differ from traditional knowledge representation formalisms in that they are web-centric languages. The main features that have been identified for web ontology languages are the following [62, 64]:

i. They extend existing web standards such as HTML, XML, RDF-S, enabling their integration with other web technologies.

Language	Score	Normalised score	Underlying formalism
XOL	2.5	33.33	Frame systems
SHOE	−1.5	3.70	Frame systems
OML	10.5	92.59	Conceptual graphs
RDF-S	−2.0	0	Semantic networks
OIL	11.5	100	Description logics
DAML+OIL	11.5	100	Description logics
OWL	11.5	100	Description logics

Table 2.1: *Expressiveness of most relevant ontological web languages.*

ii. They enable the definition of diverse, potentially conflicting ontologies to deal with a decentralised web.

iii. They enable the evolution of the vocabularies as human understanding improves in order to handle the rapid evolution of the web.

iv. They are scalable to deal with the huge size of the web.

v. They must be specified formally and must provide automated reasoning support, in order to be useful for agents.

vi. They must have an "adequate" expressive power in order to represent the knowledge residing on the web at a sensible computing cost.

There are many ontological languages that aim at specifying the knowledge on the web, e.g.: SHOE [84], RDF-S/RDF [20], XOL [73], OML [75], DAML-ONT [90], OIL [40, 41], DAML+OIL [64, 89], or OWL [34]. Table §2.1 summarises the results of an analysis presented in Ref. [53] that was carried out to compare their main features, e.g., the possibility of defining instances of concepts, defining axioms or using cardinality constraints. A value of 1 was assigned to each supported feature, a value of -1 was assigned to each unsupported feature, and a value of 0.5 to each partially supported feature. The result of this numerical conversion is shown in the score column. The normalised score column shows the result of normalising the scores to a [0, 100] interval. It can be readily observed that the languages with the best scores are OIL, DAML+OIL and OWL; they obtain the same score due to the kinship relationships amongst them (OWL is the W3C standardisation of DAML+OIL, and DAML+OIL is an extension to OIL). Note that expressiveness and reasoning support in these languages are related with the language underlying formalism; that is, languages based on logic formalisms obtain the higher scores,

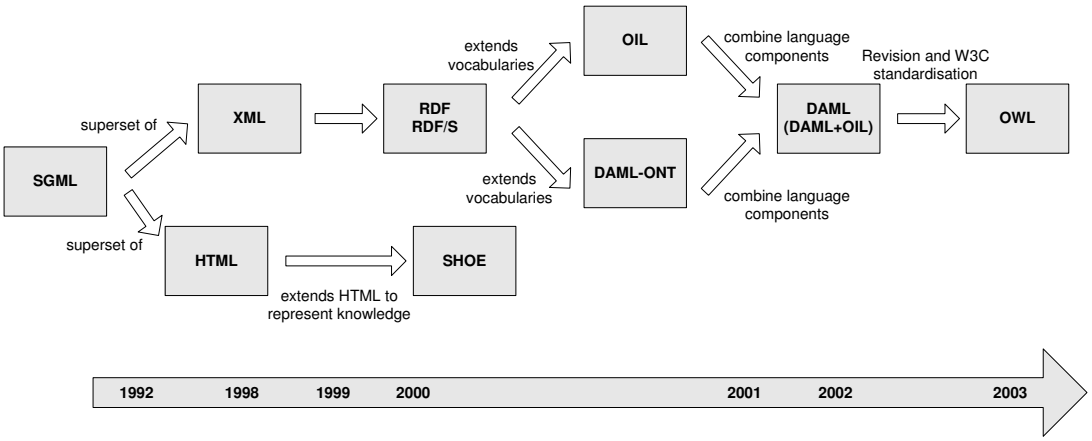


Figure 2.1: *Ontological web languages evolution.*

whereas languages based on non-logic formalisms obtain the lower scores. All of the languages, except for XOL and OML, have automatic reasoning support.

Next, we present a brief description of the most important languages in the ontological web languages evolution (c.f. Figure §2.1), namely: RDF-S/RDF, SHOE, DAML+OIL, and OWL.

RDF and RDF-S. They were developed by the W3C to provide a standard way of specifying data about something (RDF) and their interpretation (RDF-S) [20]. The main characteristic of RDF is that it is used as a general framework for the integration and exchange of knowledge described in more expressive ontological web languages, e.g., OIL, DAML+OIL, or OWL build on RDF-S and extend it with richer modelling primitives, but use RDF-S constructs as much as possible in order to maintain compatibility amongst them.

The RDF data model [106] is based on a semantic network model. It consists of a set of RDF statements, each of which is a sentence that has three parts: a subject, a predicate and an object. Both statements and predicates can be used as the subjects or objects of other statements. In RDF parlance, the subject is a resource, which is anything that can be identified on the web by a URI (web pages, people or flowers), and the object can be a resource or a literal that expresses aspects, characteristics, attributes, or relations used to describe the subject resource; the predicate is the resource that defines the relationship between the subject and object. The semantics of the data model is defined by translating RDF

data model statements into sentences in first-order logic, and giving a set of axioms that constrain the possible interpretation [43].

SHOE. It was one of the very first attempts at defining an ontology language for the web [84]. It is an extension to HTML that allows to annotate web pages with machine-readable knowledge. It is based on the frame formalism, and allows to add Horn clauses. The semantics of SHOE is defined by extending the standard model-theoretic approach for definite logic with mechanisms to handle distributed ontologies.

DAML+OIL. It is the effort of a merger between the languages DAML-ONT, which was developed by DARPA, and OIL, which was developed in the context of the European IST OnToKnowledge project [42]. DAML+OIL takes a frame approach to describe the structure of a domain, i.e., we can define the structure using classes and properties [64, 89]. An ontology consist of a set of axioms that assert relationships between these classes and properties. Formally, DAML+OIL is a syntactic serialisation of a very expressive description logic [65], with a DAML+OIL ontology corresponding to a description logic terminology (Tbox).

OWL. It is the ontology web language that is currently being developed by the W3C web ontology working group [34], and it is a W3C recommendation since February 2004. OWL was designed as a revision of the DAML+OIL web ontology language. As DAML+OIL, OWL combines the standard modelling primitives from the frame-based formalism with the formal semantics and reasoning services provided by description logics. There are only minor changes between OWL and DAML+OIL.

According to the W3C Recommendation, OWL provides three increasingly expressive sub-languages designed for use by specific communities of implementers and users, namely: OWL Lite, for users who need a classification hierarchy and simple constraints only; OWL DL, for users who want the maximum expressiveness while retaining computational completeness (all conclusions are guaranteed to be computable) and decidability (all computations are guaranteed to finish in finite time); and OWL Full, for users who want maximum expressiveness and the syntactic freedom of RDF with no computational guarantees.

2.5 Reasoning on the semantic web

Ontological web languages allow to represent the knowledge residing on the web independently from the applications that consume it, but depending

on the domain. In addition, a number of reasoners allow to perform complex tasks on this knowledge, namely: checking the ontology for consistency, i.e., to verify the syntax and usage of the ontology language and to ensure that the individual instances meet all of the constraints imposed by the ontology; computing entailments including satisfiability, i.e., checking whether a concept expression not always denote the empty concept; subsumption, i.e., checking whether a concept is considered more general than a second one; or processing queries both from human users and software agents.

Reasoners use different logic approaches to perform these tasks. Reasoners based on description logics focus on computability, which enables the definition of decidable and efficient reasoning algorithms while retaining a considerable degree of expressiveness. Reasoners based on full first-order logic theorem provers focus on expressiveness, and they are able to deal with languages that are more expressive, but reasoning on first-order logic is not computationally tractable in general.

Table §2.2 summarises the previous ideas by analysing the most popular reasoners, namely: cwm[†2], Fact[†3], F-OWL[†4], Pellet[†5], RACER[†6], Surnia[†7], TRIPLE[†8], Hoolet[†9], and XSB[†10]. Because of the similarity between DAML+OIL and OWL, we use OWL to refer to both languages. From this table, it is not difficult to realise that much work remains to be done in this area.

2.6 Summary

In this chapter, we have presented two different views of the web: the former is the current web, a huge changing repository of syntactic information for human consumption; the latter is the semantic web, a huge repository of knowledge for software agent consumption. They are not two separate and different webs, since they both are supported by the same computers as web pages. We have shown that ontological web languages play an important role to fulfill the vision of the semantic web, because they allow the specification of ontologies and the annotation of the information in web pages with semantics.

[†2] http://www.w3.org/2000/10/swap/doc/cwm.html
[†3] http://www.cs.man.ac.uk/~horrocks/FaCT
[†4] http://fowl.sourceforge.net
[†5] http://www.mindswap.org/2003/pellet
[†6] http://www.sts.tu-harburg.de/~r.f.moeller/racer
[†7] http://www.w3.org/2003/08/surnia
[†8] http://triple.semanticweb.org
[†9] http://owl.man.ac.uk/hoolet
[†10] http://www.cs.sunysb.edu/~sbprolog/xsb-page.html

	cwm	Fact	F-OWL	Pellet	RACER	Surnia	TRIPLE	Hoolet	XSB
Based on	Horn	DL	Horn, Frame, Higher order	DL	DL	FOL	Horn, DL	FOL	Horn
Support	RDF	OWL-DL	OWL-Full	OWL-DL	OWL-DL	OWL-Full	RDF	OWL-DL	SHOE
Complete consistency checker	No	No	No	OWL-Lite	OWL-Lite	No	No	No	No
Decidable	No	Yes	No	Yes	Yes	No	Yes	No	No
Query			RDQL	RDQL	RQL		Horn style		Horn style
Known limitation	RDF	No Abox support	Poor scaling			Poor scaling		Poor scaling	SHOE

Table 2.2: *Features of some reasoners for the semantic web.*

Chapter 3

Extracting information from the web

Where is the wisdom we have lost in knowledge?
Where is the knowledge we have lost in information?

Thomas Stearns Eliot, 1888–1965
British critic, dramatist and poet

Today's web is an enormous and valuable source of information. This chapter presents several proposals that aims at unveiling the information residing on the web. It is organised as follows: Section §3.1 is an introduction to the chapter; in Section §3.2, we present some features that allow to characterise information extraction proposals; then, the main proposals are presented in Section §3.3; in Section §3.4, we present the necessity of maintaining information extraction systems to deal with web changes; finally, Section §3.5, summarises the ideas in this chapter.

3.1 Introduction

The algorithms used for extracting information from the web are called wrappers [36]. They can be codified manually using properties of a web page, usually looking for strings that delimit the data to be extracted. Manual wrapper generation is the approach chosen in proposals such as TSIMMIS [24], ARANEUS [91], or JEDI [68]. Their goal is to integrate heterogeneous information sources such as traditional databases and web pages so that the user can work on them as if they were homogeneous information sources. An important contribution to the field of information extraction was provided by Kushmerick [78]. He introduced induction techniques and defined a new class of wrappers called inductive wrappers, which are algorithms that use a number of extraction rules generated by means of automated learning techniques, e.g., inductive logic programming, statistical methods, and inductive grammars. These rules allow to set up a generic algorithm to extract information from similar pages automatically. Figure §3.1 illustrates the life cycle of an inductive wrapper.

Unfortunately, wrappers have a limited life period, and they need maintenance. The wrapper maintenance problem consists of two parts: wrapper verification and wrapper reconstruction. A wrapper verification system determines if a wrapper is broken, which happens if the information extracted is not the expected, which may be due to changes on the web that have invalidated the logic of the wrapper; wrapper reconstruction consists of building a new wrapper adapted to the changes. The wrapper maintenance problem has been paid little attention, since, there exist very few proposals in the literature. Therefore, this task is usually achieved by developing domain-dependent verifiers manually.

3.2 Characterising wrappers

In the literature concerning information extraction, the terms record, frame, and template refer to the same idea: a data structure for representing the information extracted, which typically consists of slots or attributes; furthermore, these terms are used indistinctly to refer to an extracted feature. In this dissertation, we use the frame and attribute terms. Next, we present a simple classification of wrappers according to the kind of frames and web pages with which they can work.

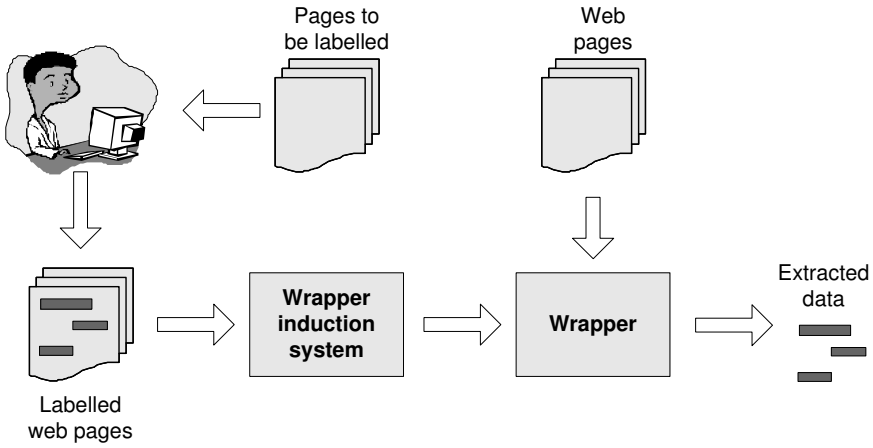
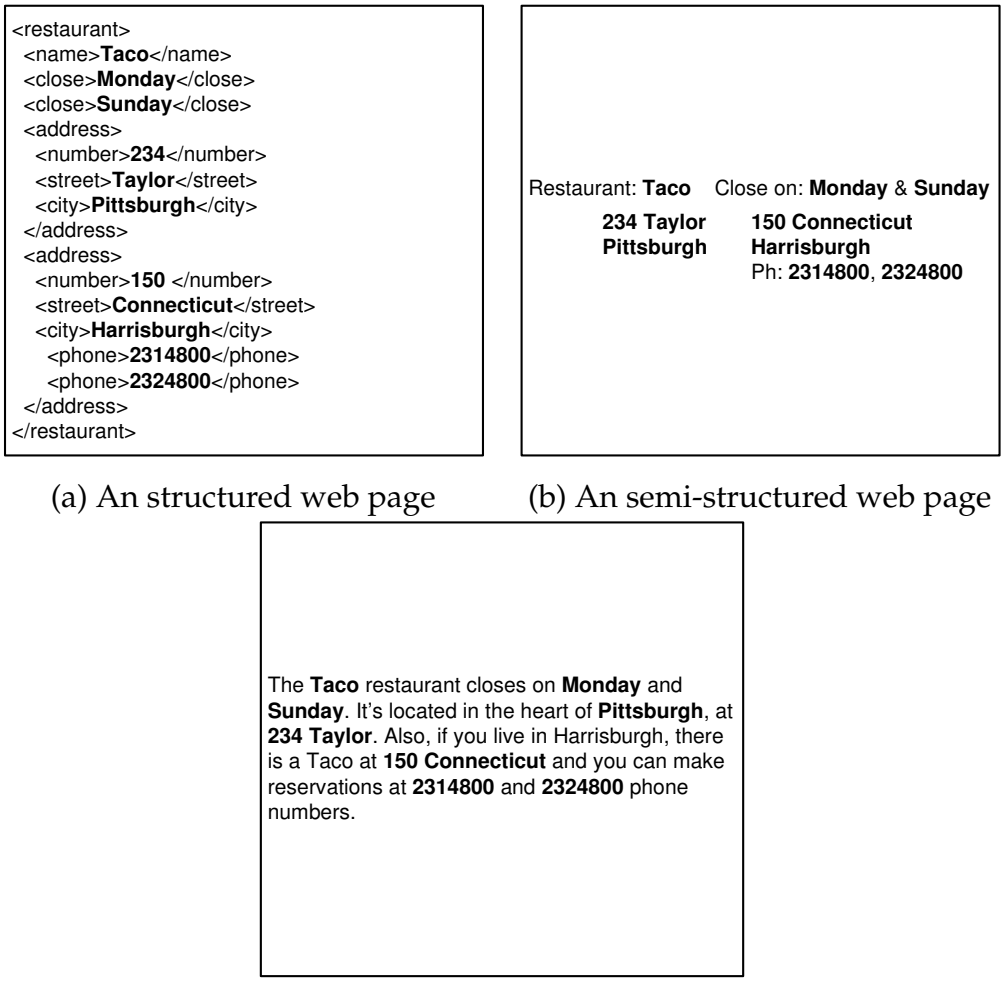


Figure 3.1: *Life cycle of an inductive wrapper.*

Single-slot vs. multi-slot extraction. A wrapper is able to perform single-slot extraction if it can only extract isolated attributes from a web page. A wrapper is able to perform multi-slot extraction if it can locate a pattern and searches recursively for similar patterns in a web page and links together attributes into frames.

Structured, semi-structured and unstructured web pages. Hsu and Dung introduced a categorisation of web pages based on the structuredness of attributes that users want to extract from them [67], namely: structured, semi-structured and unstructured web pages. A web page is structured if it provides itemised information; that is, the information has a predefined, strict format that allows to define the delimiters of every piece of information easily. A web page is semi-structured if it may contains frames with missing attributes, or attributes with multiple values, or attribute permutations; usually, inductive techniques are used to extract information from semi-structured web pages. A web page is considered unstructured if linguistic knowledge is required to extract frames correctly. Wrappers for these pages typically obtain patterns that build on syntactic relations between words or semantic classes of words. Figure §3.2 illustrates an example of structured, semi-structured and unstructured web pages.

This categorisation is interesting because the techniques used in information extraction determine the kind of web pages that can be handled. For instance, if the system uses natural language processing techniques, it shall work fine on unstructured pages, but not with web pages in which information is not in sentential form [32]; since it is not easy to



(a) An structured web page (b) An semi-structured web page

(c) An unstructured web page

Figure 3.2: *Structured, semi-structured and unstructured web pages.*

Name	Struc.	Semi-struc.	Unstruc.	S-slot	M-slot	Hi.
RAPIER	✓	✓		✓		
SRV	✓	✓		✓		
WHISK	✓	✓	✓	✓	✓	
WIEN	✓			✓	✓	
SoftMealy	✓	✓		✓		
STALKER	✓	✓		✓		✓

Table 3.1: *Features of some inductive wrappers.*

delimit the scope of a piece of data because of the HTML tags used to specify how to render it (which implies these techniques are not appropriate in general [60] to extract information from structured and semi-structured web pages).

Tabular vs. hierarchical information. Wrappers can also be classified based on how the information is structured on web pages. Most wrappers extract information structured in a relational or tabular manner. There is also a few able to extract hierarchically nested structured information, that is, values for attributes are presented with the information organised as if they were trees rather than tabulary.

3.3 Common inductive wrappers

Next, we describe the most relevant inductive wrappers systems, namely: RAPIER, SRV, WHISK, WIEN, SoftMealy, and STALKER. Their characteristics are summarised in Table §3.1. The first three columns indicate the type of web pages with which they can deal; the following two columns indicate whether the wrapper can perform single-slot and/or multi-slot extraction, respectively; the last column indicates whether the wrapper can extract hierarchical information.

RAPIER. It takes pairs of documents and filled frames and induces extraction rules that directly extract fillers for the attributes in the frame [22]. The inductive algorithm used by RAPIER is based on several inductive logic programming systems. The extraction rules generated with RAPIER are based both on syntactic information (delimiters) and semantic information (content description). They are indexed by a frame name and an

attribute name and have three parts: a pre-filler pattern that must match the text preceding the information to be extracted (a left delimiter), a filler pattern that describes the structure of the information to be extracted, and a post-filler pattern that must match the text following the information to be extracted (a right delimiter).

SRV. It generates first-order logic extraction rules that are based on features that are simple, e.g., length, character type or relational, e.g., next and previous token to the attribute to be extracted [47]. Using first-order logic to represent rules makes them very expressive. The learning procedure consists of identifying and generalising the features found in the training examples, which need to be many or otherwise the procedure does not produce good results.

WHISK. It generates rules that are based on regular expression patterns that have two components: one that describes the context that makes an attribute relevant and another that specifies the exact delimiters of the attribute to be extracted [113]. Depending on the structure of the web pages, WHISK generates patterns that rely on either of the components: for unstructured web pages it uses context-based patterns; for structured pages, it uses delimiter-based patterns; and for semi-structured web pages, both approaches are used.

WIEN. It introduces several types of wrappers that assume that items are always in fixed, known order, and that web pages have an HLRT organisation [78] . This means that there is a Head delimiter, a set of Right and Left delimiters for each attribute to be extracted, and a Tail delimiter at the end. This makes WIEN unable to handle permutations or missing attributes. The extraction rules generated by WIEN are similar to those generated by WHISK, the difference being that WIEN only uses delimiters that immediately precede and follow the data to be extracted.

SoftMealy. It generates wrappers that are non-deterministic finite automata in which the states represent the facts to be extracted and the transitions amongst them represent contextual rules that define the separators between them [66]. It allows both semantic classes and disjunctions; thus, it works well with web pages that have attribute permutations or missing attributes; however, in order to deal with attribute permutations, SoftMealy must be trained with samples that include each possible ordering of the attributes to be extracted.

STALKER. It performs hierarchical information extraction on structured and semi-structured web pages [94]. To address the problem of hierarchical information extraction, STALKER uses a formalism based on trees called

ECT that specifies the output schema for the extraction task. For each node on ECT, STALKER generates extraction rules and an iteration rule if it represents an attribute that can be repeated. The extraction process is performed in a hierarchical manner.

3.4 Wrappers maintenance

A wrapper may broke due to changes on web pages. Francisco-Revilla et al. [46] presented a classification of web changes based on the nature of the change. They distinguish four kinds of change, namely: content or semantic, presentation, structural, and behavioral. We use it to illustrate the kinds of changes that can break a wrapper:

Content or semantic changes. This refers to modifications of the page contents from the reader's point of view. For instance, `Amazon.com` changed the way it presents the authors of a book by adding the preposition "by" before the name of the first author. A wrapper, affected by this change, might extract "by Erich Gamma" instead of "Erich Gamma".

Presentation changes. They are related to the way the contents are rendered, but not to the contents themselves. This kind of changes can break a wrapper that uses tag-based extraction rules to analyse the structure of the web page.

Structural changes. This refers to the underlying connection of the web page to other web pages. Because wrappers extract information from web pages and not from the destination of links in web pages, this kind of changes do not usually affect them.

Behavioral changes. This refers to modifications to the active components of a web page, e.g., scripts, plug-ins or applets. The active components are ignored by wrappers, thus, this kind of changes do not affect them.

Wrapper maintenance consists of two tasks, namely: wrapper verification and wrapper reconstruction. The former consist of examining the information extracted by a wrapper and deciding whether it is working correctly. The latter consist of rebuilding a new wrapper when it is broken; the new wrapper can be rebuilt by hand (manual), training it again with new sample data (semi-automatically) if we are using an inductive wrapper, or by automatically constructing a set of sample labelled web pages and applying re-induction (automatically). Figure §3.3 illustrates a wrapper system that uses a verifier and that re-generates wrappers automatically if they are broken.

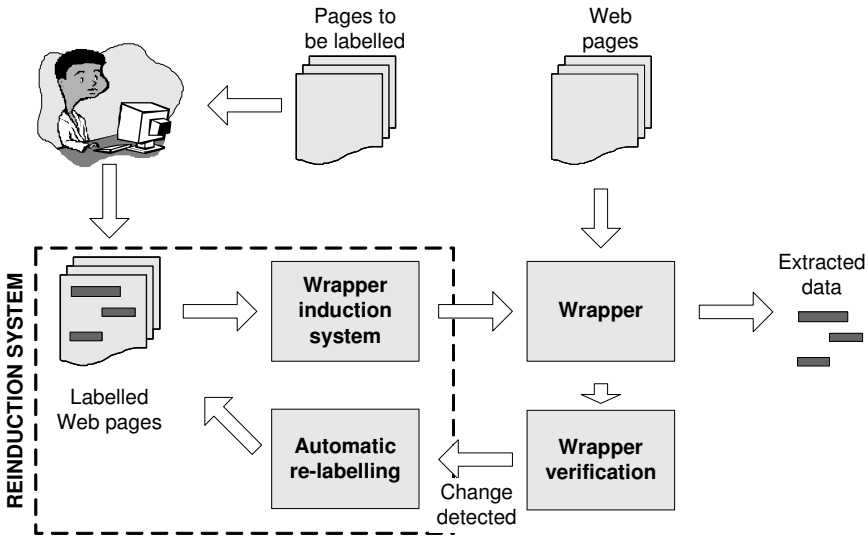


Figure 3.3: *Life cycle of the maintenance of inductive wrappers.*

To the best of our knowledge, there exist only two proposals that aim at solving the problem of wrapper maintenance, namely: RAPTURE and DataProG. The former deals with wrapper verification, and the latter deals with wrapper verification and wrapper reconstruction.

RAPTURE. It is a domain-independent verifier [77] that describes each attribute by a collection of statistical features, such as average word length, word count, number of special characters, or number of digits. It learns the parameters of normal distributions describing the feature distributions of the extracted information using positive training examples. Individual attribute probabilities are then combined to produce an overall probability that the wrapper extracted the information correctly. If this probability exceeds a threshold specified by the user, RAPTURE decides that the wrapper is correct; otherwise, it is broken.

DataProG. It is a domain-independent wrapper maintenance system [80] that verifies wrappers by applying machine learning techniques. It learns a set of patterns that describe the features of attributes from positive training examples. These features include the patterns that describe the common beginnings (or endings) of an attribute. During the verification phase, the wrapper is used to generate a new set of test examples from the same web pages used in the training phase. Then, it computes the features associated with the attributes of the test examples. If the distributions of both sets are statistically the same at some significance

level, the wrapper is judged to be extracting correctly; otherwise, it is judged to have failed. DataProG achieves the wrapper re-induction task by implementing an automatic labelling algorithm. This algorithm is a mixture of supervised and unsupervised learning algorithm that uses a set of patterns to identify examples of the data field on web pages. The patterns used are the ones learned in the verification task, and the ones obtained from other features related with the structural information of web pages.

3.5 Summary

In this chapter, we have unveiled the web as a rich source of information. We have presented syntactic wrappers as algorithms for the extraction of syntactic information from the web. Finally, we have shown that, due to the inherent web dynamism, they have a limited life period and need maintenance.

Chapter 4

Extracting knowledge from the web

The beginning of knowledge is the
discovery of something we do not understand.

Frank Herbert, 1920–1986
American science fiction novelist

Today's web is an enormous and valuable source of knowledge. This chapter classifies and presents several proposals that aim at unveiling the knowledge residing on the web. It is organised as follows: in Section §4.1, we introduce and classify the knowledge extraction proposals; then, they are summarised in Sections §4.2, §4.3, and §4.4; finally, Section §4.5 summarises the ideas in this chapter.

4.1 Introduction

Knowledge extraction systems deal with exploiting the semantic aspects of the information residing on the Web. The research work done in this area can be classified into three categories: ontology extraction, instance extraction or knowledge base extraction. The former aims at extracting models (ontologies) underlying the information residing on the Web; the second aims at extracting a set of individual instances that relates the information on the Web with the concepts specified in an external ontology; the latter extracts ontologies as well as individual instances.

Table §4.1 summarises the characteristics of the knowledge extraction systems we have studied. The first two columns indicate the kind of knowledge extraction system, and their main objective, respectively; the third column refers to the name of the proposal; the fourth column presents the type of web pages that are handled (unstructured, semi-structured or structured); the last column indicates the main methods underlying the proposal.

4.2 Common ontology extraction systems

Ontology extraction systems focus on discovering the semantic information residing on web pages. According to their goal, they are classified into ontology learning and schema learning, and a summary is presented in Table §4.1.

Ontology learning systems deal with the extraction of ontological knowledge from information sources in which this knowledge is considered to be implicit; they mainly focus on the extraction of ontologies from natural language documents. Relevant proposals in ontology learning are WebOntEx, Text-To-Onto, ASIUM, and INTHELEX, namely:

WebOntEx. It is a system to extract web ontologies semi-automatically by analysing unstructured and semi-structured web pages that are in the same application domain, and to convert the ontology into XML DTD [57]. The main module of WebOntEx is a heuristic analyser module, which uses inductive logic programming to classify the concepts into entity types, to get the relationships, attributes, superclass/subclass hierarchies, and to store this knowledge in a relational database.

Text-To-Onto. It is a framework for maintaining and extracting ontologies from unstructured web documents [86]. It uses data mining and natural

Purpose	Objective	Proposal	Domain	Methods
Ontology	Ontology learning	WebOntEx	Unstructured, semi-structured	Inductive logic programming
		Text-to-Onto	Unstructured	Natural language processing, data mining
		ASIUM	Unstructured	Conceptual and hierarchical clustering
		INTHELEX	Unstructured	Inductive logic programming
	Schema learning	Xtract	Structured	Decomposition and factorisation of expressions
		Treeminer	Semi-structured	Search on trees
		G. Cong et al.	Semi-structured	Search on trees
Instances	Question answering	Start/Omnibase	Unstructured	Natural language processing
		Squeal	Structured	Database
	Ontology population	WEB→KB	Structured, semi-structured	Machine-learning
		ArtEquAKT	Unstructured	Natural language processing
	Semantic annotation	CREAM	Unstructured	Natural language processing, data mining
		MnM	Unstructured	Natural language processing
KB	Information integration	Ontominer	Semi-structured	Hierarchical clustering

Table 4.1: *Knowledge extraction proposals.*

language processing techniques to assist the ontology engineer in importing and reusing existing ontologies, extracting ontologies from web documents, adapting them to their prime purpose, refining them, and validating the result.

ASIUM. It is a cooperative machine learning system that is able to acquire subcategorisation frames and ontologies for specific domains from syntactically parsed technical texts in natural language [39]. It is based on an unsupervised clustering method.

INTHELEX. It is defined as a fully incremental, multi-conceptual closed loop learning system for the induction of hierarchical theories from examples. This means that the learned theory is checked to be valid on any new example available, and a revision system is activated in case of failure [37].

Schema learning systems deal with the extraction of knowledge about the internal structure of documents. Schema learning for structured data [50, 99] focuses on the inference of XML DTDs, XML-Schemas, or RDF-Schemas that describe a set of tree-structured XML documents. Schema learning for semi-structured data [26, 96, 123, 124] focuses on the discovering of common subtrees in web pages. In Table §4.1, we present a couple of schema learning systems along with their main features.

4.3 Common instance extraction systems

Instance extraction systems focus on extracting individual instances of concepts and properties specified in an external ontology. They are classified into question answering, ontology population, and semantic annotation systems.

Question answering systems attempt to provide interfaces by means of which an agent can query the web as if it was a knowledge base. Common answering systems are Squeal, Omnibase and START, namely:

Squeal. It aims at viewing the web as an enormous database in which structural relationships are represented as database relationships [115]. This goal is achieved by means of the Squeal ontology, which is specified as an SQL database schema. SQL can be used to get knowledge about the structure of a web page, which is possible because Squeal relates hypertext conventions to semantic relationships.

Omnibase and START. START is a natural language question answering system and Omnibase is a virtual database that provides uniform access to web resources [74]. The authors argue that their main goal is to develop a "smart reference librarian" that knows where to find relevant knowledge, even if it is not able to answer a question directly. START uses natural machine-parseable language annotations to describe the kinds of questions that some knowledge is able to answer.

