Masaryk University
Faculty of Informatics

Building of domain-specific semantic networks from web pages

Bachelor's thesis

Ron Šmeral

Brno, spring 2011

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Ron Šmeral

Advisor: RNDr. Zuzana Nevěřilová

Acknowledgement

I would like to thank the supervisor of my bachelor's thesis, RNDr. Zuzana Nevěřilová, for her time and valuable feedback. I would also like to thank my parents for their patience and the effort they put into me, promising it was not in vain. Last but not least, my thanks go to my sister for always helpfully answering my English language-related questions.

Abstract

The aim of the thesis is to describe the issues related to design of semantic networks focused on a specific domain and to develop tools for creation, maintenance and querying of such networks. Specifically, the method of HTML web page crawling and scraping is described in detail. The implemented tools are modular and extensible, forming a framework for further development.

Keywords

semantic network, associative network, crawler, scraping, RDF, RDFS, ontology, Sesame

Contents

1 **Introduction** 1
2 **Data, information, knowledge** 2
  2.1 *Knowledge representation* 3
    2.1.1 Metadata 4
    2.1.2 Ontology 4
    2.1.3 Upper ontology 5
  2.2 *Semantic networks* 8
    2.2.1 History 9
    2.2.2 Types 9
  2.3 *The road to semantic web* 10
    2.3.1 Contemporary web 10
    2.3.2 Web as a graph 11
    2.3.3 Resource identification 11
    2.3.4 RDF(S) 12
    2.3.5 OWL 13
3 **Semantic network building toolset** 14
  3.1 *Requirements* 14
  3.2 *Platform* 15
  3.3 *Third-party software* 15
  3.4 *Piped object processor* 15
    3.4.1 Architecture 16
    3.4.2 Development 18
  3.5 *Crawler* 21
    3.5.1 Addressed issues 21
    3.5.2 Design 24
  3.6 *Other modules* 26
    3.6.1 Scrapers 26
    3.6.2 StatementMapper 29
    3.6.3 SesameWriter 29
  3.7 *Usage* 30
4 **Semantic network of artworks** 33
  4.1 *Design* 33
    4.1.1 Selected sources 35
  4.2 *Properties* 35
5 **Conclusion** 36

1 Introduction

When a human comes across a sentence like *"Yesterday, I watched I Served the King of England"*, he may be surprised at first, by the unusual structure, but provided he has got some common sense, very soon he will find out what it is meant to express. On the other hand, a computer trying to analyse such sentence might not reach success at all, without knowing that *I served the King of England* is a proper name of a man-made abstract object – a film, and a work of literature. In order to make a machine capable of understanding the meaning of natural language expressions, it is first necessary to translate the concepts occurring in real world to the language of the machine. This is the subject of study of knowledge representation.

Automatic analysis of natural language is only one of the many reasons for construction of knowledge representations. One significant aspect of knowledge represented in a well-defined structured form is the possibility to infer new pieces of information by utilising a reasoning facility. There are also many different types of knowledge representations, and the one that is the subject of this thesis is called a *semantic network*. This structure can be used for various purposes, it can either capture the relations between general concepts, thus working as a schema, or it can contain individual instances and their attributes, or both at the same time.

The aim of this thesis is to provide facilities for creation of semantic networks designed to contain knowledge from one specific domain. Moreover, the knowledge that is to be contained in these networks will be collected automatically, with human intervention only required at the beginning of the process, to define the patterns, according to which the data will be collected. The World Wide Web is used as the source of information.

The following chapter provides more insight into the field of knowledge representation, and the semantic networks in particular. In the third chapter, the outcome of this thesis – the semantic network building toolset – is described in detail, followed by the fourth chapter, where a semantic network of artworks, produced using the implemented tool, is characterised.

2 Data, information, knowledge

In all parts of this thesis, certain key concepts inherently involved in the field of information science will be dealt with, understanding of and proper distinction of which is crucial for any further elaboration. The terms *data*[1], *information* and *knowledge*, often juxtaposed with *understanding* and *wisdom* as well, are the ones in question. Only the former three, however, are of specific interest to the field of knowledge representation and artificial intelligence, the latter two being of rather philosophical nature. These concepts are often referred to as the "Information hierarchy" with *data* and *knowledge* being at the lowest and the highest level of abstraction (among these three), respectively.

According to [1], *data* are symbols, *information* is data that are processed to be useful and *knowledge* is application of data and information. More specifically, data have no significance beyond their existence and no meaning. For example, the word "blue", without a context or association does not convey any information. The phrase "The sky is blue", however, is a valid proposition and thus constitutes a piece of information. The relational association to the word "sky" is what adds the informational value. The transition from information to knowledge is not as explicit as the one from data to information. As defined by Childers in [14, p. 481], knowledge is that which is known and can only be thought of in relation to a particular knower, existing as electrical pulses, and it can be disembodied into symbolic representation, thusly becoming a particular kind of information, not knowledge. Since the term knowledge can be further divided into three main kinds: practical knowledge, knowledge by acquaintance, and propositional knowledge [14], it should be stated that the knowledge that is the subject of this thesis and ultimately, of all academic fields, is inferential propositional knowledge. That is such knowledge which is a product of inferences, such as induction and deduction.

1. In the ongoing dispute concerning the proper usage of the word *data*, this thesis takes the side of grammatical correctness and throughout the text, the term is used as a plural form.

2.1 Knowledge representation

As mentioned earlier, knowledge can only exist in relation to a particular knower. In terms of computer science, this knower is some sort of intelligent agent that has a knowledge base (KB) and an inferencing facility, enabling it to draw conclusions from facts already present in the KB. Having defined the concept of knowledge, a proper definition of *knowledge representation* is still due. Despite it being one of the core concepts in the field of artificial intelligence, its definition is similarly to that of knowledge, not a simple one. Davis, Shrobe and Szolovits in [3] try to answer the question – *What is knowledge representation?* – by defining five distinctly different roles it plays. First, KR is most fundamentally a *surrogate*, enabling an entity to determine consequences by thinking rather than acting. Then it is also (2) a set of ontological commitments, (3) certain fragmentary theory of intelligent reasoning, (4) a medium for pragmatically efficient computation and lastly (5) a medium of human expression. Further explication of these roles is, however, beyond the scope of this thesis.

There are many factors to consider when designing a knowledge representation, one of the most important being the expressivity of the KR, with consistency and completeness immediately following. Generally, it holds that expressivity of the KR is inversely proportional to effectivity of automatic inferencing. Therefore, one of the key problems is finding a KR with sufficient expressive power, capable of reasoning within given resource constraints.

Various kinds of representation techniques have been developed and used, each suitable for a specific group of problems. For example *frame systems*, introduced by Minsky, where a frame is defined as "a datastructure for representing a stereotyped situation" [10], aim to provide a framework for representing knowledge, based on the notion of frames, slots and fillers. Another approach to KR is that of semantic networks. These are based on interrelation of concepts using binary predicates and are most commonly implemented using directed graphs. Semantic networks are discussed in more detail further in this chapter.

2.1.1 Metadata

One of the most important information science-related concepts is metadata. As the prefix *meta* suggests, metadata can be defined as data *about* data. More precisely, metadata is "structured information that describes, explains, locates, or otherwise makes it easier to retrieve, use, or manage an information resource." [11] This term is used for two significantly different concepts, one is specification of data structures, called *structural metadata*, the other is *descriptive metadata*, for describing resources. This latter kind can be further separated into at least two other types, one that describes the data themselves, and the other, called *administrative metadata*, which describes information concerning management of the resource, like access rights or creation date.

Metadata was traditionally found in library catalogues, but is today, in the age of digital media, of much higher utility than it was in the past. To provide interoperability between disparate metadata producers and consumers, schemas, vocabularies and taxonomies were introduced, like the MPEG-7[2] standard for multimedia or the Dublin Core[3] metadata set for general description of resources. Some metadata schemes also specify the syntax for encoding of their elements, which is often based on SGML or XML, but some schemes are syntax independent.

