Michael Cochez

Semantic Agent Programming Language: use and formalization.

Master's Thesis
in Information Technology
March 13, 2012

UNIVERSITY OF JYVÄSKYLÄ
DEPARTMENT OF MATHEMATICAL INFORMATION TECHNOLOGY

Jyväskylä

Author: Michael Cochez
Contact information: michaelcochez@gmail.com
Title: Semantic Agent Programming Language: use and formalization.
Työn nimi: Semantic Agent Programming Language: käyttö ja formaalistaminen.
Project: Master's Thesis in Information Technology
Page count: 92
Abstract: This thesis gives an overview of languages used in the Semantic Web for data representation and querying. Then it gives a formalization of the Semantic Agent Programming Language (*S-APL*), which is a Semantic Web language for agent programming. The formalization consists of syntax and query definition, and definition of the dynamic structure of a *S-APL* document. Further, it is shown why the formalization is needed.
Suomenkielinen tiivistelmä: —
Keywords: ontology, logic based languages, *S-APL*, semantic web
Avainsanat: ontologia, logiikkapohjaiset kielet, *S-APL*, semanttinen web

Glossary

ANTLR ANother Tool for Language Recognition — A parser library and language

ASCII American Standard Code for Information Interchange — A character encoding scheme

DR Description Resources — A way of describing a set of resources on the web using RDF

EBNF Extended Backus-Naur Form — A formal model for describing context-free grammars

FIPA Foundation for Intelligent Physical Agents — A standard body specialised in agent systems

FOAF Friend of a Friend — An ontology specifying concepts for describing people, their relations and their occupations.

$\mathcal{G}$ General Context — the root of the hierarchy in a *S-APL* document

HTML HyperText Markup Language — Markup language for web pages

IETF Internet Engineering Task Force — A standard body

IP address Internet Protocol address — An address from the addressing scheme used in the Internet

IRI International Resource Identifier — A type of identifier used on the Internet

ISO International Organization for Standardization — A standard body

ITU International Telecommunication Union — A standard body

MAC address Media Access Control address — Unique address for communication on the physical network layer

N3 Notation3 — An expressive concrete syntax for RDF graphs

OWL Web Ontology Language — A framework for defining ontologies in RDF

RDF Resource Description Framework — A language for describing resources

RDF/XML RDF encoded as XML

RDFS Resource Description Framework Schema — A schema language for RDF

RFC Request for Comments — An IETF memorandum on Internet standards and protocols

S-APL Semantic Agent Programming Language — A concrete syntax for the RDF language, *S-APL* adds possibilities for dynamic documents and agent pro-

gramming

SPARQL SPARQL Protocol and RDF Query Language — A query language for RDF graphs

SQL Structured Query Language — A management and query language for relational databases

SWRL Semantic Web Rule Language — A rule language used in the semantic web

UBIWARE A semantic agent platform where agents use *S-APL* for beliefs storage and messaging

UCS Universal Character Set — Set of characters which is aiming to cover all symbols used in written and visual communication

URI Universal Resource Identifier — A type of identifier used on the Internet

URL Universal Resource Locator — A subclass of URI, which contains web addresses of resources

URN Uniform Resource Name — A type of identifier used on the Internet

UUID universally unique identifier — A set of identifiers which are very likely to be unique

W3C World Wide Web Consortium — A standard body

XHTML eXtensible HyperText Markup Language — Attempt to integrate HTML into an XML document

XML eXtensible Markup Language — A format for representation of structured data

XRI Extensible Resource Identifier — A type of identifier intended for use on the Internet, but not an accepted standard

Contents

1 Introduction

"The Semantic Web is a web of data" [1]. Data is produced at a very high rate nowadays and this data is not available enough. The data is produced and used by applications, often in formats unreadable by or unreachable for other ones. Another problem is that the data is not linked, i.e., there is no way to relate fragments of information to each other.[1] The Semantic Web aims "To do for machine processable information (application data) what the World Wide Web has done for hypertext"[2].

In order to reach the goals of the Semantic Web, several standards and languages have been introduced. Among these are languages to represent data like RDF, schema languages like RDFS, query languages like SPARQL and even rule languages like SWRL. These and others are described in section 2.

The UBIWARE platform (see also section 3.5.2) is a multi-agent platform which is based on semantic technologies. A multi-agent platform is a software platform on which independent software components (agents) perform certain tasks. While this platform was being developed, it was noticed that the existing languages for the semantic web were not sufficient for the purposes of the platform. One reason is that the different available languages are not interchangeable with each other, since their encoding is different. This is, however, only a practicality and could be ignored in theory. The main shortcoming of the existing languages is that they do not allow removal and change of information. It is for instance impossible to first state the capital of a country to be X and then redifine it to become Y. The problem is that it is impossible to state that certain information has become invalid. The agents on the UBIWARE platform need this capability, since an agent needs to have an updated view on the current state of its environment. Therefore, a new language for the semantic web was developed and named Semantic Agent Programming Language (*S-APL* language). Next to having possibilities for removal of invalid information, the language also provides advanced constructs for agent programming as described below in section 3.5.

The aim of this thesis is twofold. To begin with, it tries to show that the *S-APL* language is not restricted to agent programming and that the language needs to be formalized. In the second place, a formalization of the *S-APL* language is elabo-

rated. A formalization is defined on WordNet as "the act of making formal (as by stating formal rules governing classes of expressions)" [3]. The point of a formalization is thus to state formal rules which should enforce certain properties. In the case of *S-APL*, the formalization means the statement of a mathematical description of the language and its properties.

Because it is more logical and easier to give examples, the answer to these two research questions is given in oposite order. The second research question which this thesis tries to answer is how one can make a formalization of the *S-APL* language. This research question is answered in chapter 3. The first question about the need for the formalization is elaborated in chapter 4.

The rest of this chapter describes mathematical preliminaries needed for the thesis and prefixes used in examples.

1.1 Mathematical preliminaries

This section contains a description on the mathematics needed in this thesis. Much of the information of this section is taken directly from the book "Calculus" by James Stewart [4], and Wikipedia articles on graph theory [5] and [6].

set As set is a collection of objects which are called the elements of the set. If S is a set, then the notation $a \in S$ means that a is an element of the set and $a \notin S$ means that a is not an element of the set S. The empty set, i.e., the set without any elements is denoted $\varnothing$. A set can be described by listing its elements between braces or by using set-builder notation. An example of set-builder notation could be

$$\{x | x \text{ is a car } \}$$

Which is the set of all x such that x is a car.

size For a finite set S, the number of elements in S is denoted | S | and is always a natural number.

union The union of two sets A and B, denoted $A \cup B$ is the set which contains an element if it is an element of either A or B.

intersection The intersection of two sets, denoted $A \cap B$ is the set which contains an element if it is an element of both A and B.

subset S is a subset of a set A, denoted $S \subset A$, if all element of S are also elements of A.

power set The power set (or powerset) of any set S, which I will denote 2^S, is the set of all subsets of S. For instance, the power set of the set $\{a, b\}$ is $\{\{\}, \{a\}, \{b\}, \{a, b\}\}$. It can be shown that $|2^S| = 2^{|S|}$

partitioning A set of nonempty subsets is called a partition of a set A if every element x in A is in exactly one of the subsets.

tuples A tuple is an ordered list of elements enclosed by braces and separates by commas. For instance $(1, 2, 3)$ is the tuple containing the numbers 1, 2 and 3 in that order. When I use the word "n-tuple", I mean a tuple which has n elements. A tuple of two elements is sometimes called a pair, for more elements there are words like triple, quadruple, quintuple, etc...

functions A function f is a rule that assigns to each element x in a set A exactly one element, called $f(x)$, in a set B. Here, A is the domain of the function and B is the range. A function can be denoted by writing the assign rule directly or by a set of 2-tuples which have all a different first element from the set A and a second element from the set B. The value of a function defined with tuples, for an element a, is the second component from the tuple where the first component is a.

directed graph A directed graph or digraph is a pair $G = (V, A)$ where

- V a set whose elements are called vertices or nodes,
- A a set of ordered pairs of vertices, called arcs, directed edges, or arrows.

Label A label is a value associated with a node. It uniquely identifies the node.

Reachable Reachable is the ability to get somehow from one node to another node. One can state that a node A is reachable from a node B if one can traverse the graph, following edges and nodes, from A to B.

1.2 Definitions of used prefixes

For the remainder of this document, I define the following namespaces to be used in different examples and code listings. If the form prefix:suffix is used, where the prefix is one of the ones defined here, it should be interpreted as the concatenation of URI associated with the prefix, and the suffix.

prefix	URI associated with prefix
ex	http://www.example.org/
jyu	http://www.jyu.fi/concepts#
owl	http://www.w3.org/2002/07/owl#
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs	http://www.w3.org/2000/01/rdf-schema#
sapl	http://www.ubiware.jyu.fi/sapl#
saplvar	http://www.ubiware.jyu.fi/saplvar#
xsd	http://www.w3.org/2001/XMLSchema#

2 Languages used for representation of semantic data

For the representation of data in the Semantic Web, i.e., semantic data, several languages and models have been developed. In this chapter, I will give an overview of the main languages which are actively used in Semantic Web development. Other notable languages have been developed which overlap in features with the languages described here. One criteria used for the inclusion here is whether the language is accepted as a standard by the World Wide Web Consortium (W3C), which attempts to guide and standardize developments in the Semantic Web area.

2.1 Resources and identifiers

When using the Internet, the need arises to refer to 'things' in the real world. There are things which are tangible like food, furniture, buildings and so on. Others, however, are non-tangible e.g. feeling, weather, service, temperature and digital documents. All these 'things', both tangible and non-tangible are known as resources. In order to refer to objects and concepts in the real world, one needs some kind of identifier. This identifier should uniquely and unambiguously refer to the real world concept. In this section, I will give an overview of different technologies and standards which are used as identifiers for resources.

2.1.1 URL, URI, URN and family ...

With the appearance of the Internet, there was a need for identifying resources over the network. Initially, computers on the Internet network had an Internet Protocol address (IP address) consisting of 32 bits.[7] One way of representing these addresses is by grouping them eight bits at a time (4 octets or bytes) and then using the decimal representation of the integer represented by the byte. An example of this kind of address would be *130.234.4.129* . This way of representing addresses made memorizing easier but was still too difficult for humans to remember. The main problem was that the Internet started to grow exponentially and more and more addresses came into use and also the machine identified by a given address changed once in a while. A solution was found by assigning a name to each com-

puter in the network which mapped host names to the numerical addresses. Initially, this mapping was centrally maintained and became known as the Domain Name System. The Internet, however, grew exponentially and a centralized maintenance of this mapping became unfeasible. Therefore, a decentralized system was elaborated, which is still in use in the Internet nowadays. [8, 9] In 1994, the URI working group together with Sir Tim Berners-Lee created RFC 1738 "Uniform Resource Locators". [10] This document specifies the syntax and semantics for a compact string representation for location and access of resources via the Internet. These strings are known as URLs. An example of a URL would be `http://www.jyu.fi` . The specification of URLs is derived from concepts defined in RFC 1630 "Universal Resource Identifiers in WWW" [11], which defined a much wider class of identifiers. The identifiers specified in this document, known as URIs, are used to encode the names and addresses of objects on the Internet. It must be noted that the specification was explicitly made open for future extension. As said literally in the request for comments 1630 :

> "The web is considered to include objects accessed using an extendable number of protocols, existing, invented for the web itself, or to be invented in the future. Access instructions for an individual object under a given protocol are encoded into forms of address string. Other protocols allow the use of object names of various forms. In order to abstract the idea of a generic object, the web needs the concepts of the universal set of objects, and of the universal set of names or addresses of objects."[11]

Later on, in RFC 1737 "Functional Requirements for Uniform Resource Names" [12], specified URNs. A URN is a URI which is using the urn scheme. These identifiers are used for identification, as opposed to URLs which are used for locating or finding resources. Later RFCs like 2141 [13] "URN syntax" suggest requirements for presentation, equivalence and transmission of URNs.

In an attempt to make URIs more international, International Resource Identifiers (IRI) where proposed in RFC 3987 "Internationalized Resource Identifiers (IRIs)" [14]. IRIs are defined as a complement to URIs and add support for characters from the Universal Character Set, also known as UCS [15]. The RFC also describes how IRIs can be mapped to URIs. Because of software compatibility reasons, it was decided that a new protocol element would be defined instead of changing the existing definition of URIs. IRIs are currently the biggest accepted superset of the original URIs. The syntax of an IRI is as follows

IRI = scheme ":" ihier-part ["?" iquery] ["#" ifragment]

Where *schema* is the schema in use, like for example http, ftp, gopher, mailto, telnet, file, . . . *ihier-part* contains first two forward slashes then possible authorization information and a possible hierarchical identifier for the path. Then follows optionally an encoded list of query parameters in *iquery* and a fragment identifier in *ifragment*

Some examples of IRIs follow:

- `http://xn--rsum-bpad.example.org/`
- `http://résumé.example.org/`
- `URN:ISBN:0-395-36341-1` see also [16]
- `http://users.jyu.fi/~miselico`
- `mailto:john@example.com?body=send%20info` see also [17]
- `ftp://user:password@host:21/path`

Another notable attempt to identify resources in the Internet was done by the "OASIS Extensible Resource Identifier (XRI) Technical Committee" [18]. This specification defines URI in the xrn: scheme and extends the syntax of IRIs. The extensions provided by XRIs over IRIs are:

- Persistent and re-assignable segments. The XRI syntax does allow the internal components of an XRI reference to be persistent or re-assignable. A re-assignable component can be reassigned by by an identifier authority. The meaning of the XRI can thus dynamically change at any point in time. This gives the benefit that of time of creation, the whole resource identifier does not have to be know. One could for example specify that the identifier refers to the home page of the current boss of a certain firm.