Ontology population systems attempt to feed ontologies with instances extracted from web pages. WEB→KB, and ArtEquAKT are popular ontology population systems, namely:

WEB→KB. This project aimed at developing a probabilistic, symbolic knowledge base that mirrors the contents of the web [31]. WEB→KB works on an ontology that defines the concepts and relations of interest, and a set of training data composed of web pages labelled with semantic information that represents instances of the ontology. With these inputs, and applying several machine learning algorithms, the system learns to extract information from similar pages and hyperlinks on the web.

ArtEquAKT. This project aims at extracting individual instances about artists from the web to populate an ontology automatically [2]. The individual instances and the ontology form a knowledge base that is used to generate dynamic personalised presentations tailored to the user needs (narrative bibliographies). Although the application domain is quite restricted, we mention it since it is one of the very few proposals that exists in the field of ontology population systems.

Semantic annotation systems attempt to enrich web pages with semantic annotations, which is the motto of the semantic web initiative. Some annotation tools like Yawas[†1], CritLink[†2], or Annotea[†3] aim at creating user comments on web pages to help users have a better understanding of web contents. However, we are not interested in such tools, but in others that allow to annotate web pages with ontological web languages, which are very suited for web agents. CREAM and MnM range amongst the most important, namely:

CREAM. It is an annotation workbench that allows to construct metadata using a domain ontology [58, 59]. The workbench comprises inference services, a crawler, a document management system, ontology guidance,

[†1] `http://www.fxpal.com/people/denoue/yawas`
[†2] `http://crit.org`
[†3] `http://www.w3.org/2001/Annotea`

document editors, document viewers, and a meta-ontology. The implementation of CREAM supports the semi-automatic annotation of web pages. It is based on the Amilcare information extraction system, which is an inductive wrapper able to perform single slot extraction on unstructured web pages. CREAM annotates web pages by using knowledge extraction rules that are learned from a marked-up set of annotated web pages.

MnM. It provides both automatic and semi-automated support for marking-up web pages with semantic contents [119]. It integrates a web browser, an ontology editor, and an information extraction tool into a workbench that helps users to annotate their web pages semantically.

MnM defines a process model composed of five main activities, namely: browse, markup, learn, test, and extract. In the browse activity, a specific set of knowledge components is chosen from a library of knowledge models on an ontology server. During the markup activity, the chosen set of knowledge components is selected to form the basis of an information extraction mechanism; a set of documents are manually marked up. During the learn activity, a learning algorithm is run over the marked up set of documents to learn the extraction rules. During the test activity, the information extraction mechanism is run over a set of test documents to assess its precision and recall measures. Last, during the extract activity, an information extraction mechanism is selected and run on a set of documents.

4.4 Common knowledge base extraction systems

To the best of our knowledge, OntoMiner is the only proposal that aims at extracting both ontology and individual instances from web pages [33]. It is a system that uses datamining techniques to construct specialised domain ontologies and populating them by organising and mining a set of user-supplied web sites. It is presented as an information integration system that transforms legacy HTML documents into semantic web documents and encodes domain knowledge to facilitate automated reasoning.

OntoMiner works on the URLs of the home pages of 10 to 15 domain specific taxonomy-directed web sites that characterises the domain of interest. Next, OntoMiner uses several algorithms that detect the HTML regularities to identify hierarchical relationships amongst the most important key domain concepts and turns them into hierarchical structures encoded as XML. Then, it

expands the mined concept taxonomy with sub-concepts by selectively crawling through the links corresponding to key concepts. Finally, OntoMiner extracts instances from web pages by obtaining the hierarchically partitioned instance segments and then finding the corresponding children.

4.5 Summary

In this chapter, we have unveiled the web discovering a rich source of knowledge. We have classified and presented systems that exploit the semantics aspects of the information residing on the web. These systems are of interest because, nowadays, it is not realistic to assume that the semantic web has already lived up to the hype.

Chapter 5

Web services

It would appear that we have reached
the limits of what it is possible to achieve with
computer technology, although one should be
careful with such statements, as they
tend to sound pretty silly in five years.

Johann von Neumann, 1903–1957
Computer scientist and mathematician

Our goal in this chapter is to provide the reader with an insight into the web services technology. It is organised as follows: Section §5.1 introduces this chapter; Section §5.2 presents the concept of web service and the family of standards that are current trends in their development; Section §5.3 briefly describes the semantic web services technology; finally, Section §5.4 summarises the main ideas in this chapter.

5.1 Introduction

Web services have emerged as the next generation of web-based technology for exchanging information. They support interoperable machine-to-machine interaction over the web, which enables applications to connect to each other in a platform and programming language independent manner. Web services are defined as units of work, each handling a specific functional task with universally defined interfaces. Tasks can be combined into business-oriented tasks to handle particular business transaction which propitiates the vision of web services as a technology that enables dynamic business-to-business interactions.

In the context of the semantic web, some researches are working on semantic web services, which aim at automating web services by employing semantic web technology for service description. The challenge is the serendipitous interoperability, i.e., to enable software agents to automatically locate and interact with services provided by unknown partners.

5.2 Nowadays web services

The term web service is fairly self-explanatory since it refers to having access to services on the web, but there is more to it than that since the current use of the term refers to the architecture, standards, technology and business models that make web services possible. According to one of the most common definitions [97]:

> *Web services are a new breed of web applications. They are self-contained, self-describing, modular applications that can be published, located, and invoked across the web using standard protocols and languages such as XML, WSDL, SOAP, or UDDI.*

The infrastructure required to support web services builds on three roles: service provider, service consumer and registry; and three verbs that describe the interactions between them: register, find, and bind. Figure §5.1 illustrates this idea. A service is an implementation of a service description, and a service description is the metadata that describes the service. This metadata must include sufficient information for a service consumer to have access to the service it describes, including its interface and its location. A service provider registers a service description into a service registry and a service consumer

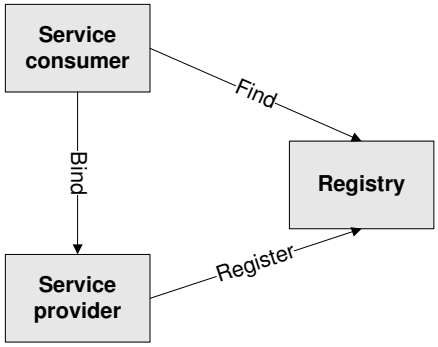


Figure 5.1: *The infrastructure needed to support web services.*

can then find the service description and its implementations by means of a registry. The interface part of the service description is known to an application developer so that the application can be implemented to communicate with services of that type.

Previous three interactions of register, find and bind in an interoperable manner are achieved using a layered web services architecture that is illustrated in Figure §5.2. The upper layers build on the capabilities provided by the lower layers. The vertical towers represent requirements that must be addressed at every level of the stack. In this section, we use this architecture to guide the presentation of the family of standards, which are current trends in the development of web services.

The network layer. It is the foundation of the architecture stack. Web services must be network-accessible so that they can be invoked by a consumer. The ubiquity characteristic of the HTTP protocol makes it the standard network protocol, although others are possible, e.g., SMTP, IIOP, or RMI.

SOAP. It is a W3C standard for XML-based messaging [112]. It is defined as a lightweight protocol for exchange of information in a decentralised, distributed environment.

WSDL. It is a W3C standard that builds on XML schema to define an XML vocabulary for defining the interfaces of web services. A WSDL description provides two pieces of information: an application-level service description, or abstract interface, and the specific protocol-dependent details that consumers must follow to access the service. This separation allows the reusing of abstract definitions amongst different providers.

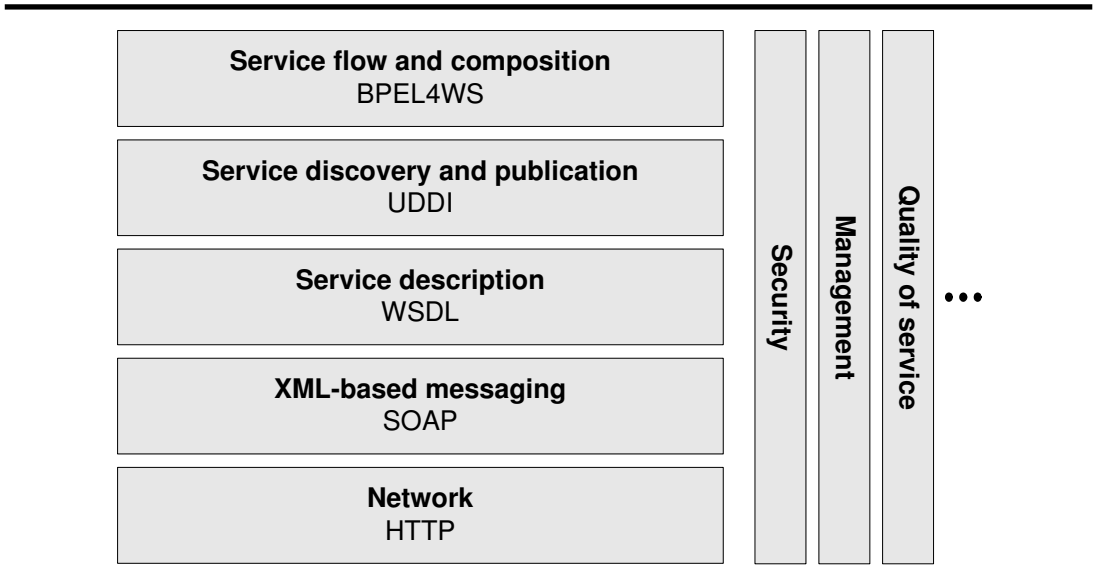


Figure 5.2: *The IBM web services architecture stack.*

UDDI. It provides a mechanism for consumers to find businesses worldwide and a mechanism for registering the products and services of a business for others to discover [5]. A key point is that the UDDI specification was developed to enable businesses to quickly, easily and dynamically find and transact business with one another, where transactions do not refer to software transactions only, but traditional brick-and-mortar transactions as well.

BPEL4WS. It builds on IBM's WSFL and Microsoft's XLANG to provide a language for the formal specification of business processes.

5.3 Semantic web services

The semantic web has motivated some researches to work on the semantic web services as an attempt to overcome the previous limitations. Semantic web services aim at mechanising web services by employing semantic web technology [4]. This refers to the ability to discover, invoke, compose and monitor their execution automatically. Automatic discovery involves the automatic location of web services that provide a particular service and that adhere to some constraints; automatic invocation involves the automatic execution of an identified web service by a software agent; automatic composition

involves the automatic selection, composition and interoperation of web services to perform some task, given a high-level description of an objective; finally, automatic execution monitoring refers to the ability to find out in which state a request is and whether any problems have appeared.

The three main proposals in this field are OWL-S, SWWS, and METEOR-S, namely:

OWL-S/DAML-S. It is an OWL-based web service ontology that supplies web service providers with a core set of markup language constructs for describing the properties and capabilities of their web services in unambiguous, computer-interpretable form [87].

SWWS. Its main goal is to establish comprehensive frameworks to support semantic web services [21]. The authors are working on providing a comprehensive web service description framework defining a web service discovery framework, and developing a scalable web service middleware.

METEOR-S. Its main goal is to add semantics to the complete web process life cycle by providing constructs for adding semantics to WSDL, UDDI, and BPEL4WS [110, 111, 120].

5.4 Summary

In this chapter, we have presented web services as a promising design approach for making computing systems more flexible and cost-effective; then, we have presented semantic web services, which aim at automating web services by employing semantic web technology for service description.

Part III

Our approach

Chapter 6

Motivation

Ability is what you're capable of doing.
Motivation determines what you do.
Attitude determines how well you do it.

Lou Holtz, 1937–
American football coach

Recently, unveiling the web has been one of the most interesting research areas in the computer science or engineering fields. However, current solutions are not practical enough since they are not able to cope with a user-friendly, changing and huge web. Our goal in this chapter is to present these problems and motivate the need for a new solution. It is organised as follows: in Section §6.1, we introduce the chapter; in Section §6.2, we present these problems in detail; in Section §6.3, we prove that none of the current solutions solve these problems at a time; in Section §6.4, we discuss our results; finally, Section §6.5 summarises the main ideas in this chapter.

6.1 Introduction

In recent years, the design of reference architectures for systems distributed on the Internet has attracted an increasing number of researchers and practitioners who have focused on platforms, languages, middlewares, or interoperability concerns. The main reason for such a great interest is that this network has experienced a rapid shift from information and entertainment to electronic commerce, which has gained importance and grown exponentially [72, 108, 118].

A major challenge for marketspace participants has become sifting through an unwieldly amount of information to find useful products and services. Fortunately, the technology has evolved and the Internet has matured to a point in which sophisticated new generation agents exist. They enable efficient, comprehensive searches on the vast web information repository, and can circumvent some problems related to slow Internet access and free up prohibitively expensive surf time by operating in the background. In order for such agents to be reliable and interoperable, they must be able to access the knowledge the web contains, which is difficult because the web is user-friendly, changing, huge and distributed.

The semantic web shall bring meaning to the web, making it possible for software agents to understand the information it contains. However, current trends seem to suggest that it is not likely to be adopted in the immediate future. Thus, the extraction of meaningful information from the web becomes a serious problem for current software agents. Some authors are working on proposals that aim at unveiling the information or knowledge residing on the web to allow software agents to retrieve and manipulate pertinent information. Unfortunately, no solution seems to be appropriate enough in the context of software engineering. The reason is that they do not address the aforementioned problems at a time, but independently.

6.2 Problems

Three main problems make it difficult to develop software agents able to exploit the information residing on the nowadays web, namely:

The web is user-friendly. Software agents need to have a common understanding of their working domains in order to communicate with each other, and perform complicated task on behalf of their users. The gap between what a software agent understands and the available user-friendly

web content may prevent software agents from being able to reason automatically about the knowledge residing on the web.

On account of this observation, it is clear that it would be desirable for a practical solution to enable agents to understand the information residing on the web. An agent that has an ontology that specifies the semantic information about the information on a web site, should have access to a set of individual instances that relates the information on the web site with the concepts specified in the ontology.

The web changes continuously. The web is dynamic, changes continuously and evolves unexpectedly, which makes it difficult for software engineers to develop software agents able to take advantage of the web as an information source. Software agents use this information to accomplish their goals, then, if the information is corrupted it can result in malfunction with unexpected effects.

Since web changes are not avoidable, it is clear that it would be desirable for a practical solution to embrace web changes and to minimise the cost of maintenance when they occur.

The web is huge and distributed. It connects many heterogeneous information sources, and thereby, a potentially immense number of sites from which an agent can extract knowledge. Dealing with the extraction of this information manually is infeasible and automatisation and distribution are necessary features to take advantage of this potential.

The previous desiderata are necessary to provide reliable support for developing web agents that need to be able to understand the information on the web at a sensible cost. By "sensible cost", we mean the there should be a balance amongst automatic reasoning, adaptability, and automatisation and distribution that minimises their development and maintenance costs.

6.3 Analysis of current solutions

Our goal in this section is to prove that none of the systems presented in Chapter §3 and Chapter §4 address the aforementioned problems at a time.

6.3.1 The semantic web

The semantic web shall simplify and improve the knowledge extraction process by changing it into querying a semantically annotated information

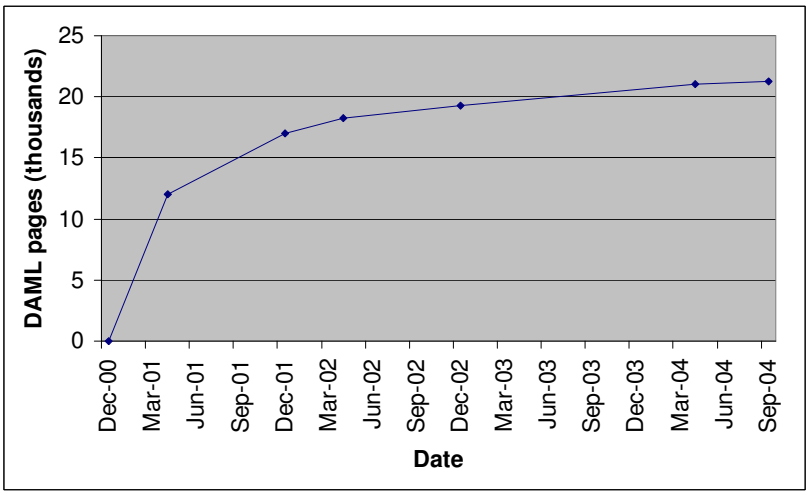


Figure 6.1: *Semantic web evolution.*

source. Nevertheless, it requires a great deal of effort to annotate current web pages with semantics. Figure §6.1 shows the semantic web evolution as of the time of writing this document. The figures suggest that there are very little annotated pages available if we compare them with the number of non-annotated pages. The DAML crawler[†1] reports 21 025 annotated web pages in September 2004, which is a negligible figure if we compare it with 550 billion, which is the estimated total number of web pages[†2]. Current trends seem to suggest that the semantic web is not likely to be widely adopted in the forthcoming years [63].

Therefore, today, it is not realistic to assume that software agents can understand the web. This argues for researchers to continue working on developing new proposals for extracting knowledge from the web, and on improving the existing ones in the meanwhile. These proposals will be useful as long as the semantic web is not in widespread use.

6.3.2 Inductive wrappers

Induction wrappers are suited to extract information from the web at a sensible cost. They automatise the process of building wrappers, and they can be combined with verifiers to embrace web changes; they can even be

[†1] `http://www.daml.org/crawler`
[†2] `http://www.cyveillance.com`

combined with re-induction systems to recover from errors without human intervention (self-healing). However, they do not associate semantics with the data they extract, this being their major drawback. Thus, they allow to use the extracted data in other applications by embedding all the knowledge about them into their functional logic.

6.3.3 Ad-hoc solutions

Other solutions are the ad-hoc applications with built-in knowledge to translate the information from today's non-semantic web into semantically-meaningful information. The problem here is due to the dynamism inherent to web sites. They are wired to properties of web pages or characteristics of the information to extract; changes on the web break them easily, which increases their maintenance costs.

6.3.4 Web knowledge extraction systems

As we mentioned in Chapter §2, neither proposals in schema learning (from ontology extraction systems) nor in question answering (from ontology population systems) are related with our work. Ontology learning systems deal with the extraction of concepts and relationships amongst them, which is still an open problem. In general their accuracy is not enough for typical applications. To increase the accuracy, they usually reduce their scope of application by focussing on a concrete domain and handling only one kind of web pages (unstructured web pages). These ideas justifies that the semantic information about data on a web page must be described by an human knowledge engineer, and this is the approach we use in our framework.

Our work is closely related to the proposals in ontology population, semantic annotation and knowledge base extraction. However, our focus is clearly different since we are concerned with the development of software applications, that is, with software engineering.

Ontology population. Feeding external ontologies with instances extracted from web pages is the proposal by WEB→KB [31] and ArtEquAKT [2].

In WEB→KB, rules learned can be used in others web sites in the same application domain, which makes this solution able to deal with web changes. It automatises the process of extracting instances from the web but it can be achieved only after a difficult training phase, which makes the cost of this

automatisation excessive. It uses the SRV wrapper to extract syntactic information, which constraints it to single slot extraction on structured and semi-structured web pages. Furthermore, the synchronisation of the knowledge on a web site with the knowledge on the knowledge base, makes it not suitable for web sites in which the information changes continually, e.g., stock values.

ArtEquAKT works on a concrete domain (biographies of artists) and handles unstructured web pages, which reduces their scope of application. Furthermore, ArtEquAKT is not able to deal with changes on the web.

Semantic annotation. CREAM [58, 59] and MnM [119] are similar systems, they provide a workbench so that users can annotate web pages with metadata. The last versions are able to semi-automatically annotate web pages by using inductive wrappers. Their major drawback is that web sites to annotate are supposed to be static. Furthermore, they focus on unstructured web pages, which reduces their scope of application significantly.

Summing up, they are solutions from the web site provider perspective, and not from the consumer of the information residing on web site. In general, it may not be appropriate to wait for the provider of a web site to decide to annotate semantically its web pages, because we are looking for solutions that work in the current non-semantic web.

Knowledge base extraction. Ontominer [33] is a solution for information integration that aims at converting traditional legacy web sites into semantic enabled web sites. It automatically extracts a taxonomy from web sites that are taxonomy-directed, and, thus, has the same drawbacks as ontology learning systems. As semantic annotation systems, it is a solution from the web site provider perspective. With respect to individual instances extraction, it also works on a static environment and it is addressed by the taxonomy obtained automatically, which precludes the reusing of existing ontologies or selecting the chunks of knowledge of interest from a web site.

6.4 Discussion

From the previous discussion, it follows that a framework able to extract knowledge, to embrace web changes, and to automatise the development of distributed knowledge extractors is very desirable from a practical point of view.

WebMeaning is our approach to extract semantically-meaningful information from today's non-semantic web. It provides engineering support for software developers to build web information agents at a sensible cost. WebMeaning achieves separation of issues in developing knowledge extractors that we call semantic wrappers. Such wrappers are composed of a syntactic wrapper to extract syntactic information from the Web, a syntactic verifier to verify the information extracted, a semantic translator to give semantics to the extracted information, and a semantic verifier to check the knowledge semantically. These elements may be freely integrated as a whole to produce semantic wrappers. WebMeaning allows to substitute them easily, which reduces the impact of changes and improves reusability.

In order to extract information from the web and to verify it, WebMeaning can use existing inductive wrappers and wrapper verifiers, respectively. This allows us to take advantage of all the work developed in these fields, to exploit their automatisation in dealing with syntactic information, and to cover a wide range of web sites. Our proposal for building semantic translator is formalised in Chapter §8 and a materialisation is given in Chapter §9; here, we present several guides that help develop domain-independent semantic translators semi-automatically. Knowledge is semantically checked using reasoners that allows to detect inconsistencies in the information source used to feed a web site. Finally, we present a proof-of-concept implementation in Chapter §10, this can be seen as a comprehensive SOA-based architecture that promotes distribution of the elements of which a semantic wrapper is composed.

6.5 Summary

Our goal in this chapter was to motivate the reason why we embarked on the development of this thesis. We have analysed the problems involved in extracting knowledge from the web, and have proved that none of them succeeds in addressing them at a time. This proves that our contribution is original and advances the state of the art a step forward.

Chapter 7

The WebMeaning framework

If you have knowledge, let others light their candles at it.

Margaret Fuller, 1810–1850
American transcendentalist author and editor

Our goal in this chapter is to present the framework we have devised to deal with the problem of developing software agents able to understand the current syntactic web. It is organised as follows: Section §7.1 presents an informal introduction by mean of a simple activity diagram; Section §7.2 introduces some preliminary concepts; Section §7.3 defines the elements of framework rigorously; finally, we summarise our contributions and publications in Section §7.4.

7.1 Introduction

As we state in the previous chapter, the primary focus of our research work is to provide engineering support so that software developers can develop agents that need a piece of knowledge residing on web pages to satisfy their goals. This is materialised as an abstract framework called WebMeaning that provides a foundation for developing semantic wrappers. Semantic wrappers are designed on the principle of separation of issues. The criterion used is to make each major step in the knowledge extraction process an activity, which is performed by a different artefact; thus, a semantic wrapper is the artefact responsible for orchestrating these artefacts.

Figure §7.1 uses the SPEM notation based on UML 2.0 to give a good intuitive understanding of the activities involved in extracting knowledge from the web; namely: information extraction, information verification, translation of information into knowledge, and semantic verification. These activities are performed by the syntactic wrapper, syntactic verifier, semantic translator, and semantic verifier artefacts, respectively. The semantic wrapper orchestrates to all them as follows: given a web page, the semantic translator invokes the syntactic wrapper to extract the syntactic information of interest from it; then, the syntactic verifier checks whether the information extracted is syntactically valid; if it is not valid, it then reports a syntactic error; on the contrary, the information is translated into instances of an ontology by the semantic translator artefact. Again, the knowledge is checked to detect semantic errors; if the knowledge is not semantically valid, the semantic verifier reports a semantic error; otherwise the knowledge is returned.

In this chapter, we present a complete formalisation of the WebMeaning framework. It is a foundation intended to be instantiated to develop a family of knowledge extraction systems. It defines the basic building blocks to create them and makes explicit where adaptations for specific functionality should be made.

7.2 Preliminaries

To define the inputs and outputs of the elements in Figure §7.1 we define four data types, namely: web page, structured information (the output format of syntactic wrappers), Abox (the output format of semantic translators), and the result of the process of knowledge extraction.

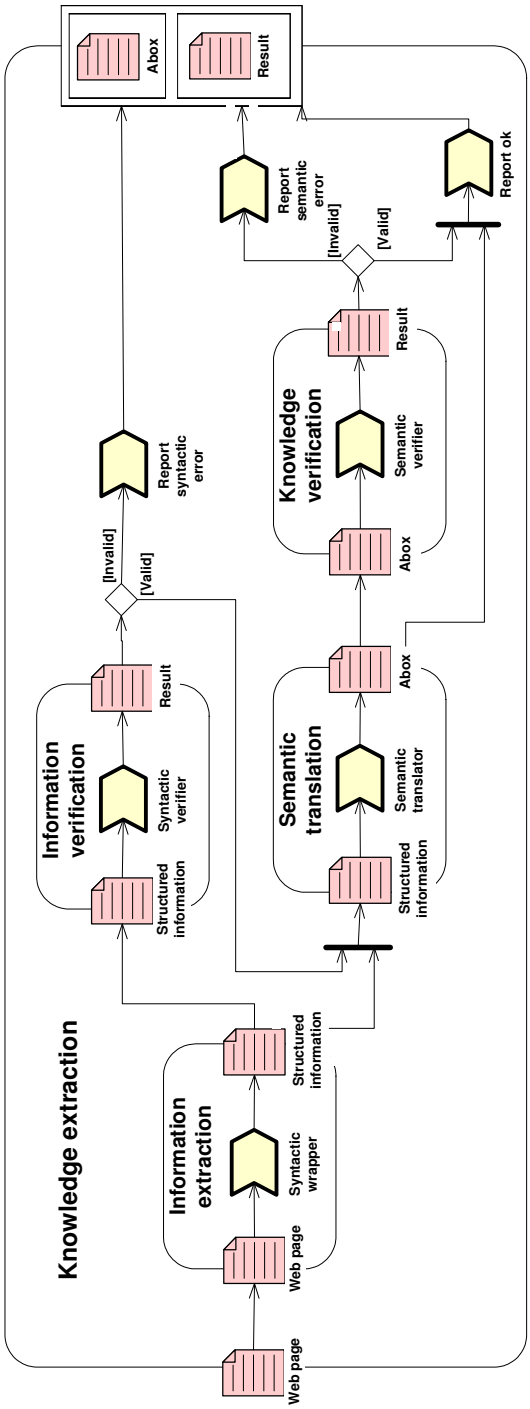


Figure 7.1: *Semantic translator workflow.*

7.2.1 Web pages

Web pages are modelled by introducing a basic type called *WebPage*. It represents the set of all existing web pages.

$[WebPage]$

Note that we not delve into the structure of web pages since, although it is well defined by W3C, it is irrelevant for us. As long as syntactic wrappers deal with them, our proposal is completely independent from actual web pages.

7.2.2 Output format of syntactic wrappers

We define the *StructuredInformation* data type to represent the output format for wrappers, c.f. Figure §7.1. It is defined as a set of *HierarchicalSlot*s, which allows us to deal with wrappers able to perform multi-slot extraction. The *HierarchicalSlot* data type is composed of a tree called t and a pair of functions: *vertexAttributes* and *vertexHLevel* that allows us to represent hierarchical information. The *Tree* data type is defined in Appendix §A, and it specifies a rooted connected acyclic graph composed of a set of vertices and a set of edges. Function *vertexHLevel* maps edges onto values of the *Label* free type, which represents the set of all labels. Each label represents a hierarchical level. Function *vertexAttributes* maps vertices onto $\mathrm{seq}\, Attribute$, which represents a piece of information provided in a hierarchical level and it allows us to set a location for each attribute, which solves the problem of representing attribute permutations. *Attribute* represents an attribute to be extracted, and it is specified as a set of literals, which allows us to deal with missing attributes, i.e., the empty set, or attributes with multiple values, i.e., a set with cardinality greater than one. Furthermore, four predicates formalise the requirements for a *StructuredInformation*: the first and the second state that functions *vertexAttributes* and *vertexHLevel* are defined for all vertices in t; the third states that all of the slots at a same hierarchical level are at the same depth in the tree and they have the same number of attributes; the latter constraints that all vertices must have one attribute at least.

$[Literal, Label]$
$StructuredInformation == \mathbb{F}\, HierarchicalSlot$
$Attribute == \mathbb{F}\, Literal$

HierarchicalSlot

$t : Tree$
$vertexAttributes : Vertex \pfun \operatorname{seq} Attribute$
$vertexHLevel : Vertex \pfun Label$

$\operatorname{dom} vertexAttributes = t.vertices$
$\operatorname{dom} vertexHLevel = t.vertices$
$\forall w_1, w_2 : Vertex \mid w_1 \neq w_2 \land vertexHLevel(w_1) = vertexHLevel(w_2) \bullet$
$\quad \#vertexAttributes(w_1) = \#vertexAttributes(w_2) \land sameDepth(t, w_1, w_2)$
$\forall w : \operatorname{dom} vertexAttributes \bullet$
$\quad \#vertexAttributes(w) > 0$

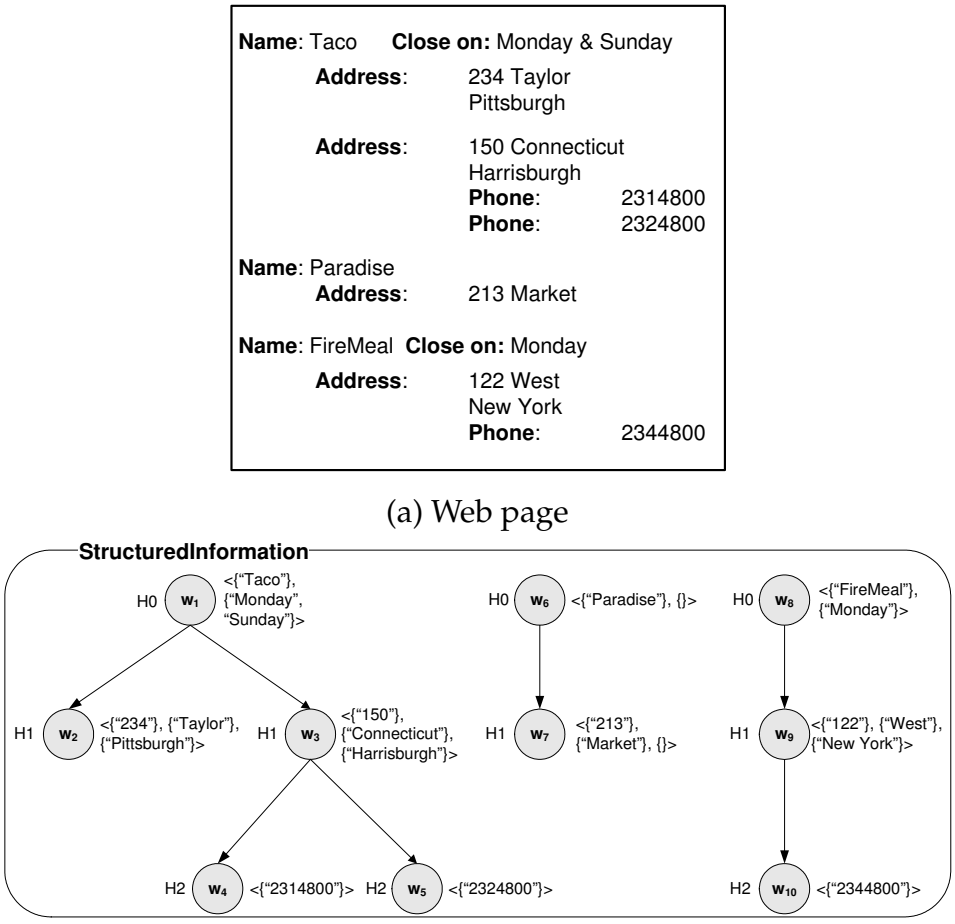
Example 7.1 *Figure §7.2(b) shows the StructuredInformation extracted by using a multi-slot wrapper from the web page in Figure §7.2(a). There are three hierarchical levels, namely: H0, which abstracts the information about a restaurant (name and closed days), H1, which abstracts information about addresses (number, street, and city), and H2, which encapsulates some information about phones (number). Each hierarchical slot represents information about one restaurant, its addresses, and phones per address; for instance, the first hierarchical slot represents a restaurant with two addresses where the second address has two phone numbers; the second hierarchical slot represents a restaurant with only one address and no phone numbers. The third hierarchical slot represents a restaurant with one address and one phone number.*

7.2.3 Assertions about individuals

Individual instances are represented in WebMeaning using the description logics formalism, i.e., they are specified as Aboxes. As mentioned in Subsection §2.2.1, an Abox is a set of axiomatic assertions. There exist two kinds of assertions: concept assertions and property assertions. A concept assertion states that an individual is a concept instance; a property assertion states that an individual is related to another individual (relationship between two concept instances), or to a literal by a given property name (value of an attribute associated with a concept).