2.1.2 Ontology

In the broader meaning, an ontology is the study of the categories of being, or in other words, the fundamental classes of entities, and their relations. In terms of computer science, ontology is commonly defined, for the sake of brevity (but also lack of clarity), simply as "explicit specification of a conceptualization" [5]. This definition, however, does not shed much light on the computer science-related meaning of this term adopted from philosophy. Author of the above mentioned terse definition fortunately offers an elucidation: "an ontology defines a set of representational primitives with which to model a domain of knowledge

2. MPEG-7 is an ISO/IEC standard, also known as *Multimedia Content Description Interface*, used for embedding descriptive metadata in multimedia.
3. Dublin Core is an ISO metadata standard containing only 15 basic elements for description of resources like books, digital video, sound, image, text, and web pages.

or discourse” [6]. These primitives are classes, attributes and relationships. Definitions of these elements can contain information about their meaning and constraints on their application.

Ontologies are usually formulated by means of representation language of some sort, which is close in its expressive power to logical formalisms. In practice, these languages are usually based on predicate logic or description logic. Some examples of the former are Common Logic, CycL or KIF, and instances of the latter are KL-ONE, or two that will be discussed further in this text – RDFS and OWL.

In summary, an ontology can act as a skeletal structure of a knowledge representation, providing a conceptual and computational model of certain problem domain, focusing on its form rather than the content.

2.1.3 Upper ontology

An upper ontology, sometimes called foundation ontology or top-level ontology, is an ontology which expresses relations between the most general concepts common across all domains of knowledge. It usually takes the form of a hierarchy of classes of entities with associated rules constraining their application. The main function of an upper ontology is providing a semantic interoperability between many otherwise disparate ontologies.

Many attempts have been made to create a single all-encompassing upper ontology but there are arguments standing against feasibility of such structure. In order to create such hierarchical system of concepts, the state of the world has to be observed from many perspectives, since the views on precise definition and classification of certain general concepts might differ across disciplines and even across cultures. Thus, finding one such representation, which would embrace all conceivable concepts with all their relations and still remain logically consistent, would be an effort that is very likely to turn out ultimately pointless.

For computational use, however, the requirement is not to have a single ultimate upper ontology, but rather one that is *sufficiently* expressive and detailed to support all its functions. Various such ontologies have been developed, some of which have been reviewed and compared in [8]. There have also been efforts to develop a standard upper ontology, specifically it was the main objective of the IEEE SUO working group[4], but there are indicators[5] suggesting this goal will not be met. The upper ontologies currently in use vary greatly in their dimensions, degree of abstraction and elaboration, and also, despite the concept of an upper ontology inherently involving universal applicability, in their intended application.

Cyc

One of the earliest successful attempts was that of the *Cyc* project, founded in 1984. The Cyc KB is a modular ontology and a knowledge base divided into many smaller *microtheories* which group contained statements into logically consistent units focused on a specific realm of knowledge. Extent of the knowledge base is one of the larger among those reviewed in [8], containing some 300 000 concepts, 3 000 000 facts and rules and 15 000 relations.

SUMO

The *Suggested Upper Merged Ontology* is an ontology created by merging publicly available ontological content into single comprehensive structure. SUMO consists of multiple modules: the SUMO upper ontology, the MILO (Mid-level Ontology) and several domain ontologies. All of the modules together consist of 20 000 terms and 60 000 axioms, making it one of the largest public formal ontologies. All of the contained concepts are mapped to WordNet synsets.

WordNet [4]

Notwithstanding the original designation of WordNet as a lexical database, it has been used extensively in many applications in various other ways, including its use as a simple class hierarchy or as an upper ontology. Additionally, WordNet contains glosses for many of the terms, making it also a human-readable lexicon of English language. It has

4. SUO WG, also known as IEEE 1600.1. Website of the project, `http://suo.ieee.org/`, seems to be rather out of date, last updated in 2003.
5. Judging by a message in the mailing list of SUO WG, `http://suo.ieee.org/email/msg13625.html`, written by a respected computer scientist John F. Sowa, the efforts of the working group are getting steered towards a different goal – creating a registry of ontologies.

been employed in a wide range of research and commercial applications and has also given rise to many extensions and similar projects.

Terms in WordNet are organised so as to capture the complex structure of natural language, capturing various semantic relations between concepts. The main constituent element of the lexicon called *synset* is a set of synonymous words or collocations, and from the perspective of ontology engineering is analogous to a class. Synsets are interconnected by semantic relations, some of which are:

hyponymy: Analogous to an *IS-A* relationship. X is a hyponym of Y if every X is a kind of Y. For example, *actor* is hyponym of *person*.

hypernymy: Inverse of the hyponymy relationship. X is a hypernym of Y if every Y is a kind of X.

holonymy: The whole–part relationship. X is a holonym of Y if Y is a part of X. For example, *body* is a holonym of *head*.

meronymy: Inverse of the holonymy relationship. X is a meronym of Y if X is a part of Y.

WordNet contains other relations as well, but those listed here are the most important for this text. The hypernymy relationship organises the synsets into a hierarchy of classes. An example of the hierarchy can be seen in Figure 2.1 where *entity* is the top-level element. This hierarchy is what enables the use of WordNet as an ontology. This application is, however, not straightforward since the lexicon was not originally designed for this purpose and contains certain semantic inconsistencies and insufficiencies such as the lack of distinction between the *subtypeOf* and *instanceOf* meanings of the hyponymy relation. Despite these deficiencies, WordNet has been successfully employed in several projects as an ontology, in some cases even without corrections to the mentioned insufficiencies.

Originally, the lexicon is contained in a custom knowledge representation format, however, many other are currently used, for example the OWL representation[6] which emerged as a part of the W3C Semantic Web Activity. The original WordNet database only covers English language but projects like EuroWordNet or BalkaNet have created

6. `http://www.w3.org/TR/wordnet-rdf/`

```
person, individual, someone, somebody, mortal, soul
  => organism, being
    => living thing, animate thing
      => object, physical object
        => physical entity
          => entity
```

Figure 2.1: Word sense hierarchy in WordNet

wordnets for several other languages. The Global Wordnet Association coordinates production and linking of these mutations.

2.2 Semantic networks

A semantic network, also called *associative network* is a form of knowledge representation which expresses semantic relations among concepts in certain domain of knowledge. Most commonly it is specified using edge-labeled directed graphs where vertices represent concepts and edges correspond to the relations. A semantic network is a form of logic and the information contained in the structure can be expressed in predicate logic using binary predicates. An example of a semantic network can be seen in Figure 2.2. It is worth noting, that despite most commonly being notated as graphs, semantic networks are not a data structure, but merely a representation.

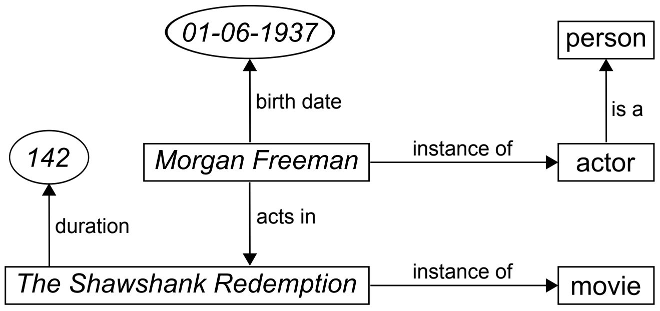

Figure 2.2: A simple semantic network. Depicts three classes, two instances, two attribute values and relations.

Semantic networks are used in several disciplines for many purposes.