- Cross-references. XRI references can recursively contain other XRI or IRI references. This way, XRIs can contain certain meta-data or semantic information.

- Additional authority types. However not commonly encountered by users, IRI and older schemes allow for authorization. This is mainly an artifact of the Internet Protocol allowing for authorization. XRIs support a superset of the authorization schemes used supported by IRIs. The extension is twofold:

  - global context symbols (GCS). These symbols are used to indicate the global context of the identifiers. Symbols in use are (=, @, + and $) which refer to Person, Organization, General public and Standards body respectively.

  - cross-references, which enable any identifier to be used as the specification of an XRI authority. This way, an authority can be identified by any other XRI.

- Standardized federation. URI syntax does not give requirements for federated identifiers. The specification of these is then done in specific schemes. XRI syntax standardizes federation of both persistent and re-assignable identifiers at any level of the path.

Despite the many benefits it would have offered, the proposed XRI standard is rejected by a ballot. [19]

2.1.2 UUID

The Universally Unique IDentifier (UUID) is defined as an ISO standard [20] and as an request for comment RFC 4122 [21]. Fortunately, all these definitions are technically compatible. It should be noted that the RFC defines the UUID in function of defining a URN namespace for UUIDs.

A UUID is a 128 bit long identifier, which means that 2^{128} or 16^{32} different identifiers can be made. This enormous amount has many benefits. Firstly, a centralized authority for administration is not needed since even at very high allocation rates, the probability of a collision is negligible. Secondly, UUIDs are unique and persistent which makes them useful as Uniform Resource Names. And at last, the length of UUIDs is fixed and can be aligned in the memory of most modern computer architectures, which makes comparing, sorting, hashing and storing in databases a lot easier and more efficient when for example compared to IRIs.

When represented in string form, a UUID looks for example like this

f81d4fae-7dec-11d0-a765-00a0c91e6bf6

UUIDs come in different versions and variants, depending on the variant the parts of the UUID have a certain meaning or are random generated. Different variants have different ways of generating the UUID. One uses the MAC addresses of the network interface to guarantee uniqueness, some use pseudo-random number generators, and also cryptographic hashing and application-provided text strings are used. One drawback of UUID is that implementers may wrongly assume that they provide some kind of security. For example, using a predictable random number

source (as most pseudo-random number generators are) for generating the UUIDs will result in a security flaw.

2.1.3 IRI vs. UUID

One might ask the question whether a system should use UUIDs or IRIs to represent external resources. The first point which should be made is that IRIs can be seen as a superset of UUIDs, because all UUIDs can be represented as a URN which is an IRI. On the other hand, the representation of UUIDs might give sufficient benefits for the implementer in terms of both space and time efficiency to prefer the presentation over IRIs. However, when IRIs are used and in the implementation pointers to these memory addresses are used (and re-used by for example interning), the argument of speed efficiency is void. Further, when the internal UUID representation has to be mapped to an external IRI for translation, the system might also greatly use the memory benefit of UUID. One more argument in favor of IRIs is that they are easily human interpretable. A system which needs to be programmed by a person, might benefit from using IRIs. The conclusion is that UUIDs are a good idea when the system does assign ids to resources itself or when the system does not have to communicate about those identifiers to other systems. Otherwise, the use of IRIs will be more advantageous or at least not cause considerable overhead.

2.2 RDF for data representation

Resource Description Framework (RDF) is a framework which is defied by W3C as a recommendation and used for representation of information. The first specification was written by Lassila et. all in "Resource Description Framework (RDF) Model and Syntax Specification" [22]. This specification got together with the older specification of RDFS (see section 2.4.1) replaced by six recommendations in 2004. These are called in short Primer [23], Concepts [2], Syntax [24], Vocabulary [25], Semantics [26] and Test Cases [27]. This review of RDF focuses on the later revisions and relevant parts of these recommendations are described in further sections. The information represented by RDF is mostly located on the web, but can also be stored offline. An abstract syntax is defined to link concrete syntaxes to formal semantics.

The same recommendation provides a motivation on why RDF is needed. The first reason is to provide meta-data for web resources. A concrete implementation

providing this functionality is RDFa, which is described further in section 2.3.1. The second motivation from the recommendation is that RDF should provide a way of defining data in an open data model. Moreover, RDF should allow data from different sources to be processed in varying contexts leading to new information. Lastly, it should enable automated processing of Web information by software agents. Making the Web a world-wide network of cooperating processes.

The designers of RDF had in mind the creation of a simple data model suitable for formal semantics and provable interference. The used vocabulary would consist of URIs and the syntax would make be XML and make use of XML schema datatypes. Some parts of RDF are, however, not URIs; since there is a broad support for literals. The XML datatypes are used in conjunction with these literals and are the ones defined in the first version of "XML Schema Part 2: Datatypes" [28] which got revised later in "XML Schema Part 2: Datatypes Second Edition" [29]. The benefit of using XML and XML schema datatypes is that RDF and XML data are easier to be transformed into each other. One final design goal was that anyone could make statements about any resource. Thus allowing explicitly that data can be separated over different locations and added to existing data at any time. On the other hand, this also allows one to produce statements that are inconsistent with other statements or plain incorrect.

In the further extend of this section, I will give a more concrete view on RDF. The first subsection gives a view on the abstract syntax defined for RDF. Further subsections give a view on RDF-XML, N3 and the Turtle language as concrete syntaxes for RDF.

2.2.1 RDF abstract syntax

RDF uses an abstract syntax which is used for defining what a concrete implementation must be able to handle and for formal proofs. This means that a concrete implementation can do optimizations or use any internal format as long as it is able to achieve the same results. Concrete implementations are described in further subsections. RDF uses a graph data model consisting of nodes, which can be subject, object or both. The nodes are connected trough directed arcs which labels are predicates. A node from which an arc leaves, is a subject for that predicate and a node to which an arc arrives, is an object for that predicate. Every arc (or equivalently predicate) thus connects a subject to an object. This arc can be denoted as the triple $(Subject, Predicate, Object)$.

Figure 2.1: RDF graph representing one statement.

Let us take a look at the example graph shown in figure 2.1. We see a node with label `http://users.jyu.fi/~miselico`, from which an arc leaves. This must thus be a subject node. The second node, which has the label `http://www.jyu.fi`, has an arc arriving to it from which we know that it is an object node. The arc connecting the two nodes is a predicate and has label `http://www.jyu.fi/concepts/studiesAt`. We can encode this information as the triple (`http://users.jyu.fi/~miselico`, `http://www.jyu.fi/concepts/studiesAt`, `http://www.jyu.fi`). Note here that the direction of the arc is important. Also when defining triples the order of the elements is significant. The meaning of this triple is that the relationship given by the predicate holds between the subject and the object, but not necessarily the other way around. When the graph consists of more nodes and arcs, the meaning of the graph is the conjunction of the meaning of all the triples.

When a certain thing in the world is unknown, but one still would want to make statements about it, blank nodes can be used. A blank node also called an anonymous node can be seen as a node without a label, but still unique in the graph, i.e., no two empty nodes are equal in the graph. This does, however, not imply that they cannot refer to the same resource the real world.

There is a restriction on the labels allowed in the graph. For a subject, the only allowed labels are a URI reference (see section 2.1.1) or a blank node, the label of a predicate can only be a URI reference and the label of an object can be a URI, a blank node or a literal. Important to note is that the URI is in most cases not to be interpreted as the location of anything but as an identifier for something.

Literals are used to indicate values e.g. a number, date, name or binary data. A literal can have an XML schema datatype, which puts the literal in the datatype's value space. Literals without any datatype are considered to be of type xsd:string and can have an optional language tag indicating the language of the literal. The data encoded in a literal could also be indicated by URIs, but literals are considered

more convenient. On the other hand, this adds complexity for concrete implementations.

It is useful to have a concrete definition which tells when two graphs are equivalent. This definition is adapted from [2, 6.3 graph equivalence]

1. M maps blank nodes to blank nodes.
2. M(lit)=lit for all RDF literals lit which are nodes of R.
3. M(uri)=uri for all RDF URI references uri which are nodes of R.
4. The triple (s, p, o) is in R if and only if the triple (M(s), p, M(o)) is in R'

This definition assumes a definition of equivalence of URIs and literals. These are described in the standard but not included here for brevity.

Further paragraphs will describe concrete implementations of the RDF abstract syntax. The RDF standard also includes models for adding meaning to specific RDF graphs. [26] This includes support for some type of reification (see section 2.2.7), containers, collection and others (see also section 2.2.4). One important part of the RDF standard is RDFS which is further described in section 2.4.1.

2.2.2 N-Triples

N-Triples, which is not a recommended syntax for RDF is defined in "RDF Test Cases" [27, 3. N-Triples]. The N-Triples language was created for definition of easy test cases. The reason for inclusion in this thesis is that the N-triples format is the most plain model which can be used to express RDF, resulting in a model which allows simpler proofs. N-Triples is a subset of N3 (see section 2.2.4), leaving out any construct which can be simplified. A exact EBNF is available from the standard. Simplified, the structure of a N-triples document can be stated as follows: The document has 1 statement per line. Each statement describes a triple, i.e., subject predicate and object with the same limitations as the abstract RDF data model. In order to encode a blank node, the notation "_:" followed by an identifier local to the document is used. To refer to the same blank node from another statement, the same identifier has to be used. Literals are denoted as an ASCII string surrounded by quotation (") marks and contain an optional datatype or language tag. The following example is adapted from the test cases collection [27, rdf-charmod-literals/test001.nt] :

```
_:a <http://example.org/named> "D\u00FCrst" .
<http://w3.org/test> <http://example.org/Creator> _:a .
```

Note that URIs have to be enclosed in angular brackets and literals, which can contain escaped characters, by quotation marks. Statements are finalized with a dot.

2.2.3 RDF/XML

The concrete syntax for RDF which got endorsed by the World Wide Web Consortium, together with the revised RDF standard, is RDF/XML as defined in "RDF/XML Syntax Specification (Revised)" [24]. This syntax is encoded as XML, which is a standard language used for encoding structured data. XML is also a W3C standard and the last revision got defined in "Extensible Markup Language (XML) 1.0 (Fifth Edition)" [30]. XML is a subset of an older standard called SGML, applying restrictions on allowed document trees. The main goal of XML is to allow data to be served, received and processed on the web. I assume the basic concepts of XML to be known to the reader. I do not include them here for sake of brevity. Next to the published standard, there exists several books explaining it. One source covering XML is for example the book "Learning XML, Second Edition" [31].

The encoding of RDF in XML needs a mapping from the statements represented by the abstract graph to XML components. Then, these components have to be put in one valid XML document. Concrete, RDF/XML uses the XML QNames to represent the URIs used in the abstract graph. All QNames have a namespace and a short local name. QNames in XML can have a prefix which is resolved against the prefixes valid in the scope. Otherwise, the QName is declared in the default namespace of that context in which is is used. Another way to represent URIs of subjects and objects is in attributes of an XML element. Literals can only be stored as element text or attribute.

The conversion between the abstract graph and RDF/XML is further described in the standard. The result of mapping the graph to XML is supposedly easily machine and human readable. However, regarding the many proposals which appeared later, one could argue that the RDF/XML does not fulfill that promise. I will not include details about the actual conversion between the abstract graph and the XML/RDF notation, since it is of limited relevance to this thesis.

2.2.4 N3

The Notation3 (N3) language is not accepted as a recommendation. Its latest Team Submission at W3C was in 2011 "Notation3 (N3): A readable RDF syntax" [32] The N3 language is an assertion and logic language. N3 is able to describe more as the abstract RDF syntax and provides thus an expressiveness beyond the graphs possible in RDF. The reason why I describe this language is because it has had a strong influence on the *S-APL* language which I will describe in chapter 3. N3 extends RDF by adding the possibility to add formulae, variables, logical implication and functional properties. The N3 syntax is not XML based and has plenty of syntactic sugar, aiming at a higher readability.

I will not give a complete coverage of all syntactic features in the N3 language. The features described here are in my opinion the most interesting ones or had the biggest influence on the *S-APL* language and are therefore most relevant for this thesis work. A basic N3 document looks like an N-Triples document. However, a lot of syntactic sugar is put on top and structures are added.

Namespaces Namespaces are defined using the @prefix directive. A directive of this form looks for example like this:

```
@prefix jyu: <http://www.jyu.fi/concepts#>
```

After this directive, the prefix bar: is said to be defined and has value `<http://www.jyu.fi/concepts#>` . When statements are declared after this directive, they can use the prefix. For example, jyu:professor , would be a shorthand for `<http://www.jyu.fi/concepts#professor>` , i.e., the prefix got replaced by its value.

Base URIs A feature similar to @prefix in the sense that it is a directive which changes the meaning of statements following the directive. The @base directive sets the URI to be used as a base URI when parsing relative URIs.

Shorthands The following shorthands are defined:

a <http://www.w3.org/1999/02/22−rdf−syntax−ns#type>

= <http://www.w3.org/2002/07/owl#sameAs>

=> <http://www.w3.org/2000/10/swap/log#implies>

<= <http://www.w3.org/2000/10/swap/log#implies> but in the inverse direction

Formulae An RDF document is equivalent to a set of statements like N-triples or its graph. The graph cannot have another graph as value for a subject or object. This is exactly where N3 extends RDF; a graph can itself be used as the value of a node in another graph. Put another way, a graph can be put as the subject or object of a statement which itself belongs to another graph. The nested graph is referred to as formula. To nest a graph, it has to be written between curly brackets and put where normally a subject or object would appear. The meaning of a subgraph, is the logical conjunction of the statements. The statements in the subgraph form an unordered set. An example of subgraphs could be as follows:

```
{ jyu:miselico jyu:studiesAt <www.ub.tg> } a n3:falsehood .
```

Which means that the conjunction of the statements in the subgraph is false. A formula is only defined by its contents. The description of N3 does not provide precise semantics for formulae, i.e., the interpretation is left open.