The scheme below sketches the structure an Abox. It has a declarative part containing a set of concept names, a set of property names, a set of individual names, a set of concept assertions, and a set of property assertions. The predicate part is empty, because this part is well-formalised in the literature, so there is no need to repeat it here [13].

$[ConceptName, PropertyName, IndividualName]$



(a) Web page

(b) *StructuredInformation*

Figure 7.2: *Sample of StructuredInformation.*

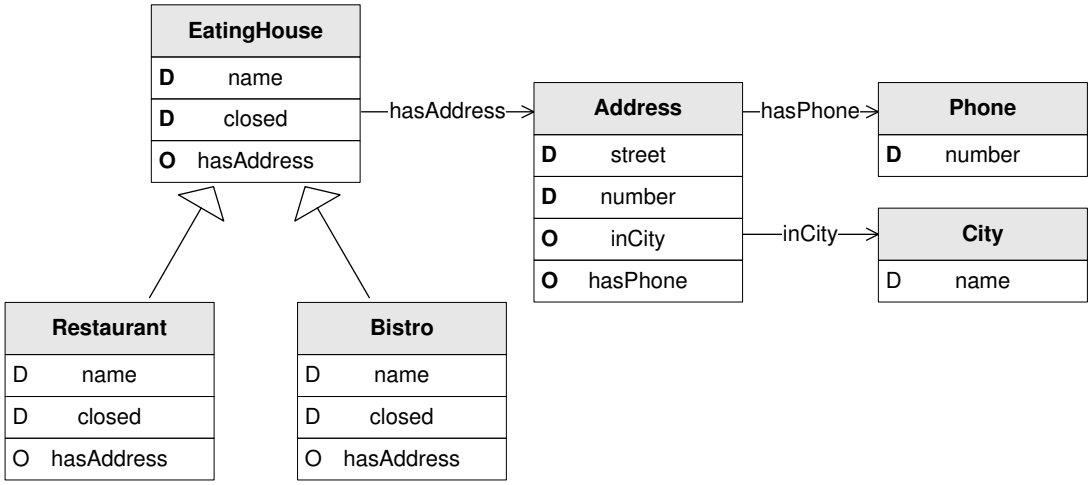


Figure 7.3: *An ontology about eating houses.*

Abox

$$
\begin{array}{l}
conceptNames : \mathbb{F}\, ConceptName \\
propertyNames : \mathbb{F}\, PropertyName \\
individualNames : \mathbb{F}\, IndividualName \\
conceptAssertions : \mathbb{F}(conceptNames \times individualNames) \\
propertyAssertions : \mathbb{F}(propertyNames \times individualNames \times (individualNames \cup Literal))
\end{array}
$$

Example 7.2 *Figure §7.3 illustrates graphically a Tbox about eating houses. Using this Tbox, we can define an Abox for our case study as follows:*

$$
\begin{array}{l}
a = (\!| \ \textbf{\textit{conceptNames}} \rightsquigarrow \{Restaurant, Address, City, Phone\}, \\
\quad \textbf{\textit{propertyNames}} \rightsquigarrow \{name, closed, hasAddress, street, number, inCity, hasPhone\}, \\
\quad \textbf{\textit{individualNames}} \rightsquigarrow \{RID_1, AID_1, AID_2, CID_1, CID_2, PID_1, PID_2\}, \\
\quad \textbf{\textit{conceptAssertions}} \rightsquigarrow \{(Restaurant, RID_1), (Address, AID_1), \\
\qquad (Address, AID_2), (City, CID_1), (City, CID_2), (Phone, PID_1), \\
\qquad (Phone, PID_2)\} \\
\quad \textbf{\textit{propertyAssertions}} \rightsquigarrow \{(name, RID_1, \text{“}Taco\text{”}), (closed, RID_1, \text{“}Monday\text{”}), \\
\qquad (closed, RID_1, \text{“}Sunday\text{”}), (hasAddress, RID_1, AID_1), (street, AID_1, \text{“}Taylor\text{”}), \\
\qquad (number, AID_1, \text{“}234\text{”}), (inCity, AID_1, CID_1), (name, CID_1, \text{“}Pittsburgh\text{”}), \\
\qquad (hasAddress, RID_1, AID_2), (street, AID_2, \text{“}Connecticut\text{”}), (number, AID_2, \text{“}150\text{”}), \\
\qquad (inCity, AID_2, CID_2), (name, CID_2, \text{“}Harrisburgh\text{”}), (hasPhone, AID_2, PID_1), \\
\qquad (number, PID_1, \text{“}2314800\text{”}), (hasPhone, AID_2, PID_2), \\
\qquad (number, PID_2, \text{“}2324800\text{”})\} \ |\!)
\end{array}
$$

This Abox uses the vocabulary in a.conceptNames $\cup$ *a.propertyNames to state assertions about one individual of concept Restaurant. For instance, concept assertion* $(Restaurant, RID_1)$ *means that* RID_1 *is an instance of the concept Restaurant; the* $(hasAddress, RID_1, AID_1)$ *property assertion means that* AID_1 *is an address of the restaurant* RID_1*, and* $(name, RID_1, \text{“Taco”})$ *asserts that the name of* RID_1 *is “Taco”.*

Last, according to the description logics formalism, a knowledge base is modelled as the cartesian product of *Tbox* and *Abox* types. Note that introducing the *Tbox* as a free type provides maximal freedom in choosing a concrete description logic.

$$[Tbox]$$
$$KnowledgeBase == Tbox \times Abox$$

7.2.4 Result of the knowledge extraction process

The result of the knowledge extraction process is an enumerated data type. We introduce a set called *Result* that is the smallest set containing the three distinct constants *OK*, *SYNTFAIL*, and *SEMFAIL*. *OK* means that there were not any failures in the knowledge extraction process. *SYNTFAIL* informs of a syntactic failure, and *SEMFAIL* informs of a semantic failure.

$$Result ::= OK \mid SYNTFAIL \mid SEMFAIL$$

7.3 Core definitions

The core of our framework is a set of definitions by means of which we define the elements presented in Figure §7.1 rigorously.

7.3.1 Syntactic wrappers

Definition 7.1 (Syntactic wrappers) *A syntactic wrapper is a function that takes a web page as input and returns a structured view of the information of interest.*

The axiomatic function below formalises a syntactic wrapper. It is modelled as a partial function because its domain is a subset of web pages that defines its scope, i.e., the web pages with which the wrapper can be used.

$$Wrapper : WebPage \nrightarrow StructuredInformation$$
$$\text{dom}\, Wrapper \neq \varnothing$$

WebMeaning is open to use any existing inductive wrapper system, handcrafted wrappers, or APIs provided by web sites to extract information. Implementing inductive wrappers generator allows to atomatise the development of wrappers. Handcrafted wrappers are rapidly developed, but maintaining them is expensive; nevertheless, this strategy can be interesting when changes on a web site are not frequent or are kept under control. Furthermore, some web sites provide APIs to access the information they offer. For instance `Google.com` for information retrieval, `Amazon.com` for books, or `Imdb.com` for movies.

When selecting the implementation of an inductive wrapper or a handcrafted wrapper, developers should ask themselves a few questions about the characteristics of the web site in which the information resides and how the information is presented, namely:

- Is it needed single-slot or multi-slot extraction?
- Are web pages structured, semi-structured or unstructured?
- Do web pages contain slots with missing attributes, multi-valuated attributes, or attribute permutations?
- Is the information on web page structured hierarchically or tabularily?

For instance, regarding the web page in Figure §7.2(a), it is clear that we need to deal with semi-structured web pages and hierarchical information extraction. These features argue for the development of a handcrafted wrapper or the use of an inductive wrapper generator. In table §3.1, we summarise the main features of most common current wrappers. Thus, a good election might be STALKER for single-slot extraction or a handcrafted wrapper for multi-slot extraction.

7.3.2 Syntactic verifiers

Definition 7.2 (Syntactic verifiers) *A syntactic verifier is a predicate that takes the information extracted by a wrapper as input and holds if the wrapper is working correctly. We call them syntactic verifiers because the decision about the wrapper's being valid is based solely on syntactic properties of the information extracted.*

$$| \mathit{SyntacticVerifier} : \mathit{StructuredInformation}$$

Thus, wrapper W is reliable if it satisfies the following formula:

$$\forall p : \mathit{WebPage} \mid p \in \operatorname{dom} W \bullet \mathit{SyntacticVerifier}(W(p))$$

WebMeaning addresses the problem of web changes by defining syntactic verifiers. It is open to use any existing generator of wrapper verifiers as RAPTURE and DataProG or handcrafted syntactic verifiers. As in inductive wrappers systems, implementing a generator allows to automatise the development of syntactic verifiers. When developing handcrafted syntactic verifiers, a result from Kushmerick's work is interesting [77]: he found that the HTML density feature (fraction of '<' and '>' characters in the extracted information) alone can correctly identify almost all of the changes in the sources he monitored. If the syntactic wrapper used is an API provided by a web site, it is not necessary to implement a syntactic verifier since predicate *SyntacticVerifier* should hold trivially for any web page in web site.

7.3.3 Semantic translators

Definition 7.3 (Semantic translators) *For each si : StructuredInformation, the goal of a semantic translator is to find a semantically equivalent a : Abox defined in terms of the vocabulary of an ontology using the available information (such as individual instances, an external ontology, or mapping information supplied by users).*

$$\begin{array}{|l} \mathit{SemanticTranslator} : \mathit{WebPage} \pfun \mathit{Abox} \\ \hline \operatorname{dom} \mathit{SemanticTranslator} \neq \varnothing \end{array}$$

Before implementing a semantic translator it is necessary to determine the most suitable language for our needs. We can decide to bet for maximum

expressiveness, or for computational issues, or to keep a balance between expressiveness and computational issues. The ideas in Chapter §2, summarised in Tables §2.1 and §2.2, can help in this selection. Depending on the language used we can take advantage of using different reasoners. For instance, if we use OWL-Lite, then we can use the RACER reasoner; but if we use OWL-Full then we must be aware that there does not exist a complete and decidable reasoner for this language.

We present further details on building semantic translators in the forthcoming chapters, namely: Chapter §8 introduces them abstractly, and Chapter §9 presents concrete algorithms to guide in their development.

7.3.4 Semantic verifiers

Definition 7.4 (Semantic verifiers) *A semantic verifier checks the satisfiability of sets of object instances of the ontologies. That is, it checks whether the instances are consistent with the constraints defined in the ontology.*

It is specified as a predicate that checks the knowledge base satisfiability using a complete and decidable reasoner.

$$| \mathit{SemanticVerifier} : \mathit{KnowledgeBase}$$

Note that semantic verifiers detect inconsistencies in the information source used to feed a web site, whereas syntactic verifiers detect changes in the layout of a web site that invalidate the extraction process. The solution to semantic errors is not to rebuild the syntactic wrapper since it works well, but to wait for the information to be corrected or to look for another site that offers the same information.

As mentioned previously, if there are not reasoners for the ontological language used, then the knowledge extracted cannot be checked, i.e., predicate *SyntacticVerifier* would hold trivially for any Abox. In table §2.2, we summarise the main features of most common current reasoners; thus, it can help in the selection of a suitable reasoner.

7.3.5 Bringing it all together

Definition 7.5 (Semantic wrappers) *A semantic wrapper is a function that takes a web page as input, and returns a set of instances of concepts defined in an ontology that represents the information of interest.*

Below, we specify a semantic wrapper as an axiomatic function. It rigorously presents the collaboration amongst concrete elements, namely: a wrapper (W), a syntactic verifier (V), a semantic translator (ST) and a semantic verifier (SV). The collaboration was informally introduced in Section §7.1. The *NULLABOX* represents an empty Abox.

$$NULLABOX == \langle\!| \, conceptNames = \varnothing, propertyNames = \varnothing, individualNames = \varnothing, \\ conceptAssertions = \varnothing, propertyAssertions = \varnothing \, |\!\rangle$$

$$SemanticWrapper : WebPage \pfun Abox \times Status$$

$$\begin{aligned}
&\forall\, p : WebPage;\ a : Abox;\ si : StructuredInformation \mid p \in \operatorname{dom} Wrapper \bullet \\
&\quad (SemanticWrapper(p) = (a, OK) \land \\
&\qquad si = W(p) \land V(si) \land a = ST(si) \land SV(a)) \\
&\quad \lor \\
&\quad (SemanticWrapper(p) = (NULLABOX, SYNTFAIL) \land \\
&\qquad si = W(p) \land \neg\, V(si)) \\
&\quad \lor \\
&\quad (SemanticWrapper(p) = (NULLABOX, SEMFAIL) \land \\
&\qquad si = W(p) \land V(si) \land a = ST(si) \land \neg\, SV(a))
\end{aligned}$$

We present further details on building semantic wrappers in Chapter §10, which presents a comprehensive architecture that provides support for developing semantic wrappers.

7.4 Summary

In this chapter, we have presented a formalisation of an abstract framework that defines a family of knowledge extraction systems. Chapter §10 complements this chapter by providing a proof-of-concept implementation that maps the framework elements into software elements. The main results were published at the 2003 IEEE/WIC International Conference on Web Intelligence (WI'03) [11].

Chapter 8

Semantic translation

Everywhere one seeks to produce meaning, to make the world signify, to render it visible. We are not, however, in danger of lacking meaning; quite the contrary, we are gorged with meaning and it is killing us.

Jean Baudrillard, 1929–
French semiologist

In this chapter, we present a proposal for semantic translation. It is organised as follows: in Section §8.1, we introduce the main ideas; Section §8.2 defines the problem of semantic translation and overviews our solution; Section §8.3 introduces a notation to represent individuals instances; Section §8.4 specifies formally semantic descriptions, a notation to represent the semantics of the information extracted from the web; Section §8.5 defines how semantic descriptions are built and Section §8.6 how they are related with the information extracted from the web; Section §8.7 formalises our solution to semantic translation; finally, Section §8.8 summarises the ideas in this chapter.

8.1 Introduction

In this chapter, we present our solution for semantic translation from an abstract point of view. It involves the unravelling of the semantics of the information to be extracted and the relation of previous unravelled semantics with pieces of extracted data. The results from these tasks allow us to set up a domain independent algorithm, to which we refer to as semantic translator, that automatically gives semantics to the information extracted by a wrapper. Thus, the abstractness with which we illustrate our proposal provides us with a maximal degree of implementation freedom, as we illustrate in Chapter §9. There we report on an efficient implementation, but others can be integrated smoothly.

8.2 Problem definition

To pursuit the problem of semantic translation, we make some assumptions about the attributes in a hierarchical slot, the relationships amongst hierarchical levels, and the way in which hierarchical slots are translated, namely:

Assumption 1. We assume that each attribute in a hierarchical slot is the value of one property associated with a concept. That is, the value of one property cannot be computed from other attributes. For instance, if the ontology in Figure §7.3 had only one property to refer to the number and street of concept *Address*, then, the wrapper should extract two attributes for hierarchical level $H1$, i.e., $\langle\{\text{“234 Taylor”}, \text{“Pittsburgh”}\}\rangle$ instead of $\langle\{\text{“234”}, \text{“Taylor”}, \text{“Pittsburgh”}\}\rangle$. Overcoming this assumption will require preprocessing the *StructuredInformation* extracted by the wrapper, which is not difficult, but irrelevant for this proposal.

Assumption 2. Each edge between two vertices in a hierarchical slot represents a property with multiple values that relates two different concepts. This is not a shortcoming since missing properties are represented by means of an empty set, and single-value properties are represented by means of a singleton.

Assumption 3. The information to be extracted must deal with only one concept, which cannot be further subclassified or specialised. For instance, all the information extracted from web site in Figure §7.2(a), refers to restaurants or bistros, but never to restaurants and bistros at the same time.

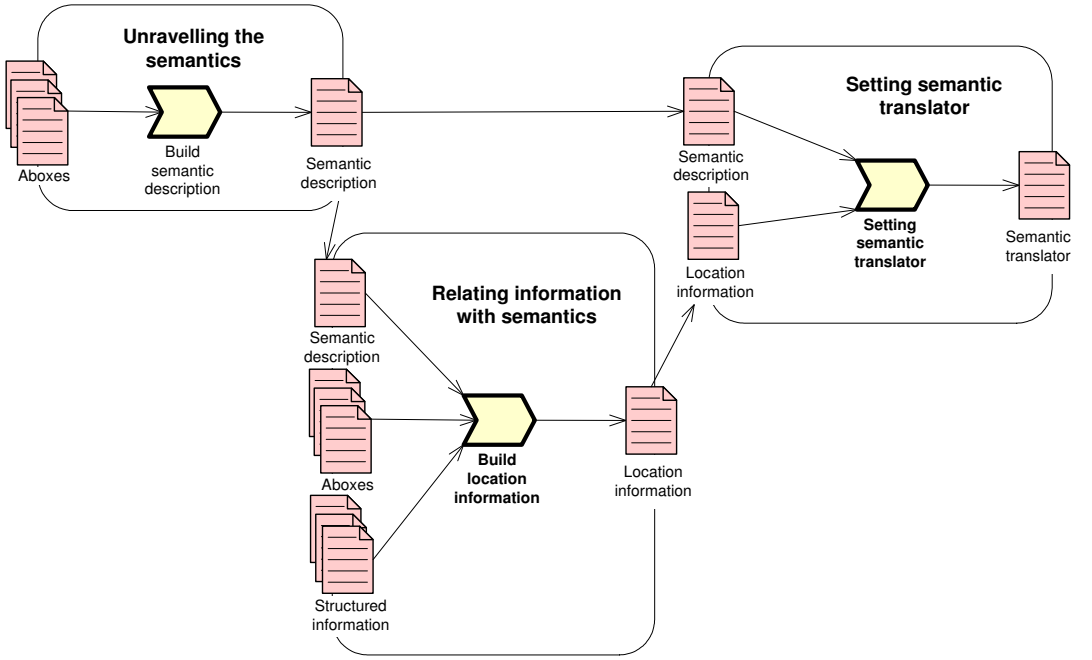


Figure 8.1: *Activities to build a semantic translator.*

Given the definition of semantic translator in Chapter §7 and the previous assumptions, our solution to semantic translation involves three tasks, which are sketched in the activity diagram in Figure §8.1, namely:

Unravelling the semantics of the information to be extracted. We use a set of examples of assertions of individuals to get a description of the semantics involved in defining these individuals.

Relating the information extracted with semantic information. We use a set of pairs of structured information and Aboxes to infer the location information that allows to relate the semantic description obtained in previous task with the output format of the wrapper. The location provides information about how properties in a semantic description are related with attributes in a hierarchical slot.

Setting up a domain independent semantic translator. The semantic translator is provided with the global semantic description and the location information obtained in the previous tasks to give semantics to the information extracted by the wrapper automatically.

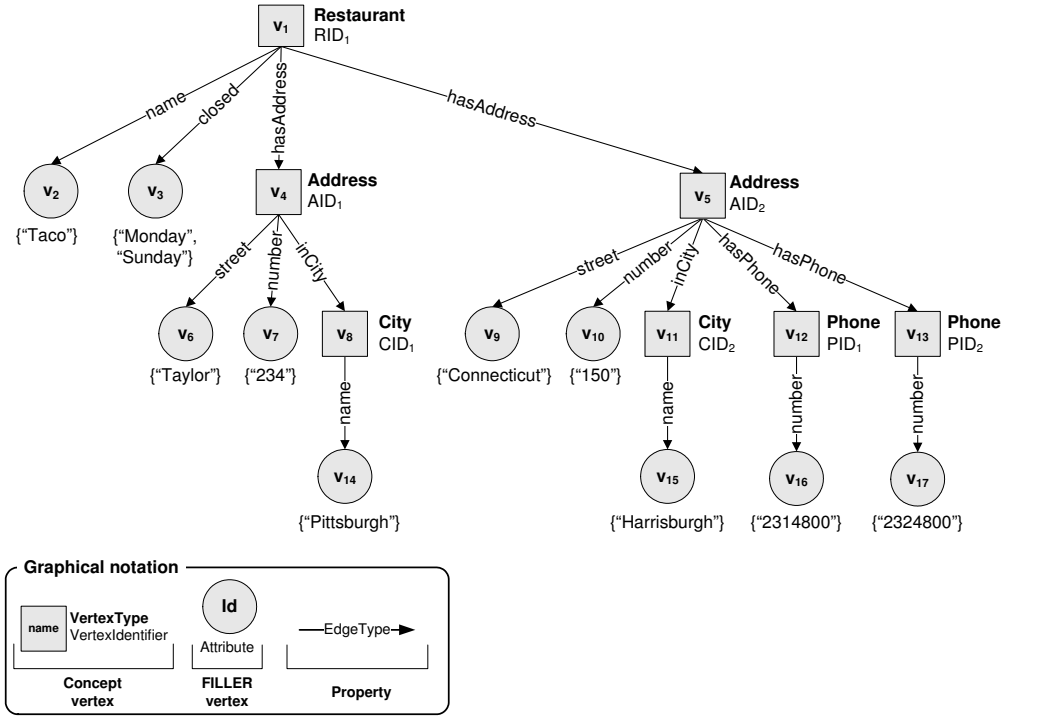


Figure 8.2: *An individual tree.*

8.3 A representation for individuals

The *Abox* specification in Chapter §7 is an abstract specification that defines assertional knowledge accurately. Now, we present a concrete specification of assertions about one individual in an Abox based on trees (the equivalence between these specifications is proved in Appendix §B).

Definition 8.1 (Individual trees) *An individual tree is a representation of an individual in some Abox as a tree. There are two types of vertices: vertices that represent concepts assertions and vertices that represent attributes. Edges represent properties and they establish relationships between two concept instances vertices, or between a concept instance vertex and a vertex that represents literal values.*

Example 8.1 *Figure §8.2 illustrates the individual referred to as RID_1 in the Abox in Example §7.2.*

For the sake of readability, we define a new type that is widely used in this chapter. The *LabelledTree* data type describes a tree in which vertices are

labelled with concept names and edges are labelled with property names. We use the special concept name *FILLER* to represent literal values. The assertions in the predicate part of the scheme constrain that filler vertices do not have any children and leaves are labelled with *FILLER*. Predicate *isLeaf* is specified in Appendix §A.

LabelledTree

$t : Tree$
$vertexType : Vertex \pfun ConceptName$
$edgeType : Edge \pfun PropertyName$

$\mathrm{dom}\, vertexType = t.vertices$
$\mathrm{dom}\, edgeType = t.edges$
$\forall (v_1, v_2) : t.edges \bullet vertexType(v_1) \neq FILLER$
$\forall v : t.vertices \mid isLeaf(t, v) \bullet vertexType(v) = FILLER$

Below, we define the *IndividualTree* scheme, which includes the variable *lt* whose type is a *LabelledTree* and two new functions that add two new labels to vertices: function *vertexIndividualName* maps vertices onto the individuals names defined in the Abox and function *vertexAttribute* maps vertices onto *Attribute*. The predicates constraint the domain of *vertexIndividualName* and *vertexAttribute* to concept vertices and filler vertices, respectively.

IndividualTree

$lt : LabelledTree$
$vertexIndividualName : Vertex \pfun IndividualName$
$vertexAttribute : Vertex \pfun Attribute$

$\mathrm{dom}\, vertexIndividualName = \{v : lt.t.vertices \mid lt.vertexType(v) \neq FILLER \bullet v\}$
$\mathrm{dom}\, vertexAttribute = \{v : lt.t.vertices \mid lt.vertexType(v) = FILLER \bullet v\}$

8.4 Semantic descriptions

Semantic descriptions allow to represent the semantics of the information extracted from the web as a tree. Thus, they define the semantics involved in the individual trees that represent the individuals instances offered by a web site.

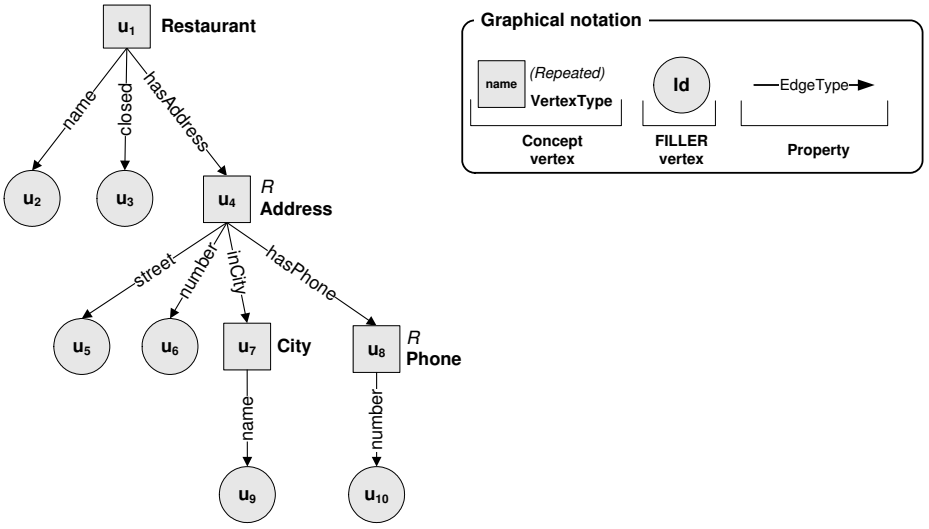


Figure 8.3: *A semantic description.*

Definition 8.2 (Semantic descriptions) *A semantic description is a simple notation to represent the semantics of the information extracted by a wrapper as a tree. It defines concepts that represent entities in a domain, properties describing relationships amongst concepts or attributes, and cardinality constraints on properties.*

Example 8.2 *Figure §8.3 shows a semantic description for our case study. It is rendered to English as follows: an instance of the Restaurant concept has three properties, namely: name, closed, and hasAddress. The first two properties are attributes. Property hasAddress relates two concepts, a Restaurant with one or more Address. Concept Address is defined by properties street, number, inCity, and Phone. inCity relates the Address concept with concept City (an address optionally is located in a city). The City concept has a Literal as name. Finally, a Phone has a Literal as number. Note that literals are always associated with filler vertices.*

The following scheme specifies a semantic description. It is composed of a labelled tree and a set of vertices (*repeated*) that defines cardinality constraints on properties. The predicates state that only concept vertices can be in the *repeated* set and that there are not any collapsable vertices on labelled trees. The concept of collapsable vertices is formalised in Section §8.5. Intuitively, the predicate constraints that there is not more than one vertex representing the same piece of semantics.

SemanticDescription

$lt : LabelledTree$
$repeated : \mathbb{F}\ vertices$

$\forall\, u : lt.t.vertices \mid lt.vertexType(u) = FILLER \bullet u \notin repeated$
$\forall\, su : \mathbb{F}\, lt.t.vertices \bullet \neg\ collapsableVertices(lt, su)$

8.4.1 Cardinality constraints on properties

Figure §8.4 illustrates graphically the meaning of cardinality constraints on edges of a semantic description. Edges are classified according to the type of vertices and whether they are in the *repeated* set. The classification is as follows:

i. An edge P between two concept vertices C_x and C_y in which the vertex associated to C_y is in the *repeated* set of the semantic description states that P represents a multiple relation ($[0\,..\,n]$, $n \geq 0$), in other words, P relates one instance of C_x with zero or more instances of C_y.

ii. An edge P between two concept vertices C_x and C_y that are not in the *repeated* set of the semantic description states that P is an optional relation ($[0\,..\,1]$), in other words, P optionally relates one instance of C_x with one instance of C_y.

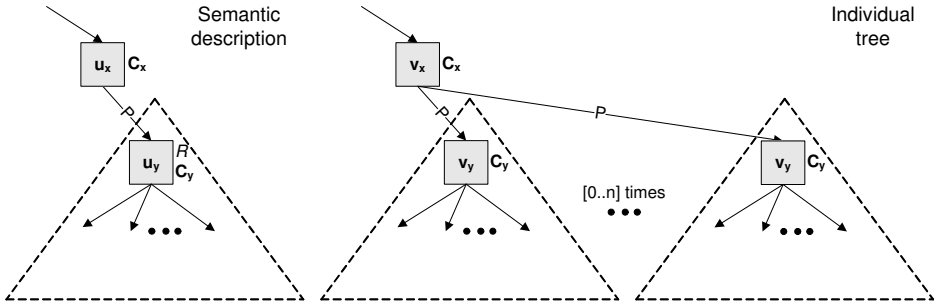
iii. An edge P between a concept vertex C_x and a filler vertex states that P is a multiple relation ($[0\,..\,n]$), in other words, P relates one instance of C_x with zero or more *FILLER*s.

8.4.2 Semantics of a semantic description

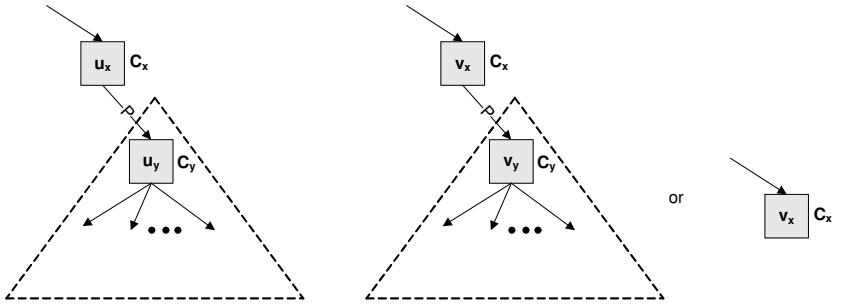
The formal semantics of a semantic description is specified using first-order interpretations. Let $\mathcal{L}$ be a logical language; a semantic description is interpreted as a tuple (P, A), where P is a subset of the vocabulary of predicate symbols of $\mathcal{L}$ and A is a subset of well-formed formulae in $\mathcal{L}$. Thus, a semantic description is interpreted as a subset of $\mathcal{L}$ in which concepts and properties are specified by unary and binary predicates, respectively.

Given $sd : SemanticDescription$, then, the translation is as follows:

- Each concept name in the semantic description is interpreted as a unary predicate symbol. Furthermore, *Literal* is considered a concept name that represents the set of all possible strings.