In psychology, they were implemented as a mechanism for modelling human cognitive processes and the concept of semantic memory. In linguistics, the inheritance of properties is among the more significant qualities of semantic networks, supporting disambiguation in natural language processing. Among their more important applications is that in artificial intelligence, where they are used as a form of knowledge representation supporting intelligent reasoning.

2.2.1 History

The first use of semantic networks dates back to 300 AD and is attributed to Greek philosopher Porphyry, which also illustrates the fact, that the idea of using nodes and arcs for representation of interrelation of concepts has been employed in philosophy, psychology and linguistics long before their contemporary computer-related applications. Charles S. Peirce has proposed this way of notation in 1909, calling it *existential graphs*. The term 'semantic network' has first been used by cognitive scientist Ross Quillian in 1968 in his thesis where he used it to describe the organisation of human semantic memory.

The first application in computers is ascribed to Richard H. Richens who used semantic networks as an *interlingua* for machine translation of natural language. Along with the advent of hypertext systems in the 1980s, the idea of *semantic links* was brought to light, providing simple semantics to links between hypertext documents, thus creating a network of documents. This idea was incorporated into the HTML standard in the form of `rel` and `rev` attributes of hyperlinks. However, this effort partly missed its aim, since HTML does not restrict the values of these attributes, it merely suggests the correct usage. In the recent years, new approach for describing semantic properties of hypertext content has emerged, called *microformats*, which reuses existing HTML tags to convey metadata.

2.2.2 Types

According to an article [13] by John F. Sowa, semantic networks can be classified into following categories:

Definitional networks form a generalisation hierarchy by emphasizing the subtype relation and support *inheritance* of properties from a supertype.

Assertional networks used primarily to assert propositions.

Implicational networks with implication as the primary relation.

Executable networks include a mechanism which can perform inferences, pass messages or search for patterns and associations.

Learning networks are capable of acquiring knowledge from examples and may modify the network in the process.

Hybrid networks combine some of the above mentioned techniques.

2.3 The road to semantic web

Semantic web is a name (quickly becoming a buzzword) for a number of technologies and standards sharing the common goal of making the internet easier to comprehend by non-human agents, which in turn should lead to the internet being more effectively usable by humans. Although the internet was originally designed for exchange of documents containing human readable information, today many benefits arise from enabling the web with technologies that make it more comprehensible for insentient agents, one of the most important being substantially improved effectivity of search engines.

2.3.1 Contemporary web

Web pages nowadays are composed mainly of semi-structured documents marked up using HTML. Documents marked up in this language have a tree structure with elements containing text content or other HTML elements. The structure describes mainly presentational qualities of the document. Even though HTML has some elements that partly describe the semantics of the content, the granularity of information provided about its meaning is not sufficient for machine processing and automatic data extraction. Moreover, many of the elements are often misused, not containing the information they were designed to contain. All of this implies the need to use specialised tools in order to extract information from websites in an automated fashion.

2.3.2 Web as a graph

One of the key features of hypertext is the ability to enrich text with links to other hypertext documents, which can be immediately accessed, thereby forming a network of documents – a *web*. These links are called *hyperlinks* and in HTML they are represented by the anchor element with the `href` attribute, like so:

```
<a href="http://example.com/some/resource">Link text</a>
```

By virtue of this network structure, the web can be seen as a graph, with documents as nodes and links as edges, making it possible to automatically collect information from online resources by walking the graph and continuously visiting all encountered links. This process called *crawling* has been used extensively mainly in search engines, which crawl the web in order to index its contents. The agents performing this task are usually called *web robots* or *crawlers.*

2.3.3 Resource identification

The resources composing the web need to be uniquely identified, exactly addressed and named. There are three main standards serving this purpose – *URL*, *URI* and *URN*. Often, even in technical literature, the terms URI and URL are incorrectly interchanged [9, p.2]. A clarification is in order.

URI

The *Uniform Resource Identifier* is a standard for resource identification in the internet with a prominent role in the semantic web. The URI only serves the purpose of unique identification of resources (however, not necessarily globally unique), not their location. The syntax of URI consists of a scheme name followed by a colon character and a scheme-specific part. URIs are either absolute, fully identifying a resource, or relative, containing only some trailing component of the URI. Relative URIs can be *resolved* to absolute URIs by merging with some absolute base URI according to a fixed algorithm.

URL

The *Uniform Resource Locator* is the prominent means of resource addressing used in the internet. It is a subclass of URI and is specified by an absolute path to a resource, located on a machine connected to the internet, that can be dereferenced by issuing a specific request to this address. The syntax of URL consists of the scheme name (also called *protocol*), host name and optional port, path of the document and possibly a query string and a document fragment.

$$\underbrace{\texttt{http}}_{\text{scheme}}\texttt{://}\underbrace{\texttt{www.example.com}}_{\text{host}}\texttt{:}\underbrace{\texttt{80}}_{\text{port}}\underbrace{\texttt{/index.php}}_{\text{path}}\texttt{?}\underbrace{\texttt{arg=value}}_{\text{query string}}\texttt{\#}\underbrace{\texttt{part1}}_{\text{fragment}}$$

Figure 2.3: An example of a URL and its components.

URN

The *Uniform Resource Name* is a subclass of URI that uses the `urn` scheme and is a persistent location-independent resource identifier. The syntax is as follows: `urn:`***`nid`***`:`***`nss`***, where `nid` is a namespace identifier and `nss` is a namespace specific string. It is often compared to ISBN, which too uniquely identifies a resource (a book), but conveys no information about its availability and location.

2.3.4 RDF(S)

The *Resource Description Framework* is the name for a set of W3C specifications originally designed as a metadata model, now used as a general method for conceptual description of web resources. These descriptions are realised by statements, called *triples* in RDF, which have the form subject–predicate–object.

RDFS stands for *RDF Schema* and together with RDF forms an extensible language for knowledge representation. It provides a vocabulary for description of ontologies. Both, RDF and RDFS are principal elements of Semantic Web. These specifications are not bound to any specific formats, but the most common serialisation format is XML. URIs are used for resource identification.

2.3.5 OWL

The *Web Ontology Language* is an ontology language with strong logical foundations, having model-theoretic formal semantics. More precisely, it is a family of languages with different levels of expresiveness – OWL Lite, OWL DL and OWL Full. The DL and Lite variants are based on description logics and thus have well-understood computational properties and sound, complete, terminating reasoners exist for these languages. OWL is designed to be generally compatible with RDFS, OWL Full is a semantic extension of RDF.

3 Semantic network building toolset

After having established the theoretical foundations, the stage is now set for putting the theory into practice. In this chapter, we provide detailed look at the design and implementation of a modular and extensible toolset for building domain-specific semantic networks. From now on, this tool shall be referred to simply as *SemNet*.

The main purpose of SemNet is automated collection of unstructured or semi-structured data from web resources and their transformation into a machine readable representation – a semantic network. This process is split into several stages, each carried out by a separate module. This modular system based on *object processors* provides a solid and flexible platform for development of data processing chains and is described in section 3.4. There are four of these processors in SemNet. The first and the most sophisticated is the *crawler*, whose functionality is discussed in section 3.5, followed by description of scraper, statement mapper and Sesame writer in section 3.6. Details of configuration and execution of the system are provided in section 3.7.

3.1 Requirements

The developed software is, in the first place, intended for immediate usability, not only for demonstration of its capabilities (not only a proof of concept). Therefore certain key requirements should be met, which can be summed up as follows:

independent operation: After proper configuration, the system has to be capable of unattended operation, not requiring any user interaction during processing.

robustness: Implied by the need for independent operation is the necessity of certain amount of resistance against adverse conditions. The system must handle errors gracefully and maintain a log of operation.

extensibility: The system should be designed so as to provide the potential of extending its features, thus it must be modular and well documented.