Blank nodes N3 provides several ways to represent blank nodes. Firstly, there is the _: form known from N-triples. It must be defined, however, what the meaning is of blank nodes inside formulae. The creators of N3 chose to define that blank nodes can only refer to blank nodes in the formula it occurs directly in. This means that blank node identifiers cannot refer to 'surrounding' graphs.

The second way N3 allows to define a blank node is without any identifier. Instead the so called square bracket notation is used. The notation is syntactic sugar for the _: form. A statement of the form `[a b] c [d e]` can be equivalently written as

```
_:x a b .
_:x c _:y
_:y d e
```

Where _:x and_:y are identifiers different from possible other identifiers in the document.

The last way to define blank nodes is implicit by using a feature called paths. Paths are used to describe a certain type of relation in a concise form. For instance, the statement `x!p` stands for `[x p]`. Another example is `x^p` stands for `[p x]`. From these two notations, whole chains can be build, much like in natural languages. For example `:Joe!fam:mother^fam:mother!loc:office!loc:zip` could

mean something like "The zip code of the office of someone who's mother is also the mother of Joe."

Quantification Quantification allows one to use the existential and universal quantifiers for variables which can then be used in statements. Variables quantified in outer graphs can be used in subgraphs. One example could be

```
@forAll <#person>. @forSome <#drink>. <#person> <#drinks> <#drink> .
```

This means that for all persons, there is some drink where the person drinks the drink.

Lists Representing lists in RDF is rather cumbersome, since RDF has a graph structure and thus no order between nodes.

The solution is to a use blank node, indicating the start of the list. From this blank node, two predicate arcs leave. The first one is labelled `rdf:first` and ends at the node which is the first element of the list. The second one is labelled `rdf:rest` and ends at a node indicating the tail of the list. The tail of the list is actually itself a list defined in the same way or rdf:nil, indicating that the end of the list is reached. For instance the list with elements "x1", "x2" and "x3" would be represented as depicted in figure 2.2. The complete N3 code of the picture is `<#a> <http://example.org> ("x1" "x2" "x3") .`

Repetition When a subject or a subject-predicate pair has to be repeated multiple times, one can use shorthand notation. The first version is for repeating subject where `s1 p1 o1 ; p2 o2` is shorthand for `s1 p1 o1 . s1 p2 o2 .` The second version is for repeating subject and predicate where `s1 p1 o1 ; o2` is shorthand for `s1 p1 o1 . s1 p1 o2 .`

There are plans to define other N3 notations as subsets of N3. Both N-triples described above and Turtle described in the next section are strict subsets of N3.

2.2.5 Turtle

Turtle is a language which is a superset of N-triples described in section 2.2.2 and a subset of Notation3 described in the section 2.2.4. The main design strategy of the Turtle language is to extend N-triples with the most useful things from N3. Turtle is not yet an officially endorsed standard, but has its current definition is a Team Submission in "Turtle – Terse RDF Triple Language" [33]. One problem with the

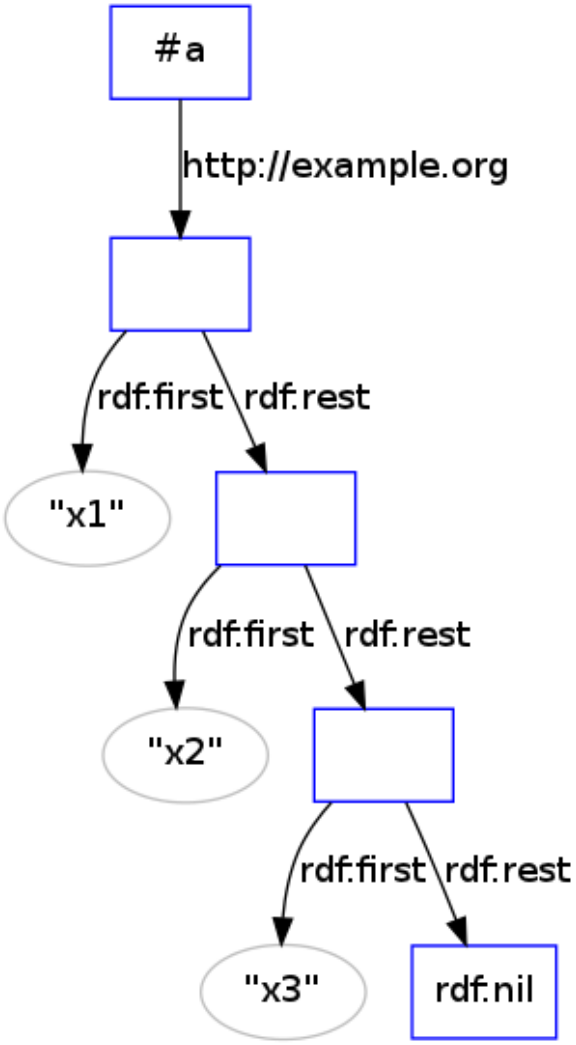

Figure 2.2: RDF graph representing a list with elements x1, x2 and x3.

officially adopted RDF/XML language as described in section 2.2.3 is that it in its current form not able to encode all possible RDF graphs. The exceptions are quite peculiar, like for example that a QName cannot start with a number and that certain UNICODE code points are not allowed in XML 1.0. These problems do not apply to Turtle (nor to N3). Another design idea is compatibility with the query language part of the SPARQL Protocol And RDF Query Language (SPARQL) which will be described in section 2.5.1. Since Turtle is a subset of N3, the following descriptions will be rather short if the information can be found in the previous section.

A document is a sequence of triples written in the form subject predicate object . Thus subject, predicate and object separated by white space and finalized with a dot. URIs are enclosed in square brackets and can use prefixes just like in N3 notation. In the newer proposed versions of Turtle, a multi-line string literal is added as a possible syntax. Blank nodes can only be added by _: notation. Both @prefix and @base are supported and so is repetition just like in N3. Furthermore, there is support for writing certain numerical types directly into the document. Thus, without the need for quoting and providing an XML schema type. This means that 5 can be the object of a statement, which is equivalent to "5"^^xsd:integer. Turtle has the same notation for lists as N3.

2.2.6 N-Triples vs. RDF/XML vs. N3 vs. Turtle.

Choosing the best language among the four languages presented here is impossible and also does not make much sense. All four language can describe quasi the same languages. (With a minor exception for RDF/XML.) The choice on whether the one or the other language is better is depending on the context in which the language will be used. N-triples, with its very simple structure, is very tempting for formal proofs. This mainly because while performing the proof there are very little exceptions to be taken into account. RDF/XML is than again easier to to interchange since XML parsers exist for all major programming languages. Another important aspect is the readability. RDF/XML although written in XML which is supposed to also assist humans in reading data, is much more complicated to understand as the syntactic sugar used in Notation3 or Turtle. Perhaps the most general purpose language is Turtle, since it tries to be both simple and advanced by taking parts of N3. However, the re-definable @base and @prefix directives make the documents a lot less human readable.

2.2.7 Reification of statements

One very relevant concept of RDF is reification of statements, which is best explained with an example for which we will use the Turtle language. Assume that we have a triple which looks like

```
@prefix  ex: <http://www.example.org/> .
ex:s ex:p ex:o .
```

Now, we want to make a statement about this triple like for example who the author of the triple is. This can be done by as follows:

```
@prefix  ex: <http://www.example.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
_:s rdf:type rdf:Statement .
_:s rdf:subject ex:s .
_:s rdf:predicate ex:p .
_:s rdf:object ex:o .

_:s ex:creator ex:author1
```

Thus, first we tell that some blank node is of type rdf:statement and then state the subject, predicate and object in separate statements. Finally, we can make more statements about the blank node like the fact that the ex:creator is ex:author1 .

2.3 Frameworks using RDF to represent data.

Some frameworks have been adapting the RDF representation of data to represent their own data. This section describes the frameworks RDFa and POWDER. The RDFa standard uses RDF representation of data to include semantic data in XHTML documents. POWDER uses it to describe the content of other documents.

2.3.1 RDFa

The RDFa standard is defined by W3C in "RDFa in XHTML: Syntax and Processing" [34]. The goal of RDFa is to make the structured data which is available on the web also accessible to tools and applications. The idea is that tools are unable to read and interpret the data which is intended for humans to read. When publishers are on the other hand able to express the content in a machine readable form, this problem would be solved.

RDFa specifies attributes to describe the structure of data independent of the actual surrounding markup language. In the referred standard, the eXtensible HyperText Markup Language (XHTML) is used, which is one standard for the markup of extensible documents for the web [35]. The data which gets included in the document is RDF. However, the RDF is not included as one block of data in the header of the document. Instead the designers opted for inclusion of attributes in tags of the existing XHTML structure. Because the data is RDF, publishers are allowed to extend on the recommendation and add their own data on top of what is standard. The rules for interpretation of the data are specified independently on the used format of representation. For instance, consider the following document fragment without any semantic information:

```
<ul>
  <li>Shoes</li>
  <li>43</li>
  <li>White</li>
</ul>
```

To a human, this document describes a pair of shoes of size 43 which have a white color. To a machine, however, this is a list of separate items. The machine is not able to put connections between the different items on the list. What can be done with RDFa is for example the following:

```
<ul xmlns:ex="http://example.org/ typeof="ex:Product">
  <li property="ex:Product">Shoes</li>
  <li property="ex:hasShoeSize">43</li>
  <li property="ex:hasColor">White</li>
</ul>
```

This data is usefully annotated and a system which has knowledge about the ontology, can use the data and give a meaning to it. Any information on the web could be annotated in a similar way like for example ratings for movies, links between pages, persons in images, etc...

2.3.2 POWDER

The Protocol for Web Description Resources (POWDER) is described in several separate recommendations [36] [37] [38]. The aim of POWDER is to aid content discovery, protection from unwanted content and increase quality of semantic searches. In order to achieve this aim, it provides a machine readable way to describe web

resources. These descriptions should then guide the user to content of interest. Further improvements could be achieved in efficiency of data retrieval, matching with user profiles, rating of trustworthiness, adaption to the used device, the users privilege level etc... [39] The semantics of POWDER (or in other words its use of RDF) lies inside the description of resources. A POWDER document is an XML document which contains an attribution section and describes so called Description Resources (DR). The attribution section contains the issuer of the information in the document, using the Friend of a Friend (FOAF) [40] or Dublin Core [41] ontology, or references to RDF/XML documents containing the information in such form. The DR contains a selector which describes to which resources (IRIs) this DR applies. Then it contains the actual description which consists of RDF/XML properties with literal values. In addition to this data, a human readable description (in the displaytext tag) and icon (in the displayicon tag) can be included. An example can be found in listing 2.1. This example was showcased in [39].

```
<?xml version="1.0"?>
<powder xmlns="http://www.w3.org/2007/05/powder#"
    xmlns:ex="http://example.org/vocab#">
  <attribution>
    <issuedby src="http://example.org/company.rdf#me" />
    <issued>2007-12-14T00:00:00</issued>
  </attribution>
  <dr>
    <iriset>
      <includehosts>example.com</includehosts>
    </iriset>
    <descriptorset>
      <ex:color>red</ex:color>
      <ex:shape>square</ex:shape>
      <displaytext>Everything here is red and square</displaytext>
      <displayicon src="http://authority.example.org/icon.png" />
    </descriptorset>
   </dr>
</powder>
```

Listing 2.1: "An example POWDER document"

The meaning of this example is that the issuer about which more information can be found from http://authority.example.org/company.rdf#me declares that any resource in the domain example.org is red and square, assuming that this are the semantics connected to ex:color and ex:shape.

2.3.3 Use of RDF as embedded information structure.

The sections on RDFa 2.3.1 and POWDER 2.3.2 where examples on how other standards make use of the extendability of RDF. Both standards chose to embed information described with RDF into other documents. This use of RDF could help the understanding of existing and newly created documents. It is perhaps questionable why the POWDER standard, which has been designed much later as the appearance of RDF is not entirely an RDF/XML document. For the RDFa standard, this is understandable since the standards from which XHTML derives are much older and generally supported.

2.4 RDF structure languages

Because the data which can be represented with RDF is in principle without any limits, there is a need to define some structure for the data. The approach taken is to define the meaning of certain parts of the data in RDF form and the combination of the data and its description forms the knowledge. Further two different technologies are described. RDFS is a language with limited expressive powers able to make statements about resources. OWL is a very expressive language which makes statements about individuals and properties. It is then possible to use the ontological information and the data to reason more information and even give answers to certain question about the data.

2.4.1 Resource Description Framework Schema (RDFS)

The Resource Description Framework Schema 1.0 (RDFS) was originally specified in a separate Candidate Recommendation called "Resource Description Framework (RDF) Schema Specification 1.0" [42]. The new recommendation is spread over the above-mentioned documents specifying the RDF standard. The parts relevant to RDFS can be found in the Vocabulary [25] and the Semantics [26] document, I will only consider the main parts of the specification. The goal of RDFS is to describe other RDF data. RDF can be seen as a language stating properties of resources. When looking at a triple, the subject is the described resource, the object is the property value and the predicate is the actual property which is being described. In RDF the property value can be either a literal or an arbitrary resource. RDF does not provide any mean to describe the properties themselves and relations among them.