(a) Repetition.



(b) Optionality.

(c) Attribution.

Figure 8.4: *Different types of edges in a semantic description.*

- Each property name is a binary predicate symbol.
- Each property name adds an axiom that constrains the domain and range of properties:

  $$\forall x \bullet \exists y \bullet P(x,y) \Rightarrow \\ (C_{x_1}(x) \bigwedge (C_{y_1}(x)) \bigvee (C_{x_2}(x) \bigwedge (C_{y_2}(x)) \bigvee \ldots \bigvee (C_{x_n}(x) \bigwedge (C_{y_n}(x))$$

  where P is the property name, and the pairs (C_{x_i}, C_{y_i}) are elements of the set $\{(v_1, v_2) : sd.lt.t.edges \mid sd.lt.edgeType(v_1, v_2) = P \bullet (sd.lt.vertexType(v_1), sd.lt.vertexType(v_2))\}$.
- Each edge (v_1, v_2) in the semantic description of type (ii) adds an axiom that states that the cardinality of the corresponding property is zero or one, namely:

  $$\forall x, y, z \bullet P(x,y) \bigwedge P(x,z) \bigwedge C_x(x) \bigwedge C_y(y) \bigwedge C_z(z) \Rightarrow y = z$$

  where

  $$P == sd.lt.edgeType(v_1, v_2), C_x == sd.lt.vertexType(v_1), \text{ and} \\ C_y == sd.lt.vertexType(v_2)$$
- Note that, edges of the type (i) and (iii) do not provide any information about cardinality on properties, therefore there can exist more than one property assertion for a same concept.

Note that, an interpretation of concepts and properties defined by previous axioms over the domain of the extracted information is a set of ground predicates.

Example 8.3 *In our study case, the interpretation is as follows:*

$P = \{$***[From concept names]***
$Restaurant, Address, Phone, City, Literal$
[From property names]
$name, closed, hasAddress, street, number, inCity, hasPhone\}$
$A = \{$***[From property names]***
$\forall x \bullet \exists y \bullet name(x,y) \Rightarrow (Restaurant(x) \bigwedge Literal(y)) \bigvee (City(x) \bigwedge Literal(y))$
$\forall x \bullet \exists y \bullet closed(x,y) \Rightarrow Restaurant(x) \bigwedge Literal(y)$
$\forall x \bullet \exists y \bullet hasAddress(x,y) \Rightarrow Restaurant(x) \bigwedge Address(y)$
$\forall x \bullet \exists y \bullet street(x,y) \Rightarrow Restaurant(x) \bigwedge Literal(y)$
$\forall x \bullet \exists y \bullet number(x,y) \Rightarrow (Address(x) \bigwedge Literal(y)) \bigvee (Phone(x) \bigwedge Literal(y))$
$\forall x \bullet \exists y \bullet inCity(x,y) \Rightarrow Address(x) \bigwedge City(y)$
$\forall x \bullet \exists y \bullet hasPhone(x,y) \Rightarrow Address(x) \bigwedge Phone(y)$
[From edge (u_4, u_7)***]***
$\forall x, y, z \bullet inCity(x,y) \bigwedge inCity(x,z) \bigwedge Address(x) \bigwedge City(y) \bigwedge City(z) \Rightarrow y = z\}$

8.5 Building semantic descriptions

At the beginning of the previous section, we present the relation between a semantic description and an individual tree informally. Now, we formalise it. Semantic descriptions are built from individual trees, which are collapsed to infer the semantic information behind individuals. Two concepts allow us to formalise and build semantic descriptions, namely: collapsable vertices and collapsable paths.

8.5.1 Collapsable vertices

Definition 8.3 (Collapsable vertices) *A set of vertices in a labelled tree are collapsable if they all have a common parent, the edges that relate the common parent with the set of vertices are labelled with the same property name, and there is not a vertex that does not belong to the set of vertices that fulfill the two previous constraints.*

Predicate *collapsableVertices* formalises this definition. It holds if a set of vertices are collapsable in a labelled tree. It is specified as follows:

$$
\begin{array}{|l}
collapsableVertices : LabelledTree \times \mathbb{F}\, Vertex \\
\hline
\forall\, lt : LabelledTree;\ sv : \mathbb{F}\, Vertex \bullet collapsableVertices(lt, sv) \Leftrightarrow \\
\quad \exists\, v : Vertex;\ pn : PropertyName \mid sv \subseteq children(lt.t, v) \bullet \\
\qquad (\forall\, v_x : sv \bullet lt.edgeType(v, v_x) = pn) \bigwedge \\
\qquad (\forall\, v_x : children(lt.t, v) \setminus sv \bullet lt.edgeType(v, v_x) \neq pn)
\end{array}
$$

The labelled tree of an individual tree may have collapsable vertices; however, a semantic description has not any collapsable vertices. For instance, the sets of vertices $\{v_4, v_5\}$ and $\{v_{12}, v_{13}\}$ in Figure §8.2 are collapsable; this means that properties *hasAddress* and *hasPhone* have multiple cardinality. The same information is provided in the semantic description in Figure §8.3 by indicating that vertices u_4, and u_8 are in the *repeated* set of the semantic description.

8.5.2 Collapsable paths

We define a path in a labelled tree as a sequence of vertices in which every vertex is connected in that tree to the succeeding vertex in the sequence. We only consider paths that connect the root vertex to leaf vertices, thus, the root

vertex belongs to every path in a tree. Every path has an associated pattern, which is an alternated sequence of vertex types and edges types such that vertices are replaced by their vertex type, and between two vertex we add the name of property that connects them. For instance, path $\langle v_1, v_4, v_5 \rangle$ from the *labelledTree* in Figure §8.2 is associated to the pattern $\langle Restaurant, hasAddress, Address, street, FILLER \rangle$. Formally, paths, and patterns are defined as data types. In addition, function *pattern* takes a labelled tree and a path that belongs to that tree as input and outputs the pattern associated to the path.

$$Path == \text{seq}\, Vertex$$
$$PathPattern == \text{seq}(ConceptName \cup PropertyName)$$

$$pattern : LabelledTree \times Path \nrightarrow PathPattern$$

$$\begin{aligned}
&\forall\, lt : LabelledTree;\ p : Path;\ q : PathPattern \mid p \in paths(lt.t) \bullet pattern(lt, p) = q \Leftrightarrow \\
&\quad \#q = 2 * \#p - 1 \bigwedge \\
&\quad \forall\, i : 1 \,..\, \#p \bullet q(2 * i - 1) = lt.vertexType(p(i)) \bigwedge \\
&\quad \forall\, i : 1 \,..\, \#p - 1 \bullet q(2 * i) = lt.edgeType(p(i), p(i+1)))
\end{aligned}$$

Different paths can fit into the same path pattern. We define a binary relation of equality that relates paths based on the pattern they follow in a labelled tree. This relation is specified as follows:

$$\_ \sim \_ : (LabelledTree \times Path) \leftrightarrow (LabelledTree \times Path)$$

$$\begin{aligned}
&\forall\, lt_1, lt_2 : LabelledTree;\ p_1, p_2 : Path;\ (lt_1, p_1) \sim (lt_2, p_2) \Leftrightarrow \\
&\quad pattern(lt_1, p_1) = pattern(lt_2, p_2)
\end{aligned}$$

Lemma 8.1 *The binary relation of equality is an equivalence relation on Path.*

Proof To prove this lemma we need to show that $\sim$ is reflexive, symmetric and transitive. These properties are easily satisfied, because $\sim$ is a restatement of the equality relation in the language of sets. The proof is as follows:

Reflexivity. For any $(lt, p) : LabelledTree \times Path$, it follows that $pattern(lt, p) = pattern(lt, p)$; therefore $(lt, p) \sim (lt, p)$.

Symmetry. Let $(lt_1, p_1), (lt_2, p_2) : \mathit{LabelledTree} \times \mathit{Path}$; it follows that

$$
\begin{array}{ll}
& (lt_1, p_1) \sim (lt_2, p_2) \\
\Leftrightarrow & \qquad \text{[by definition of } \sim\text{]} \\
& \mathit{pattern}(lt_1, p_1) = \mathit{pattern}(lt_2, p_2) \\
\Leftrightarrow & \qquad \text{[symmetry of equality in the language of sets]} \\
& \mathit{pattern}(lt_2, p_2) = \mathit{pattern}(lt_1, p_1) \\
\Leftrightarrow & \qquad \text{[by definition of } \sim\text{]} \\
& (lt_2, p_2) \sim (lt_1, p_1)
\end{array}
$$

Transitivity. Let $(lt_1, p_1), (lt_2, p_2), (lt_3, p_3) : \mathit{LabelledTree} \times \mathit{Path}$; it follows that

$$
\begin{array}{ll}
& (lt_1, p_1) \sim (lt_2, p_2) \bigwedge (lt_2, p_2) \sim (lt_3, p_3) \\
\Leftrightarrow & \qquad \text{[by definition of } \sim\text{]} \\
& \mathit{pattern}(lt_1, p_1) = \mathit{pattern}(lt_2, p_2) \bigwedge \mathit{pattern}(lt_2, p_2) = \mathit{pattern}(lt_3, p_3) \\
\Leftrightarrow & \qquad \text{[transitivity of equality in the language of sets]} \\
& \mathit{pattern}(lt_1, p_1) = \mathit{pattern}(lt_3, p_3) \\
\Leftrightarrow & \qquad \text{[by definition of } \sim\text{]} \\
& (lt_1, p_1) \sim (lt_3, p_3)
\end{array}
$$

This concludes the proof. □

Definition 8.4 (Collapsable paths) *A set of paths is collapsable if all of its paths are related by the ∼ relation.*

The equality relation allows to define a partition of a set of paths into equivalence classes, i.e., all of the elements that are equivalent to each other are put into the same class. Each equivalent class corresponds to a set of collapsable paths. Then, collapsable paths in a labelled tree are defined by the quotient set of paths in a labelled tree by the ∼ relation.

$$
\begin{array}{|l}
\_ /\sim : \mathit{LabelledTree} \rightarrow \mathbb{F}\,\mathbb{F}\,\mathit{Path} \\
\hline
\forall\, lt : \mathit{LabelledTree};\ ssp : \mathbb{F}\,\mathbb{F}\,\mathit{Path};\ sp : \mathbb{F}\,\mathit{Path} \mid sp = \mathit{paths}(lt) \bullet lt /\sim = ssp \Leftrightarrow \\
\quad (sp = \bigcup ssp) \bigwedge \\
\quad (\forall\, p : sp \bullet \nexists pp, pq : ssp \mid pp \neq pq \bullet p \in pp \bigwedge p \in pq) \bigwedge \\
\quad (\forall\, pp : ssp \bullet (\forall\, p, q : pp \bullet (lt, p) \sim (lt, q)) \bigwedge (\nexists pq : ssp, q : pq \mid pp \neq pq \bullet \\
\qquad (lt, p) \sim (lt, q))
\end{array}
$$

Example 8.4 *Figure §8.5 illustrates the partition of a labelled tree into collapsable paths. The labelled tree comes from the individual tree in Figure §8.2.*

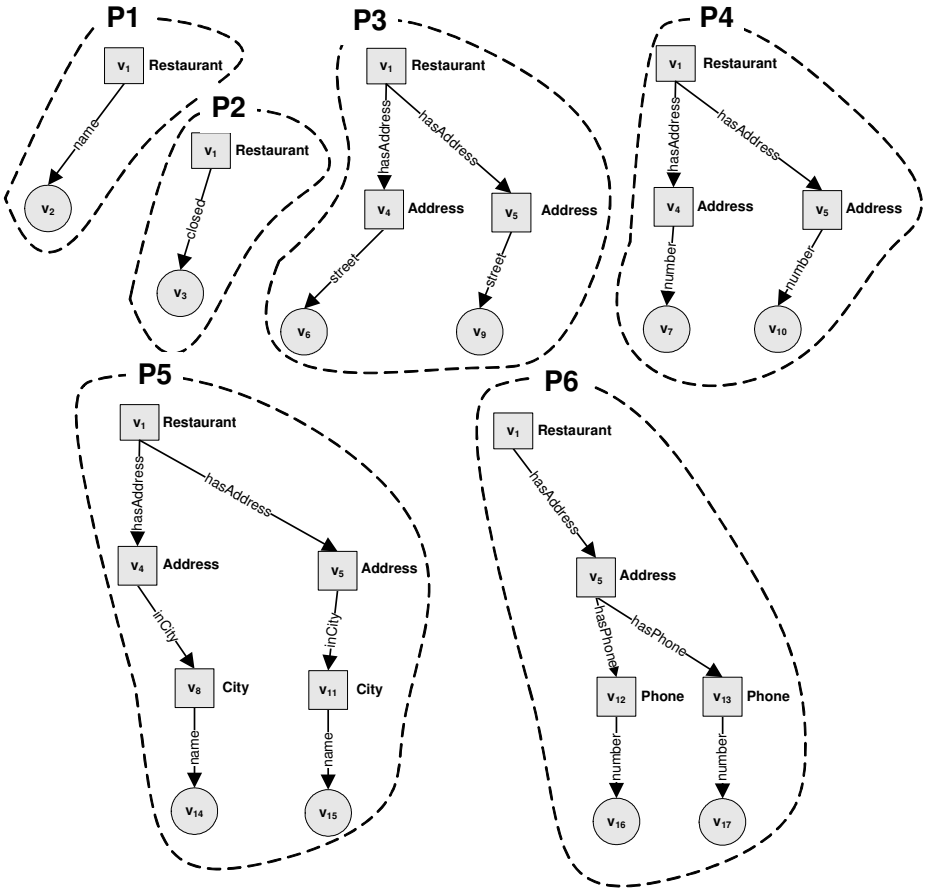


Figure 8.5: *Partitioning a labelled tree into collapsable paths.*

The labelled tree of an individual tree may have collapsable paths; however, the labelled tree in a semantic description has not any collapsable paths. For instance, the paths $\langle v_1, v_4, v_6\rangle$ and $\langle v_1, v_5, v_9\rangle$ in Figure §8.2 are collapsable because they both follow the pattern: $\langle$*Restaurant*, *hasAddress*, *Address*, *number*, *FILLER*$\rangle$. In the semantic description in Figure §8.3, they are collapsed as path $\langle u_1, u_4, u_5\rangle$.

Lemma 8.2 *The partition of a labelled tree into collapsable paths is unique.*

Proof The proof is straightforward because collapsable paths are equivalent and they are defined as a quotient set of the $\sim$ relation. Note that, the partition of a set induced by an equivalent relation is unique [25]. □

Theorem 8.1 *If a labelled tree has not any collapsable vertices, then it has not any collapsable paths.*

Proof Suppose there is it : *IndividualTree* with collapsable paths, then there exist at least two distinct paths p_1, and p_2 in $it.lt$, such that $(lt, p_1) \sim (lt, p_2)$. According to the definition of collapsable vertices, it does not exists a vertex v in p_1, and p_2 such that $p_1(i) = v \land p_2(i) = v \land p_1(i+1) \neq p_2(i+1)$, then $p_1 = p_2$. It contradicts our initial assumption that *it* has not any collapsable vertices, but has collapsable paths. So we can conclude that the original proposition must be true. □

Note that the necessary part is not true in general. For instance, if we remove vertices v_7, v_8, v_{14}, and v_9 from the individual tree in Figure §8.2, then it has not collapsable paths, but vertices v_4 and v_5 are still collapsable.

8.5.3 Collapsing individual trees

Function *buildSD* takes an individual tree as input and outputs a semantic description. It collapses paths by defining a total bijection that maps the set of paths that belongs to the partition of the labelled tree of the individual tree onto paths of the semantic description. Collapsable vertices are detected and included in the *repeated* set of the semantic description.

$$
\begin{array}{l}
buildSD : IndividualTree \rightarrow SemanticDescription \\
\hline
\forall\, it : IndividualTree;\ sd : SemanticDescription \bullet buildSD(it) = sd \Leftrightarrow \\
\quad \exists f : it.lt\,/\!\sim \twoheadrightarrow paths(sd.lt.t) \bullet \\
\quad\quad \forall\, pp : \mathrm{dom} f;\ p : pp \bullet (it.lt, p) \sim (sd.lt, f(pp)) \land \\
\quad\quad \forall\, pp : \mathrm{dom} f;\ p, q : pp;\ n : 2\,..\,\#p \mid p(i) \neq q(i) \land p(i-1) = q(i-1) \bullet \\
\quad\quad\quad f(pp)(i) \in sd.repeated
\end{array}
$$

Example 8.5 *Figure §8.6 illustrates how buildSD works. It shows the collapsable paths, the paths onto which they are mapped in the semantic description, and the repeated vertices.*

Function *buildSD* allows to build a semantic description from one individual tree. However, this might not to comprise all the semantic information needed to define all the instances of the individuals offered by a web site. We are interested in the semantic description that conforms to the semantics of any individual, which we call global semantic description and can be obtained by merging semantic descriptions. Function *mergeSDs* takes a set of semantic descriptions as input and outputs a semantic description that contains all the semantic information residing on all of the semantic descriptions that takes as input. It is specified as follows:

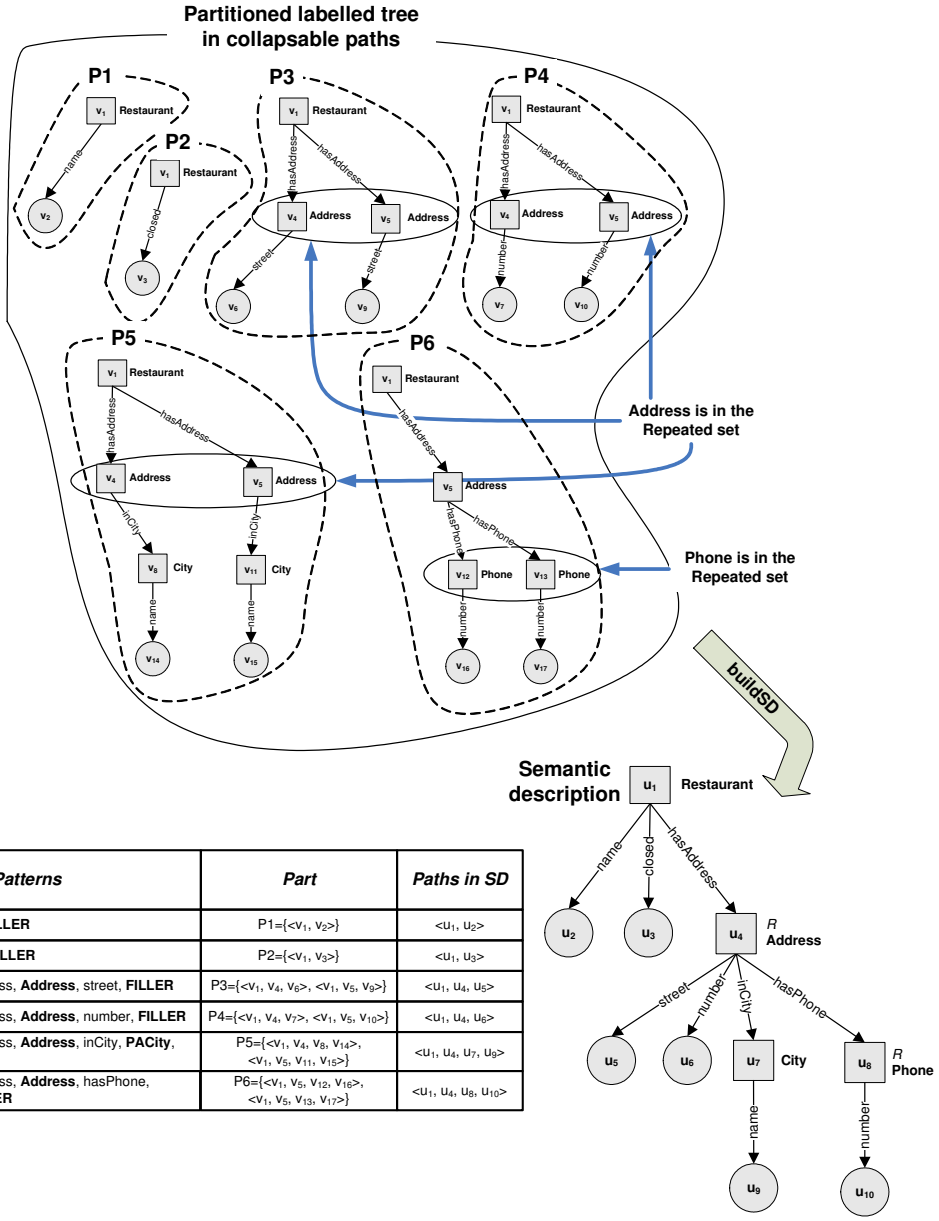
$$
\begin{array}{|l}
mergeSDs : \mathbb{F}\, SemanticDescription \rightarrow SemanticDescription \\
\hline
\forall\, ssd : \mathbb{F}\, SemanticDescription;\ sd : SemanticDescription \bullet mergeSDs(ssd) = sd \Leftrightarrow \\
\quad \forall\, sd_x : ssd;\ p : paths(sd_x.lt.t) \bullet \exists_1\, q : paths(sd.lt.t) \bullet \\
\qquad (sd_x.lt, p) \sim (sd.lt, q) \bigwedge (\forall\, i : 1 \,..\, \#p \mid p(i) \in sd_x.repeated \bullet q(i) \in sd.repeated)) \bigwedge \\
\quad \forall\, p : paths(sd.lt.t) \bullet \exists\, sd_x : ssd;\ q : paths(sd.lt.t) \bullet (sd_x.lt, p) \sim (sd.lt, q) \bigwedge \\
\quad \forall\, p : paths(sd.lt.t);\ i : 1 \,..\, \#p \mid p(i) \in sd.repeated \bullet \\
\qquad \exists\, sd_x : ssd;\ q : Path \mid q \in paths(sd.lt.t) \bullet q(i) \in sd_x.repeated
\end{array}
$$

Let $\mathcal{I}$ be the set of individuals in a web site and $\mathcal{SD}$ the global semantic description, then $\mathcal{SD} = buildGlobalSD(\mathcal{I})$, where function *buildGlobalSD* is defined as the result of merging all of the semantic descriptions obtained from the elements in $\mathcal{I}$:

$$
\begin{array}{|l}
buildGlobalSD : \mathbb{F}\, IndividualTree \rightarrow SemanticDescription \\
\hline
\forall\, pit : \mathbb{F}\, IndividualTree \bullet buildGlobalSD(pit) = mergeSDs(\{it : pit \bullet buildSD(sd)\})
\end{array}
$$

8.6 Relating information and semantic descriptions

According to the assumptions in Section §8.2, each filler vertex in a semantic description represents one attribute in the hierarchical slot that appears in a hierarchical level. Thus, the extracted information can be univocally related to a semantic description by providing location information for filler vertices. The attribute that a filler vertex represents can be referenced by a hierarchical level and by a natural index to distinguish amongst several vertices in the



Patterns	*Part*	*Paths in SD*
Restaurant, name, **FILLER**	P1={<v_1, v_2>}	<u_1, u_2>
Restaurant, closed, **FILLER**	P2={<v_1, v_3>}	<u_1, u_3>
Restaurant, hasAddress, **Address**, street, **FILLER**	P3={<v_1, v_4, v_6>, <v_1, v_5, v_9>}	<u_1, u_4, u_5>
Restaurant, hasAddress, **Address**, number, **FILLER**	P4={<v_1, v_4, v_7>, <v_1, v_5, v_{10}>}	<u_1, u_4, u_6>
Restaurant, hasAddress, **Address**, inCity, **PACity**, name, **FILLER**	P5={<v_1, v_4, v_8, v_{14}>, <v_1, v_5, v_{11}, v_{15}>}	<u_1, u_4, u_7, u_9>
Restaurant, hasAddress, **Address**, hasPhone, **Phone**, number, **FILLER**	P6={<v_1, v_5, v_{12}, v_{16}>, <v_1, v_5, v_{13}, v_{17}>}	<u_1, u_4, u_8, u_{10}>

Figure 8.6: *The builSD function.*

same hierarchical level. For instance, vertex u_2 in the semantic description of Figure §8.7 represents the first attribute of vertices at the $H0$ level of the hierarchical slot; and u_8 represents the third attribute of vertices at the $H1$ level. Two concepts allow us to formalise this, namely: influence areas and mirrored influence areas.

8.6.1 Influence areas and mirrored influence areas

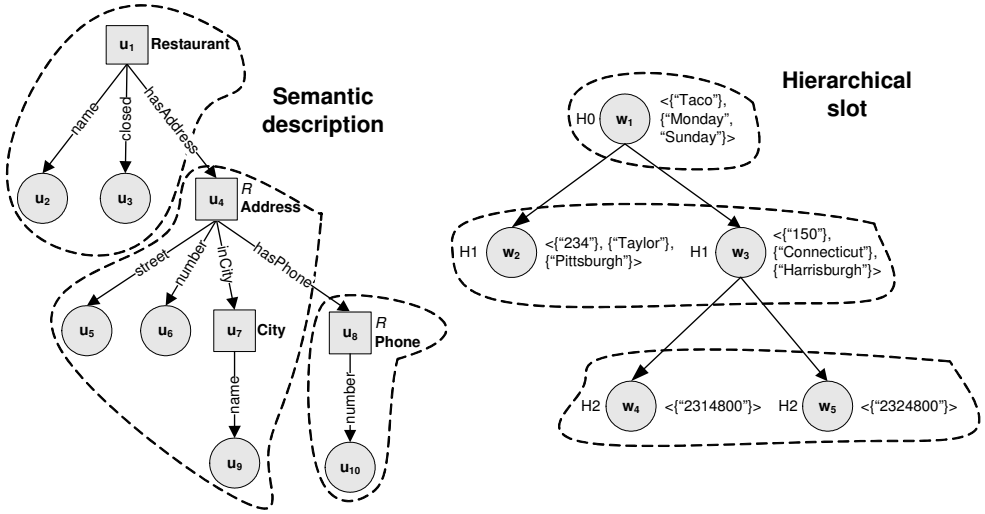
Each concept vertex in the *repeated* set in a semantic description or root vertex defines an influence area; thus, filler vertices in the same area of influence take values from the same hierarchical level.

Definition 8.5 (Influence areas) *The area of influence of a repeated or root vertex v in the labelled tree of a semantic description is composed of all of the vertices that are reachable from v without passing through any other repeated vertex. Formally:*

$$
\begin{array}{|l}
influenceArea : SemanticDescription \times \mathbb{F}\, Vertex \\
\hline
\forall\, sd : SemanticDescription;\ sv : \mathbb{F}\, Vertex \bullet influenceArea(sd, sv) \Leftrightarrow \\
\quad \exists_1\, v : sv \mid v \in sd.repeated \bigvee v = root(sd.lt.t) \bullet sv \setminus \{v\} = \\
\qquad \{p : paths(sd.lt.t);\ n, i : \mathbb{N} \mid p(n) = v \bigwedge i \in n + 1 \,..\, \#p \bigwedge \\
\qquad\quad (\nexists j : n + 1 \,..\, i \bullet p(j) \in sd.repeated) \bullet p(i)\}
\end{array}
$$

Example 8.6 *Figure §8.7 illustrates a partition of the semantic description into influence areas. There are three different areas defined by the root vertex (Restaurant) and the repeated vertices (Address and Phone). The area defined by Restaurant is associated to hierarchical level $H0$; the filler vertices from this area take their value from the vertices in the hierarchical slot at level $H0$. And so on.*

Since semantic descriptions are obtained from individual trees so that vertices and paths in individual trees are mirrored to vertices and paths in semantic descriptions, influences areas in a semantic description are mirrored also in individual trees. Thus, each concept vertex that is mirrored to a *repeated* set in the semantic description or to the root defines a mirrored influence area, i.e., thus, filler vertices in the same area of influence take values from the same vertex in the hierarchical slot.



Influence areas	Level on hierarchical slot
$\{u_1, u_2, u_3\}$	H0
$\{u_4, u_5, u_6, u_7, u_9\}$	H1
$\{u_8, u_{10}\}$	H2

Figure 8.7: *Areas of influence.*

Definition 8.6 (Mirrored influence areas) *The mirrored area of influence of vertex v in the labelled tree of an individual tree that is associated to a vertex in the repeated set of the semantic description or is the root is composed of all of the vertices reachable from v without passing through any other vertex that is mirrored to a repeated vertex in the semantic description.*

$$
\begin{array}{|l}
mirroredInfluenceArea : SemanticDescription \times IndividualTree \times \mathbb{F}\, Vertex \\
\hline
\forall\, sd : SemanticDescription;\ it : IndividualTree;\ sv : \mathbb{F}\, Vertex \bullet \\
\quad mirroredInfluenceArea(sd, it, sv) \Leftrightarrow \\
\qquad \exists_1\, su : sd.lt.t.vertices \mid influenceArea(sd, su) \bullet \\
\qquad\quad (\forall\, v : sv \bullet mirrorInSD(sd, it, v) \in su \bigwedge \\
\qquad\quad \nexists v \in it.lt.t.vertices \mid v \notin sv \bullet mirrorInSD(sd, it, v) \in su)
\end{array}
$$

Function *mirrorInSD* takes a semantic description, an individual tree, and a vertex that belongs to the labelled tree of the individual tree as input and outputs the mirrored vertex in the semantic description.

$$mirrorInSD : SemanticDescription \times IndividualTree \times Vertex \pfun Vertex$$

$$\begin{aligned}
&\forall\, sd : SemanticDescription;\ it : IndividualTree;\ v, u : Vertex \bullet \\
&\quad mirrorInSD(sd, it, v) = u \Leftrightarrow \\
&\qquad \exists\, p_1 : paths(lt.lt.t);\ p_2 : paths(sd.lt.t);\ i : \mathbb{N} \mid p_1(i) = v \bigwedge (sd.lt, p_1) \sim (it.lt, p_2) \bullet \\
&\qquad\quad p_2(i) = u
\end{aligned}$$

Example 8.7 *Figure §8.8 illustrates a partition of our running individual tree into mirrored influence areas. There are five different areas defined by the root vertex (Restaurant) and the vertices that are mirrored to vertices in the repeated set of the semantic description (Address and Phone). The area defined by Restaurant is associated to hierarchical level H0; filler vertices from this area take their values from vertex w_0 in the hierarchical slot. The areas defined by v_4 and v_5, which represent instances of concept address, are associated to vertices w_2 and w_3 in the hierarchical slot; since the Address concept in the semantic description is associated to hierarchical level H1, w_2 and w_3 are labelled with H1. We omit the remaining details, which should not be a problem to the reader.*

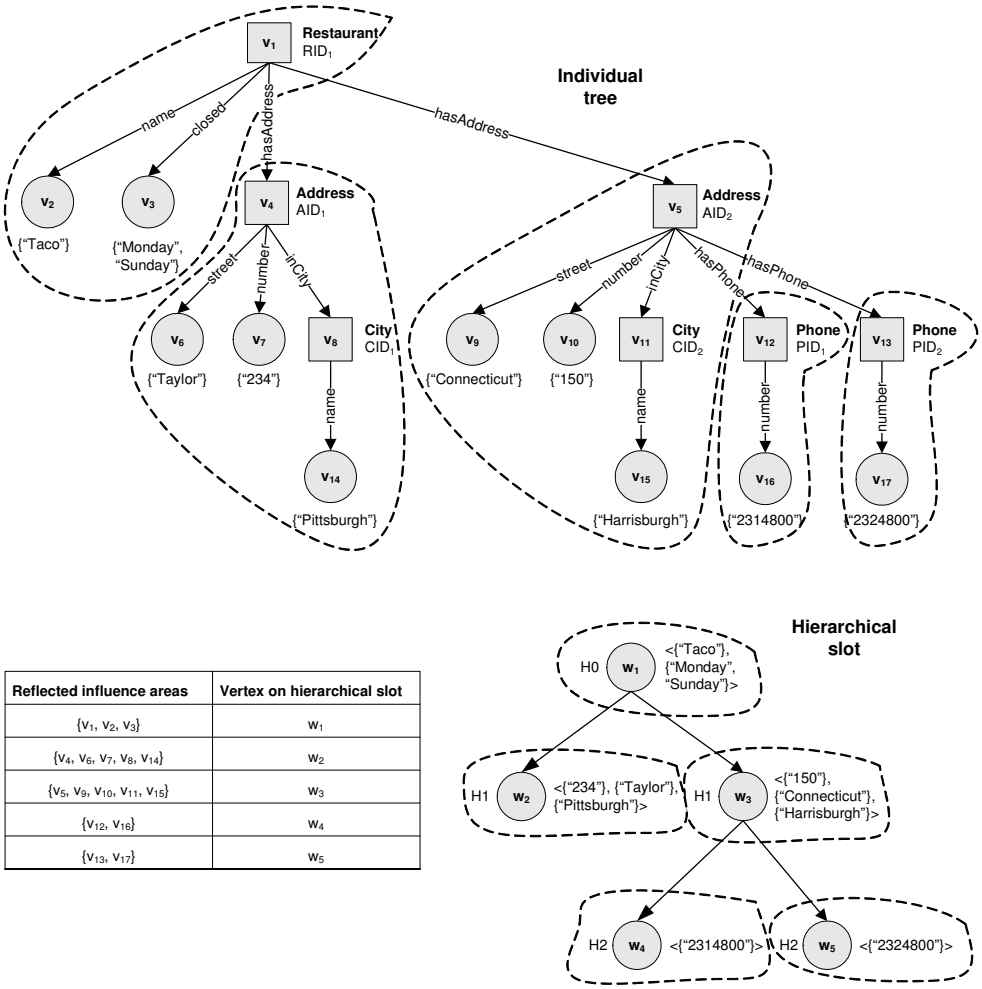
8.6.2 Building the location information

According to the previous ideas, we devise the *Location* type data. It maps filler vertices in a semantic description into hierarchical levels (*vertexLevel*) or natural indices (*vertexPosition*) to indicate univocally which attribute each filler vertex represents inside the hierarchical level.