3.2 Platform

One of the secondary objectives of SemNet is cross-platform operation. This requirement is, to a large extent, satisfied by the implementation being written completely in *Java*, a popular cross-platform, object-oriented, bytecode-compiled, strongly-typed programming language. The only other requirement is a RDBMS – a relational database management system. Consequently, the system should run on all major platforms currently in use. However, correct operation has only been tested on a system with the following parameters:

- Windows 7 64-bit operating system
- JDK 1.6.0_20
- PostgreSQL 9.0.3 64-bit
- Dual-core Intel Pentium processor, 4 GB RAM

The primary development environment used was NetBeans IDE 6.9.1 with IvyBeans plugin for dependency management.

3.3 Third-party software

As mentioned earlier, one technology SemNet depends on is a RDBMS. Specifically, the PostgreSQL database was used in the course of development. In order not to reinvent the wheel, several third-party libraries were employed for specific tasks. The RDF store used for knowledge persistence is *Sesame*[1]. Configuration of all components is contained in XML files and *XStream*[2] is used for (de)serialisation. The *HTMLCleaner*[3] library is used for parsing of HTML files.

3.4 Piped object processor

Piped object processor (POP) is the name given to the lowest layer of the implementation. It is a construct inspired by the design pattern

1. Sesame – `http://www.openrdf.org/`
2. XStream – `http://xstream.codehaus.org/`
3. HTMLCleaner – `http://htmlcleaner.sourceforge.net/`

called *Chain of Responsibility*. The POP is based on the notion of processing chains where information flows from the input to the output, passing through arbitrary number of *object processors*, each of which might perform some transformation on the received information or emit new pieces of information based on those received. Only discrete pieces of information are exchanged, not continuous data streams. Information is encapsulated in containers[4] called simply *objects*, since POP is based on Java, where the top-level element in type hierarchy is *Object*. Any Java class may serve as a container.

3.4.1 Architecture

The main reason for the chosen behavioural pattern is to support the idea of loose coupling of components by splitting functionality into smaller units – the object processors. These processors are connected to form a chain called a *pipe*, which is responsible for state management, input/output type checking and execution. An illustration of the system can be seen in Figure 3.1.

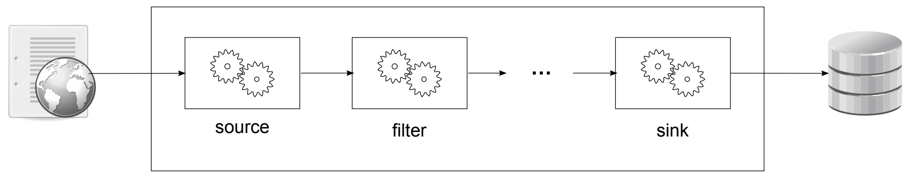

Figure 3.1: An illustration of object processors. One object source, one or more filters and a sink.

Processing context

Despite the processors operating mutually independently, a communication facility is necessary in order to ensure a certain degree of flexibility and resource efficiency of the system. For this purpose, the *processing context* has been introduced. It is a memory space shared by all the

4. Throughout this text, the term *container* will be used to denote a Java class (or its instance), which does not implement any functionality, and only holds data – other objects or literals. Such container roughly corresponds to a JavaBean.

object processors in a pipe, implemented as a map of named parameters. Its primary function is that of a container for shared resources, like database connections, but it could be possibly used also for direct communication between processors, by including some sort of message dispatcher in the context. This would, however, go even further against the principle of loose coupling and inherent linearity of the processing chain. The other function of context implemented in POP is storage of runtime statistics of processors such as performance or error counters.

There is a special type of processor in POP, designed specifically to work with the processing context – the *attached processor*. It has got read/write access to all context parameters, all object processors and the pipe. A variety of applications can be found for such facility, including persisting and analysis of statistics, state monitoring and management of the pipe and possibly resource injection as well. All of this can be accomplished by virtue of fine-grained control over the lifecycle of the pipe and the processors, which is split into several phases, described further in section 3.4.2.

Object processors

As seen in Figure 3.1 on page 16, there are three types of object processors, differentiated by their role and valid position in the chain. These types are:

Object Source: This type of processor must be placed first in the chain, since the pipe wouldn't work without one and it is not allowed on any other position. As the name implies, it is a source of objects, constructed by the processor using data from any – very likely an external – resource, and emitted at any rate. For instance, it might process user input, perform an automatic collection of data from the web, or react to remote invocation.

Object Filter: An object filter is an intermediate stage in the processing, where transformations of objects may take place. Filters can behave in several ways. It can either simply change attributes of received objects and pass them forward, or it can receive an object of one type and emit any number of objects of other type. The received objects, however, do not need to be affected at all, with the processor acting as a 'transparent filter', like a counter.

Object Sink: The object sink is the terminal stage of the pipe. Its primary purpose is to persist received objects, either to a file system, a database, or any other form of storage.

The default implementation was developed so as to support the idea of mutual independence of processors also in terms of execution and performance. All processors are executed in separate threads and objects are passed using concurrent queues. Asynchronous operation eliminates the waiting for adjacent processors. Should a computation-intensive step slow down the processing, it is advisable to parallelise the task if possible, to increase throughput of the processor.

3.4.2 Development

API

The API of POP consists of several interfaces and classes. The interface representing object processors is called, as would be expected, `ObjectProcessor` and the roles of processors are represented by the `ObjectSource` and `ObjectSink` interfaces. POP makes use of Java Annotations to improve readability of code and to simplify certain declarations. One example of this is the `ObjectProcessorInterface` annotation type, which is used to declare input and output type of a processor. The `Object` class is used as a wildcard, allowing any object in or out of the processor.

The different roles of object processors, namely object source, filter and sink are implemented by `LocalObjectSource`, `LocalObjectFilter` and `LocalObjectSink` classes, respectively[5]. Reading the above mentioned description of class hierarchy, an observation can be made, that there is no `ObjectFilter` interface for the corresponding local implementation class. Such observation would be correct and is justified by the `LocalObjectFilter` implementing both `ObjectSource` and `ObjectSink` using delegation to instances of `LocalObjectSource` and

5. The prefix '`Local`' suggests that the processors can only be executed locally, in one virtual machine, as opposed to being distributed across machines, which could be expected from a more advanced data processing system.

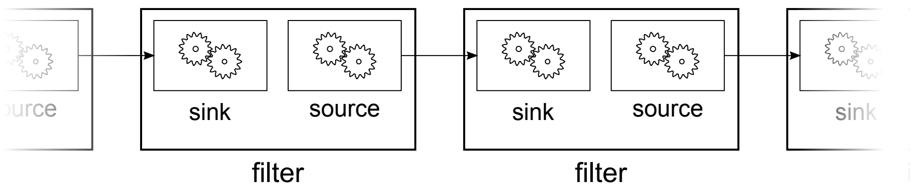

Figure 3.2: A chain of object filters, illustrating the delegation of functionality to an object source and a sink.

LocalObjectSink. This is illustrated in Figure 3.2. Common functionality of all implementing classes is included in the abstract class AbstractObjectProcessor.

Lifecycle

The main class that manages the lifecycle of object processors is Pipe. An example of how a simple Pipe is instantiated and started can be seen in Listing 3.1. Methods for creation, initialisation, starting and stopping of pipes are provided. When the pipe is started, it sets the context on all processors, and calls their callback methods, allowing them to initialise their state and access the context.

Listing 3.1: Starting a pipe

```
ObjectProcessor[] processors = {
    new MessageSource(),
    new TextFilter(),
    new PersistentStore()
};
Pipe pipe = new Pipe(processors);
pipe.start(true);
```

Initialisation of processors is carried out in several phases. Firstly, the processor can be initialised by calling its `initialize` method, which takes one argument – a map of parameters. It is a substitute for initialisation with constructor, which must be parameterless. Then, in the `start` method of `Pipe`, the `initWithContext` and `initPostContext` can be used to access the context immediately after its association to the processor, and after it has been associated with all processors, respectively.