This is where RDFS comes in by providing the concept of classes and properties giving meaning to other resources and properties. RDFS does not intend to specify specific properties which can be used in RDF documents, it provides a mean to specify your own properties and classes and their relations. RDFS is itself encoded as RDF and can thus for example accompany an existing document, which becomes self descriptive.

The basic idea of RDFS classes, is that classes are not described in terms of properties. It is the properties which are described in terms of the classes they apply to. In order to define a property, one must define the range and domain, i.e., a set of classes which can be used as subject or object of the property respectively. The main benefit of this approach is that properties can be added to classes at any point, without the need to modify the class itself. This way there is no need to have a centralized and managed repository of class descriptions. Stating that a resource is an instance of a certain class is done by adding the rdf:type property with as a value the resource representing the class. For example ex:MyInstance rdf:type ex:MyClass, makes the resource ex:MyInstance an instance of the class ex:MyClass . Interesting is that the class itself can also be member of other classes. This allows to define for example the class of all classes which define groups of people. It could even be that a class is instance of itself as for instance the class of all classes which is known as rdfs:Class. A class can be a subclass of another class. All instances of a class C, which is subclass of class D, are also instances of class D. This relation between classes is stated using the property rdfs:subClassOf .

A property is a relation between the subject and the object of a statement. RDFS adds the notion of a sub-property. If a property P is a sub-property of property P', then all subject-object pairs which are related by the predicate P are also related by the predicate P'. An intuitive example of a sub-property is when we consider a father-son relation being a sub-property of the parent-child relation, i.e., when a person is father of a certain boy, he is also the parent of that child. The sub-property relation is indicated by the property rdfs:subPropertyOf . As mentioned above properties are defined by specifying the domain and range of the property. This is indicated by the properties rdfs:range and rdfs:domain respectively. It is interesting to note that these two are properties themselves, having rdfs:Property as their domain and rdfs:Class as their range. Next to properties and classes, RDFS provides the properties rdfs:label and rdfs:comment, which allow a human-readable version of the resource name and a human-readable comment to be attached to a resource respec-

tively.

Lastly, the RDFS specification describes container and collection classes for RDF. The goal of the container classes is giving a unified way to define certain types of containers like bags, sequences, alternatives. RDFS does not specify any different formal requirements for these three types of containers, they are rather a convention for the human reader of the documents. The collection classes define lists in a similar way as described in the section about lists in Notation3 (see section 2.2.4).

2.4.2 Web Ontology Language (OWL)

Just as RDFS, the Web Ontology Language (OWL) languages has two versions. The first version was defined in 2004 in “OWL Web Ontology Language Semantics and Abstract Syntax” [43] and got redefined in 2009 in "OWL 2 Web Ontology Language Document Overview" [44] and related documents. The later version is often referred to as OWL 2 and since this version is an extension of the previous one, I will describe the later. OWL 2 is a Semantic Web language used to describe things, groups of things and the relation among them. The knowledge described is logic-based in order to enable computers to reason based on this data. A computer program could for example determine the consistency of a set of data or infer knowledge only implicitly available in the data set. OWL 2 can be encoded as an RDF graph, which makes it possible to write OWL in the various concrete syntaxes described above in section 2.2. The designers of OWL used a Functional-Style syntax to describe OWL. The reason for this is that that syntax is supposedly more convenient for specification and implementation of various tools. The functional-style syntax and the RDF representation are equivalent as is shown in "OWL 2 Web Ontology Language – Mapping to RDF Graphs" [45]. The OWL 2 language has an overlap with RDFS which was previously described in section 2.4.1. The authors, however, decided to not reuse it entirely and defined for instance the resource owl:Class which is the type of classes in OWL 2.

The "OWL 2 Web Ontology Language Primer" [46] gives a concise description of the OWL 2 language and its intentions. OWL 2 is created to express ontologies, i.e., a set of descriptive statements about some domain of interest. These statements can be of different kinds e.g. natural language definitions of terms, their interrelation with other terms and assertional knowledge about the considered domain. Having said that an OWL 2 document consists of a set of statements, it should become clear that OWL 2 is not a programming language. OWL 2 declares, i.e., it represents the

current state of an environment in a logical way without giving any information about how this state is reached or modified. Moreover, next to not being a programming language, OWL 2 is also not a schema language for syntax conformance nor a database language. The problem for enforcing syntax conformance is caused by the open world assumption used in RDF and the Semantic Web in general. In accordance to this assumption, one cannot tell that information does not exist if it is not available in the currently available data. For example, if one asserts that a certain property has one and only one value associated with it, it is impossible to tell whether a data set is conform or not if that property is not present for that instance in the available part of the data. OWL 2 is not a database, because it does not define in any way the form the data should have, nor does it give any mean for storing data.

The modeling of data in OWL 2 is based on three basic notions. There are axioms which are basic statements of the ontology, entities which are references to real-world objects and expressions which combine entities in complexer descriptions. Axioms are statements which can be true or not true as opposed to entities and expressions for which a truth value does not make sense. Entities can be either objects (called individuals), categories (called classes) or even relations (called properties). Also expressions are some kind of entity, but instead of being atomic they are defined by their structure.

The way OWL 2 works reminds partially of the working of RDFS. Therefore, I will not provide as much details about the exact RDF triples used to describe the statement as given in the RDFS section. The notation is somewhat similar, but does not add directly to the scope of this thesis.

First, one can make class hierarchies and assign individuals to classes. One can say that classes are subclasses of each other, equivalent, disjoint, etc... Furthermore, one can define a class as an enumeration or an intersection, union or complement of classes defined elsewhere, which is much more as anything RDFS provides. Then, properties can be assigned to individuals just like a normal RDF statement. Moreover, it is possible to state that an individual does not have a certain property, which is a very strong tool. Note that in normal RDF it is not possible to state information not being true.

OWL 2 provides constructs similar to RDFS to define properties with certain hierarchy, cardinality, domain and range. The possibilities for ranges also include restrictions, intersections, union, complements and enumerations of values of XML

Schema Datatypes. This can be illustrated by restricting the xsd:integer Datatype which represents the whole numbers to a certain allowed range. This is done in a similar way to XML Schema Datatypes facets. Also properties provide a mean to define classes. One could for example define the class of teachers to be all individuals that are linked to a student by the hasStudent property, which would be a sound definition. Further, one can state that two entities in the data referred to with different identifiers are the same in the real world or just different.

On top of all this, OWL 2 provides a way to define characteristics of properties. It is for example possible to define a property being the inverse of another one or being the result of a chaining of properties, e.g., chaining a property standing for a father of relation two times, results in a grandfather of relation. Furthermore, let A, B and C be individuals, then it is possible to state that that a property is ...

symmetric If A is connected to B trough this property, then B is connected trough this property to A. An example could be the property linking siblings together.

asymmetric If A is connected to B trough this property, then B is not connected trough this property to A. An example could be the property linking children to their parents.

disjoint No two individuals are linked by both properties. For instance, the property linking a man to his parents and the property linking a woman to her parents.

reflexive The property relates everything to itself. As an example, one could take the property which connects individuals with the same last name.

irreflexive The property does never relate individuals to themselves. For instance a property which connects A with B, if A was created before B.

functional If A is linked to B trough a property which is functional, then A cannot be linked to another individual trough the same property. An example of a functional property is the property connecting a person with his/her mother.

inverse functional If A is linked to B trough a property which is inverse functional, then B cannot be linked to from another individual trough the same property. An example could be the property connecting a company to its address. Assuming that two companies cannot share an address.

transitive If A is linked trough the property to B and B is linked trough the property to C, then A is linked trough the property to C. For instance a property which connects A with B if A was created before B or the property relating siblings to each other.

Direct model-theoretic semantics as specified in "OWL 2 Web Ontology Language Direct Semantics" [47] and RDF-based semantics as specified in "OWL 2 Web Ontology Language RDF-Based Semantics" [48] are two different ways of defining the semantic meaning of an OWL ontology. The difference is that the former is using a descriptive model to define the meaning while the later is using RDF graphs as a model for the ontology. The differences between both are very technical and not relevant enough for this thesis. However, interesting to note is that the interpretation given by the Direct model-theoretic semantics is decidable, i.e., it can find any answer which can be found in the given data set.

OWL is further divided in different so called profiles of which a few predefined ones are described in "OWL 2 Web Ontology Language Profiles" [49]. A profile is a restriction on the expressive power of the OWL language and so describes a subset of the language. The reason for these restrictions are mainly because the expressive power of OWL makes computation too hard, i.e., both time and space complexity go beyond reasonable limits. Examples of restrictions are disallowing of negations and disjunction. Specific profiles have a specific set of restrictions and are designed with specific use cases in mind which do not benefit from the excluded possibilities.

2.5 Query languages for Semantic data

In this section, I will introduce the SPARQL query language for RDF data. Many query languages have been elaborated for data retrieval from RDF graphs. Moreover, not all semantic data is represented by a concrete RDF syntax. When also considering semantic data which is not RDF, also languages which do not query RDF data can be seen as Semantic Web query languages. [50]

Because the *S-APL* language, which is the main topic of this thesis, is mainly concerned with RDF-like data, languages which query RDF are most relevant. Many of this type of languages like for instance SquishQL, RDQL, TriQL and SPARQL are much influenced by the SQL relational database query language. I decided to describe the SPARQL language, since it is the query language which has had the strongest influence on *S-APL* and is a W3C recommendation.

2.5.1 SPARQL

When data is stored, it is often needed to extract very specific information from it. The same is true for the data stored in an RDF graph. The most popular query language for RDF graphs is 'SPARQL Query Language for RDF' (SPARQL) which is a W3C Recommendation described in "SPARQL Query Language for RDF" [51]. SPARQL is developed alongside other languages used in the Semantic Web and has for instance had a strong influence on the Turtle language discussed in section 2.2.5. SPARQL is strongly influenced by the select statement of the SQL language. The current SPARQL recommendation does not include any way of updating or adding data to data sets. The newer version which is not a recommendation yet will include a way to update the data set as well. [52] The most relevant part of SPARQL for this thesis are its querying abilities, because they have had a strong influence on the queries used in the *S-APL* language. I will focus on those features which are also available for querying in *S-APL* and leave out many significant features of SPARQL. Syntactic sugar used in SPARQL is similar to Turtle's and includes predicate-object lists, object lists, RDF collections and the use of the 'a' as a shorthand for 'rdf:type'. The way queries are performed in the *S-APL* language is described in section 3.2.

SPARQL queries can be grouped according to the type of result they return. Four different forms are distinguished:

SELECT
: Returns values bound to variables.

CONSTRUCT
: Returns an RDF graph based on filling of variables in a user specified graph template.

ASK
: Returns whether the pattern in the query could be matched.

DESCRIBE
: Returns an RDF graph containing data associated with given resources.

I will focus on the 'CONSTRUCT' type of query since this is the one which is similar to the queries used in *S-APL*. In some examples, I left the prefix declarations out for brevity, they are the same as the ones defines in section 1.2.

First, we can look at an example of such a query. Assume we have the following RDF data, here in Turtle notation:

```
@prefix ex: <http://www.example.org/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

ex:student1 foaf:name "Michael Cochez" .
ex:student2 foaf:name "John Doe" .
```

We can then write a SPARQL CONSTRUCT query as follows:

```
PREFIX ex:      <http://www.example.org/>
PREFIX foaf:     <http://xmlns.com/foaf/0.1/>

CONSTRUCT { ?x a ex:student . ?x ex:hasName ?name }
WHERE  { ?x foaf:name ?name }
```

This query consists of two main parts. At first, between the curly brackets after the CONSTRUCT keyword, the graph template is defined. The graph template looks like normal Turtle notation except that some or all terms can be replaced by variables which are denoted by a question mark and the name of the variable. Secondly, between the curly brackets after the WHERE keyword, the query is defined by a graph pattern. This graph pattern also looks (in this example) like Turtle notation with variables added. It is also possible to use a dollar sign ($) instead of the question mark to denote variables. Blank nodes in the graph pattern are scoped locally to the pattern.

The first step in evaluation of the query is searching all possible so called binding sets for the variables. A binding set can be seen as a set of tuples each containing a variable name and the associated value. I chose this way of representing the results instead of the table representation used in the recommendation because I will use a similar way to represent the variable binding in *S-APL* queries.

In this case the binding sets look as follows:

$$\{(?x, ex:student2), (?name, "JohnDoe")\}$$

and

$$\{(?x, ex:student1), (?name, "MichaelCochez")\}$$

Then these binding sets, in this case two, can next be used to fill the variables in the graph template. The graph template is repeated for each of the binding sets with the variables replaced. In case the graph template contains blank nodes, a new blank node is created for each solution. In this concrete example, we get:

```
@prefix ex: <http://www.example.org/> .
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

ex:student2 rdf:type ex:student .
ex:student2 ex:hasName "John Doe" .
ex:student1 rdf:type ex:student .
ex:student1 ex:hasName "Michael Cochez"
```

Starting from this basic query structure, more advanced queries can be build using syntax available in SPARQL. Note that the form of the query does not any longer correspond to a Turtle document. In fact, the query is not even a representation of an abstract RDF tree. The following possibilities modify the query to get a different binding set, many of them combine queries recursively to create more complicated patterns.