Location

$$vertexPosition : Vertex \pfun \mathbb{N}$$
$$vertexLevel : Vertex \pfun Label$$

Before specifying how to obtain the location information, we formalise the relationships amongst an individual tree, a hierarchical slot, a global semantic description, and the location function. Predicate *translation* allows to relate them. It is based on the idea that a semantic description can be viewed as a container of patterns, where each pattern is defined by an influence area; hierarchical slots materialise influence mirrored areas from these patterns. Thus, *translation* uses a total bijection function that maps vertices in the hierarchical slots into mirrored influence areas in the individual tree, in such a way that filler vertices in the individual tree take their attributes from the location information provided by their mirrored vertices in the semantic description. Furthermore, the edges in the hierarchical slot are mapped onto the edges of



Reflected influence areas	Vertex on hierarchical slot
$\{v_1, v_2, v_3\}$	w_1
$\{v_4, v_6, v_7, v_8, v_{14}\}$	w_2
$\{v_5, v_9, v_{10}, v_{11}, v_{15}\}$	w_3
$\{v_{12}, v_{16}\}$	w_4
$\{v_{13}, v_{17}\}$	w_5

Figure 8.8: *Mirrored areas of influence.*

the individual tree that connects vertices defining mirrored influence areas (the types of the edges are obtained from the semantic description). The auxiliary function *subtreeRoot* outputs the root of a subtree which is represented as a set of vertices, and it is defined in Appendix §A.

$$
\begin{array}{|l}
translation : IndividualTree \times HierarchicalSlot \times SemanticDescription \times Location \\
\hline
\forall\, it : IndividualTree;\ hs : HierarchicalSlot;\ sd : SemanticDescription;\ loc : Location \bullet \\
\quad translation(it, hs, sd, loc) \Leftrightarrow \exists f : hs.t.vertices \twoheadrightarrow \mathbb{F}\, it.lt.t.vertices \bullet \\
\qquad \forall\, sv : \mathrm{ran} f \bullet mirroredInfluenceAreas(sd, it, sv) \bigwedge \\
\qquad \forall\, w : \mathrm{dom} f;\ v : Vertex \mid v \in f(w) \bigwedge it.lt.vertexType(v) = FILLER \bullet \\
\qquad\quad it.vertexAttribute(v) = \\
\qquad\qquad hs.vertexAttributes(w)(loc.vertexPosition(mirrorInSD(sd, it, v))) \bigwedge \\
\qquad\quad loc.vertexLevel(mirrorInSD(sd, it, v)) = hs.vertexHLevel(w))) \bigwedge \\
\qquad \forall (w_1, w_2) : hs.t.edges \bullet \exists_1 v_1, v_2 : Vertex \bullet \\
\qquad\quad v_1 = subtreeRoot(it.lt.t, f(w_1)) \bigwedge v_2 = subtreeRoot(it.lt.t, f(w_2)) \bigwedge \\
\qquad\quad (v_1, v_2) \in it.lt.edges \bigwedge \\
\qquad\quad sd.lt.edgeType(mirrorInSD(sd, it, v_1), mirrorInSD(sd, it, v_2)) = \\
\qquad\qquad it.lt.edgeType(v_1, v_2)
\end{array}
$$

Example 8.8 *Figure §8.9 illustrates two examples of the relationships amongst an individual tree, a hierarchical slot, a global semantic description and the location function. An X in the location information represents any value (it is not of interest in the example).*

Let $\mathcal{IH}$ be the set of pairs of individual trees and hierarchical slots in a web site, and $\mathcal{SD}$ its corresponding global semantic description, then the location information $\mathcal{LOC}$ is obtained as $\mathcal{LOC} = buildLocation(\mathcal{IH}, \mathcal{SD})$, where function *buildLocation* is defined from predicate *translation*, and it ensures that the location information is well-defined for every element in $\mathcal{IH}$:

$$
\begin{array}{|l}
buildLocation : \mathbb{F}(IndividualTree \times HierarchicalSlot) \times SemanticDescription \rightarrow Location \\
\hline
\forall\, ih : \mathbb{F}(IndividualTree \times HierarchicalSlot);\ sd : SemanticDescription;\ loc : Location \bullet \\
\quad buildLocation(ih, sd) = loc \Leftrightarrow \\
\qquad \mathrm{dom}\, loc.vertexPosition = \{u : sd.lt.t.vertices \mid sd.lt.vertexType(u) = FILLER \bullet u\} \bigwedge \\
\qquad \mathrm{dom}\, loc.vertexLevel = \{u : sd.lt.t.vertices \mid sd.lt.vertexType(u) = FILLER \bullet u\} \bigwedge \\
\qquad \forall (it, hs) : ih \bullet translation(it, hs, sd, loc)
\end{array}
$$

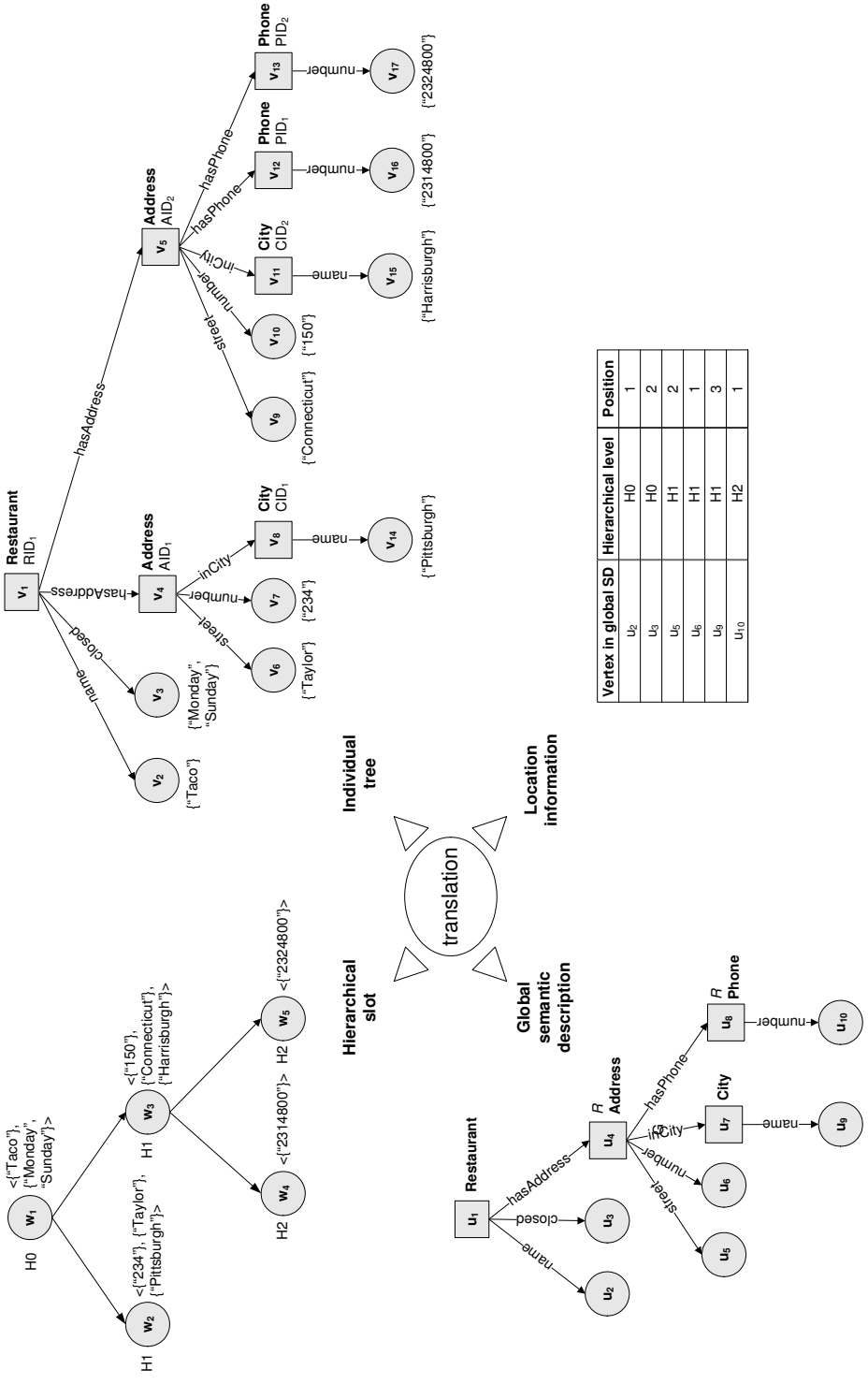
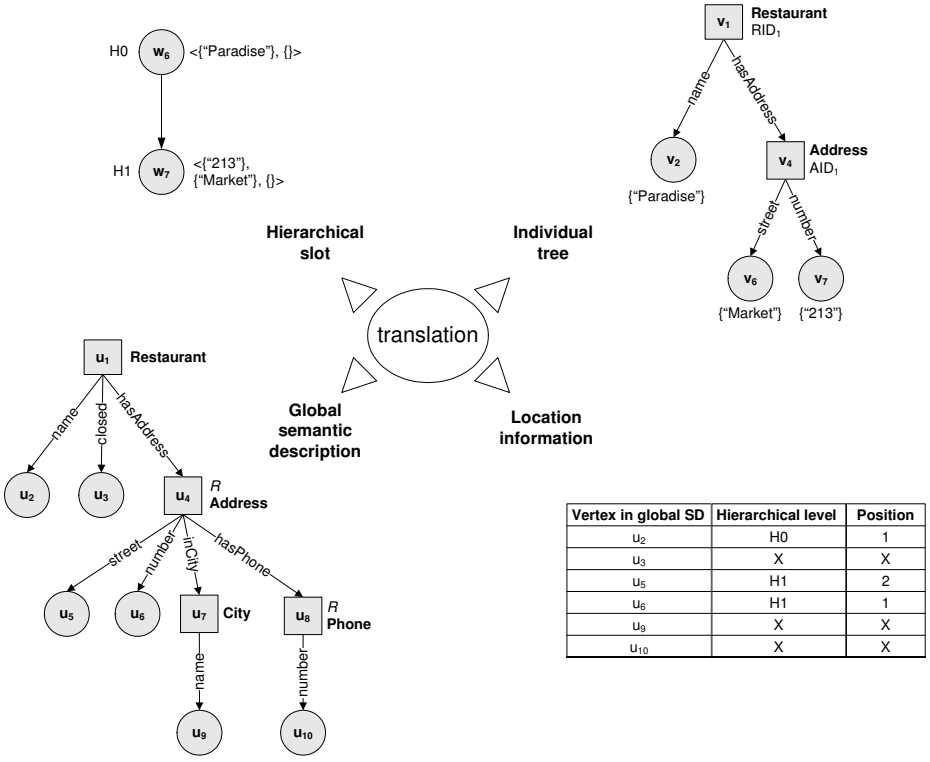


Figure 8.9: *The translation relation.*



Vertex in global SD	Hierarchical level	Position
u_2	H0	1
u_3	X	X
u_5	H1	2
u_6	H1	1
u_9	X	X
u_{10}	X	X

Figure 8.9: *The translation relation (Cont'd).*

8.7 Semantic translators

The problem of semantic translation is solved by setting up a generic algorithm, called semantic translator, with the global semantic description and the location information obtained previously. It uses these data to automatically give semantics to any *StructuredInformation* extracted by a wrapper. Since we defined *StructuredInformation* to be a set of *HierarchicalSlot*, the semantic translator translates each tree in the set; furthermore, it uses *toAbox*, which is defined Appendix §B).

$$
\begin{array}{|l}
semanticTranslator : SemanticDescription \times Location \times StructuredInformation \rightarrow Abox \\
\hline
\forall\, sd : SemanticDescription;\ loc : Location;\ si : StructuredInformation;\ a : Abox \bullet \\
\quad semanticTranslator(sd, loc, si) = \\
\qquad \bigcup \{ \forall\, hs : si;\ it : IndividualTree \mid translation(it, hs, sd, loc) \bullet toAbox(it) \}
\end{array}
$$

8.8 Summary

In this chapter, we have defined our solution for the problem of semantic translation. The main results were published at the 2003 IEEE/WIC International Conference on Web Intelligence (WI'03) [11], the First International Atlantic Web Intelligence Conference (AWIC 2003) [7], and the 15th Conference on Advanced Information Systems Engineering (CAiSE'03) [10].

Chapter 9

A materialisation of the semantic translation problem

The worst thing one can do is not to try,
to be aware of what one wants and not give in to it,
to spend years in silent hurt wondering if
something could have materialised, never knowing.

David Viscott, 1938–
American psychiatrist

In this chapter, we present an implementation of semantic translators. It is organised as follows: Section §9.1 presents a brief introduction; Section §9.2 illustrates the implementation of an algorithm to build semantic descriptions; Section §9.3 presents an algorithm to obtain the location information; Section §9.4 focuses on the implementation of the semantic translator generic algorithm; finally, in Section §9.5, we summarise the chapter.

9.1 Introduction

Abstractness provides us a maximal degree of implementation freedom, but it is not practical enough unless we devise an implementation. In this chapter, we present a materialisation of the semantic translation problem. It consists of three algorithms: the first takes care of building global semantic descriptions, the second is responsible for calculating location information, and the third is a domain-independent semantic translator. We also prove that these algorithms are correct implementations with respect to specification in Chapter §8.

We describe the materialisation by means of Plotkin's transition rules since this technique has proved to be both simple and powerful [102], c.f. Appendix §A. Note that our main goal is not to report on the many optimisations that are possible, but to demonstrate that the framework can be implemented using finite-state machines. Nevertheless, we point out some tips that may help optimise the implementation, but we do not present many details since they fall outside the scope of this dissertation.

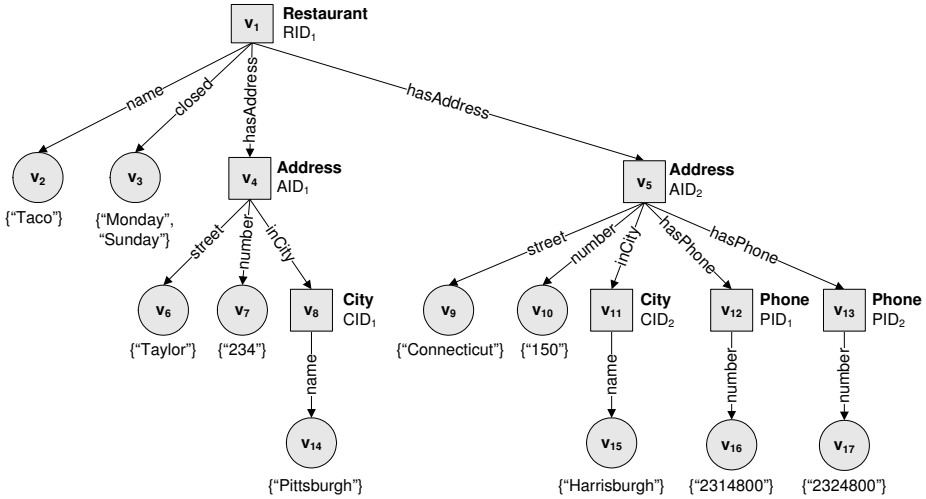
9.2 Building semantic descriptions

In the previous chapter, we explain how a semantic description is built by merging the semantic information provided by all of the individuals in a web site. Inductive methods seem to be a good election for implementing an algorithm to solve this problem. However, in this materialisation, we follow a simpler strategy that requires users to have a good understanding of the semantics relations residing on a web site. It requires the user to build an Abox with information about one individual that fulfills the following requirements:

Requirement 1. All of the properties and concept names that are needed to render the information on the web page to instances appear in the Abox.

Requirement 2. Each property with multiple cardinality ($[0..n]$ or $[1..n]$) must appear two times at least. That is, there exists more than one property assertion in the Abox that state that the same concept instance is related to other concept instances.

Requirement 3. Each optional property ($[0\,..\,1]$) appears once. That is, there exists only one property assertion in the Abox that states that a concept instance is related to only one concept instance or to literal values.



(a) An individual tree (repetition of Figure §8.2)

Name: Taco **Close on:** Monday & Sunday
Address: 234 Taylor, Pittsburgh
Address: 150 Connecticut, Harrisburgh
Phone: 2314800
Phone: 2324800
Name: Paradise
Address: 213 Market
Name: FireMeal **Close on:** Monday
Address: 122 West, New York
Phone: 2344800

(b) Web page (repetition of Figure §7.2(a))

Figure 9.1: *Requirements for algorithm* `buildSD`.

The previous requirements ensure that the Abox comprises all the semantic information needed for the translation of all of the individuals in the web site. For the sake of convenience, the algorithm is presented in terms of individual trees instead of Aboxes (as mentioned in the previous chapter, they are equivalent notations for representing individuals, but the former is better suited to design our algorithms).

Example 9.1 *The indvidual tree in Figure §9.1(a) does not fulfill the previous requirements because address AID_1 has not any phone numbers. It should have more than one phone number because the hasPhone property has multiple cardinality according to the information provided by the web page in Figure §9.1(b).*

9.2.1 Algorithm

The algorithm `buildSD` takes an individual tree that fulfills the previous constraints as input, and outputs a global semantic description. It works as follows: initially, the labelled tree of the semantic description is a clone of the labelled tree of the individual tree, and the set of repeated vertices is empty. Then, it traverses the cloned labelled tree by processing concept vertices. The processing consists of computing the collapsable vertices that are children of the concept vertex being studied. For each set of collapsable vertices with more that one element, the algorithm prunes the subtrees associated to all of the collapsable vertices except for one, which is added to the set of repeated vertices. The algorithm terminates when the labelled tree in the semantic description is traversed completely.

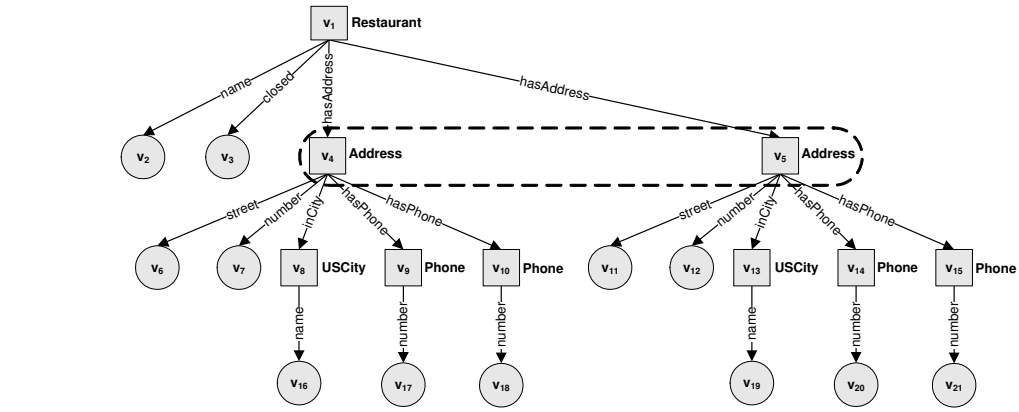
Figure §9.2 shows a trace of the `buildSD` algorithm. The first vertex to be studied is the root vertex v_1; Figure §9.2(a) shows the unique part that belongs to the collapsable vertices of v_1. The algorithm prunes the subtree associated to v_5 (it is indifferent to take v_4 or v_5) and adds the other vertex to the repeated set of the semantic description and to set of vertices to be studied. In Figure §9.2(b), vertex v_4 is studied. Concept vertices of v_4 are partitioned into two parts. The first part has a concept vertex only, and it is added to the set of vertices to be studied. The second part has two concept vertices. Again, the subtree associated to one of these vertices is pruned and the other vertex is added to the set of repeated vertices and to set of vertices to be studied. Figure §9.2(c) shows the semantic description obtained.

Configurations: The configurations we need to define the `buildSD` algorithm are four-tuples defined as:

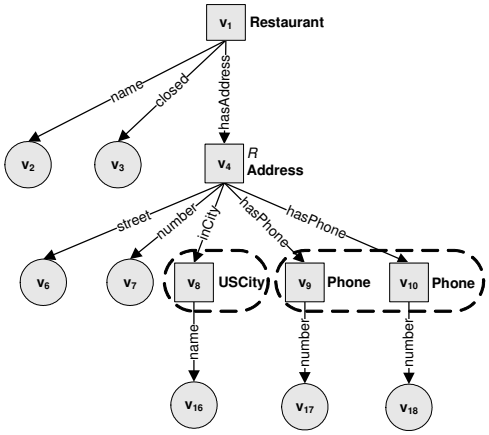
$$Config_{BSD} == IndividualTree \times LabelledTree \times \mathbb{F}\, Vertex \times \mathbb{F}\, Vertex$$

where the first element refers to an individual tree that fulfills the previous requirements; the second and third elements compose the semantic description for which we are looking (a labelled tree and a set of repeated vertices); the last element records the vertices to be studied.

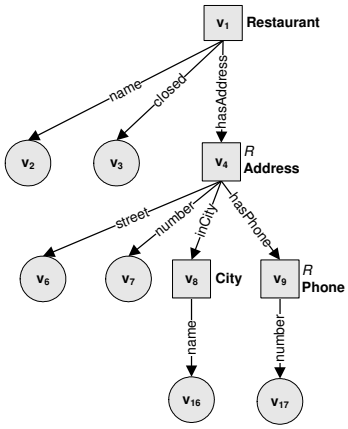
Predicates and functions: Two functions are used to update configurations, namely: *pruneSubtrees* and *partitionChildren*. In addition, function *cloneLt* is used to define the initial configuration of the algorithm. They are specified as follows:



(a) Studying vertex v_1



(b) Studying vertex v_4



(c) Resulting semantic description

Figure 9.2: *Example of how* `buildSD` *works.*

i. Function *pruneSubtrees* prunes the subtrees associated to a set of vertices in a given labelled tree. It removes the set of vertices taken as input, and the vertices reachable from them (all of their descendants). The edges that hit one of previous vertices are removed, too. Furthermore, functions *vertexType* and *edgeType* are updated to maintain information about the remaining vertices and edges only.

$$
\begin{array}{|l}
pruneSubtrees : LabelledTree \times \mathbb{F}\, Vertex \pfun LabelledTree \\
\hline
\forall\, lt_1, lt_2 : LabelledTree;\ sv_1, sv_2 : \mathbb{F}\, Vertex \mid sv_1 \subseteq lt_1.t.vertices \bullet \\
\quad pruneSubtrees(lt_1, sv_1) = lt_2 \Leftrightarrow \\
\qquad sv_2 = \{p : paths(lt_1.t);\ v : sv_1;\ i, j : \mathbb{N} \mid v \in \operatorname{ran} p \land p(i) = v \land j \geq i \bullet p(j)\} \land \\
\qquad lt_2.t.vertices = lt_1.t.vertices \setminus sv_2 \land \\
\qquad lt_2.t.edges = \{(v_1, v_2) : lt_1.t.edges \bullet v_1 \notin sv_2 \land v_2 \notin sv_2 \bullet (v_1, v_2)\} \land \\
\qquad lt_2.vertexType = \{v : lt_2.t.vertices \bullet v \mapsto lt_1.vertexType(v)\} \land \\
\qquad lt_2.edgeType = \{e : lt_2.t.edges \bullet e \mapsto lt_1.edgeType(e)\}
\end{array}
$$

ii. Function *partitionChildren* partitions filler children of a concept vertex, and outputs a set of sequences of vertices. The range of each sequence represents a maximal set of collapsable vertices. A vertex with filler children only results in the empty set.

$$
\begin{array}{|l}
partitionChildren : LabelledTree \times Vertex \pfun \mathbb{F} \operatorname{seq} Vertex \\
\hline
\forall\, lt : LabelledTree;\ v : Vertex;\ ssv : \mathbb{F} \operatorname{seq} Vertex \mid v \in lt.t.vertices \bullet \\
\quad (notFillerChildren(lt, v) \neq \varnothing \land partitionChildren(lt, v) = ssv \Leftrightarrow \\
\qquad \forall\, sv : ssv \bullet collapsableVertices(lt, \operatorname{ran} sv) \land \\
\qquad \#notFillerChildren(lt, v) = \bigcup \{sv : ssv \bullet \operatorname{ran} sv\} \land \\
\qquad \forall\, sv : ssv \bullet \#sv = \# \operatorname{ran} sv) \lor \\
\quad (notFillerChildren(lt, v) = \varnothing \land partitionChildren(lt, v) = \varnothing)
\end{array}
$$

$$
\begin{array}{|l}
notFillerChildren : LabelledTree \times Vertex \pfun \mathbb{F}\, Vertex \\
\hline
\forall\, lt : LabelledTree;\ v : lt.t.vertices \bullet notFillerChildren(lt, v) = \\
\quad \{v_x : children(v) \mid lt.vertexType(v_x) \neq FILLER\}
\end{array}
$$

iii. Function *cloneLt* clones a labelled tree. It is specified by defining a total bijection function that maps vertices between labelled trees. Note that this definition is just a semantic characterisation, and that it is not intended to be executable since we think that this topic is well-covered in the literature.

$$
\begin{array}{|l}
cloneLt : LabelledTree \rightarrow LabelledTree \\
\hline
\forall\, lt_1, lt_2 : LabelledTree \bullet cloneLt(lt_1) = lt_2 \Leftrightarrow \\
\quad \exists f : lt_1.t.vertices \rightarrowtail lt_2.t.vertices \bullet \\
\quad\quad \forall (v_1, v_2) : lt_1.t.edges \bullet \\
\quad\quad\quad (f(v_1), f(v_2)) \in lt_2.t.edges \bigwedge \\
\quad\quad\quad lt_1.edgeType(v_1, v_2) = lt_2.edgeType(f(v_1), f(v_2)) \bigwedge \\
\quad\quad \forall\, v : lt_1.t.vertices \bullet lt_1.vertexType(v) = lt_2.vertexType(f(v))
\end{array}
$$

Rules: The `buildSD` algorithm is defined by the following rules that define a homogeneous relation between configurations.

i. If the vertex being visited has children of type concept, then we use function *partitionChildren* to partition them into collapsable vertices. The first element of each sequence is added to the tail of *sv* (the set of vertices to be studied); furthermore, if the sequence to which the vertex belongs has more than one element, it is added to the *rep* set. Subtrees associated to vertices at the tail of the sequence are pruned from *lt*.

$$
(it, lt, rep, sv) \rightarrow_{BSD} (it, lt', rep', sv') \quad \left[\begin{array}{l} sv \neq \varnothing \\ pc \neq \varnothing \end{array} \right]
$$

$$
where \quad \left\{ \begin{array}{l}
v == member(sv) \\
pc == partitionChildren(lt, v) \\
sv' == (sv \setminus \{v\}) \cup \{s : pc \bullet head(s)\} \\
lt' == pruneSubtrees(lt, \bigcup \{s : pc \bullet \operatorname{ran} tail(s)\}) \\
rep' == rep \cup \{s : pc \mid \#s > 1 \bullet head(s)\}
\end{array} \right.
$$

ii. If the vertex being visited has not any filler children, we update the vertices to be studied by removing it from the sequence.

$$
(it, lt, rep, sv) \rightarrow_{BSD} (it, lt, rep, sv') \quad \left[\begin{array}{l} sv \neq \varnothing \\ pc = \varnothing \end{array} \right]
$$

$$
where \quad \left\{ \begin{array}{l}
v == member(sv) \\
pc == partitionChildren(lt, v) \\
sv' == sv \setminus \{v\}
\end{array} \right.
$$

Algorithm: The algorithm is specified as the application of the previous rules as many times as necessary to reach the final configuration $(it, lt, rep, \langle\,\rangle)$ from the initial configuration $(it, cloneLt(it.lt), \varnothing, \langle root(it.lt.t)\rangle)$. It is formalised as follows:

$$\texttt{buildSD} : IndividualTree \nrightarrow SemanticDescription$$

$$\begin{aligned}
&\forall\, it : IndividualTree;\ sd : SemanticDescription;\ lt : LabelledTree;\ rep : \mathbb{F}\, Vertex \bullet \\
&\quad \texttt{buildSD}(it) = sd \Leftrightarrow \\
&\qquad (it, cloneLt(it.lt), \varnothing, \langle root(it.lt.t)\rangle) \xrightarrow{!}_{BSD} (it, lt, rep, \langle\,\rangle) \bigwedge \\
&\qquad sd.lt = lt \bigwedge sd.repeated = rep
\end{aligned}$$

9.2.2 Correctness

Since the individual tree taken as input fulfills the requirements presented at the beginning of this section, the semantic description comprises all the semantic information needed for the translation of all of the individuals on a web site.