In order to stop the pipe, the `requestStop` method of only the first processor is called. Subsequently, the following processor encounters an exception while trying to read an object from the empty buffer of the preceding processor, and handles it by stopping as well, which causes a chain reaction leading to stopping of all processors.

Developing an object processor

When a new object processor is developed, it should extend, depending on its type, one of `LocalObjectSource`, `LocalObjectFilter` or `LocalObjectSink`, from which it inherits all the core functionality. For a minimal implementation, the only method that needs to be overridden is `process`, which, in the default implementation of the `run` method (which can be overridden if necessary) is called in an infinite loop, until the processor is stopped. In the `process` method, the expected behaviour of the processor is such, that it uses the `read` (in case of filter and sink) and `write` (in case of source and filter) methods, to retrieve and submit objects from and to the buffers of the adjacent processors. To illustrate how little code is required to implement a simple processor, Listing 3.2 shows the `process` method of a putative object filter (whose input and output type is `String`), which simply converts a string to upper case.

Listing 3.2: The `process` method of a simple object filter

```
String inStr = read();
inStr = inStr.toUpperCase();
write(inStr);
```

There are, however, several other overridable methods, which provide fine-grained control over the behaviour of the processor. The initialisation-time methods were already described in the Lifecycle section. Another two methods that can be used to perform actions during certain lifecycle phases are `preRun` and `postRun`, called as the first and the last statement in the default `run` method, respectively. Also, the option to react to the event of an adjacent processor having stopped is provided by means of `handleStoppedSource` and `handleStoppedSink`.

3.5 Crawler

The central part of SemNet is the crawler, implemented in the class `HTMLCrawler`. It is a web robot, which collects data from HTML web pages according to user-defined rules. Being an internet agent, the crawler has to cope with the vast diversity of the medium in many aspects, be it the various character encodings in use, different URLs corresponding to the same resource, or the countless ways of writing faulty HTML code. The methods applied to tackle these and some other issues are described in the following section.

3.5.1 Addressed issues

The crawler is designed to handle the most common situations that can be encountered while walking the web graph and the peculiarities pertaining to many web technologies.

URL normalisation

Ideally, a one-to-one correspondence between a URL and a resource would exist in the internet. However, in reality, it is seldom the case, despite the word *uniform* in URL suggesting some sort of homogeneity. On the contrary, resources can be very often accessed by many different URLs. For example, these two URLs reference the same resource:

```
HTTP://WWW.EXAMPLE.COM/some/../resource_1?a=1&b=2

http://www.example.com:80/resource%5f1?b=2&a=1&#part2
```

And since in the course of its operation, the crawler searches for, stores and dereferences URLs, these variations in resource identification are undesirable, at least in order to eliminate duplicate requests. A process known as *URL normalisation* is used to compensate for this deficiency.

Even though there is no general agreement on what a normal form of a URL would be, there are several most commonly used techniques for making the resource locators somewhat more uniform. In SemNet, URL normalisation is performed by the `normalize` method of the `URLUtil` class, which performs the following steps, some of which are based on the suggestions found in [2] and [7]:

- **case normalisation** – since the host part of URL is case insensitive, it is transformed to lower case,
- **removal of standard port** – certain protocols are associated to specific port numbers (e.g. HTTP has port number 80), in which case the port number need not be specified,
- **decoding of unreserved characters in path** – characters that are encoded despite being unreserved are decoded to their ASCII representation,
- **parameter sorting** – since the order of parameters in the query string is arbitrary, they are sorted by a fixed algorithm,
- **capitalisation of percent-encoded octets** – letters in the encoded representation of a character are converted to upper case (`%3f` becomes `%3F`),
- **path normalisation** – relative path references, also called *dot segments*, are resolved (`/some/../path` becomes `/path`),
- **removal of document fragment** – the fragment only marks a position inside the referenced document, therefore it is irrelevant in terms of resource identification.

Character set detection

Due to the internet being an international medium containing documents written in many different languages, various coding schemes called *character sets* (or simply *charsets*) have been developed to encode the characters of various scripts. The UTF-8 charset has become dominant for the web, albeit not ubiquitous. Therefore, knowing what encoding a web page uses is critical for correct decoding of characters. The class `CharsetDetector` is responsible for detecting the charset used in a resource, using the following three methods, in the given order, returning the first found result:

1. reading the `Content-Type` HTTP header,
2. parsing the HTML `meta` tag with `http-equiv="Content-Type"`,

3. guessing the charset using an external library[6].

Invalid HTML markup

Despite HTML being a standard defined by W3C, the official authority for web, many web pages contain markup that does not comply to the rules defined in the specification. Occurrence of these syntax errors is further sustained by the fact, that all the popular web browsers have overly lenient HTML parsers. Thus the HTML parsing module of the crawler needs to possess similar qualities in terms of error tolerance as the ones in the browsers do. And due to such parsers being very complex, an external library – *HTMLCleaner* is utilised for this task.

Connection management and HTTP requests

Throughout the process of crawling, thousands of HTTP connections to servers are established. There are several parameters that can be adjusted, like the connection and read timeout, whether or not to follow redirects, or the user agent string. In order to provide consistent parameters for all connections and to simplify these adjustments, a simple class called `ConnectionManager` is used. This class also takes care of connection retrying, in case of an I/O error.

Since servers do not have unlimited processing power and network capacity, internet agents should respect certain rules of fair use. One of the most important parameters that need to be regulated is the rate at which an agent issues HTTP requests. Also, certain pages of a web host might not be suited for automated processing. For this purpose, one de facto standard has been around for many years – the *Robots Exclusion Protocol*[7], also called *robots policy*, which offers a way for host owners to set rules for the behaviour of web robots. The SemNet crawler uses the class `RobotsPolicy` to obey this policy, while still providing the option to ignore it.

6. The third-party library used for charset detection is *juniversalchardet*, available at `http://code.google.com/p/juniversalchardet/`.
7. Robots Exclusion Protocol – `http://www.robotstxt.org/`

3.5.2 Design

The SemNet crawler is not a typical crawler inasmuch as it is not designed to crawl the whole web and index its contents, but its designation is much more specific. Only user-defined hosts are visited and only links matching defined patterns are collected. Processing of data retrieved from the visited pages is carried out in a different module – in the scrapers.

Since the primary purpose of the crawler is to collect data for building of domain-specific semantic networks, it is first necessary to identify the hosts which contain information pertaining to the domain. Secondly, a description of the entity types that are to be retrieved must be provided. It is worth noting, that functionality of the crawler has two important preconditions:

- the entity types collected from the host must be distinguishable by their URLs, i.e. the URL of a resource should contain some indication of what entity type it represents (for example, in some film database, the URL pointing to a resource which describes an actor, could have the path part begin with `/actor/` followed by a unique identifier of the instance),
- every resource representing an entity must be uniquely identified by a URL.

These might seem like rather strong assumptions, but could be justified by another assumption, that once a host contains high-quality information resources, very likely the method of addressing these resources will be of similar quality. Based on these ideas, the crawler identifies entity types by their *URL patterns* – regular expressions matching the URLs of all entities of that type, on one particular host. In addition to this kind – the *entity URLs*, the crawler takes into account one other critical concept, called *source URLs* – the resources which themselves do not represent any distinguishable entity, but contain links to entities on the same host. Indexes are a good example of this concept.

Finally, the crawler provides two options that affect the progress of crawling:

- the *update frequency* in days of both, the source and the entity URLs, which signifies how often the resource should be visited and checked for new links,

- the *weight* of an entity type, which determines the ratio of the entities visited in the process. For example, if A has weight of 1 and B the weight of 2, provided there are enough A's and B's on the host, the ratio of visited links by their entity type will be in every moment close to 2:1, in favor of B.