FILTER

SPARQL FILTERs limit the binding sets to those for which the expression in the FILTER evaluates to true. The following example limits the binding set to results for which the literal bound to the variable ?o matches the regular expression ".*@jyu.fi", i.e., it returns all triples whose object is a literal ending in "@jyu.fi"

```
CONSTRUCT  ?s ?p ?o
WHERE    { ?s ?p ?o
           FILTER regex(?o, ".*@jyu.fi")
         }
```

The expression used in the FILTER, can consist of numerical operations and comparisons, time expressions, boolean operators and type tests. All operations are constrained using XML Schema Datatypes.

OPTIONAL

A graph pattern can be put as being OPTIONAL. This means that the pattern can optionally be matched. This implies that it is possible for variables to not have any value in case the optional part did not get matched. Therefore SPARQL allows the binding set to include less variables as there are altogether in the query pattern. As a result, it could happen that binding sets of the same query pattern have a different number of elements. As a result of missing variables in the binding sets, not all variable in the graph template can be filled in. SPARQL then leaves the statements using these variables out of the resulting graph. If we have for example the following data in Turtle notation:

```
@prefix ex: <http://www.example.org/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

ex:student1 foaf:name "name1" ; ex:hasCourse ex:course1 .
ex:student2 foaf:name "name2" .
ex:student3 foaf:name "name3" ; ex:hasCourse ex:course1
                              ; ex:hasCourse ex:course2 .
```

Thus student 1 has course 1, student 2 has no courses and student 3 has course 1 and 2. We now construct a query with the OPTIONAL specifier like:

```
PREFIX ex:       <http://www.example.org/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

CONSTRUCT { ?student foaf:name ?name ; ex:hasCourse ?course}
WHERE
  { ?student foaf:name ?name .
  OPTIONAL{?student ex:hasCourse ?course}
}
```

The binding sets found from the graph query are the following:

$$\left\{ \begin{array}{c} \{(?student, ex:student1), (?name, "name1"), (?course, ex:course1)\}, \\ \{(?student, ex:student2), (?name, "name2")\}, \\ \{(?student, ex:student3), (?name, "name3"), (?course, ex:course1)\}, \\ \{(?student, ex:student3), (?name, "name3"), (?course, ex:course2)\} \end{array} \right\}$$

The constructed rdf graph when filling the graph template is as follows:

```
@prefix ex: <http://www.example.org/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

ex:student1 foaf:name "name1" ; ex:hasCourse ex:course1 .
ex:student2> foaf:name "name2" .
ex:student3> foaf:name "name3" ; ex:hasCourse ex:course1
                                            , ex:course2 .
```

Note that duplicate statements are removed since they are not adding any meaning to the rdf graph.

When the optional part of the query would contain any FILTERs, they are only affecting the matching of the optional group. This means that if there is a FILTER in an optional group which fails, it does not affect the failing or succeeding of the surrounding pattern, the only difference is that variables will not be bound.

UNION

With UNION, one can join the result sets of two separate groups. The matching succeeds if either of the query patterns in the UNION succeeds. Only the variables from the matching group will be put in the result set. If both groups match, then each match forms a separate result set. For instance, we could look at all students which are taking courses and professors which are teaching the course. The information is in the following Turtle data:

```
@prefix ex: <http://www.example.org/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

ex:student1 foaf:name "student I" ; ex:hasCourse ex:course1 .
ex:student2 foaf:name "student II" ; ex:hasCourse ex:course1
                                   ; ex:hasCourse ex:course2 .
ex:professor1 foaf:name "prof I" ; ex:teachesCourse ex:course1
                                                   , ex:course2 .
```

Let us now design a query which gives us the name of all students and professors which are related to ex:course1. The problem is now that students and professors are related to courses in different ways. The UNION can help to overcome this problem. The query could for example be defined as in the listing 2.2: ."

```
PREFIX ex:      <http://www.example.org/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

CONSTRUCT { ?person foaf:name ?name }
WHERE {
  {
    ?person foaf:name ?name .
    ?person ex:hasCourse ?course
  }
  UNION
  {
    ?person foaf:name ?name .
    ?person ex:teachesCourse ?course
  }
}
```

Listing 2.2: "Example of a SPARQL query using UNION

The result sets look a follows:

$$\left\{\begin{array}{l}\{(?person, ex:student1), (?name, "studentI"), (?course, ex:course1)\},\\ \{(?person, ex:student2), (?name, "studentII"), (?course, ex:course1)\},\\ \{(?person, ex:student2), (?name, "studentII"), (?course, ex:course2)\},\\ \{(?person, ex:professor1), (?name, "profI"), (?course, ex:course1)\},\\ \{(?person, ex:professor1), (?name, "profI"), (?course, ex:course2)\}\end{array}\right\}$$

The constructed RDF graph then looks as follows and contains the expected results.

```
@prefix ex: <http://www.example.org/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

ex:student1         foaf:name         "student I" .
ex:student2         foaf:name         "student II" .
ex:professor1         foaf:name         "prof I" .
```

Note again that duplicate statements are removed from the resulting graph.

After selecting data with the query pattern graph(s) which can include FILTERs, UNIONs an OPTIONAL, a binding set is created. SPARQL allows this binding set to be modified further by applying so called solution sequence modifiers. As the name suggests, the bindings are treated as a sequence and thus not as a set without order. The solution sequence modifier are very much like their counterparts in the SQL language by which SPARQL is heavily influenced. The following modifiers are available:

Order

The first modifier puts the solutions in a certain order. The order is determined by the expression following the ORDER BY clause. In order to sort even entities which are unrelated, e.g. blank nodes and literals, the standard specifies an order for all different possibilities. I will give an example of the use of this modifier together with the example of the LIMIT modifier in listing 2.7.
The reason for not giving a specific example here is that using the ORDER BY modifier, does not change anything in the constructed RDF graph. This because the RDF graph is represented by an unordered set of statements. When the modifier is used in combination with other modifiers, a different solution graph will be created. When using the SELECT type of query, the solution set will be ordered according to the requested way of ordering.

Projection

Projection is used to only select part of the variables which are bound. I have been doing this implicitly above when not using all variables which were bound in the example about the use of UNION in listing 2.2.

Distinct and Reduced

The solution sequence modifiers SELECT DISTINCT and SELECT REDUCED remove duplicates from the solutions. Since duplicates in the RDF graph will be removed anyway, these modifiers are not of interest for this discussion.

Offset and Limit

In order to reduce the number of elements in the solution sequence, the OFFSET and LIMIT modifiers can be used. Their working is best explained by means of an example. Lets assume we have the data shown in listing 2.3.

```
@prefix ex: <http://www.example.org/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

ex:student2 foaf:name "student II" .
ex:student3 foaf:name "student III" .
ex:student1 foaf:name "student I" .
ex:student4 foaf:name "student IV" .
```

Listing 2.3: Data for examples about LIMIT, OFFSET, ORDER and BIND in SPARQL

The data contains four students with their name. I will start from a query without any solution sequence modifiers and the resulting data set. Then I will add the modifiers and discuss how the data set changes. We start from the basic query shown in listing 2.4.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

CONSTRUCT { ?x foaf:name ?name }
WHERE { ?x foaf:name ?name }
```

Listing 2.4: Basic query in SPARQL

This query will have as a result all the data which was originally available as shown in listing 2.3.

Now, I will apply the LIMIT solution sequence modifier. This modifier will limit the number of result sets used for creating the result. In the example in listing 2.5 the number of considered binding sets is reduced to 2.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
```

```
CONSTRUCT { ?x foaf:name ?name }
WHERE  { ?x foaf:name ?name }
LIMIT 2
```

Listing 2.5: LIMIT query in SPARQL

One possible result is shown in listing 2.6. There are, however, also other results possible. The set of result sets is not ordered, thus any two could have been chosen and included in the result.

```
@prefix ex: <http://www.example.org/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

ex:student2 foaf:name "student II" .
ex:student3 foaf:name "student III" .
```

Listing 2.6: LIMIT query in SPARQL – Resulting data.

The next thing we can do is make the result deterministic by first sorting the sequence and then applying the LIMIT. We have then for example the query as shown in listing 2.7, which instructs to order the result sets by the value given to the name variable in ascending order.

```
PREFIX ex:<http://www.example.org/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

CONSTRUCT { ?x foaf:name ?name }
WHERE  { ?x foaf:name ?name }
ORDER BY ASC(?name)
LIMIT 2
```

Listing 2.7: ORDER BY query in SPARQL

The (unique) result of this query is shown in listing 2.8. The data is limited to two bindings sets and the limitation is performed after the sorting, resulting in a unique result.

```
@prefix ex: <http://www.example.org/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

ex:student1 foaf:name "student I" .
ex:student2 foaf:name "student II" .
```

Listing 2.8: ORDER BY query in SPARQL – Resulting data.

Lastly, we can add an OFFSET solution sequence modifier. This will only start

the selection of result sets with a certain offset. We add to the query an OFFSET of 1 as shown in listing 2.9.

```
PREFIX ex:<http://www.example.org/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

CONSTRUCT { ?x foaf:name ?name }
WHERE { ?x foaf:name ?name }
ORDER BY ASC(?name)
LIMIT 2
OFFSET 1
```

Listing 2.9: OFFSET query in SPARQL

The unique result set is show in listing 2.10

```
@prefix ex: <http://www.example.org/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

ex:student2 foaf:name "student II" .
ex:student3 foaf:name "student III" .
```

Listing 2.10: OFFSET query in SPARQL – Resulting data.

SPARQL has support for querying from multiple RDF graphs and combining of their information. This feature is not relevant to *S-APL* and I will thus not include it in this thesis.

SPARQL 1.1 is not an accepted standard yet, but published as a W3C working draft in "SPARQL 1.1 Query Language" [53]. This new version of SPARQL intends to extend the current specification with for instance aggregation, sub-queries, negation, extensible value testing, etc...The feature I want to describe here is the possibility to create values by expressions. The reason why I pick exactly this feature, is that this is also one feature of the *S-APL* language. The most relevant form of using expressions is using the BIND form. The following example uses the same data set as we used above from listing 2.3. In the query, as shown in listing 2.11, the concatenation of whatever is bound to the variable ?name with itself will be bound to the variable ?doubleName. This variable can than be used in the construction of the RDF graph pattern.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX ex: <http://www.example.org/>

CONSTRUCT { ?x ex:doubleName ?doubleName}
```

```
WHERE  { ?x foaf:name ?name .
    BIND(CONCAT(?name, ?name) AS ?doubleName)
}
```

Listing 2.11: Example of using expressions in SPARQL 1.1

The result of execution is show in listing 2.12.

```
@prefix ex: <http://www.example.org/> .