Theorem 9.1 (Termination) *Algorithm* `buildSD` *terminates.*

Proof Initially, the labelled tree of the semantic description is a clone of the labelled tree of the individual tree. The algorithm traverses the cloned individual tree by processing concept vertices only. Vertices to be processed are recorded in the *sv* set. Once a concept vertex is studied it is removed from *sv*. Since individual trees have a finite number of vertices and cycles are not allowed, algorithm `buildSD` terminates after traversing all of the concept vertices. □

Theorem 9.2 (Correctness) *Algorithm* `buildSD` *is correct regarding the specification of function buildSD (c.f. Subsection §8.5.3 for its definition).*

Proof The proof consists of two parts: first, we have to show that collapsable vertices in the individual tree taken as input are collapsed in the resulting semantic description (i); and that collapsable paths in the individual tree are collapsed in the semantic description (ii).

i. The algorithm uses the *partitionChildren* function to partition the children of the vertex being studied into sequences of collapsable vertices. For each sequence, only one vertex is recorded in the set of vertices to be studied. The subtrees represented by the rest of vertices in the same partition are pruned. In other words, collapsabe vertices are collapsed. Because, all of the concept vertices in the output labelled tree are studied, the semantic description has not any collapsable vertices.

ii. It follows from Lemma §8.1 and (i).

This concludes the proof. □

9.2.3 Complexity

Algorithm `buildSD` visits all of the vertices in the individual tree, and for each concept vertex, it partitions children concept vertices into collapsabe vertices. Then, the asymptotic complexity is $O(n \cdot b)$, where n and b are the total number of vertices and the breadth of the individual tree, respectively.

Note that, n is always greater than b, since $b \in 1 \,..\, n-1$. For a tree with $b = 1$ (each vertex in tree has one child at most) the complexity of algorithm is approximated by $O(n)$; if $b = n - 1$ (only the root vertex has children, that is, all of the properties relate a concept assertion with literals) the complexity is approximated by $O(n^2)$.

9.3 Calculating locations

The algorithm takes an Abox, a hierarchical slot, and a global semantic description as input and outputs the location information. The Abox represents the information in the hierarchical slot, and the global semantic description comprises all of the semantic information needed for the translation of all of the individuals in a web site. The location information that it outputs relates the information extracted with a global semantic description. As was the case for the `buildSD` algorithm, this algorithm requires the user tu supply an Abox and a hierarchical slot that fulfills the following requirements:

Requirement 1. All of the properties and concept names that are needed to render the information on web pages to instances appear in the Abox.

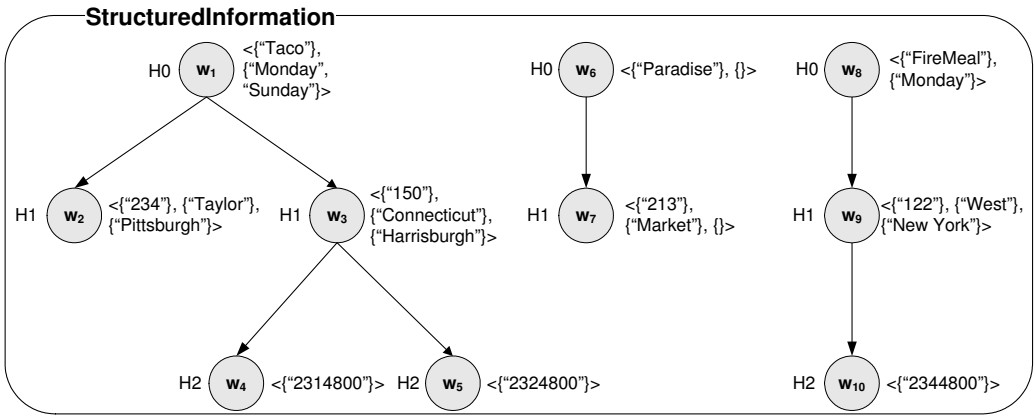


Figure 9.3: *StructuredInformation (repetition of Figure §7.2(b)).*

Requirement 2. Each optional or multiple cardinality property appears only once.

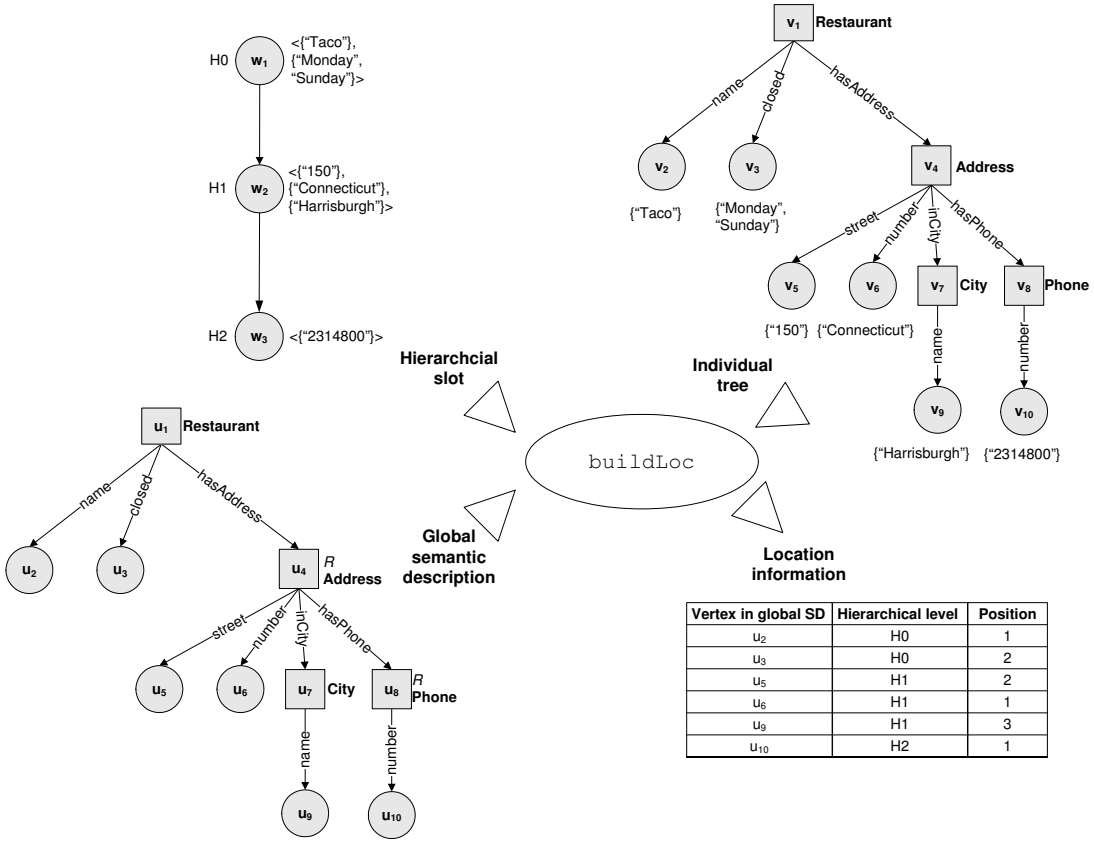
Requirement 3. There are not any empty attribute in the hierarchical slot, and all of the attributes are distinct.

Previous requirements avoid ambiguity in obtaining the location information. Note that the first two requirements are equivalent to stating that labelled tree of the individual tree must be the same as the labelled tree of the global semantic description.

Example 9.2 *The individual tree in Figure §9.1(a) does not fulfill the first and the second requirements. Restaurant RID_1 has more than one address, address AID_2 has more than one phone, and address AID_1 has not a phone number. Then, this individual tree and the hierarchical slot from which this individual tree is expected are not valid for building the location information. Similarly, the second hierarchical slot in he structured information in Figure §9.3 does not fulfill the third requirement. It contains empty attributes. Then, the hierarchical slot and the individual tree obtained, are not valid for building the location information.*

9.3.1 Algorithm

The algorithm `buildLoc` builds the location information as follows: it traverses the labelled tree of the semantic description by processing filler vertices.



Vertex in global SD	Hierarchical level	Position
u_2	H0	1
u_3	H0	2
u_5	H1	2
u_6	H1	1
u_9	H1	3
u_{10}	H2	1

Figure 9.4: *Example of how* `buildLoc` *works.*

The processing consist of looking for the vertex mirrored in the individual tree (it is unique by the previous requirements), and looking for its attributes in the corresponding hierarchical slot. The algorithm terminates when the labelled tree is traversed completely.

Example 9.3 *Figure §9.4 illustrates the location information obtained for a global semantic description, an individual tree, and a hierarchical slot. For instance, the hierarchical level and position for the semantic description vertex u_2 is obtained by looking for attribute {"Taco"} in the hierarchical slot.*

Configurations: The configurations we need are five-tuples defined as follows:

$$Config_{BL} == IndividualTree \times HierarchicalSlot \times SemanticDescription \times Location \times \mathbb{F}\, Vertex$$

where the first and second elements refer to an individual tree and a hierarchical slot that fulfills the previous restrictions; the third element is the global semantic description, the fourth element is the location structure that we are looking for; the last element records the name of the vertices to be studied.

Predicates and functions: We need a couple of auxiliary functions to define this algorithm, namely:

i. Function *sameVertex* looks for a vertex in the individual tree that is the mirror of a semantic description vertex. The mirrored vertex is obtained by determining the unique path in the semantic description to which the path containing the vertex of the individual tree is equivalent.

$$
\begin{array}{|l}
sameVertex : Vertex \times LabeledTree \times LabeledTree \pfun Vertex \\
\hline
\forall\, v_1, v_2 : Vertex;\ lt_1, lt_2 : LabeledTree \mid v_1 \in lt_1 \bullet sameVertex(v_1, lt_1, lt_2) = v_2 \Leftrightarrow \\
\quad \exists\, p_1 : paths(lt_1.t);\ p_2 : paths(lt_2.t);\ i : \mathbb{N} \mid \\
\qquad (p_1, lt_1) \sim (p_2, lt_2) \bigwedge p_1(i) = v_1 \bullet p_2(i) = v_2
\end{array}
$$

ii. Function *findAttribute* looks for an attribute into a hierarchical slot. It returns the position the attribute occupies into the *vertexAttributes* and the hierarchical level in which it is defined.

$$
\begin{array}{|l}
findAttribute : HierarchicalSlot \times Attribute \pfun \mathbb{N} \times Label \\
\hline
\forall\, hs : HierarchicalSlot;\ a : Attribute;\ n : \mathbb{N};\ l : Label \bullet \\
\quad findAttribute(hs, a) = (n, l) \Leftrightarrow \\
\qquad \exists_1\, v : hs.t.vertices;\ sa : \mathrm{seq}\, Attribute;\ a : Attribute \bullet \\
\qquad\quad hs.vertexAttributes(v) = sa \bigwedge sa(n) = a \bigwedge hs.vertexHLevel(v) = l
\end{array}
$$

Rules: The algorithm is defined by the following rules:

i. If the vertex v being visited is a filler, we also need to update sv (set of vertices to be studied) by removing v.

$$(it, hs, sd, loc, sv) \rightarrow_{BL} (it, hs, sd, loc', sv') \quad \left[\begin{array}{l} sv \neq \varnothing \\ sd.lt.vertexType(v) = FILLER \end{array} \right]$$

$$where \quad \left\{ \begin{array}{l} v == member(sv) \\ a == it.vertexAttribute(sameVertex(v, sd.lt, it.lt)) \\ (pos, hlevel) == findAttribute(hs, a) \\ loc' == (\!| \, vertexPosition \rightsquigarrow loc.vertexPosition \cup \{v \mapsto hlevel\}, \\ \qquad\qquad vertexLevel \rightsquigarrow loc.vertexLevel \cup \{v \mapsto hlevel\} \, |\!) \\ sv' == sv \setminus \{v\} \end{array} \right.$$

ii. If v is not a filler, then we update sv by removing v and adding its children.

$$(it, hs, sd, loc, sv) \rightarrow_{BL} (it, hs, sd, loc, sv') \quad \left[\begin{array}{l} sv \neq \varnothing \\ sd.lt.vertexType(v) \neq FILLER \end{array} \right]$$

$$where \quad \left\{ \begin{array}{l} v == member(sv) \\ sv' == (sv \setminus \{v\}) \cup children(sd.lt.t, v) \end{array} \right.$$

Algorithm: The algorithm is specified as the application of the previous rule as many times as necessary to reach the final configuration $(it, hs, sd, loc, \langle\,\rangle)$ from the initial configuration $(it, hs, sd, (\!| \, vertexPosition \rightsquigarrow \varnothing, vertexLevel \rightsquigarrow \varnothing \, |\!), \langle root(sd.lt.t)\rangle)$. It is formalised as follows:

$$\texttt{buildLoc} : IndividualTree \times HierarchicalSlot \times SemanticDescription \nrightarrow Location$$

$$\forall\, it : IndividualTree;\ hs : HierarchicalSlot;\ sd : SemanticDescription;\ loc : Location \bullet$$
$$\texttt{buildPos}(it, hs, sd) = loc \Leftrightarrow$$
$$(it, hs, sd, (\!| \, vertexPosition \rightsquigarrow \varnothing, vertexLevel \rightsquigarrow \varnothing \, |\!), \langle root(sd.lt.t)\rangle) \overset{!}{\rightarrow}_{BL} (it, hs, sd, loc, \langle\,\rangle)$$

9.3.2 Correctness

Theorem 9.3 (Termination) *Algorithm* `buildLoc` *terminates.*

Proof The algorithm traverses all of the vertices of the labelled tree in a semantic description. Once a vertex is processed, it is removed from the record of vertices to be studied (sv). Concept vertices add their children to sv. Since individual trees have a finite number of vertices, and cycles are not allowed, the algorithm `buildLoc` terminates. □

Theorem 9.4 (Correctness) *Algorithm* `buildLoc` *is correct regarding the specification of buidLoc (c.f. Subsection §8.6.2 for its definition).*

Proof It is immediate from the assumptions made at the beginning of this section. For each influence area in a semantic description, there exist only one mirrored influence area in the individual tree; furthermore, all of the attributes at the same hierarchical slot are distinct, which makes it possible to unambiguously compute the positions and hierarchical levels for all of the filler vertices in the labelled tree of the semantic description. □

9.3.3 Complexity

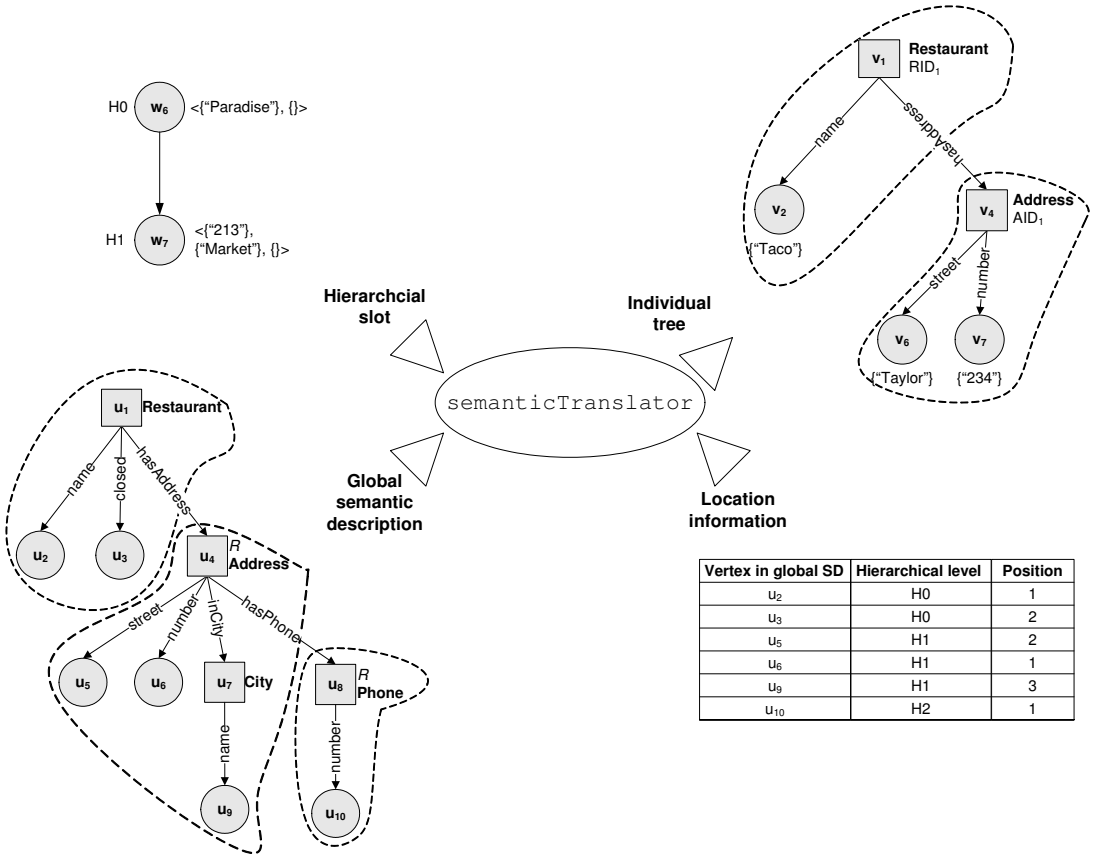
Algorithm `buildLoc` visits all of the vertices in the semantic description, and for each filler vertex, it looks for its attribute in the hierarchical slot; the location information is associated with the mirrored vertex in the semantic description. Therefore, the complexity of `buildSD` is $O(n \cdot m \cdot d)$ in the worst case, where n is the number of vertices in the individual tree, m is the number of vertices in the hierarchical slot, and d is the depth of the semantic description labelled tree.

9.4 Semantic translator

The algorithm `semanticTranslator` takes an *StructuredInformation* as input and it uses a global semantic description and the location information, to output an Abox that represents this information semantically. The algorithm is presented in terms of translating a hierarchical slot, then, it is iteratively applied to all of the hierarchical slots in a *StructuredInformation*.

9.4.1 Algorithm

It is implemented as a recursive algorithm based on the idea that each vertex in a hierarchical slot is associated to a mirrored influence area, and that each mirrored influence area can be represented as an individual tree. Recursion is applied over hierarchical slots. The stopping condition is that the hierarchical slot vertex being studied has not any children, in which case, the individual tree corresponding to its associated mirrored influence area is returned. The recursive call happens if the hierarchical slot vertex being studied



Vertex in global SD	Hierarchical level	Position
u_2	H0	1
u_3	H0	2
u_5	H1	2
u_6	H1	1
u_9	H1	3
u_{10}	H2	1

Figure 9.5: *Example of how* `sematicTranslator` *works.*

has children, in which case, the individual tree corresponding to the mirrored influence area of the parent vertex is attached to the individual trees obtained recursively for its children.

Example 9.4 *Figure §9.5 illustrates the individual tree obtained for a given global semantic description, a location information, and a hierarchical slot. Note that the individual tree has two mirrored influence areas that are obtained by cloning the influence areas represented by vertex u_1 and u_4 in the semantic description; furthermore, not all of the vertices and edges in one influence need to be cloned, e.g., vertex u_3 is not cloned because there is not information about the days the corresponding restaurant closes; finally, there are not any mirrored influence areas for the influence area represented by u_8 because the hierarchical slot does not contain any information about phones.*

Configurations: Three types of configurations are needed to define the semantic translation algorithm. Their meaning is shown later.

i. $Config_{ST_1} == SemanticDescription \times HierarchicalSlot \times Location \times Vertex$

ii. $Config_{ST_2} == IndividualTree$

iii. $Config_{ST_3} == SemanticDescription \times HierarchicalSlot \times Location \times \mathbb{F}\, Vertex \times \mathbb{F}\, IndividualTree \times \text{seq}\, Label$

Predicates and functions:

i. Function *cloneInfluenceArea* takes a semantic description, a hierarchical slot, a location information, and a vertex from the previous hierarchical slot as input and outputs an individual tree. It obtains the individual tree by cloning a set of nodes of the labelled tree in the semantic description and by computing the *vertexIndividualName* and *vertexAttribute* functions for cloned vertices. Vertices to be cloned are returned by function *verticesToClone*. *vertexIndividualName* is obtained by the function *getIdentifier*. *vertexAttribute* is obtained from the location information and the hierarchical slot. These functions are specified as follows:

$$
\begin{array}{|l}
cloneInfluenceArea : SemanticDescription \times HierarchicalSlot \times Location \times \\
\qquad\qquad\qquad\qquad Vertex \rightarrow IndividualTree \\
\hline
\forall\, sd : SemanticDescription;\ hs : HierarchicalSlot;\ loc : Location;\ w : Vertex; \\
\quad it : IndividualTree \bullet cloneInfluenceArea(sd, hs, loc, w) = it \Leftrightarrow \\
\qquad \exists f : verticesToClone(sd, hs, loc, hs.vertexHLevel(w), w) \rightarrowtail\!\!\!\!\!\!\twoheadrightarrow it.lt.t.vertices \bullet \\
\qquad\quad \forall (u_1, u_2) : sd.lt.t.edges \mid u_1 \in \text{dom}\, f \land u_2 \in \text{dom}\, f \bullet \\
\qquad\qquad (f(u_1), f(u_2)) \in it.lt.t.edges \land \\
\qquad\qquad sd.lt.edgeType(u_1, u_2) = it.lt.edgeType(f(u_1), f(u_2)) \land \\
\qquad\quad \forall\, u : \text{dom}\, f \bullet \\
\qquad\qquad sd.lt.vertexType(u) = it.lt.vertexType(f(u)) \land \\
\qquad\qquad (sd.lt.vertexType(u) = FILLER \land \\
\qquad\qquad\quad it.vertexAttribute(f(u)) = \\
\qquad\qquad\qquad hs.vertexAttributes(w)(loc.vertexPosition(u)) \lor \\
\qquad\qquad sd.lt.vertexType(u) \neq FILLER \land \\
\qquad\qquad\quad it.vertexIndividualName(f(u)) = getIdentifier(it, f(u)))
\end{array}
$$

Function *verticesToClone* outputs the set of vertices in a semantic description that belong to the influence area associated to a hierarchical slot

vertex, so that they are filler vertices to which the location information associates a non-empty attribute or are concept vertices that belong to a path that ends in a vertex that has a reference to a non-empty attribute as location information. It is specified as follows:

$$
\begin{array}{|l}
verticesToClone : SemanticDescription \times HierarchicalSlot \times Location \times Label \times \\
\qquad\qquad Vertex \rightarrow \mathbb{F}\, Vertex \\
\hline
\forall\, sd : SemanticDescription;\ hs : HierarchicalSlot;\ loc : Location;\ lb : Label; \\
\quad w : Vertex \bullet verticesToClone(sd, lb, w) = \\
\qquad \{v : influenceAreaLabel(sd, loc, lb) \mid sd.lt.vertexType(v) = FILLER \bigwedge \\
\qquad\quad hs.vertexAttributes(w)(l.vertexPosition(v)) \neq \varnothing\} \cup \\
\qquad \{v : influenceAreaLabel(sd, loc, lb) \mid sd.lt.vertexType(v) \neq FILLER \bigwedge \\
\qquad\quad (\exists\, p : paths(sd.lt.t);\ v_x : Vertex \mid v_x = p(\#p) \bigwedge v \in ran(p) \bullet \\
\qquad\qquad hs.vertexAttributes(w)(loc.vertexPosition(v_x)) \neq \varnothing)\}
\end{array}
$$

Function *influenceAreaLabel* outputs the vertices in a semantic descriptions that belong to the influence area associated to a given label.

$$
\begin{array}{|l}
influenceAreaLabel : SemanticDescription \times Location \times Label \rightarrow \mathbb{F}\, Vertex \\
\hline
\forall\, sd : SemanticDescription;\ loc : Location;\ lb : Label;\ su : \mathbb{F}\, Vertex \bullet \\
\quad influenceAreaLabel(sd, loc.lb) = su \Leftrightarrow \\
\qquad influenceArea(sd, su) \bigwedge \\
\qquad \exists\, u : su \mid sd.lt.vertexType(u) = FILLER \bullet loc.vertexLevel(u) = lb
\end{array}
$$

Function *getIdentifier* uses a hashing algorithm to provide a unique identifier name for a concept vertex. The hashing algorithm takes the attributes of filler vertices in individual tree that are reachable from the concept vertex as input. Note that the hashing definition is just a semantic characterisation, and that it is not intended to be executable since we think that this topic is well-covered in the literature. It is specified as follows:

$$
\begin{array}{|l}
getIdentifier : IndividualTree \times Vertex \rightarrow IndividualName \\
\hline
\forall\, it : IndividualTree;\ v : Vertex \mid v \in it.lt.t.vertices \bullet getIdentifier(it, v) = \\
\quad hashing(\bigcup\{p : paths(it.lt.t) \mid v \in \operatorname{ran} p \bullet it.vertexAttribute(p(\#p))\})
\end{array}
$$

$$
\begin{array}{|l}
hashing : X \rightarrow IndividualTree \\
\hline
\forall\, x, y : X \mid x \neq y \bullet hashing(x) \neq hashing(y)
\end{array}
$$

ii. Function *attach* takes a semantic description, a location information, an individual tree, a label, a sequence of individual trees, and a sequence of labels as input, and it outputs a new individual tree. The individual tree represents a mirrored influence area (of the influence area identified by the label) obtained from a vertex in the hierarchical slot; the sequence of individual trees represent the mirrored influence areas (of the influence areas identified by the sequence of labels) of the children of the previous vertex. Attaching is done by adding new edges that connect the root vertex of the individual tree with the root vertices of all of the individual trees in the sequence. The property types of the new edges are returned by function *propertyType*. Function *attach* is defined as follows:

$$
\begin{array}{l}
attach : semanticDescription \times Location \times IndividualTree \times Label \times \\
\qquad \text{seq}\, IndividualTree \times \text{seq}\, Label \rightarrow IndividualTree \\
\hline
\forall\, sd : semanticDescription;\ loc : Location;\ it, it_x : IndividualTree;\ lb : Label; \\
\quad sit : \text{seq}\, IndividualTree;\ slb : \text{seq}\, Label \bullet attach(sd, it, lb, sit, slb) = it_x \Leftrightarrow \\
\qquad it_x.lt.t.vertices = it.lt.t.vertices \cup \bigcup\{ita : \text{ran}\, sit \bullet ita.lt.t.vertices\} \\
\qquad it_x.lt.t.edges = it.lt.t.edges \cup \bigcup\{ita : \text{ran}\, sit \bullet ita.lt.t.edges\} \cup \\
\qquad\quad \{ita : \text{ran}\, sit \bullet (root(it.lt.t), root(ita.lt.t))\} \\
\qquad it_x.lt.vertexType = it.lt.vertexType \cup \bigcup\{ita : \text{ran}\, sit \bullet ita.lt.vertexType\} \\
\qquad it_x.lt.edgeType = it.lt.edgeType \cup \bigcup\{ita : \text{ran}\, sit \bullet ita.lt.edgeType\} \cup \\
\qquad\quad \{i : 1 \,..\, \#sit \bullet (root(it.lt.t), root(ita.lt.t)) \mapsto propertyType(sd, loc, lb, slb(i))\} \\
\qquad it_x.lt.vertexIndividualName = it.lt.vertexIndividualName \cup \\
\qquad\quad \bigcup\{ita : \text{ran}\, sit \bullet ita.lt.vertexIndividualName\} \\
\qquad it_x.lt.vertexAttribute = it.lt.vertexAttribute(v) \cup \\
\qquad\quad \{ita : \text{ran}\, sit \bullet ita.lt.vertexAttribute\}
\end{array}
$$

Function *propertyType* takes a semantic description and two labels as input, and outputs the type of property in the semantic description that connects the influence areas identified by these labels. The *propertyType* function is specified as follows:

$$
\begin{array}{l}
propertyType : SemanticDescription \times Location \times Label \times Label \rightarrow PropertyName \\
\hline
\forall\, sd : SemanticDescription;\ loc : Location;\ lb_1, lb_2 : Label;\ u_1, u_2 : Vertex \mid \\
\quad u_1 = subtreeRoot(influenceAreaLabel(sd, loc, lb_1)) \bigwedge \\
\quad u_2 = subtreeRoot(influenceAreaLabel(sd, loc, lb_2)) \bullet \\
\qquad propertyType(sd, lb_1, lb_2) = sd.lt.edgeType(u_1, u_2)
\end{array}
$$

Rules: The `semanticTranslator` algorithm is defined by the following rules, which are mutually recursive:

i. For a vertex w with children in the hierarchical slot, we obtain the individual tree that represents its mirrored influence area by cloning the semantic description influence area associated to the hierarchical level to which w belongs (*cloneInfluenceArea*); furthermore, it is attached to the individuals trees obtained for its children (*attach*). The individual trees (*sit*) for its children and their labels (*slb*) are obtained in the antecedent of the rule, c.f. the next rule for an explanation.

$$\frac{(sd, hs, loc, sw, \langle\,\rangle, \langle\,\rangle) \xrightarrow{!}_{ST_1} (sd, hs, loc, \varnothing, sit, slb)}{(sd, hs, loc, w) \rightarrow_{ST_2} it} \quad [\ sw \neq \varnothing\ ]$$

$$where \begin{cases} sw == children(hs, w) \\ itc == cloneInfluenceArea(sd, hs, loc, w) \\ lb == hs.vertexHLevel(w) \\ it == attach(sd, loc, itc, lb, sit, slb) \end{cases}$$

ii. For each children of a vertex in the hierarchical slot, the individual tree that represents a mirrored influence area is recursively computed. The individual trees are recorded in a sequence (*sit*), as well as the labels (*slb*) associated to them all.

$$\frac{(sd, hs, loc, w) \rightarrow_{ST_2} it}{(sd, hs, loc, sw, sit, slb) \rightarrow_{ST_1} (sd, hs, loc, sw', sit', slb')} \quad [\ sw \neq 0\ ]$$

$$where \begin{cases} sit' == sit \frown \langle it \rangle \\ slb' == slb \frown \langle hs.vertexHLevel(w) \rangle \\ w == member(sw) \\ sw' == sw \setminus \{w\} \end{cases}$$

iii. If w has not any children, then the individual tree is obtained by cloning the influence area of the semantic description that is associated to the hierarchical level to which w belongs.

$$(sd, hs, loc, w) \rightarrow_{ST_2} it \quad [\ sw = \varnothing\ ]$$

$$where \begin{cases} sw == children(hs, w) \\ it == cloneInfluenceArea(sd, HS, loc, w) \end{cases}$$

Algorithm: The algorithm is specified as the process of applying the rule $\rightarrow_{ST_2}$ for each *HierarchicalSlot* in an *StructuredInformation*. The initial configuration is defined as $(sd, hs, loc, root(hs.t))$. The algorithm is specified as follows:

$$\texttt{semanticTranslator} : \mathit{StructuredInformation} \times \mathit{SemanticDescription} \times \mathit{Location} \rightarrow \mathit{Abox}$$

$$\forall\, si : \mathit{StructuredInformation};\ sd : \mathit{SemanticDescription};\ loc : \mathit{Location} \bullet$$
$$\texttt{semanticTranslator}(si, sd, loc) =$$
$$\bigcup \{hs : si;\ it : \mathit{IndividualTree} \mid (sd, hs, loc, root(hs.t)) \rightarrow_{ST_2} it \bullet toAbox(it)\}$$

9.4.2 Correctness

Theorem 9.5 (Termination) *Algorithm* `semanticTranslator` *terminates.*

Proof The algorithm traverses a hierarchical slot. It begins with the root vertex and children are studied recursively; every time the algorithm is required for a vertex without children, recursion ends. Since hierarchical slots have a finite number of vertices, and cycles are not allowed, the algorithm `semanticTranslator` terminates. □

Theorem 9.6 (Correctness) *The* `semanticTranslator` *algorithm is correct regarding the specification of the semanticTranslator function (c.f. Section §8.7 for its definition).*

Proof The proof follows from how the algorithm has been constructed. According to predicate *translation*, each vertex in a hierarchical slot is associated to a mirrored influence area. Filler vertices in an individual tree take their attributes from the location information provided by its mirrored vertex in the corresponding semantic description. Furthermore, edges in a hierarchical slot are mapped onto those edges of the individual tree that connect vertices defining mirrored influence areas. □

9.4.3 Complexity

For each vertex with children in a hierarchical slot, the algorithm computes its associated influence area, whose upper bound should be a constant time denoted as *tc*, and attaches the mirrored influence area of the parent vertex to the individual trees obtained for its children recursively, whose upper bound should be a constant time denoted as *ta*. If a slot vertex has not any children, the individual tree corresponding to the mirrored influence area is returned

(tc). In the worst case, the hierarchical slot tree is a balanced k-ary tree, i.e., the same number of recursive calls is made from each vertex. The temporal function for complexity of `semanticTranslator` algorithm is defined recursively as:

$$T(m) = \begin{cases} k \cdot T(\frac{m}{k}) + tc + ta & \texttt{if } m > 1 \\ \\ tc & \texttt{if } m = 1 \end{cases}$$

Where m is the number of vertices of a hierarchical slot, and k is the arity of the corresponding tree.