These design decisions are reflected in the model by means of the `HostDescriptor` and `EntityDescriptor` classes, which act as containers for the description of hosts and entities. The entity descriptor applies to one specific entity type of one host and contains:

- URL pattern of the entity type (as a path relative to the base address of the associated host) and its update frequency,
- set of scraper configurations,
- 'weight' of the entity.

The host descriptor describes one web host and consists of the following fields:

- base URL of the host – the address against which the relative paths of patterns are resolved,
- set of source URLs and their update frequencies,
- set of entity descriptors,
- other optional parameters, such as a fallback charset (used in case no charset is found using the default methods), the crawl delay (time interval between two requests to the same host) and an option to visit the source URLs first in every run of the crawler.

There is one more container class used for configuration of the crawler, called `CrawlerConfiguration`, which contains a set of host descriptors and some additional settings of the crawler. One crawler configuration should correspond to one domain-specific semantic network that is built using SemNet. An example of the configuration in XML format can be seen in Appendix A.

Low-level design

With modularity in mind, the *URL frontier* – a core part of a crawler's functionality, is implemented separately from `HTMLCrawler`, in class `URLManager` that works in cooperation with `HostManager`, which provides methods for working with host and entity descriptors. The URL manager is responsible for storage and retrieval of the URLs used by the crawler. The SemNet implementation employs a relational database to maintain the store.

Even though SemNet is not specifically tuned for high performance, certain measures have been applied to increase effectivity of the processing. Most importantly, the crawler is multithreaded, with one or more crawling threads per host. This allows for simultaneous crawling of several hosts without unnecessary waiting, and ensures optimal throughput of the crawler. Moreover, URLs that are to be visited are prefetched in a separate thread, in order to eliminate prolonged waiting for database operations. However, concurrency comes at a cost of increased complexity of the implementation and higher probability of errors arising thereof. The main technique used to avoid common problems related to concurrent operation – memory inconsistencies and thread interference – is the use of locks on various levels, like the *monitor locks* intrinsically present in Java.

The process of collecting and visiting internet links is pointless without processing the resources represented by the collected URLs. Again, for the sake of modularity, this processing step called *scraping* has been detached from the crawler into a separate processor, described in the *Scrapers* section along with the type of objects flowing from the crawler to a scraper.

3.6 Other modules

3.6.1 Scrapers

Once the crawler visits a web page, parses its HTML code, and collects matching URLs, the parsed document is passed forward in the processing chain, to the scraper. The `EntityDocument` class is the container used for this purpose and contains:

- base URL of the document, which is either specified by the

HTML `<base>` tag, or else the URL of the document is used,

- absolute URL of the online resource, where the document originated,
- a descriptor of the entity (instance of `EntityDescriptor`),
- a DOM tree representation of the document.

To explain the functioning of these processors, a short introduction to its core principle follows.

Scraping

Scraping is the process of extracting useful data from a web resource. The main purpose of scraping is the transformation of information expressed in natural language or in some non-standard structure into a machine-processable form, with the objective of creating a knowledge base of some sort. The following example illustrates why this process is necessary: suppose there is a web page about a movie, which contains the following text:

> **The Shawshank Redemption**
> *USA, 1994, 142 min.*

A human reading this text would assume, that there is a movie called *The Shawshank Redemption*, which was shot in USA in 1994 and is 142 minutes long. Such assumption is based on the experience, that when within a text, a name of a country, a year and a temporal quantity are situated next to a name of a movie, they usually denote the country of origin, the year of release and the duration of the movie, respectively. An insentient agent, however, lacks such experience and thus can not derive any meaning from the text. This is the point where human intervention comes into play, to describe the patterns found in given resources and to relate the extracted data to predefined set of concepts. The specification of patterns in resources and the methods for their extraction is implemented in *scrapers*.

Scrapers in SemNet

SemNet provides the base functionality for implementation of a scraper in the class `AbstractScraper`, which is an object filter with the container `EntityDocument` on input and `Statement`[8] on the output. There are only two methods to implement in a scraper: the `scrape` method, which takes an `EntityDocument` as the argument and should use the `write` method to output `Statements`, and `getNamespace`, which should return the namespace in which the terms of the scraper's vocabulary reside. As to techniques used for the extraction, these are not limited in any way. However, the use of XPath (potentially in conjunction with regular expressions) is recommended and supported by means of the `XPathUtil` class, which simplifies querying of the DOM tree.

As mentioned earlier, the scraped data must be related to some concepts, usually defined in an ontology or a controlled vocabulary. To support flexibility, the scrapers in SemNet do not describe data by terms of the vocabulary of the built network, but use custom terms residing in the namespace of the scraper. The vocabulary is declared very simply, by annotating fields of type `URI` with the `Term` annotation, which takes an optional `String` argument – textual definition of the term. In summary, the conceptual independence allows for implementing of the scraper even prior to making decisions about the ontology of the network.

In processing chains, it will hardly ever be the case, that only a single scraper is used, mainly because of the design decision of using one scraper per entity type. To allow for using of multiple scrapers in one chain, the `ScraperWrapper` processor is used, which acts as a router, dispatching `EntityDocuments` to appropriate scrapers, based on the set of scrapers listed in the entity descriptor contained in the received `EntityDocument`. The option to have multiple scrapers process one entity type makes it possible to develop a generic scraper, which could be used to scrape certain structures common across various entity types. An example of this could be a processor which would extract data described by microformats.

8. The `Statement` class is one of the few classes, used in SemNet for working with the RDF data model, that are a part of the Sesame API.

3.6.2 StatementMapper

This object filter, the `StatementMapper`, is what enables the scrapers to be agnostic of the network's ontology, by performing translation of terms between vocabularies. The terms in question are objects of type `URI`[9]. Configuration of this module is realised using the `Mapping` class, which is a simple wrapper for a set of key–value pairs with the optional specification of the association roles[10] it pertains to. As implied by the processor's function, its input and output type is `Statement`.

3.6.3 SesameWriter

At the end of the chain for creation of a semantic network, the data extracted from web resources by the crawler and given meaning to by the scrapers, are persisted in a *triple store*[11] – a database designed specifically for storage of knowledge in the form of *triples*, which are statements consisting of a subject, predicate and object. A simple configuration is necessary for this processor's functioning – the specification of the type and parameters of the Sesame store to use. By default, the native and RDBMS stores are supported. There is also an option to enable reasoning, by specifying the inferencer class to use. This processor is an adapter for the Sesame database, but by virtue of the modularity provided by POP, the system can easily be adapted to a different product by implementing an object sink.

The operation of this processor is very simple – it is only responsible for adding every received statement to the store. But it has one more feature – *bootstrapping*. Prior to starting, it checks for the `bootstrap` initialization parameter holding the name of a file containing statements, all of which will be added to the database. This may be useful for preparing the store for reasoning by preloading it with a knowledge base.

9. The `URI` class is a part of the model of Sesame API.
10. The term *association role* is used here to denote one of *subject*, *predicate* or *object*.
11. The Sesame database used in SemNet is actually a *quad store*, since in addition to subject, predicate and object, it supports the storing of one more attribute – the *context*. This feature is, however, not currently made use of by SemNet.

3.7 Usage

The processors implemented as a part of SemNet are intended for, but not limited to utilisation in ways described in this text. In fact, potential users are encouraged to employ them for different purposes, or as a part of processing chains other than the default, which is illustrated in Figure 3.3. It is also possible to modify the default chain by adding different processors.