ex:student2 ex:doubleName "student IIstudent II" .
ex:student3 ex:doubleName "student IIIstudent III" .
ex:student1 ex:doubleName "student Istudent I" .
ex:student4 ex:doubleName "student IVstudent IV" .
```

Listing 2.12: Example of using expressions in SPARQL 1.1 - Resulting data.

2.6 Semantic Web Rule Language

This section describes Semantic Web Rule Language (SWRL) as an exmple of a rule language which can be used in the semantic web. SWRL is not a recommendation, but is filed as a Member Submission under the title "SWRL: A Semantic Web Rule Language – Combining OWL and RuleML" [54] to the World Wide Web Consortium.

A rule language is a language which formally states rules. In this case a rule is something which has a antecedent and a consequent. The antecedent is the condition which must be true in order for the consequent to be true. The following example in "Human Readable Syntax" [1] shows the main feautures of SWRL.

$$Student(?s) \wedge hasCourse(?s, ?c) \wedge Course(?c) \wedge hasStudent(?u, ?s)$$
$$\Rightarrow Institution(?u) \wedge offersCourse(?u, ?c)$$

The first line is the antecedent and the second line contains the consequent. The rule means that if

- there is an instance of class *Student* which we will call *s* and
- *s* has a property *hasCourse* with value *c* and
- *c* is of type *Course* and
- there is a *u* which has a property *hasStudent* with value *s*

[1] SWRL has, just like RDF, one abstract and several concrete syntaxes.

then

- u is of type $Institution$ and
- u has the property $offersCourse$ with value c.

It is thus possible to infer new knowledge from existing knowledge using these rules. One more feauture is the use of algebraic expressions on the variables. This can be illustrated as follows:

$$Event(?e) \wedge hasDurationSeconds(?e, ?l)$$
$$\Rightarrow hasDurationMinutes(?e, ?lm) \wedge ?x = \text{op:numeric-divide}(?l, 60)$$

In this case, it is stated that if a given event has a *hasDurationSeconds* property with value l, then it also has a *hasDurationMinutes* property which has value l divided by 60.

In short, SWRL is able to infer new knowledge from existing knowledge. There is, however, no way of making information invalid or remove knowledge which has been reasoned, but is not valid any longer.

Another related effort is AIR (The Accountability In RDF language), which is an N3 based Semantic Web rule language.[55]. It has some features in common with the below described *S-APL* language, like for instance nested rules, negation and scoping. However, it still does not have the power to do more as adding facts and new rules. Thus it is not able of removal of facts, nor can external code be executed.

3 S-APL language and its formalization

This chapter tries to answer the second research question of this thesis, i.e.,"How can one make a formalization of the *S-APL* language?". The goal is thus to give a sound and formal definition of the *S-APL* language. The structure of this chapter is as follows: First, in section 3.1, I give a definition of the structure of a *S-APL* document. This structure will be an abstract syntax for *S-APL* which is stricter as the one used in the UBIWARE agent platform for which the *S-APL* language was initially developed. Then, I will define how queries can be made using *S-APL* in section 3.2 and describe the limitations compared to query constructs available for agent programming in UBIWARE in section 3.3. The goal of the query definition is to prepare for the rules and dynamics chapter 3.4, where I will define how a *S-APL* document changes itself dynamically. The next section 3.5 discusses the use of the language in agents by showing what the UBIWARE version of S-APL needed to become usable as an agent programming language. The last section 3.6 is a discussion about the use of variables in *S-APL*.

3.1 Syntax definition

In this thesis, I consider a *S-APL* document, not as a concrete tangible document. It is more an abstract syntax for which concrete syntaxes can be made. One concrete notation for *S-APL* documents is the *S-APL* language as used in the UBIWARE multi-agent platform as described in "Semantic Agent Programming Language (*S-APL*): A Middleware Platform for the Semantic Web" [56], which I will refer to as UBIWARE *S-APL*. The language is in that context used to represent the current memory state of an agent including beliefs, desires and intentions, as well as an encoding to exchange messages between agents. *S-APL* can, however, be seen from a more abstract perspective as a language representing a self modifying graph, i.e., a graph in which the way the graph must be modified is described in the graph itself. The first subsections of this section describe the syntax as defined for the original *S-APL* language. In subsections 3.1.3 and 3.1.4 the abstract syntax is formally introduced. Subsections 3.1.5 and 3.1.6 discuss the equivalence between *S-APL* and

RDF, and the benefits thereof. The final subsection 3.1.7 defines an operator which will be used in the next chapter.

3.1.1 Original UBIWARE S-APL definition

The original definition of the *S-APL* language is most detailed described in "Deliverable D1.1 The Central Principles and Tools of UBIWARE" [57] and "Semantic Agent Programming Language (*S-APL*) Developer's Guide"[58]. The developer's guide describes what is know as *S-APL* axioms. The description goes verbatim as follows:

- Everything is a belief. All other mental attitudes such as desires, goals, commitments, behavioral rules are just complex beliefs.
- Every belief is either a semantic statement (subject-predicate-object triple) or a linked set of such statements.
- Every belief has the context container that restricts the scope of validity of that belief. Beliefs have any meaning only inside their respective contexts.
- Statements can be made about context, i.e. contexts may appear as subjects or/and objects of triples. Such statements give meaning to contexts. This also leads to a hierarchy of contexts (not necessarily a tree structure though).
- There is the General Context G, which is the root of the hierarchy. G is the context for the global beliefs of the agent (what it believes to be true here and now). Nevertheless, every local belief, through the hierarchical chain of its contexts, is linked to G.
- Making statements about other statements directly (without mediation of a context container) is not allowed. The only exception is when a statement appears as the object of one of the following predicates: sapl:add, sapl:remove and sapl:erase.

The same document gives also a concrete syntax (which I will call UBIWARE *S-APL*) for the *S-APL* language as follows:

> " The description of *S-APL* notation follows:
>
> - A statement is a white-space-separated sequence of subject, predicate and object

- Dot (.) followed by a white space separates statements of the same level, i.e. S P O . S P O
- Semicolon (;) followed by a white space allows making several statements about the same subject, i.e. S P O ; P O
- Comma (,) followed by a white space allows making several statements having common subject and predicate, i.e. S P O , O
- { } denotes reification, it may appear as the subject or the object of a statement and has to include inside itself one or more other statements, e.g. S P { S P O } or { S P O } P { S P O }. Reification always implies a context; however, the relation is not necessarily 1-to-1. E.g. {S P O} P O ; P O implies that the statement in {} is linked to two different contexts defined as given.
- Colon (:) is used to specify an URI as a combination of the namespace and the local name, i.e. ns:localname There can be default namespace, the colon is used anyway, i.e. :localname.
- @prefix prefix: namespace links a prefix to a namespace.
- URIs given directly are to be inside < >, i.e. <http://someaddress>.
- Literals containing whitespaces, {, }, <, >, ", or : are to be inside " ", i.e. "some literal".
- Comments are java-style, i.e. /* comment */ as well as // comment <end of line>
- Character escaping is java-style as well, i.e. e.g. for " symbol, use \ " while for backslash symbol itself, use \\
- N3 syntax for anonymous nodes with [ ], i.e. S P [P O] or [P O] P O is also supported.
- N3 syntax for RDF lists of resources with ( ), i.e. (R R R ..) P O or S P (R R R ..) is also supported.

c ”

The biggest drawback of this concrete syntax is that there is no formal way of making statements about the document or precise parts of it. Therefore, the developer's guide, this definition is taken from, limits itself to giving a partially informal discussion of the language. Another problem of this concrete syntax is that certain syntactic tricks have to be used to refer to containers from multiple places in the document (see also section 3.5.10).

3.1.2 Removal of syntactic sugar

The *S-APL* syntax as described in [58] allows for the declaration of namespaces. These namespaces are used similarly to the way they are handles in N3 and Turtle (see sections 2.2.4 and 2.2.5). For the rest of this formalization, we assume the prefixes to be expanded completely, i.e., all qualified names are replaced by their IRI and are put between angular brackets. Furthermore, the shorthands used in *S-APL* are replaced with their IRI counterparts according to table 3.1.

Table 3.1: Shorthands in *S-APL*

shorthand	**IRI**
a	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
=>	<http://www.ubiware.jyu.fi/sapl#implies>
>>	<http://www.ubiware.jyu.fi/sapl#achievedBy>
->	<http://www.ubiware.jyu.fi/sapl#impliesNow>
==>	<http://www.ubiware.jyu.fi/sapl#infers>
>	<http://www.ubiware.jyu.fi/sapl#gt>
<	<http://www.ubiware.jyu.fi/sapl#lt>
>=	<http://www.ubiware.jyu.fi/sapl#gte>
<=	<http://www.ubiware.jyu.fi/sapl#lte>
!=	<http://www.ubiware.jyu.fi/sapl#neq>
=	<http://www.ubiware.jyu.fi/sapl#eq>

Then, a unique IRI is assigned for each different blank node in the document. This could for instance be a UUID as described in section 2.1.2 or any other IRIs which can be guaranteed to be unique. The blank nodes are then replaced by there assigned IRIs.

Lastly, *S-APL* allows subject-object, object lists and RDF lists like in Notation3. We assume all these to be expanded. The first deliverable of the UBIWARE project described a way of considering variables and the * symbol to be shorthand notations of certain resources. This, however, lead to some problems when considering expressions in *S-APL* . The problem was that expressions in *S-APL* are literals which contain variables inside e.g. "?variable1+?variable2", which imposes difficulties on considering '?' a shorthand. The solution used, did no longer replace only variables on

the right hand side of rules with their values, but also variables mentioned inside all string literals. In this thesis, I will propose another way of handling the problem with variables used inside expressions in section 3.2.7, the main point, for now, is that variables are not considered shorthand notations.

We will call the combination of prefix expansion and removal, shorthand replacement and list expansion *normalisation* of the document. A document which has been treated this way will be called *normalized*.

Consider for instance the document in listing 3.1 which is using the UBIWARE *S-APL* syntax. In this document, there are prefixes, shorthands and different lists. The normalized form of this document shown in listing 3.2

```
@prefix ex: <http://www.example.org/> .

ex:student2 a ex:Student .
{ ?student a <http://www.example.org/Student> . ?student = ex:student2 }
=>
{ ?student ex:name "Student II"}
```

Listing 3.1: *S-APL* document using syntactic sugar.