To solve this equation, we calculate the time contribution of vertices at the same depth in hierarchical slot. At depth i, the number of vertices in a hierarchical slot is k^i. The time contribution of each vertex is $\frac{m}{k^i} + tc + ta$; then, the total time contribution of vertices at depth i is $(\frac{m}{k^i} + tc + ta) \cdot k^i$. The depth of the tree is $\lg_k m$, so if $c = tc + ta$, then:

$$T(m) = \sum_{i=0}^{\lg_k m} (m + c \cdot k^i) = \sum_{i=0}^{\lg_k m} m + c \cdot \sum_{i=0}^{\lg_k m} k^i = m \cdot \lg_k m + c \cdot (\frac{k^{\lg_k m + 1} - 1}{k - 1})$$

Therefore, the complexity of `semanticTranslator` algorithm is approximated by $O(m \cdot \lg_k m + k^{\lg_k m}) = O(m \cdot \lg_k m + m) = O(m \cdot \lg m)$ in the worst case.

9.5 Summary

In this chapter, we have presented a materialisation for semantic translation. We have devised three algorithms called `buildSD`, `buildLoc`, and `semanticTranslation`. The correctness of these algorithms has been asserted with respect to the specification in the previous chapter, which proves that our approach for semantic translation can be implemented. Furthermore, we have shown that they are quite efficient in practice.

Chapter 10

A proof-of-concept implementation

If it's your idea, you get to implement it.

Leland E. Modesitt, Jr., 1943–
Science fiction and fantasy writer

In this chapter, we present a realisation of the WebMeaning framework. It is organised as follows: in Section §10.1, we introduce the main ideas; Section §10.2 presents the WebMeaning architecture; Section §10.3 briefly describes its realisation; finally, Section §10.4 summarises the main ideas.

10.1 Introduction

The proof-of-concept implementation of the WebMeaning framework provides support for software engineers by defining a comprehensive architecture. It maps the core elements identified in Chapter §7 onto software elements that cooperatively implement the functionality defined in the framework.

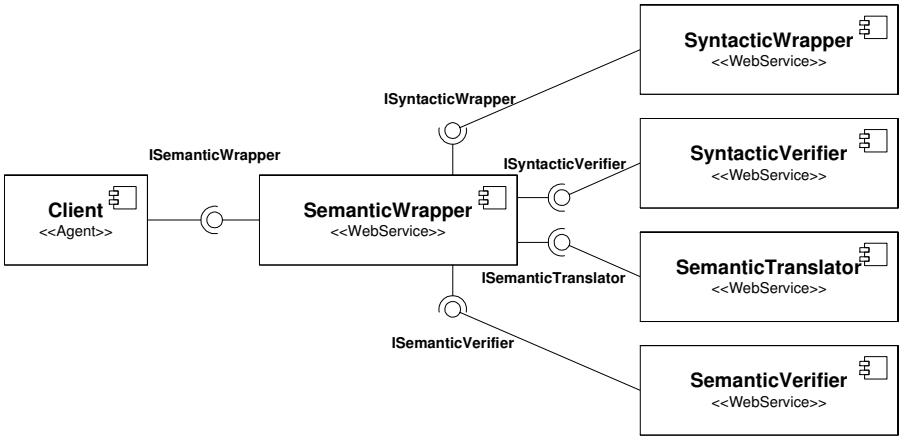
The design criterion for implementing the WebMeaning framework was the principle of modular composability, that is, the development of syntactic wrappers, syntactic verifiers, semantic translators, and semantic verifiers in such a way that they may be freely integrated as a whole to produce semantic translators. Thus, the implementation allows to substitute them very easily, which reduces the impact of changes, improves reusability, and leads to a better return on the investment. The criterion was achieved by using a Service-Oriented Architecture (SOA) in which each part of the system was implemented using web services.

We describe the implementation by means of UML 2.0 since this notation has proved to be valuable in representing models for software development [101]. Our main goal is to provide an understanding, but we do not present many details since they fall outside the scope of this dissertation. The full implementation of WebMeaning is available at `http://www.tdg-seville.info/tools/WebMeaning`.

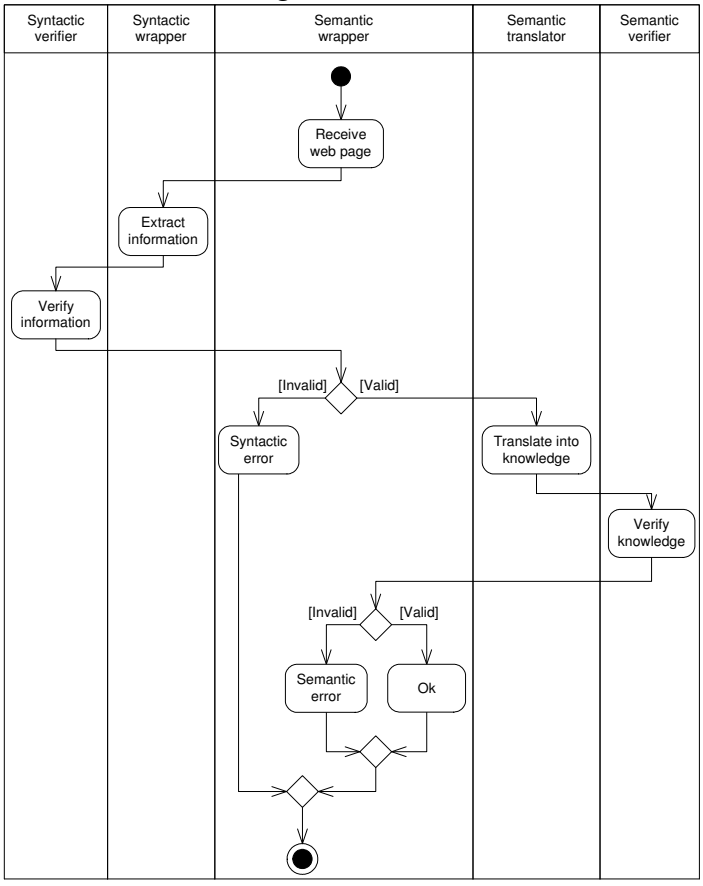
10.2 The architecture

In Chapter §7, we divide the functionality of a knowledge extractor into several chunks. The WebMeaning architecture maps them onto web services. Figure §10.1(a) illustrates this idea. The semantic wrapper is mapped onto a composite service that acts as a façade for the rest of elements, which are implemented as individual web services. The semantic wrapper uses them to extract syntactic information (*SyntacticWrapper*), to verify the extracted syntactic information (*SyntacticVerifier*), to translate the information into knowledge (*SemanticTranslator*), and to verify the knowledge (*SemanticVerification*). In addition, Figure §10.1(b) illustrates an activity diagram that explains the interaction between these elements.

Separation of issues is enforced by requiring these web services to interact only via a defined set of public facilities (their interfaces). A semantic wrapper requires *ISyntacticWrapper*, *ISyntacticVerifier*, *ISemanticTranslator*, and *ISemanticVerifier* interfaces to orchestrate the elements implementing them. A



(a) WebMeaning elements as web services



(b) Activities in knowledge extraction

Figure 10.1: *The WebMeaning architecture.*

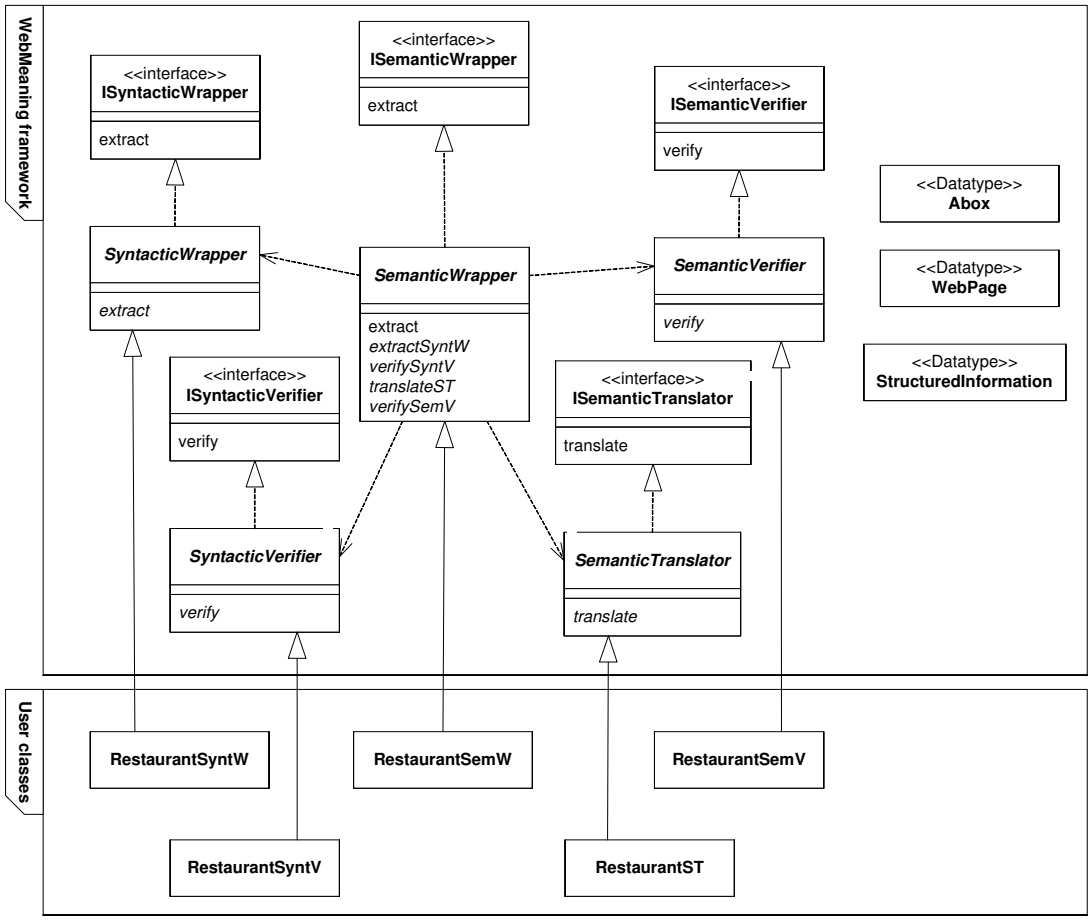


Figure 10.2: *A realisation of WebMeaning.*

semantic wrapper provides the interface *ISemanticWrapper*, which is the only means for a software agent to extract knowledge from a web page.

10.3 Realisation

The class diagram in Figure §10.2 illustrates the logic view of the realisation of the WebMeaning architecture. It is divided into two parts: the former corresponds to the WebMeaning core classes and interfaces, whereas the latter sketches user classes that extend the framework. In addition, Figure §10.3 illustrates the sequence diagrams for three distinct scenarios: the first represents a scenario in which the knowledge is correctly extracted; the second represents a syntactic fail, in which case neither the semantic translator nor the

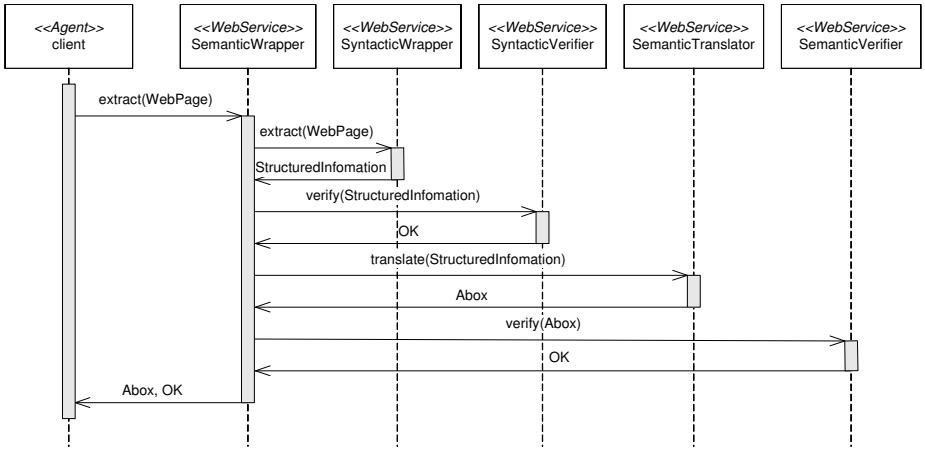
semantic verifier are invoked by the semantic wrapper; the last corresponds to a semantic fail.

Web services are specified as core abstract classes that implement the interfaces they provide. Abstract methods *extractSyntW*, *verifySyntW*, *translateST*, *verifySemV* of abstract class *SemanticWrapper*, encapsulate the invocation of *extract* in a *SyntacticWrapper*, *verify* in a *SyntacticVerifier*, *translate* in a *SemanticTranslator*, and *verify* in a *SemanticVerifier*, respectively. The method *extract* of abstract class *SemanticWrapper* is a template method [49].

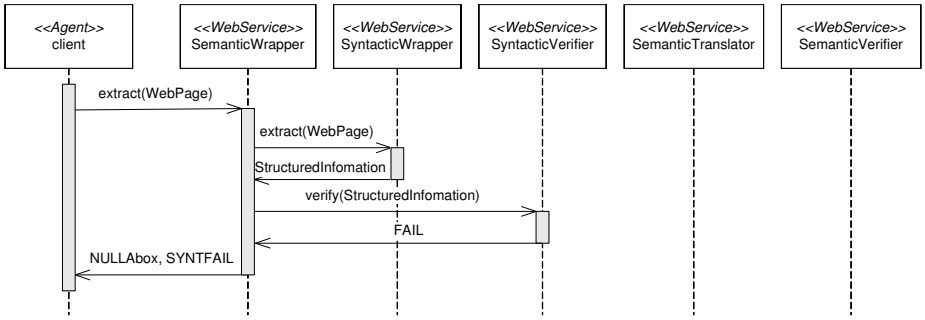
Users are able to extend the WebMeaning framework by specialising abstract classes. The extension is controlled by forcing them to implement the interfaces associated to the abstract classes. Reusing is enforced by the whole-part relation of web services as containers of the required functionality for extracting knowledge; for instance, a user can define two semantic wrappers to extract information about the some concern from two different web sites, and they might orchestrate two different syntactic wrappers, but the same syntactic verifier, semantic translator, and semantic verifier.

10.4 Summary

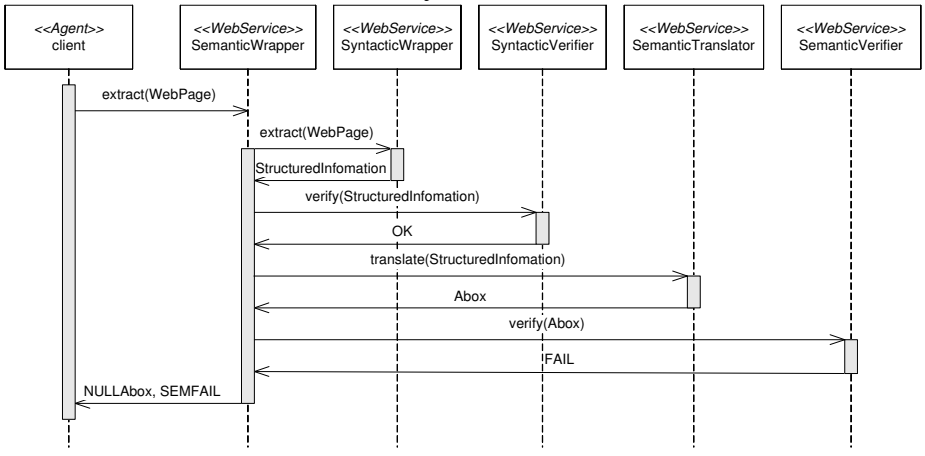
In this chapter, we have sketched a proof-of-concept implementation of WebMeaning. Given the definition of WebMeaning framework in Chapter §7, we devised a comprehensive reference architecture based on web services.



(a) Knowledge correctly extracted



(b) Syntatic fail



(c) Semantic fail

Figure 10.3: *Three distinct scenarios in WebMeaning.*

Part IV

Final remarks

Chapter 11

Conclusions and future work

When people agree, it is only in their conclusions; their reasons are always different.

Jorge A.N. de Santayana, 1863–1952
Spanish philosopher

The incredible successfulness of the Internet world has paved the way for technologies whose goal is to enhance the way humans and computers interact on the web. Unfortunately, the information a human user can easy interpret is usually difficult to be extracted and interpreted by a software agent. This is the reason why such enhancements are usually viewed as problems from an agent programmer's point of view. The semantic web shall help extract information with well-defined semantics, regardless of the way it is rendered, but it does not seem it is going to be adopted in the immediate future, which argues for another solution to the problem in the meanwhile.

The goal of this dissertation was to support the idea that information residing on web pages is not understood by software agents at a sensible cost. In the body of this dissertation, we presented strong motivation for this idea, and described the problems that appear when an agent is willing to retrieve the knowledge on web. These problem were due to the fact that the current web is mostly user-oriented, changing, huge and distributed (c.f. Chapter §6 for the description of these problems and the proof that none of the existing proposals succeeded in addressing them at a time). WebMeaning is our approach to extract semantically-meaningful information from today's non-semantic web. Its main advantages are that it associates semantics with the information extracted, which improves agent interoperability (c.f. Chapter §8

for our proposal for semantic translation); it can also deal with changes to the structure of a web page, which improves adaptability (c.f. Chapter §7 for how WebMeaning embraces web changes by using syntactic verifiers), and it achieves a complete separation of issues in the task of knowledge extraction, automating the development of distributed knowledge extractors (c.f. Chapters §9 and §10 for the automatic algorithms and the services-based architecture of WebMeaning, respectively).

Anyhow, the results in this dissertation cannot be seen as the concluding end of a path, but as the motivation for further research on this topic. Amongst the many issues that remain open or can be improved, we think that the most exciting is to extend the framework so as to have access to heterogeneous information sources, e.g., federated databases or knowledge bases. Accessing to database implies translating the relational model into an ontology, but in a federated environment there can be many ontologies for the same domain. In order to achieve semantic interoperability, these different ontologies must be able to interoperate or integrate with each other. Having access to knowledge bases implies defining translation schemes between ontologies, and the same problem appears, i.e., semantic interoperability.

Part V

Appendices

Appendix A

Mathematical notes

A.1 Notation

The Z formal specification language is based on set theory and first-order predicate calculus [1]. It extends the use of these languages by allowing an additional mathematical type known as the schema type. Z schemas have two parts: the upper declarative part, which declares variables and their types, and the lower predicate part, which relates and constrains those variables. The type of any schema can be considered as the Cartesian product of its variables, without any notion of order, but constrained by the predicates. Modularity is facilitated in Z by allowing schemas to be included within other schemas. We can select a variable of an schema instance by writing *schemaInstance.var*.

To introduce a type in which we wish to abstract away from the actual elements of the type, we use the notion of a given set. We write $[Vertex]$ to represent the set of all vertices. If we wish to state that a variable ranges over some finite set of values or an ordered pair of values we write $x : \mathbb{F}\, Vertex$ and $x : Vertex \times Vertex$, respectively.

A summary of the notation to be used is given in Table §A.1. For a more complete treatment of the Z language, the reader is referred to one of the numerous texts, c.f. Refs. [35, 70, 103]

A.2 Plotkin's method

We use the popular Plotkin's method to define our algorithms [102] since it is simple, yet powerful. It relies on inference rules of the following forms:

Notation	Description
Definitions and declarations	
$a : A$	Declarations
$A == B$	Abbreviated definition
Logic	
$p \land q$	Logical conjunction
$p \lor q$	Logical disjunction
$\forall X \bullet p$	Universal quantification
$\exists_1 X \bullet p$, $\exists X \bullet p$	(Unique) existencial quantification
Sets	
$x \in A$	Set membership
$\varnothing$	Empty set
$A \subseteq B$	Set inclusion
$\{x, y, \ldots\}$	Set of elements
(x, y)	Ordered pair
$A \times B$	Cartesian product
$\mathbb{F} A$	Finite set
$A \cap B$	Set intersection
$A \cup B$	Set union
$\bigcup A$	Generalised or distributive union
$\#A$	Size of a finite set
Sequences	
$\langle x, y, \ldots \rangle$	Sequence of elements
$\langle\,\rangle$	Empty sequence
$tail\, s$	Tail of a sequence
$head\, s$	Head of a sequence
$s \frown t$	Sequence concatenation
$\{a : A \mid P(a) \bullet f(a)\}$	Set comprehension
Relations and functions	
$\mathrm{dom}\, R$	Domain of a relation
$\mathrm{ran}\, R$	Range of a relation
$A \pfun B$	Partial function
$A \fun B$	Total function
$A \rightarrowtail B$	Total bijection
$\mathrm{dom}\, R$	Domain of a relation
$\overset{!}{\fun}$	Normalisation of a relation

Table A.1: *Summary of the notation used in this dissertation.*

$$\frac{\textit{Antecedent}}{\textit{Consequent}} \left[\begin{array}{l} \textit{Applicability} \\ \textit{conditions} \end{array} \right] \qquad \textit{Consequent} \left[\begin{array}{l} \textit{Applicability} \\ \textit{conditions} \end{array} \right]$$

$$\textit{where} \left\{ \textit{Definitions} \right. \qquad \textit{where} \left\{ \textit{Definitions} \right.$$

To define an algorithm using this method, it is necessary to identify the data on which it works and model it as a tuple that is usually referred to as the configuration of the algorithm.

For instance, to model a simple producer/consumer system, we need configurations of the form (p, c, τ), where p denotes the state of the producer, c the state of the consumer, and τ is a fixed-sized queue that helps store the items that are ready to be consumed. Obviously, we also need a couple of rules denoted as $\rightarrow_{PROD}$ and $\rightarrow_{CONS}$ to describe how the producer and the consumer change their state, respectively. These rules may be left unspecified since the abstraction level at which we are describing the system does not require in-depth knowledge of their semantics. Using this information, the system can be described by means of the following rules:

The production rule. It controls how new items are stored in the queue as long if there is room for them. (We assume that MAX denotes the maximum number of items τ can store.)

$$\frac{p \xrightarrow{i}_{PROD} p'}{(p, c, \tau) \rightarrow_{PC} (p', c', \tau')} \quad \left[\mid \tau \mid < MAX \right]$$

$$\textit{where} \left\{ \begin{array}{l} c' == c \\ \tau' == \textit{enqueue}(\tau, i) \end{array} \right.$$

$p \xrightarrow{i}_{PROD} p'$ means that the producer works locally to produce an item denoted as i, and this makes it to transit from state p to state p'. Note that the rule can be applied as long as $\mid \tau \mid < MAX$; otherwise, the producer has to wait until the consumer removes an item from the queue.

The consumption rule. It controls how items are removed from the queue and consumed.

$$\frac{c \xrightarrow{i}_{CONS} c'}{(p, c, \tau) \rightarrow_{PC} (p', c', \tau')} \quad \left[\mid \tau \mid > 0 \bigwedge i = \textit{Head}(\tau) \right]$$

$$\textit{where} \left\{ \begin{array}{l} p' == p \\ \tau' == \textit{dequeue}(\tau) \end{array} \right.$$

The burglar rule. It models a burglar who steals an item from the queue. This rule has not an antecedent.

$$(p, c, \tau) \rightarrow_{BURGLAR} (p, c, \tau') \quad [\ |\tau| > 0\ ]$$

$$\textit{where} \quad \{\ \tau' == \textit{dequeue}(\tau)$$

A.3 The *Tree* data type

The *Tree* data type describes a tree as a set of vertices and a set of edges. The assertions in the predicate part of the schema constrain that there must not be any cycles (there is no path that includes at least one edge that can return to the starting vertex), and that each must be in a path to the root.

$$[\textit{Vertex}]$$
$$\textit{Edge} == \textit{Vertex} \times \textit{Vertex}$$

Tree

$$\textit{vertices} : \mathbb{F}\ \textit{Vertex}$$
$$\textit{edges} : \mathbb{F}\ \textit{Edge}$$

$$\forall (v_1, v_2) : \textit{edges} \bullet v_1 \in \textit{vertices} \land v_2 \in \textit{vertices}$$
$$\forall p : \textit{paths}(\langle\!|\ \textit{vertices} \rightsquigarrow \textit{vertices}, \textit{edges} \rightsquigarrow \textit{edges}\ |\!\rangle);\ i, j : \mathbb{N} \mid i, j : 1 \mathinner{..} (\#p) \land i \neq j \bullet p(i) \neq p(j)$$
$$\forall v : \textit{vertices} \bullet \exists p : \textit{paths}(\langle\!|\ \textit{vertices} \rightsquigarrow \textit{vertices}, \textit{edges} \rightsquigarrow \textit{edges}\ |\!\rangle) \bullet v \in \operatorname{ran} p$$

Next, we define a predicate and some functions on this data type that are commonly used in this dissertation:

i. Predicate *isLeaf* holds if a vertex in a given tree is a leaf.

$$\textit{isLeaf} : \textit{Tree} \times \textit{Vertex}$$

$$\forall t : \textit{Tree};\ v : t.\textit{vertices} \bullet \textit{isLeaf}(t, v) \Leftrightarrow \nexists v_x : t.\textit{vertices} \bullet (v, v_x) \in t.\textit{edges}$$

ii. Function *root* returns the root of a tree.

$$\textit{root} : \textit{Tree} \rightarrow \textit{Vertex}$$

$$\forall t : \textit{Tree};\ v : \textit{Vertex} \bullet \textit{root}(t) = v \Leftrightarrow \nexists v_x \bullet (v_x, v) \in t.\textit{edges}$$

iii. Function *children* returns the children (set of vertices) in a tree for a given tree and vertex.

$$children : Tree \times Vertex \pfun \mathbb{F}\, Vertex$$
$$\forall\, t : Tree;\ v : t.vertices \bullet children(t, v) = \{v_x : Vertex \mid (v, v_x) \in t.edges \bullet v_x\}$$

iv. Function *paths* returns all of the paths in a tree.

$$paths : Tree \fun \mathbb{F}\, \mathrm{seq}\, Vertex$$
$$\forall\, t : Tree \bullet paths(t) = \{p : \mathrm{seq}\, Vertex \mid p(1) = root(t) \land isLeaf(t, p(\#p)) \land (\forall\, i : 1 \,..\, \#p - 1 \bullet (p(i), p(i+1)) \in t.edges) \bullet p\}$$

v. Function *subtreeRoot* outputs the root of a subtree. Subtree is represented as a set of vertices.

$$subtreeRoot : Tree \times \mathbb{F}\, Vertex \fun Vertex$$
$$\forall\, t : Tree;\ sv : \mathbb{F}\, Vertex;\ v : Vertex \bullet subtreeRoot(t, sv) = v \Leftrightarrow \forall\, v_x : sv \setminus \{v\} \bullet \exists\, p : paths(t);\ i, j : \mathbb{N} \bullet p(i) = v \land p(j) = v_x \land j > i$$

Appendix B

Equivalence between Aboxes and IndividualTrees

Before proving the equivalence between *Abox*es and *IndividualTree*s we introduce a couple of helpful functions that allow to differentiate between the property assertions that establish relationships amongst concept instances and those that give values to attributes associated with concepts, namely: function *getRelations* takes an *Abox* as input and outputs the property assertions with individual names as fillers (tuples in *propertyAssertions* with an element from *IndividualName* at the third position); on the contrary, function *getAttributes* outputs property assertions with literals as fillers (tuples in *propertyAssertions* with a literal at the third position). They are specified as follows:

$$
\begin{array}{|l}
getRelations : Abox \rightarrow \mathbb{F}(PropertyName \times IndividualName \times IndividualName) \\
\hline
\forall\, a : Abox \bullet getRelations(a) = \\
\quad \{x : a.pn;\ y, z : a.IndividualNames \mid (x, y, z) \in a.propertyAssertions \bullet (x, y, z)\}
\end{array}
$$

$$
\begin{array}{|l}
getAttributes : Abox \rightarrow \mathbb{F}(PropertyName \times IndividualName \times Literal) \\
\hline
\forall\, a : Abox \bullet getAttributes(a) = \\
\quad \{x : a.pn;\ y : a.IndividualNames;\ l : Literal \mid (x, y, l) \in a.propertyAssertions \bullet (x, y, l)\}
\end{array}
$$

Note that, if a is an *Abox*, then $getRelations(a)$ and $getAttributes(a)$ define a partition of $a.propertyAssertions$.