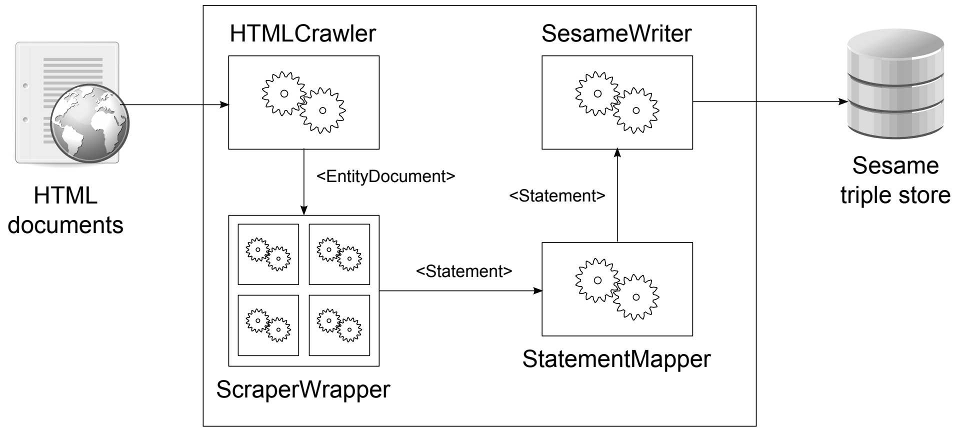

Figure 3.3: The default processing chain.

Processing jobs

This amount of flexibility in adjustment of the processing chains is possible due to the execution being realised in terms of *processing jobs*. First, it might be worth to make a terminological distinction for clarification of matters: the term *processing chain* is used to represent a constellation of processors assembled for certain purpose, e.g. the SemNet default chain for building of semantic networks, composed of the crawler, scraper, mapper and a triple store writer. On the other hand, a *processing job* is a specific arrangement of processors *configured* for a specific task, like the building of a network of artworks. In short, a processing job is an instance of a processing chain.

There is an executable class called `JobRunner`, responsible for the execution of processing jobs. It takes one argument – the name of the

XML file describing the configuration of processors. When started, it ensures the processors are of matching types (output of one matches the input of the following), assembles them into a `Pipe` and starts it. The processing halts either naturally, when the object source stops emitting objects, or forcibly, by sending the interrupt signal to the program. This design makes it possible to run processing jobs completely automatically, using an operating system-provided task scheduler.

Host management

The crawler operates automatically, finding links, visiting web pages periodically based on their update frequency, and thus incrementally building the network. The database of links maintained by the crawler's `URLManager` can not be accessed directly using the SemNet API, apart from one exception: the `HostManager` class has a command line interface, providing the option to remove all URLs of one host, thus causing a complete re-crawling of that host, and one other option, to completely reset the state of the crawler by removing all managed hosts.

Querying

The Sesame database which contains the built network, offers a comprehensive set of functions for querying of the stored RDF graphs, with the prominent means being the support for *SPARQL* and *SeRQL* query languages. For convenience, in SemNet there is a simple command line interface to some functions of the Sesame API, in class `QueryInterface`. The functionality provided is:

- listing resources with a given label,
- querying the type of a resource, either only the direct type, or the full type,
- description of a resource – returning all statements with the given resource in the role of subject or object,
- execution of tuple queries expressed in SeRQL.

In regard to querying the database with a RDF query language, there is an observation that might seem obvious, but is still worth noting. It is very important to formulate the queries reasonably, since one set of tuples can be retrieved by many different queries, and there can be tremendous differences in terms of performance, between the variations. For example, to retrieve the name of the longest movie (M) of some director (D), the query might be expressed in these two ways (phrased in natural language): (1) *select movie M directed by D, where M is longer than every other movie directed by D*, and (2) *select all movies directed by D, sort the results by duration in descending order, return the first result*. In the first query, many comparisons have to be made to get the result, since all of the movies need to be compared with each other. The second query, however, only performs sorting and limiting, which makes it much easier to evaluate.

4 Semantic network of artworks

Besides the semantic network building toolset SemNet, the other outcome of the thesis is a semantic network of works of art, hereinafter called *ArtNet*. The purpose of this product is at least twofold. Firstly, since it was built using the tools of SemNet, it is an informal proof of its functionality. Secondly and more importantly, it finds its use in methods of natural language processing and some other fields as well. Applications, qualities and extent of this network are further discussed in this chapter.

4.1 Design

The network was designed to meet the requirements established in the assignment of this work. It was supposed to contain information on works of any kind of art, with concepts mapped to WordNet synsets. The kinds of art chosen for ArtNet are film and literature, mainly because of wide availability of online resources concerning these fields. Two hosts have been selected as the sources of data for the network. One of the main criteria for the choice of the hosts was introduced in section 3.5.2 – the distinguishability of the entity type by URL, to which both of the selected hosts comply. The scrapers used for data extraction utilise mainly the XPath technology, combined with regular expressions in cases where the data elements were not addressable by XPath. The database of the network resides in a Sesame native store – a file based triple store.

Applications

The network was primarily designed for natural language processing applications, mainly to aid in the automatic processing and understanding of free text, acting as a lexical resource containing proper names of entities from the domain of art. However, it could find use in other areas of research as well. For example, it could be used as a part of a knowledge base of some intelligent agent, to support reasoning. Provided there were no legal issues concerning the licensing of the contained data, it could also serve as a public resource of information on works of art.

WordNet mapping

The extracted information is mapped to WordNet concepts using the mapping defined in the `wn_map.xml`[1] file, which is a configuration for the `StatementMapper`. For this purpose, the OWL/RDF representation[2] of WordNet 2.0 is used, where the terms are represented by dereferencable URIs and the rich structure of relationships of WordNet is expressed in the OWL language and some of them also in RDF(S), to support compatibility with agents incapable of OWL reasoning (which is also the case of the Sesame framework used in SemNet).

To allow for wider spectrum of applications, the `bootstrap` option of `SesameWriter` was used to preload the network with the WordNet hyponymy set, which acts as a class hierarchy. This application of the WordNet data is not available directly, but can be achieved by first adding the following two statements to the database [12]:

1. `wn20s:Synset rdfs:subClassOf rdfs:Class`

2. `wn20s:hyponymOf rdfs:subPropertyOf rdfs:subClassOf`

The statements in the example[3] express the facts that: (1) a WordNet synset should be considered a class in terms of RDFS, and (2) the hyponymy relationship should be treated as the subclass relationship. The benefit arising of having the database aware of this hierarchy, is the possibility to automatically infer supertypes of every entity added to the database. For example, in the case of statement: "the type of 'Morgan Freeman' is *actor*", the system automatically infers, that "Morgan Freeman" is also a *person* and a *living thing*.[4]

1. Locations of files on the attached medium can be found in Appendix B.
2. The OWL representation of WordNet is created by W3C and is available at `http://www.w3.org/2006/03/wn/wn20/instances/`.
3. The namespace `wn20s` refers to the WordNet 2.0 OWL representation schema and stands for `http://www.w3.org/2006/03/wn/wn20/schema/`, and `rdfs` is the namespace of RDF Schema – `http://www.w3.org/2000/01/rdf-schema`.
4. This example is simplified for easier comprehension. In fact, the statements in the store are related to resource identifiers, not directly to literals, so they look more like this: *The resource `http://www.csfd.cz/tvurce/92-morgan-freeman/` is of type `actor`, and is labelled "Morgan Freeman".*

4.1.1 Selected sources

ČSFD[5]

The acronym stands for *Česko-Slovenská filmová databáze* (Czechoslovak film database) and despite its name, the database is not limited to movies shot in Czech and Slovak Republic. At the time of writing, it contained roughly 270 000 movies, 45 000 actors and 18 250 directors. All of the content is maintained by a community of volunteers.

Most of the factual information available on the pages were collected, including all available names of individual movies (the primary and localised variants), genres, country of origin, year of release, duration, respective directors and actors and their birth dates. The classes of objects collected from the host – film, actor and director are mapped to WordNet senses[6] film#1, actor#1 and director#3, respectively.