```
<http://www.example.org/student2>
<http://www.w3.org/1999/02/22-rdf-syntax-ns\#type>
<http://www.example.org/Student> .
{ ?student
  <http://www.w3.org/1999/02/22-rdf-syntax-ns\#type>
  <http://www.example.org/Student> .
  ?student
  <http://www.ubiware.jyu.fi/sapl#eq>
  <http://www.example.org/student2> }
<http://www.ubiware.jyu.fi/sapl\#implies>
{ ?student <http://www.example.org/name> "Student II" }
```

Listing 3.2: *S-APL* document with syntactic sugar expanded.

Note that also the prefixes declarations have been removed from the document.

In the next sections, I will introduce the definition for *S-APL* documents which is used in this thesis, which is a subset of the documents considered in the UBIWARE *S-APL* definition. The differences are clarified in the footnotes. In some examples, I will use a notation which resembles the UBIWARE *S-APL* closely, since UBIWARE *S-APL* syntax is intuitive to understand and no other concrete syntax is available.

The semantics of the examples further in this thesis will, however, use the semantics as described in the following sections and chapters.

3.1.3 S-APL supergraph definition

In order to define what a *S-APL* document is, I will start from an infinite (directed) graph, from which each *S-APL* document will be a finite subset. The infinite graph, which I will call $\hat{\mathcal{S}}\hat{\mathcal{A}}\hat{\mathcal{P}}\hat{\mathcal{L}}$, consists of nodes and vertices denoted by $\hat{N}$ and $\hat{V}$ respectively.

The set of nodes $\hat{N}$ is partitioned in 5 subsets, i.e., $\hat{N} = \hat{\mathcal{C}} \cup \hat{\mathcal{S}} \cup \hat{\mathcal{R}} \cup \hat{\mathcal{L}} \cup \hat{\mathcal{V}}$ where the intersection of each pair of sets is empty. The subsets are defined as follows:

$\hat{\mathcal{R}}$ = { Resource node with label $IRI|IRI$ is an IRI as defined in RFC 3987 [14] } In other words, $\hat{\mathcal{R}}$ contains one node for each possible IRI. This implies that the graph is infinite, since there is an infinite number of IRIs.

$\hat{\mathcal{L}}$ = {$literal|literal$ is any literal allowed in N3 (see section 2.2.4) } This set contains a node for every possible literal.

$\hat{\mathcal{V}}$ = { Variable node with label $x|x$ matches regex $'\backslash?[a-zA-Z]+|\backslash*'$} Thus $\hat{\mathcal{V}}$ contains the nodes representing all possible variables. The node with label '*' indicated as v_* is the universal matching variable.

$\hat{\mathcal{S}}$ is the set of all possible statements in a *S-APL* document. In order to ease its definition, I introduce three new sets: [1]

S = $\hat{\mathcal{R}} \cup \hat{\mathcal{C}} \cup \hat{\mathcal{V}}$, i.e., the set of possible subjects of statements.

P = $\hat{\mathcal{R}} \cup \hat{\mathcal{V}}$, i.e., the set of possible predicates of statements.

O = $\hat{\mathcal{R}} \cup \hat{\mathcal{C}} \cup \hat{\mathcal{V}} \cup \hat{\mathcal{L}}$, i.e., the set of possible objects of statements.

When we now introduce the notation of a statement which has subject s, predicate p and object o as $st_{(s,p,o)}$, then we can define

$$\hat{\mathcal{S}} = \left\{ st_{(s,p,o)} | s \in S, p \in P, o \in O \right\}$$

[1] The implementation used in the UBIWARE platform uses a slightly bigger set by also allowing Literals in the Subject and Predicate.

$\hat{\mathcal{C}}$ is the set containing a node for every possible context in an *S-APL* document. A container node is defined by the statement arcs which leave from it, i.e., the statements in the context. We denote the context node from which there is an arc to statements $s_1, s_2, \ldots, s_n$ as $c_{\{s_1,s_2,\ldots,s_n\}}$. This way, we can define

$$\hat{\mathcal{C}} = \left\{ c_{statements} | statements \in 2^{\hat{\mathcal{S}}} \right\}$$

The set of vertices of the infinite graph can be partitioned in 4 subsets. $\hat{V} = \hat{\mathcal{CS}} \cup \hat{\mathcal{SS}} \cup \hat{\mathcal{SP}} \cup \hat{\mathcal{SO}}$ These subsets are defined as follows:

$\hat{\mathcal{CS}}$ is the set of arcs connecting contexts to statements. Formally, using previously introduced notation, it is defined as $\hat{\mathcal{CS}} = \left\{ < c_{\{s_1,s_2,\ldots,s_n\}}, s_m > | m \in [1,n] \right\}$ with other words, there is an arc from the context to the statement if the statement is a member of the context.

$\hat{\mathcal{SS}}$, $\hat{\mathcal{SP}}$ and $\hat{\mathcal{SO}}$ are similar sets. They connect the statement nodes with their content. They are formally defined as:

$$\hat{\mathcal{SS}} = \left\{ \left\langle st_{(s,p,o)}, s \right\rangle \right\}$$
$$\hat{\mathcal{SP}} = \left\{ \left\langle st_{(s,p,o)}, p \right\rangle \right\}$$
$$\hat{\mathcal{SO}} = \left\{ \left\langle st_{(s,p,o)}, o \right\rangle \right\}$$

Put another way, $\hat{\mathcal{SS}}$ connect each statement to its subject, $\hat{\mathcal{SP}}$ each statement to its predicate and the arcs in $\hat{\mathcal{SO}}$ make the connection between the statements and their object.

3.1.4 S-APL document definition

In the previous section, I defined a graph which is the supergraph of all possible *S-APL* document graphs. In this section, I will define how a *S-APL* document is defined as a subset of this supergraph. I denote the set $\mathcal{SAPL}$ as the set of all valid *S-APL* documents. A *S-APL* document is a subgraph of $\hat{\mathcal{S}}\hat{\mathcal{A}}\hat{\mathcal{P}}\hat{\mathcal{L}}$ and is defined as a 9 tuple $(\mathcal{C}, \mathcal{S}, \mathcal{R}, \mathcal{L}, \mathcal{V}, \mathcal{CS}, \mathcal{SS}, \mathcal{SP}, \mathcal{SO}, \mathcal{G})$ where

$$\mathcal{C} \subset \hat{\mathcal{C}}, \mathcal{S} \subset \hat{\mathcal{S}}, \mathcal{R} \subset \hat{\mathcal{R}}, \mathcal{L} \subset \hat{\mathcal{L}}, \mathcal{V} \subset \hat{\mathcal{V}},$$
$$\mathcal{CS} \subset \hat{\mathcal{CS}}, \mathcal{SS} \subset \hat{\mathcal{SS}}, \mathcal{SP} \subset \hat{\mathcal{SP}}, \mathcal{SO} \subset \hat{\mathcal{SO}} \text{ and } \mathcal{G} \in \hat{\mathcal{C}}$$

and the following assertions hold:

1. Every node of $\mathcal{C}, \mathcal{S}, \mathcal{R}, \mathcal{L}$ and $\mathcal{V}$ is reachable from $\mathcal{G}$ and all arcs in $\mathcal{CS}, \mathcal{SS}, \mathcal{SP}$ and $\mathcal{SO}$ can be traversed in the process. Informally, this means that all parts of the document are used, i.e., there are no dangling nodes nor edges.

2. $\forall c_{\{s_1,s_2,\ldots,s_n\}} \in \mathcal{C} : \forall k \in [1,n] : s_k \in \mathcal{S}$ and $\left\langle c_{\{s_1,s_2,\ldots,s_n\}}, s_k \right\rangle \in \mathcal{CS}$. Informally, all statements referenced from containers are available and linked by an arc.

3. $\forall st_{(s,p,o)} \in \mathcal{S}$:

   - $s \in \mathcal{R} \cup \mathcal{C} \cup \mathcal{V}$ and $\left\langle st_{(s,p,o)}, s \right\rangle \in \mathcal{SS}$
   - $p \in \mathcal{R} \cup \mathcal{V}$ and $\left\langle st_{(s,p,o)}, p \right\rangle \in \mathcal{SP}$
   - $o \in \mathcal{R} \cup \mathcal{C} \cup \mathcal{V} \cup \mathcal{L}$ and $\left\langle st_{(s,p,o)}, o \right\rangle \in \mathcal{SO}$

   This last assertion holds if all statements are linked to their content and their content is part of the document.

For closer equivalence to the language used in the UBIWARE platform, it would be needed to add the following assertion. This one is, however, not taken into account in this thesis.

1. $\mathcal{G}$ is not reachable from any other node in $\mathcal{C}$, i.e., there is no recursive reference to the 'General Context'.

3.1.5 S-APL document and RDF graph equivalence

The *S-APL* document might, at first sight, look as a superset of the RDF abstract syntax. One can write any valid RDF document graph as a *S-APL* document without any containers. Thus it is obvious that *S-APL* can represent any RDF graph. However, also the opposite is true. The *S-APL* language as defined, can also be represented by an RDF graph. The way this is done, is by what is called reification of statements as was described above in subsection 2.2.7 and is hinted in the "Semantic Agent Programming Language (*S-APL*) Developer's Guide" [58], but not defined. Information about transformation of variable names can be found from "The Central Principles and Tools of UBIWARE"[57]. The following procedure convert any *S-APL* document to RDF by describing how to create a Turtle document (see also section 2.2.5):

1. Add the following prefixes:

@prefix sapl: <http://www.ubiware.jyu.fi/sapl#> .

@prefix saplvar: <http://www.ubiware.jyu.fi/saplvar#> .

@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

2. Assign a unique blank node to each element of $\mathcal{C}$ and $\mathcal{S}$.

3. For each $c \in \mathcal{C}$ and its assigned blank node $\overline{c}$, add an RDF statement of the form " $\overline{c}$ rdf:type sapl:Container ".

4. For each $< c, s > \in \mathcal{CS}$ with $\overline{c}$ and $\overline{s}$ the blank nodes assigned to c and s respectively, add an RDF statement of the form "$\overline{c}$ sapl:hasMember $\overline{s}$".

5. For each $st_{(s,p,o)} \in \mathcal{S}$ with assigned blank label $\overline{st_{(s,p,o)}}$ if

   - $s \in \mathcal{R} \cup \mathcal{L}$ and $\dot{s}$ the label of s, add statement "$\overline{st_{(s,p,o)}}$ rdf:subject $\dot{s}$"
   - $p \in \mathcal{R} \cup \mathcal{L}$ and $\dot{p}$ the label of p, add statement "$\overline{st_{(s,p,o)}}$ rdf:predicate $\dot{p}$"
   - $o \in \mathcal{R} \cup \mathcal{L}$ and $\dot{o}$ the label of o, add statement "$\overline{st_{(s,p,o)}}$ rdf:object $\dot{o}$"
   - $s \in \mathcal{V}$ and $\dot{s}$ the label of s, add statement "$\overline{st_{(s,p,o)}}$ rdf:subject saplvar:$\dot{s}$"
   - $p \in \mathcal{V}$ and $\dot{p}$ the label of p, add statement "$\overline{st_{(s,p,o)}}$ rdf:predicate saplvar:$\dot{p}$"
   - $o \in \mathcal{V}$ and $\dot{o}$ the label of o, add statement "$\overline{st_{(s,p,o)}}$ rdf:object saplvar:$\dot{o}$"
   - $s \in \mathcal{C}$ and $\overline{s}$ the blank node assigned to s, add "$\overline{st_{(s,p,o)}}$ rdf:subject $\overline{s}$"
   - $p \in \mathcal{C}$ and $\overline{p}$ the blank node assigned to p, add "$\overline{st_{(s,p,o)}}$ rdf:predicate $\overline{p}$"
   - $o \in \mathcal{C}$ and $\overline{o}$ the blank node assigned to o, add "$\overline{st_{(s,p,o)}}$ rdf:object $\overline{o}$"

6. Let $\overline{\mathcal{G}}$ be the blank node assigned to $\mathcal{G}$, add statement "$\overline{\mathcal{G}}$ rdf:type sapl:G".

It is obvious to see from this procedure that the length of the document in RDF is much bigger and that the structure is harder to discover. Also concrete implementations will have more difficulties to recover the structure of the *S-APL* document when only given the RDF document. As can be seen from the procedure, one will need 3 RDF statements for each *S-APL* statement, and on top of that 2 statements for describing the membership relations of the statement. If the *S-APL* statement happens to have a container as subject and/or object, then one resp. two extra statements are needed to describe these.

3.1.6 Benefits of equivalence

The equivalence between *S-APL* and RDF, shown by the procedure in the previous section, has the benefit that everything which has been said about RDF is also valid for *S-APL*. One can for example use *S-APL* embedded in XHTML and still comply to the RDFa standard (see section 2.3.1), define an RDFS schema for describing allowed structure (see sections 2.4.1 and 4.3), use any of the described concrete syntaxes for describing an *S-APL* document, use SPARQL queries on *S-APL* documents, and use any other existing tools for handling RDF data. Of course, specific tools for the *S-APL* language will have strong benefits when speed and space constraints are taken into account.

3.1.7 Merging of containers

In this section, I will define an operator which gives a way to merge two *S-APL* containers into one. This operator will be used in further sections, but can be generaly used to merge two *S-APL* documents into one when applied on the generals contexts of the documents.

Given the containers $c = c_{\{sc_1,sc_2,...,sc_n\}} \in \mathcal{C}$ and $d = c_{\{sd_1,sd_2,...,sd_m\}} \in \mathcal{C}$. It is assumed that the containers have been themselves formed by merging of containers of one statement into each other. The merge operation "merge c and d" : $c \Cup d$ is defined in the merge procedure in algorithm 1. Let us try to explain what the result of this procedure is. The first container gets added to the second one in such a way that the resulting container contains the following (**earlier rules overrule later ones**)

- each statement from the first and second container which has a container as subject and object.
- no two statements which have the same subject and same predicate and a container as object. They are joined together by creating a statement which has the same subject and predicate and the merge of both containers as object.
- Similarly, also no two statements exist which have the same predicate and object an a container as subject. They are joined together by creating a statement which has the same object and predicate and the merge of both containers as subject.
- all statement from both documents. Duplicates are removed because contain-

ers are sets.

Note that according to these rules, the merge operator is commutative.

Further, I define the function mergeContainers which takes a set of containers as argument as follows:

$$mergeContainers\colon \{\hat{\mathcal{C}}\} \to \hat{\mathcal{C}}$$
$$mergeContainers(c_1, c_2, \ldots, c_n) \mapsto c_1 \Cup c_2 \Cup \cdots \Cup c_n$$

Algorithm 1 Procedure for merging one container into another one.

procedure MERGE($c_{cContent} = c_{\{sc_{(s,p,o)}, sc_2, \ldots, sc_n\}}$, $d_{dContent} = c_{\{sd_1, sd_2, \ldots, sd_m\}}$)
  if $cContent = \varnothing$ **then** ▷ This is the basic case of the recursion
    return $d_{dContent}$
  end if
  if $s, o \in \hat{\mathcal{C}}$ or $s, o \notin \hat{\mathcal{C}}$ **then**
    return $merge(c_{\{sc_2, \ldots, sc_n\}}, c_{\{sc_{(s,p,o)}, sd_1, sd_2, \ldots, sd_m\}})$
  else if $s \in \hat{\mathcal{C}}$ **then**
    if $\exists sd_{(ds,dp,do)} \in dContent, dp = p, do = o, ds \in \hat{\mathcal{C}}$ **then** ▷ merge is needed
      return $merge(c_{\{sc_2, \ldots, sc_n\}}, c_{\{sd_1, sd_2, \ldots, st_{(merge(s,ds),p,o)}, \cancel{sd_{(ds,dp,do)}}, \ldots, sd_m\}})$
    else
      return $merge(c_{\{sc_2, \ldots, sc_n\}}, c_{\{sc_{(s,p,o)}, sd_1, sd_2, \ldots, sd_m\}})$
    end if
  else ▷ $o \in \hat{\mathcal{C}}$
    if $\exists sd_{(ds,dp,do)} \in dContent, ds = s, dp = p, do \in \hat{\mathcal{C}}$ **then** ▷ merge is needed
      return $merge(c_{\{sc_2, \ldots, sc_n\}}, c_{\{sd_1, sd_2, \ldots, st_{(s,p,(merge(o,do)))}, \cancel{sd_{(ds,dp,do)}}, \ldots, sd_m\}})$
    else
      return $merge(c_{\{sc_2, \ldots, sc_n\}}, c_{\{sc_{(s,p,o)}, sd_1, sd_2, \ldots, sd_m\}})$
    end if
  end if
end procedure

3.2 Queries – binding of variables

The general goal of querying is extracting a subset of information out of a (bigger) set of data. In *S-APL*, the data is represented by a graph and hence, querying means

the selection of a subgraph of it. As described in the previous section, there are different types of nodes in the *S-APL* graph. There are resource, literal, variable, statement and container nodes. Querying in *S-APL* will thus be making a certain selection of these nodes. In order to give names to these selections, we will use variable nodes. To summarise, this section will tell how to formally assign parts (or better nodes) of the *S-APL* graph to variables.

In order to query RDF data, there was the SPARQL language which is described in section 2.5.1. The designers of *S-APL*, however, decided that the best way to query a *S-APL* document is by using another *S-APL* document. The *S-APL* language is expressive enough to be used as a query language itself. In order to make *S-APL* a usable query language some strict semantics had to be introduced, which give certain graphs a certain meaning when used as a query for another graph. Furthermore, a lot of syntactic sugar was introduced to make the writing of queries more convenient and readable. In this section, we will look at how we can define the semantic meaning of certain query constructs. In the next section 3.3, we will look at the limitations of the language described here compared to UBIWARE *S-APL*and how the syntactic sugar can be defined in function of the in this section defined constructs. The reason for working this way is that we should avoid defining too much independent constructs. Defining all construct separately might make the writing of proofs which should cover all cases more cumbersome and lengthy.

3.2.1 Definition of a query, bindingset and operators

In order to talk about queries, we must first define what a query is. We define it as follows:

$$\text{The set of queries } \mathcal{Q} = \hat{\mathcal{C}}$$

This is the same as saying that any container can be a query. I will, however, use the notation $\mathcal{Q}$ in the context of queries to make the difference clear to the reader. The notation used to represent a certain query extends from the notation used to specify a certain container, i.e., $q_{\{qs_1,qs_2,\ldots,qs_m\}} = c_{\{qs_1,qs_2,\ldots,qs_m\}}$ thus it is the query specified by the container containing the statements qs_1 till qs_m.

When querying a *S-APL* document, the goal is to bind certain nodes to variables. A query answer can, however, contain multiple mappings between the variables and nodes. Therefore, we will introduce a notation for the result of a query which is able to represent the whole solution. We call the one possible mapping of a query, a

bindingset of the query and the *S-APL* document. A bindingset looks as follows:

$$bindingset \subset \{(var, value) \,|var \in \hat{\mathcal{V}} \text{ and } value \in \hat{N})\}$$
$$\text{where } \forall (var1, val1), (var2, val2) \in bindingset\colon var1 = var2 \Rightarrow val1 = val2$$

It is thus a set of tuples in which the first component is a variable and the second component a value. Such a tuple with components var and value, we will call a binding for var. Furthermore, there cannot be two tuples defining the same variable in a bindingset. Another way to look at this is that binding is function which maps a subset of the variables $\hat{\mathcal{V}}$ to nodes in $\hat{\mathcal{N}}$

The complete answer to a query is a set of bindingsets. The set of bindingsets looks as follows:

$$queryresult = \overline{QR} \subset \{bindingset | bindingset \text{ is a bindingset } \}$$

Next on, I will define a few operators which act on queryresults and which we will use in further sections. The first operator is Υ which joins two queryresults in the following way:

$$\Upsilon\colon (\overline{QR}, \overline{QR}) \to \overline{QR}$$
$$\alpha \Upsilon \beta \mapsto \left\{ \begin{array}{l} b\alpha \cup b\beta | b\alpha \in \alpha, b\alpha \in \beta\colon \\ \forall\, (var\alpha, val\alpha) \in b\alpha, \forall\, (var\beta, val\beta) \in b\beta : \\ var\alpha = var\beta \Rightarrow val\alpha = val\beta \end{array} \right\}$$

In other words, the function joins two query results together by taking the union of these bindingsets where all bindings which are in both bindingsets have the same value.

It is directly visible from the definition that this operator is commutative and it can be shown that the operator is also associative. Note that for the working of the Υ operator, an operand of $\{\}$ is very different from $\{\{\}\}$. In the former, there are no results, while in the later there is one result which does not contain any bindings. Concrete, when applying Υ on α and $\{\}$, we will get $\{\}$ while when applying it on α and $\{\{\}\}$ we will get α.

Mathematically speaking, querying in *S-APL* is a function which takes as its arguments a container in which the query takes place and a container representing the query. We define the query function as follows:

$$query\colon (\mathcal{C}, \mathcal{Q}) \to \bar{BS}$$

We will define the actual mapping of the function in the next sections. Note that is possible that $\mathcal{G}$ is used as a query or as the container from which the query selects data.

3.2.2 Filling variables

In this subsection, we will define one more function which maps a container and a bindingset to a container. The meaning of this operator is the filling of variables in a container with the bindings from the bindingset. The function is called $fill$ and is defined as follows:

$$fill : (\hat{\mathcal{C}}, \text{ set of bindingsets }) \rightarrow \hat{\mathcal{C}}$$
$$fill\left(c_{\{qs_1,qs_2,\ldots,qs_n\}}, bs\right) \mapsto c_{\{fillStat(qs_1,bs),fillStat(qs_2,bs),\ldots,fillStat(qs_n,bs)\}}$$

where $fillStat$ is defined as

$$fillStat : (\hat{\mathcal{S}}, \text{ set of bindingsets }) \rightarrow \hat{\mathcal{S}}$$
$$fillStat\left(st_{(s,po)}, bs\right) \mapsto st_{(\hat{s},\hat{p},\hat{o})}$$
$$\text{where } \hat{s} = \begin{cases} bs(s) & \text{if } bs(s) \text{ is defined} \\ s & \text{otherwise} \end{cases}$$
$$\hat{p} = \begin{cases} bs(p) & \text{if } bs(p) \text{ is defined} \\ p & \text{otherwise} \end{cases}$$
$$\hat{o} = \begin{cases} bs(o) & \text{if } bs(o) \text{ defined} \\ \text{in place replacement} & \text{see below} \\ o & \text{otherwise} \end{cases}$$

In place replacement happens when the statement has as object a literal of XML schema datatype "sapl:ExpressionLiteral". For instance a statement of the form