Example B.1 *The outputs of getRelations and getAttributes for the Abox in Example §7.2, page 69 are as follows:*

$$\textbf{\textit{getRelations}}(a) = \{(hasAddress, RID_1, AID_1), (inCity, AID_1, CID_1), (hasAddress, RID_1, AID_2), (inCity, AID_2, CID_2), (hasPhone, AID_2, PID_1), (hasPhone, AID_2, PID_2)\}$$

$$\textbf{\textit{getAttributes}}(a) = \{(name, RID_1, \text{“Taco”}), (closed, RID_1, \text{“Monday”}), (closed, RID_1, \text{“Sunday”}), (street, AID_1, \text{“Taylor”}), (number, AID_1, \text{“234”}), (name, CID_1, \text{“Pittsburgh”}), (street, AID_2, \text{“Connecticut”}), (number, AID_2, \text{“150”}), (name, CID_2, \text{“Harrisburgh”}), (number, PID_1, \text{“2314800”}), (number, PID_2, \text{“2324800”})\}$$

B.1 Building an *IndividualTree* from an *Abox*

Theorem B.1 *Given an Abox with assertions about only one individual, there exists a unique IndividualTree that represents this individual.*

Proof To proof this theorem, we use a constructive approach that consists of the specification of an abstract algorithm that takes an *Abox* as input and outputs an *IndividualTree*. The algorithm is as follows:

i. For each concept assertion in the *Abox*, there is one concept vertex in the tree:

$$\begin{array}{l}\forall (c, id) : a.conceptAssertions \bullet \\ \quad \exists_1 v : Vertex \bullet \\ \qquad v \in it.lt.t.vertices \bigwedge \\ \qquad it.lt.vertexType(v) = c \bigwedge \\ \qquad it.vertexIndividualName(v) = id\end{array}$$

ii. For each property assertion (p, id_1, id_2) in $getRelations(a)$, there is one edge in the tree that connects the vertices obtained in step (i) for individual names id_1 and id_2:

$$\begin{array}{l}\forall (p, id_1, id_2) : getRelations(a) \bullet \\ \quad \exists_1 v_1, v_2 : Vertex \bullet \\ \qquad (v_1, v_2) \in it.lt.t.edges \bigwedge \\ \qquad it.lt.edgeType(v_1, v_2) = p \bigwedge \\ \qquad it.vertexIndividualName(v_1) = id_1 \bigwedge \\ \qquad it.vertexIndividualName(v_2) = id_2\end{array}$$

iii. For each set of properties with both the same property and individual names in $getAttributes(a)$, there is one edge in the tree that relates the vertex obtained in step (i) for individual name *id* with a new filler vertex:

$$
\begin{array}{l}
\forall\, p : PropertyName;\ id : IndividualName;\ att : Attribute \mid \\
\quad att = \{(p, id, l) : getAttributes(a) \bullet l\} \bigwedge att \neq \varnothing \bullet \\
\quad\quad \exists_1\, v_1, v_2 : Vertex \bullet \\
\quad\quad\quad (v_1, v_2) \in it.lt.t.edges \bigwedge \\
\quad\quad\quad it.lt.edgeType(v_1, v_2) = p \bigwedge \\
\quad\quad\quad it.vertexIndividualName(v_1) = id \bigwedge \\
\quad\quad\quad it.lt.vertexType(v_2) = FILLER \bigwedge \\
\quad\quad\quad it.vertexAttribute(v_2) = att
\end{array}
$$

Function *toIndividualTree* returns the *IndividualTree* corresponding to *Abox* as shown previously. It is specified as follows:

$$
\begin{array}{l}
toIndividualTree : Abox \rightarrow IndividualTree \\
\hline
\forall\, a : Abox;\ it : IndividualTree \bullet toIndividualTree(a) = it \Leftrightarrow \\
\quad \forall (c, id) : a.conceptAssertions \bullet \qquad \text{[i]} \\
\quad\quad \exists_1\, v : Vertex \bullet \\
\quad\quad\quad v \in it.lt.t.vertices \bigwedge \\
\quad\quad\quad it.lt.vertexType(v) = c \bigwedge \\
\quad\quad\quad it.vertexIndividualName(v) = id \bigwedge \\
\quad \forall (p, id_1, id_2) : getRelations(a) \bullet \qquad \text{[ii]} \\
\quad\quad \exists_1\, v_1, v_2 : Vertex \bullet \\
\quad\quad\quad (v_1, v_2) \in it.lt.t.edges \bigwedge \\
\quad\quad\quad it.lt.edgeType(v_1, v_2) = p \bigwedge \\
\quad\quad\quad it.vertexIndividualName(v_1) = id_1 \bigwedge \\
\quad\quad\quad it.vertexIndividualName(v_2) = id_2 \bigwedge \\
\quad \forall\, p : PropertyName;\ id : IndividualName;\ att : Attribute \mid \qquad \text{[iii]} \\
\quad\quad att = \{(p, id, l) : getAttributes(a) \bullet l\} \bigwedge att \neq \varnothing \bullet \\
\quad\quad\quad \exists_1\, v_1, v_2 : Vertex \bullet \\
\quad\quad\quad\quad (v_1, v_2) \in it.lt.edges \bigwedge \\
\quad\quad\quad\quad it.lt.edgeType(v_1, v_2) = p \bigwedge \\
\quad\quad\quad\quad it.vertexIndividualName(v_1) = id \bigwedge \\
\quad\quad\quad\quad it.lt.vertexType(v_2) = FILLER \bigwedge \\
\quad\quad\quad\quad it.vertexAttribute(v_2) = att \bigwedge \\
\quad (\#it.lt.t.Edges = \#getRelations(a) + \#\{(p, id, l) : getAttributes(a) \bullet (p, id)\}) \bigwedge \qquad \text{[iv]} \\
\quad (\#it.lt.t.Vertices = \#a.conceptAssertions + \#\{(p, id, l) : getAttributes(a) \bullet (p, id)\}
\end{array}
$$

Note that two new constraints are added at the end: the former states that the number of edges in the resulting tree is equal to the sum of the number of assertions obtained in steps (ii) and (iii); the latter states that the number of vertices in the tree equals the sum of concept assertions in the *Abox* and filler vertices obtained in step (iii). These constraints ensure that the *IndividualTree* obtained is unique (there does not exist another *IndividualTree* to represent this *Abox*). □

B.2 Building an *IndividualTree* from an *Abox*

Theorem B.2 *Given an IndividualTree there exists a unique Abox with assertions about only the individual represented in the IndividualTree.*

Proof Let *it* be an *IndividualTree*, then the corresponding *Abox* is built as follows:

i. For each concept vertex in *it*, there is one concept assertion in *a*.

$$\begin{array}{l} \forall\, v : it.lt.t.vertices \bullet \\ \quad \exists_1 (c, id) : a.conceptAssertions \bullet \\ \quad\quad c = it.lt.vertexType(v) \bigwedge \\ \quad\quad id = it.vertexIndividualName(v) \end{array}$$

ii. For each edge (v_1, v_2) in which v_2 is not a filler vertex, there is one property assertion in *a* between two concepts (the ones obtained in step (i) for vertices v_1 and v_2).

$$\begin{array}{l} \forall (v_1, v_2) : it.lt.t.edges \mid it.lt.vertexType(v_2) \neq FILLER \bullet \\ \quad \exists_1 (p, id_1, id_2) : a.propertyAssertions \bullet \\ \quad\quad p = it.lt.edgeType(v_1, v_2) \bigwedge \\ \quad\quad id_1 = it.vertexIndividualName(v_1) \bigwedge \\ \quad\quad id_2 = it.vertexIndividualName(v_2) \end{array}$$

iii. For each edge (v_1, v_2) in which v_2 is a filler vertex, the number of property assertions in *a* equals the number of literals in the attribute associated to v_2. These properties relate the same concept (the one obtained for v_1 in step (i)) with the different literals in the corresponding attribute.

$$\forall (v_1, v_2) : it.lt.t.edges \mid it.lt.vertexType(v_2) = FILLER \bullet$$
$$\exists (p, id, l) : a.propertyAssertions \bullet$$
$$p = it.lt.edgeType(v_1, v_2) \bigwedge$$
$$id = it.vertexIndividualName(v_1) \bigwedge$$
$$l \in it.vertexAttribute(v_2)$$

Function *toAbox* returns the *Abox* for an *IndividualTree* as showed previously. It is specified as follows:

$$toAbox : IndividualTree \rightarrow Abox$$

$$\forall it : IndividualTree;\ a : Abox \bullet toAbox(it) = a \Leftrightarrow$$
$$\forall v : it.lt.t.Vertices \bullet \quad \text{[i]}$$
$$\exists_1 (c, id) : a.ConceptAssertions \bullet$$
$$c = it.lt.vertexType(v) \bigwedge$$
$$id = it.vertexIndividualName(v) \bigwedge$$
$$\forall (v_1, v_2) : it.lt.t.edges \mid it.lt.vertexType(v_2) \neq FILLER \bullet \quad \text{[ii]}$$
$$\exists_1 (p, id_1, id_2) : a.propertyAssertions \bullet$$
$$p = it.lt.edgeType(v_1, v_2) \bigwedge$$
$$id_1 = it.vertexIndividualName(v_1) \bigwedge$$
$$id_2 = it.vertexIndividualName(v_2) \bigwedge$$
$$\forall (v_1, v_2) : it.lt.t.edges \mid it.lt.vertexType(v_2) = FILLER \bullet \quad \text{[iii]}$$
$$\exists (p, id, l) : a.propertyAssertions \bullet$$
$$p = it.lt.edgeType(v_1, v_2) \bigwedge$$
$$id = it.vertexIndividualName(v_1) \bigwedge$$
$$l \in it.vertexAttribute(v_2) \bigwedge$$
$$(\#it.lt.edges = \#getRelations(a) + \#\{(p, id, l) : getAttributes(a) \bullet (p, id)\}) \bigwedge \quad \text{[iv]}$$
$$(\#it.lt.vertices = \#a.ConceptAssertions + \#\{(p, id, l) : getAttributes(a) \bullet (p, id)\}$$

Again, two new constraints (iv) are added to ensure that the *IndividualTree* built is unique. □

Appendix C

Acronyms

BPEL4WS. Business Process Execution Language for Web Services.

CREAM. CREAtion of Metadata.

DAML. DARPA agent markup language.

DAML+OIL. DARPA Agent Markup Language plus OIL.

DARPA. Defense Advanced Research Projects Agency.

DTD. Document Type Definition.

HTML. Hypertext Markup Language.

HTTP. Hypertext Transfer Protocol.

INTHELEX. INcremental THEory Learner from EXamples.

KIF. Knowledge Interchange Format.

OIL. Ontology Interchange Language.

OML. Ontology Markup Language.

OWL. Web Ontology Language.

RACER. Renamed ABox and Concept Expression Reasoner.

RAPIER. Robust Automated Production of Information Extraction Rules.

RDF. Resource Description Framework.

RDF-S. Resource Description Framework Schema.

SHOE. Simple HTML Ontology Extensions.

SOAP. Simple Object Access Protocol.

SPEM. Software Process Engineering Metamodel.

SQL. Structured Query Language.

SRV. Sequence Rules with Validation.

SWWS. Semantic Web Enabled Web Services.

UDDI. Universal Description, Discovery and Integration.

UML. Unified Modelling Language.

URI. The Uniform Resource Identifier.

W3C. World Wide Web Consortium.

WebOntEx. Web Ontology Extraction.

WIEN. Wrapper Induction ENvironment.

WSDL. Web Service Definition Language.

XML. Extensible Markup Language.

XOL. Ontology Exchange Language.

Bibliography

[1] ISO/IEC 13568:2002. Z formal specification notation: syntax, type system and semantics, 2002. International Standard.

[2] H. Alani, S. Kim, D.E. Millard, M.J. Weal, W. Hall, P.H. Lewis, and N.R. Shadbolt. Automatic ontology-based knowledge extraction from web documents. *IEEE Intelligent Systems*, 18(1):14–21, 2003.

[3] A. Aldea, R. Bañares-Alcántara, J. Bocio, J. Gramajo, D. Isern, A. Kokossis, L. Jiménez, A. Moreno, and D. Riaño. An ontology-based knowledge management platform. In *Proceedings of Workshop on Information Integration on the Web (IIWEB'03) at IJCAI'03*, pages 177–182, 2003.

[4] A. Ankolekar, M. Burstein, J.R. Hobbs, O. Lassila, D. Martin, D. McDermott, S.A. McIlraith, S. Narayanan, M. Paolucci, T. Payne, and K. Sycara. DAML-S: web service description for the semantic web. In *Proceedings of the International Semantic Web Conference (ISWC'02)*, pages 348–363. Springer, 2002.

[5] N. Apte and T. Mehta. *UDDI: building registry-based web services solutions*. Prentice Hall, 2002.

[6] J.L. Arjona and R. Corchuelo. Extracción de información en una plataforma multiagente. In *Actas de la reunión de trabajo ZOCO*, pages 83–98, 2001.

[7] J.L. Arjona and R. Corchuelo. Coping with web knowledge. In *Proceedings of the 1st International Atlantic Web Intelligence Conference (AWIC'03)*, pages 165–178. Springer, 2003.

[8] J.L. Arjona, R. Corchuelo, and M. Toro. Automatic extraction of semantically-meaningful information from the web. In *Proceedings of the 2nd International Conference on Adaptive Hypermedia and Adaptive Web-Based Systems (AH'02)*, pages 24–35. Springer, 2002.

[9] J.L. Arjona, R. Corchuelo, and M. Toro. A practical agent-based method to extract semantic information from the web. In *Proceedings of the 14th International Conference on Advanced Information Systems Engineering (CAiSE'02)*, pages 697–700. Springer, 2002.

[10] J.L. Arjona, R. Corchuelo, and M. Toro. Knowledge channels. bringing the knowledge on the web to software agents. In *Proceedings of the short papers of the 15th Conference on Advanced Information Systems Engineering (CAiSE'03)*, pages 161–164. Technical University of Aachen, 2003.

[11] J.L. Arjona, R. Corchuelo, and M. Toro. A knowledge extraction process specification for today's non-semantic web. In *Proceedings of the IEEE/WIC International Conference on Web Intelligence (WI'03)*, pages 61–67. IEEE Computer Society, 2003.

[12] A.Y. and D.S. Weld. Intelligent internet systems. *Artificial Intelligence*, 118(12):1–14, 2000.

[13] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider. *The description logic handbook: theory, implementation and applications*. Cambridge University Press, 2003.

[14] T. Berners-Lee. WWW: past, present, and future. *Computer*, 29(10):69–77, 1996.

[15] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, 284(5):34–43, 2001.

[16] P. Bloodsworth, S. Greenwood, and J. Nealon. A generic model for distributed real-time scheduling based on dynamic heterogeneous data. In *Proceedings of the 6th Pacific Rim International Workshop on Multi-Agents (PRIMA'01)*, pages 110–121. Springer, 2003.

[17] W.N. Borst. *Construction of Engineering Ontologies*. PhD thesis, University of Twente, 1997.

[18] R.J. Brachman. On the epistemological status of semantic networks. In N.V. Findler, editor, *Associative Networks: Representation and Use of Knowledge by Computers*, pages 3–50. Academic Press, 1979.

[19] B.E. Brewington and G. Cybenko. Keeping up with the changing web. *Computer*, 33(5):52–58, 2000.

[20] D. Brickley and R.V. Guha. Resource description framework schema specification 1.0. Technical report, W3C, 2000.

[21] C. Bussler, A. Maedche, and D. Fensel. *A conceptual architecture for semantic web enabled web services*. ACM Special Interest Group on Management of Data, 2002.

[22] M.E. Califf and R.J. Mooney. Relational learning of pattern-match rules for information extraction. In *Working Notes of AAAI Spring Symposium on Applying Machine Learning to Discourse Processing*, pages 6–11. AAAI Press, 1998.

[23] C.-K. Chang. Bidding against competitors. *IEEE Transactions on Software Engineering*, 16(1):100–104, 1990.

[24] S. Chawathe, H. García-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J.D. Ullman, and J. Widom. The TSIMMIS project: integration of heterogeneous information sources. In *Proceedings of the 16th Meeting of the Information Processing Society of Japan (IPSJ'94)*, pages 7–18, 1994.

[25] K. Ciesielski. *Set theory for the working mathematician*. Cambridge University Press, 1997.

[26] G. Cong, L. Yi, B. Liu, and K. Wang. Discovering frequent substructures from hierarchical semi-structured data. In *Proceedings of the 2nd SIAM International Conference on Data Mining (SDM'02)*, 2002.

[27] O. Corcho and A. Gómez-Pérez. Evaluating knowledge representation and reasoning capabilities of ontology specification languages. In *Proceedings of the ECAI'00 Workshop on Applications of Ontologies and Problem Solving Methods*, pages 1–9, 2000.

[28] O. Corcho and A. Gómez-Pérez. A road map on ontology specification languages. In *Proceedings of the ECAI'00 Workshop on Applications of Ontologies and Problem Solving Methods*, 2000.

[29] R. Corchuelo, J.S. Aguilar, and J.L. Arjona. A framework for extracting information with semantics from the web. an application to knowledge discovery for web agents. *The International Journal of Computers, Systems and Signals*, 3(2):12–28, 2002.

[30] R. Corchuelo and J.L. Arjona. Automatic extraction of semantically-meaningful information from the web. *UPGRADE: The European Online Magazine for the Information Technology Professional)*, 3(3):44–51, 2002.

[31] M. Craven, D. DiPasquo, D. Freitag, A.K. McCallum, T.M. Mitchell, K. Nigam, and Seán Slattery. Learning to construct knowledge bases from the world wide web. *Artificial Intelligence*, 118(1/2):69–113, 2000.

[32] R. Dale, H. Moisl, and H. Somers. *A handbook of natural language processing: techniques and applications for the processing of language as text*. Marcel Dekker, 2000.

[33] H. Davulcu, S. Vadrevu, S. Nagarajan, and I.V. Ramakrishnan. OntoMiner: bootstrapping and populating ontologies from domain-specific web sites. *IEEE Intelligent Systems*, 18(1):24–33, 2003.

[34] M. Dean and G. Schreiber. OWL web ontology language reference, 2004.

[35] A. Diller. *Z: an introduction to formal methods*. John Wiley & Sons, 1994.

[36] L. Eikvil. Information extraction from world wide web - a survey. Technical Report 945, Norwegian Computing Center, 1999.

[37] F. Esposito, G. Semeraro, N. Fanizzi, and S. Ferilli. Multistrategy theory revision: induction and abduction in INTHELEX. *Machine Learning*, 38 (1-2):133–156, 2000.

[38] A. Farquhar, R. Fikes, and J. Rice. The Ontolingua server: a tool for collaborative ontology construction. *International Journal of Human Computer Studies*, 46(6):707–727, 1997.

[39] D. Faure and C. Nédellec. A corpus-based conceptual clustering method for verb frames and ontology acquisition. In *Proceedings of the LREC workshop on Adapting Lexical and Corpus Resources to Sublanguages and Applications*, pages 5–12, 1998.

[40] D. Fensel, I. Horrocks, F. Van Harmelen, S. Decker, M. Erdmann, and M. Klein. OIL in a nutshell. In *Proceedings of the 12th European Workshop on Knowledge Acquisition, Modelling and Management (EKAW'00)*. Springer, 2000.

[41] D. Fensel, I. Horrocks, F. van Harmelen, D. L. McGuinness, and P. F. Patel-Schneider. OIL: an ontology infrastructure for the semantic web. *IEEE Intelligent Systems*, 16(2):293–310, 2001.

[42] D. Fensel, F. van Harmelen, M. Klein, and H. Akkermans. OnToKnowledge: ontology-based tools for knowledge management. In *Proceedings of the eBusiness and eWork 2000 Conference (EMMSEC'00)*, pages 1–7, 2000.

[43] R. Fikes and D. McGuinness. An axiomatic semantics for RDF, RDF-S, and DAML+OIL. Technical report, W3C, 2001.

[44] D. Florescu, A.Y. Levy, and A. Mendelzon. Database techniques for the world wide web: a survery. *ACM SIGMOD Record*, 27(3):59–74, 1998.

[45] UCLA Center for Communication Policy. The UCLA internet report. Surveying the digital future: year three. Technical report, 2003.

[46] L. Francisco-Revilla, F. Shipman, R. Furuta, U. Karadkar, and A. Arora. Managing change on the web. In *Proceedings of the 1st ACM/IEEE-CS Joint Conference on Digital Libraries (JCDL'01)*, pages 67–76. IEEE Press, 2001.

[47] D. Freitag. Information extraction from HTML: application of a general machine learning approach. In *Proceedings of the 15th Conference on Artificial Intelligence (AAAI'98) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI'98)*, pages 517–523. American Association for Artificial Intelligence, 1998.

[48] L.M. Fuld. *The new competitor intelligence: the complete resource for finding, analyzing, and using information about Your competitors*. John Wiley & Sons, 1994.

[49] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[50] M. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. DTD inference from XML documents: the XTRACT approach. *IEEE Data Engineering Bulletin*, 26(3):18–24, 2003.

[51] A. Gilbert, M. Gordon, M. Paprzycki, and J. Wright. The world of travel: A comparative analysis of classification methods. Technical report, 2003.

[52] M.L. Ginsberg. Knowledge interchange format: the KIF of death. *Artificial Intelligence*, 12(3):57–63, 1991.

[53] A. Gómez-Pérez and O. Corcho. Ontology specification languages for the semantic web. *IEEE Intelligent Systems*, 17(1):54–60, 2002.

[54] T.R. Gruber. Towards principles for the design of ontologies used for knowledge sharing. *International Journal of Human Computer Studies*, 43 (5/6):907–928, 1993.

[55] T.R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.

[56] N. Guarino. Understanding, building and using ontologies. *International Journal of Human-Computer Studies*, 46(2/3):293–310, 1997.

[57] H. Han and R. Elmasri. Ontology extraction and conceptual modelling for web information. In *Information Modelling for Internet Applications*, pages 174–188. Idea Group Publishing, 2003.

[58] S. Handschuh, S. Staab, and F. Ciravegna. S-CREAM - Semi-automatic CREAtion of Metadata. In *Proceedings of the 13th International Conference on Knowledge Engineering and Knowledge Management (CIKM'02)*, pages 358–372. Springer, 2002.

[59] S. Handschuh, S. Staab, and A. Maedche. CREAM: creating relational metadata with a component-based, ontology-driven annotation framework. In *Proceedings of the First International Conference on Knowledge Capture (K-CAP 2001)*, pages 76–83. ACM Press, 2001.

[60] F. van Harmelen and D. Fensel. Practical knowledge representation for the web. In *Proceedings of the IJCAI Workshop on Intelligent Information Integration*, 1999.

[61] J. Heflin. *Towards the Semantic Web: Knowledge Representation in a Dynamic, Distributed Environment*. PhD thesis, University of Maryland, 2001.

[62] J. Heflin and J. Hendler. Dynamic ontologies on the web. In *Proceedings of the 7th Conference on Artificial Intelligence (AAAI'00) and of the 12th Conference on Innovative Applications of Artificial Intelligence (IAAI'00)*, pages 443–449. AAAI Press, 2000.

[63] J. Hendler. Agents and the semantic web. *IEEE Intelligent Systems*, 16(2): 30–37, 2001.

[64] I. Horrocks. DAML+OIL: a reason-able web ontology language. In *Proceedings of the CAiSE 2002 workshop on Web Services, E-Business, and the Semantic Web (WES'02)*, page 174. Springer, 2002.

[65] I. Horrocks, U. Sattler, and S. Tobies. Practical reasoning for very expressive description logics. *Logic Journal of the IGPL*, 8(3):239–264, 2000.

[66] C.-N. Hsu. Initial results on wrapping semistructured web pages with finite-state transducers and contextual rules. In *Proceedings of the AAAI Workshop on AI and Information Integration*, pages 66–73, 1998.

[67] C.-N. Hsu and M.-T. Dung. Generating finite-state transducers for semistructured data extraction from the web. *Information Systems*, 23(8):521–538, 1998.

[68] G. Huck, P. Fankhauser, K. Aberer, and E.J. Neuhold. Jedi: extracting and synthesizing information from the web. In *Proceedings of the 3rd International Conference on Cooperative Information Systems (IFCIS'98)*, pages 32–43, 1998.

[69] D.E. Hussey, P.V. Jenster, and P. Jenster. *Competitor intelligence: turning analysis into success*. John Wiley & Sons, 1999.

[70] J. Jacky. *The way of Z: practical programming with formal methods*. Cambridge University Press, 1996.

[71] M. Jaeger. A logic for default reasoning about probabilities. In *Proceedings of the 10th Conference on Uncertainty in Artificial Intelligence (UAI'94)*, pages 352–359. Morgan Kaufmann Publishers, 1994.

[72] C.F. Kalmbach and D.M. Palmer. *eCommerce and alliances: how eCommerce is affecting alliances in value chain businesses*. Accenture LLP, 2002.

[73] P.D. Karp, V.K. Chaudhri, and J. Thomere. XOL: an XML-based ontology exchange language. `http://www.ai.sri.com/~pkarp/xol`, 1999.

[74] B. Katz and J.J. Lin. START and beyond. In *Proceedings of 6th World Multiconference on Systemics, Cybernetics, and Informatics (SCI'02)*, 2002.

[75] R.E. Kent. Conceptual knowledge markup language: the central core. In *Proceedings of the 12th Workshop on Knowledge Acquisition, Modelling and Management*, 1999.

[76] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42(4):741–843, 1995.

[77] N. Kushmerick. Wrapper verification. *World Wide Web Journal*, 3(2):79–94, 2000.

[78] N. Kushmerick, D.S. Weld, and R.B. Doorenbos. Wrapper induction for information extraction. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'97)*, pages 729–737, 1997.

[79] D. Lehmann. Nonmonotonic logics and semantics. *Journal of Logic and Computation*, 11(2):229–256, 2001.

[80] K. Lerman, S.N. Minton, and C.A. Knoblock. Wrapper maintenance: a machine learning approach. *Journal of Artificial Intelligence Research*, 18 (2003):149–181, 2003.

[81] A.Y. Levy and M.-C. Rousset. Combining horn rules and description logics in CARIN. *Artificial Intelligence*, 104(1-2):165–209, 1998.

[82] L. Lim, M. Wang, S. Padmanabhan, J.S. Vitter, and R. Agarwal. Characterizing web document change. In *Proceedings of the 2nd Conference on Web-Age Information Management (WAIM'01)*, pages 133–144. Springer, 2001.

[83] M. Luck and M. d'Inverno. Autonomy: A nice idea in theory. In *Proceedings of the 7th International Workshop on Agent Theories, Architectures and Languages (ATAL'01)*, pages 351–354. Springer, 2001.

[84] S. Luke, L. Spector, D. Rager, and J. Hendler. Ontology-based web agents. In *Proceedings of the 1st International Conference on Autonomous Agents (Agents'97)*, pages 59–68. ACM Press, 1997.

[85] R.M. MacGregor. Inside the LOOM description classifier. *SIGART Buletin*, 2(3):88–92, 1991.

[86] A. Maedche and S. Staab. Semi-automatic engineering of ontologies from text. In *Proceedings of the 12th International Conference on Software and Knowledge Engineering (KSI'00)*, 2000.

[87] D. Martin, M. Paolucci, S. McIlraith, M. Burstein, D. McDermott, D. McGuinness, B. Parsia, T. Payne, M. Sabou, M. Solanki, N. Srinivasan, and K. Sycara. Bringing semantics to web services: the OWL-S approach. In *Proceedings of the First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC'04)*, pages 1–12, 2004.

[88] J. McCarthy. Circumscription: a form of nonmonotonic reasoning. *Artificial Intelligence*, 13:27–39, 1980.

[89] D.L. McGuinness, R. Fikes, J. Hendler, and L.A. Stein. DAML+OIL: an ontology language for the semantic web. *IEEE Intelligent Systems*, 17(1): 72–80, 2002.

[90] D.L. McGuinness, R. Fikes, L.A. Stein, and J.A. Hendler. DAML-ONT: an ontology language for the semantic web. In *Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential*, pages 65–93. MIT Press, 2003.

[91] G. Mecca, P. Merialdo, and P. Atzeni. ARANEUS in the era of XML. *IEEE Data Engineering Bulletin*, 22(3):19–26, 1999.

[92] M. Minsky. *A framework for representing knowledge*. McGraw-Hill, 1975.

[93] E. Motta. *Reusable components for knowledge modelling: case studies in parametric design problem solving*. IOS Press, 1999.

[94] I. Muslea, S. Minton, and C.A. Knoblock. Hierarchical wrapper induction for semistructured information sources. *Autonomous Agents and Multi-Agent Systems*, 4(1-2):93–114, 2001.

[95] A. Nauli. *Using Software Agents to Index Data for an E-Travel System*. PhD thesis, Oklahoma State University, 2003.

[96] S. Nestorov, S. Abiteboul, and R. Motwani. Extracting schema from semistructured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'98)*, pages 295–306. ACM Press, 1998.

[97] E. Newcomer. *Understanding web services: XML, WSDL, SOAP, and UDDI*. Addison-Wesley, 2002.

[98] H.S. Nwana. Software agents: an overview. *Knowledge Engineering Review*, 11(3):205–244, 1995.

[99] Y. Papakonstantinou and V. Vianu. DTD inference for views of XML data. In *Proceedings of the 19th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, pages 35–46. ACM Press, 2000.

[100] M. Paprzycki and A. Abraham. Agent systems today; methodological considerations. In *Proceedings of the International Conference on Management of e-Commerce and e-Government (ICMeCG 2003)*, pages 416–4212. Jangxi Science and Technology Press, 2003.

[101] T. Pender. *UML bible*. Wiley Publishing, 2003.

[102] G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[103] B. Potter, J. Sinclair, and D. Till. *Introduction to formal specification and Z*. Prentice Hall, 1996.

[104] J. Powell. Spinning the world wide web: an HTML primer. *Database*, 18 (1):54–59, 1995.

[105] T. Powell. *HTML & XHTML: the complete reference*. McGraw-Hill, 2003.

[106] S. Powers. *Practical RDF*. O'Reilly, 2003.

[107] M.R. Quillian. Word concepts: a theory and simulation of some basic semantic capabilities. *Behavioral Science*, 12(5):410–430, 1967.

[108] J. Reynolds and R. Mofazali. *The complete e-commerce book: design, build and maintain a successful web-based business*. CMP Books, 2000.

[109] B. Richard and P. Tchounikine. Enhancing the adaptivity of an existing website with an epiphyte recommender system. *New Review of Hypermedia and Multimedia*, 10(1):31–52, 2004.

[110] K. Sivashanmugam, J. Miller, A. Sheth, and K. Verma. Framework for semantic web process composition. Technical report, LSDIS Lab, Computer Science Dept., UGA, 2003.

[111] K. Sivashanmugam, K. Verma, A. Sheth, and J. Miller. Adding semantics to web services standards. In *Proceedings of the 1st International Conference on Web Services (ICWS'03)*, pages 395–401, 2003.

[112] J. Snell, D. Tidwell, and P. Kulchenko. *Programming web services with SOAP*. O'Reilly & Associates, Inc., 2002.

[113] S. Soderland. Learning information extraction rules for semi-structured and free text. *Machine Learning*, 34(1-3):233–272, 1999.

[114] J.F. Sowa. *Conceptual structures: information processing in mind and machine*. Addison-Wesley, 1984.

[115] E. Spertus. ParaSite: mining structural information on the web. In *Selected papers from the 6th International Conference on World Wide Web*, pages 1205–1215. Elsevier Science Publishers, 1997.

[116] B. Starr, M.S. Ackerman, and M. Pazzani. Do-I-Care: tell me what's changed on the web. In *Proceedings of the AAAI Spring Symposium on Machine Learning in Information Access Technical Papers*, 1996.

[117] R. Studer, V.R. Benjamins, and D. Fensel. Knowledge engineering: principles and methods. *Data Knowledge Engineering*, 25(1-2):161–197, 1998.

[118] D. Tsichritzis. *Electronic commerce*. Centre Universitaire d'Informatique (University of Geneva), 1998.

[119] M. Vargas-Vera, E. Motta, J. Domingue, M. Lanzoni, A. Stutt, and F. Ciravegna. MnM: ontology driven tool for semantic markup. In *Proceedings of the ECAI 2002 Workshop on Semantic Authoring, Annotation & Knowledge Markup (SAAKM'02)*, 2002.

[120] K. Verma, K. Sivashanmugam, A. Sheth, A. Patil, S. Oundhakar, and J. Miller. METEOR-S WSDI: a scalable infrastructure of registries for semantic publication and discovery of web services. *Journal of Information Technology and Management*, 2004. to appear.

[121] Y. Wang and E. Shakshuki. A multi-agent system for semantic information retrieval. In *Proceedings of the 17th Conference of the Canadian Society for Computational Studies of Intelligence (AI'04)*, pages 573–575. Springer, 2004.

[122] M.J. Wooldridge and M.R. Jennings. Intelligent agents: theory and practice. *Knowledge Engineering Review*, 10(2):115–152, 1995.

[123] M.J. Zaki. Efficiently mining frequent trees in a forest. In *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'02)*, pages 71–80. ACM Press, 2002.

[124] M.J. Zaki and C.C. Aggarwal. XRules: an effective structural classifier for XML data. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'03)*, pages 316–325. ACM Press, 2003.

Index