Databáze knih[7]

Similarly to ČSFD, this host contains community-created content, with the total of 74 000 books and 24 000 authors. For the books, the collected attributes are title, year of release, author, ISBN, and for the authors it is their name, pseudonyms, date of birth and demise, and nationality. One additional relationship is that between a short story and the book containing it. The WordNet concepts corresponding to the entity types are writer#1, book#1 and short story#1.

4.2 Properties

In terms of quality of the data, the contents of the network are on the same level as the chosen source hosts. As to the extent of the network, it contains 244 000 movies, 58 000 actors/directors, 73 000 books, 23 000 literary authors and millions of relationships.

5. ČSFD is located at `http://www.csfd.cz/`.
6. The number after the hash (#) is the word sense number.
7. Databáze knih is located at `http://www.databazeknih.cz/`.

5 Conclusion

In this thesis, an approach to building of domain-specific semantic networks by using the methods of web crawling and scraping has been presented. The goals of the thesis – to create tools for building of such networks, and to build a network of works of art – have been achieved, if not surpassed, by developing a custom web crawler and using it for creation of the network. The tools have been developed with modularity and extensibility in mind and are designed as a framework, thus opening possibilities for further development.

Bibliography

[1] Russell Ackoff. From data to wisdom. *Journal of Applied Systems Analysis*, 16:3–9, 1989.

[2] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986 (Standard), January 2005.

[3] Randall Davis, Howard Shrobe, and Peter Szolovits. What is a knowledge representation? *AI Magazine*, 14(1):17–33, 1993.

[4] Christiane Fellbaum. *WordNet: an electronic lexical database*. Language, speech, and communication. MIT Press, 1998.

[5] Thomas Gruber. Toward principles for the design of ontologies used for knowledge sharing. *International Journal of Human-Computer Studies*, 43:907–928, November 1995.

[6] Thomas Gruber. Ontology. In Ling Liu and Tamer M. Özsu, editors, *Encyclopedia of Database System*. Springer-Verlag, 2008.

[7] Sang Ho Lee, Sung Jin Kim, and Seok Hoo Hong. On URL normalization. In Osvaldo Gervasi, Marina Gavrilova, Vipin Kumar, Antonio Laganà, Heow Lee, Youngsong Mun, David Taniar, and Chih Tan, editors, *Computational Science and Its Applications – ICCSA 2005*, volume 3481 of *Lecture Notes in Computer Science*, pages 122–130. Springer Berlin / Heidelberg, 2005.

[8] Viviana Mascardi, Valentina Cordì, and Paolo Rosso. A comparison of upper ontologies. Technical Report DISI-TR-06-21, Dipartimento di Informatica e Scienze dell'Informazione (DISI), Università degli Studi di Genova, Via Dodecaneso 35, 16146, Genova, Italy, 2006.

[9] M. Mealling and R. Denenberg. Report from the Joint W3C/IETF URI Planning Interest Group: Uniform Resource Identifiers (URIs), URLs, and Uniform Resource Names (URNs): Clarifications and Recommendations. RFC 3305 (Informational), August 2002.

[10] Marvin Minsky. A framework for representing knowledge. Technical report, MIT, Cambridge, MA, USA, 1974.

[11] NISO Press. Understanding Metadata. *National Information Standards*, 2004.

[12] Guus Schreiber, Mark van Assem, and Aldo Gangemi. RDF/OWL Representation of WordNet. W3C working draft, W3C, June 2006. http://www.w3.org/TR/2006/WD-wordnet-rdf-20060619/.

[13] John F. Sowa. Semantic networks. In Stuart C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*. Wiley, 2 edition, 1992.

[14] Chaim Zins. Conceptual approaches for defining data, information, and knowledge. *Journal of the American Society for Information Science and Technology*, 58:479–493, 2007.

Appendix A

Crawler configuration

```
<crawler>
  <dbLayer>
    <driver>org.postgresql.Driver</driver>
    <url>jdbc:postgresql://localhost:5432/postgres</url>
    <user>artnet</user>
    <password>artnet</password>
    <schema>artnet</schema>
    <autoCommit>true</autoCommit>
  </dbLayer>
  <threadsPerHost>2</threadsPerHost>
  <globalCrawlDelayMinimum>500</globalCrawlDelayMinimum>
  <policyIgnored>false</policyIgnored>
  <fakeReferrer>true</fakeReferrer>
  <hosts>
    <host>
      <baseURL>http://www.csfd.cz/</baseURL>
      <name>CSFD.cz</name>
      <charset>UTF-8</charset>
      <crawlDelay>1500</crawlDelay>
      <sourceFirst>true</sourceFirst>
      <source>
        <pattern update="1">/kino/?</pattern>
        <pattern update="1">/tvurci/?</pattern>
        <pattern update="60">/tvurci/.+</pattern>
      </source>
      <entities>
        <entity weight="3">
          <pattern update="365">/film/[^/]+/?</pattern>
          <scraper>
            xsmeral.artnet.scraper.CSFDScraper$Film
          </scraper>
        </entity>
        <entity weight="2">
          <pattern update="365">/tvurce/[^/]+/?</pattern>
          <scraper>
            xsmeral.artnet.scraper.CSFDScraper$Creator
          </scraper>
        </entity>
      </entities>
    </host>
    <host>
      <baseURL>http://www.databazeknih.cz/</baseURL>
      <name>DatabazeKnih.cz</name>
      <charset>UTF-8</charset>
      <crawlDelay>1500</crawlDelay>
      <sourceFirst>true</sourceFirst>
      <source>
        <pattern update="1">/</pattern>
        <pattern update="1">/dnesni-autori/?</pattern>
        <pattern update="1">/dnesni-knihy/?</pattern>
        <pattern update="1">/dnesni-povidky/?</pattern>
        <pattern update="365">/autori/?</pattern>
        <pattern update="365">
          /index.php\?(?=.*stranka=autori)(?=.*id=\d+).*
        </pattern>
        <pattern update="365">/vydane-knihy/[^/]+</pattern>
      </source>
      <entities>
        <entity weight="1">
          <pattern update="365">/knihy/[^/]+</pattern>
          <scraper>xsmeral.artnet.scraper.DBKnih$Kniha</scraper>
        </entity>
        <entity weight="1">
          <pattern update="365">/povidky/[^/]+/[^/]+</pattern>
          <scraper>xsmeral.artnet.scraper.DBKnih$Povidka</scraper>
        </entity>
        <entity weight="1">
          <pattern update="365">/autori/[^/]+</pattern>
          <scraper>xsmeral.artnet.scraper.DBKnih$Autor</scraper>
        </entity>
      </entities>
    </host>
  </hosts>
</crawler>
```

Appendix B

Contents of the attached CD

- `/`
  - `data` .... **Collected data of ArtNet**
    - `sesameData` .... The network, in Sesame native store format
    - `pg_dump` .... State of the crawler, dump of Postgres DB
  - `doc` .... **Javadoc documentation**
    - `PipedObjectProcessor`
    - `SemNet`
  - `proj` .... **NetBeans projects with source files**
    - `ArtNetScrapers` .... The scrapers for ArtNet
    - `CustomTaglets` .... Auxilliary classes for documentation
    - `PipedObjectProcessor` .... Piped object processor
    - `SemNet` .... Semantic network building toolset
  - `sample` .... **Sample semantic network – ArtNet**
    - `artnet` .... Configuration files for crawler, mapper, Sesame writer
    - `bin` .... The runtime of SemNet, JAR files
      - `lib` .... Third-party libraries
    - `scrapers` .... JARs of the ArtNet scrapers
    - `serql` .... Sample SeRQL queries
    - `sql` .... SQL scripts for creation of DB schema and monitoring