```
ex:subject ex:predicate "?x+?y"^^sapl:ExpressionLiteral
```

The effect of in place replacement is that all variables which are inside the literal string and which are in the bindingset are replaced by their values and put back in the literal.

3.2.3 Selection of Literals, Resources, Variables and Containers

This section will show the most essential part of querying *S-APL* graphs, namely querying for literals, resources, variables and even containers in a given container,

by a query which contains only one statement. The fact that this section is placed first, does not imply that it is the first step taken when deciding the result of the query. In some cases, described in further subsections, constructs are recognised in the query and treated first.

Given a query $q_{\{qs_1\}} \in \mathcal{Q}$ and a container $c_{\{s_1,s_2,\ldots,s_n\}} \in \hat{\mathcal{C}}$ where $qs_1 = st_{(qs,qp,qo)}$ is not one of the specially recognised statements described in further sections.

First we define the "matching statement set" (mss) as :

$$mss(c_{\{s_1,s_2,\ldots,s_n\}}, qs_1,) = \left\{ s \;\middle|\; \begin{array}{l} s \in c_{\{s_1,s_2,\ldots,s_n\}} \text{ and} \\ testStatement(qs_1, s) \text{ (see 2) returns } true \end{array} \right\}$$

From mss, we can give a partial definition of the query function, which is:

$$query\colon (\hat{\mathcal{C}}, \mathcal{Q}) \to \bar{BS}$$

$$query(c, q_{\{(qs,qp,qo)\}}) \mapsto \left\{ \begin{array}{r} \{(qs, s) | qs \in \hat{\mathcal{V}} \setminus \{v_*\}\} \cup \\ \{(qp, p) | qp \in \hat{\mathcal{V}} \setminus \{v_*\}\} \cup \\ \{(qo, o) | qo \in \hat{\mathcal{V}} \setminus \{v_*\}\} \\ |(s, p, o) \in mss(c, (qs, qp, qo)) \end{array} \right\}$$

Let us try to analyze what happens here. We have a container with statements which is the container from which we want to query data. Then we have a container with one statement which is the query which we need to solve. The first thing we do is applying the testStatement procedure to each statement from the queried container and the query statement. This procedure will only return true if the statement matches the query, i.e., all variables can be bound and all non-variables are equal. Then we need to find the bindingsets. A bindingset should be added for every possible set of bindings one can make between the variables and the values. We thus add a set which contains the subject, predicate and/or object and their value if the subject, predicate and/or object were variables in the query. The fact that the queryresult and bindingset are sets, removes duplicates. One more remark should be made: when the query does not contain any variables and an exact match is found, the bindingset will contain one binding namely the empty set. This detail is important for the Υ operator which was defined and discussed in section 3.2.1.

3.2.4 Selection of nested nodes

Next up, I will show how data can be addressed inside a container referenced from one of the statements of the container being queried. There are three possible ways

Algorithm 2 Procedure for calculation of bindingset from a query containing one statement.

procedure TESTSTATEMENT($(qs, qp, qo), (s, p, o)$)
  if $qs \notin \hat{\mathcal{V}}$ and $qs \neq s$ **then** ▷ Not a variable and also not equal.
    return false
  end if
  if $qp \notin \hat{\mathcal{V}}$ and $qp \neq p$ **then** ▷ Not a variable and also not equal.
    return false
  end if
  if $qp \in \hat{\mathcal{V}}$ **then**
    if $qp = qs \neq v_*$ and $p \neq s$ **then** ▷ v_* is the universal matching variable ▷ The variable occurs twice in the query, but the s and p of the statement are not equal.
      return false
    end if
  else if $qp = p$ **then** ▷ Nothing must be done in this case
  else ▷ Not a variable and also not equal.
    return false
  end if
  if $qo \in \hat{\mathcal{V}}$ **then**
    if $qo = qs \neq v_*$ and $o \neq s$ **then** ▷ The variable occurs twice in the query, but the s and o of the statement are not equal.
      return false
    else if $qo = qp \neq v_*$ and $o \neq p$ **then** ▷ The variable occurs twice in the query, but the p and o of the statement are not equal.
      return false
    end if
  else if $qo = o$ **then** ▷ Nothing must be done in this case
  else ▷ Not a variable and also not equal.
    return false
  end if
  return true
end procedure

for a statement to refer to a container; either the subject refers to a container node, the object refers to a container node or both. For that reason, the mathematical representation will be very similar for the three cases. First, let us look at the case where the subject (and only the subject) of the query refers to a container:

Given a query $q_{\{qs_1\}} \in \mathcal{Q}$ and a container $c_{\{s_1,s_2,\ldots,s_n\}} \in \hat{\mathcal{C}}$ where $qs_1 = st_{(qs,qp,qo)}$ and $qs \in \hat{\mathcal{C}}$ and $qo \notin \hat{\mathcal{C}}$. We first define a new function *subConQuery* which selects all statements from the container which match the query at least partially.

$$subConQuery(qs, qp, qo, newvar) = \left\{ \begin{array}{r} bs | bs \in query(c_{\{st_{(newvar,qp,qo)}\}}), \\ bs(newvar) \in \hat{\mathcal{C}} \end{array} \right\}$$

Using this definition, the result of the query is defined as follows:

$$\bigcup_{sq \in subConQuery(qs,qp,qo,newvar)} (\{sq \setminus (newvar, sq(newvar))\} \curlyvee \{query\,(qs, sq(newvar))\})$$

where *newvar* is any element of $\hat{\mathcal{V}}$ which is not reachable from qs_1.

Before giving more explanation about what this all means, I will define what happens when the object (but also possibly the subject) of the query is a container. The reason why the subject can be a container in this case, is because in order to define the result of the query, I will use queries where the object is not a container and which will then be handled by the previous case. Therefore, this part defines what happens in the two last possible cases of statements referring to containers. First we define a function *objConQuery* which is similar to the function *subConQuery* defined above.

$$objConQuery(qs, qp, qo, newvar) = \left\{ \begin{array}{r} bs | bs \in query(c_{\{st_{(qs,qp,newvar)}\}}), \\ bs(newvar) \in \hat{\mathcal{C}} \end{array} \right\}$$

Using this definition, the result of the query is defined as follows:

$$\bigcup_{sq \in objConQuery(qs,qp,qo,newvar)} (\{sq \setminus (newvar, sq(newvar))\} \curlyvee \{query\,(qo, sq(newvar))\})$$

where *newvar* is any element of $\hat{\mathcal{V}}$ which is not reachable from qs_1.

Let us have a look what this all means by looking at the first possibility. We have a query consisting of one statement called qs_1, of which the subject of the first

statement refers to a container, which we will call qs. That container, can itself contain an arbitrary query. The query is performed in a container which we will call $c = c_{\{s_1,s_2,\ldots,s_n\}}$. First the subConQuery function must be computed. The result of this function, is a queryresult containing a bindingset for each statement in c which has a container as its subject and a predicate and object matching the predicate and object of the query. That bindingset contains a binding of the container of that statement to the variable *newvar*, and bindings to the predicate and object of the query if they are variables.

The actual definition of the query tells us that, for each bindingset in the previously calculated queryresult, we should use the Υ operator on

- the bindingset from which we remove the binding to the *newvar* and
- the query result from querying for the query defined in qs in the container which is bound to *newvar*, i.e., the subject of the statements which got selected in the computation of subConQuery.

This way, we make sure that if a variable is used in the subject container of the query and in the predicate and/or object, the binding will be identical and only existent if all of the variable locations could be bound.

3.2.5 Construct for conjunction

In this section, I will give meaning to a query which consists of multiple statements. Given a query $q_{\{qs_1,qs_2,\ldots,qs_m\}} \in \mathcal{Q}$ and a container $c = c_{\{s_1,s_2,\ldots,s_n\}} \in \hat{\mathcal{C}}$, i.e., a query with m statements from a container with n statements. The semantic meaning given to the query is that the query result, will contain bindingsets for which when each variable in the query is replaced by the value given by the binding, each statement from the query can be found from the container. This is achieved by using the Υ operator as follows:

$$query(c, q_{\{qs_1,qs_2,\ldots,qs_m\}}) = query\left(c, q_{\{qs_2,qs_3,\ldots,qs_m\}}\right) \Upsilon query\left(c, q_{\{qs_1\}}\right)$$

As mentioned above, the Υ operator is both commutative and associative which shows that this operation is independent on the order of the statements in the container. What this does, is querying the container for the first statement of the query and combine that queryresult with the result of querying the container with the rest of the query statements.

3.2.6 Construct for optionality

The construct for optionality in *S-APL* reminds of the SPARQL OPTIONAL construct (see section 2.5.1). The point is that certain variables will only be bound if possible, otherwise they will not be bound but the query still has the bindingset where that part of variable bindings is missing. A part of the *S-APL* query can be indicated as being optional by making it the subject of a statement which has predicate "sapl:is" and object "sapl:Optional". Formally, let the query $q_{\{qs_1\}} \in \mathcal{Q}$ and the container in which we query be $c = c_{\{s_1,s_2,\ldots,s_n\}} \in \hat{\mathcal{C}}$ where $qs_1 = st_{(qs,qp,qo)}$ with $qs \in \hat{\mathcal{C}}$, qp the resource node for "sapl:is" and qo the resource node for "sapl:Optional". Then the query result is given by:

$$query(c, q_{\{st_{(qs,qp,qo)}\}}) = \begin{cases} \{\varnothing\} & \text{if } query(c, qs) = \varnothing \\ query(c, qs) & \text{otherwise} \end{cases}$$

Hence, when the query from the subject container gives a result, then it is used. Otherwise an empty result is added. Note that a result of $\{\}$ is not the same as a result of $\{\varnothing\}$ as discussed in 3.2.1. In the former, there are no results, while in the later there is one result which does not contain any bindings.

3.2.7 Creating new nodes from expressions

The *S-APL* query can contain expressions from which data which was not available in the original data set can be constructed. For instance, we can sum two numbers by defining a function on two literal nodes containing an integer, resulting in in another literal node, which can then be bound to a variable. We will give a mathematical definition of this construct which is slightly different from how this construct is defined in the *S-APL* version used in the UBIWARE platform. [2] Formally, let the query $q_{\{qs_1\}} \in \mathcal{Q}$ and the container in which we query be $c = c_{\{s_1,s_2,\ldots,s_n\}} \in \hat{\mathcal{C}}$ where $qs_1 = st_{(qs,qp,qo)}$ with $qs \in \hat{\mathcal{V}}$, qp the resource node for "sapl:expression" and qo a literal node representing an expression depending on the set of variables $v_1, v_2, \ldots, v_m \subset \hat{\mathcal{V}}$ and has XML schema datatype "sapl:ExpressionLiteral". The notation $expression(val_1, val_2, \ldots, val_m)$ is used to indicate the result of evaluation of the expression when var_i is substituted by val_i. We will now define the result of the query as being all possible bindings we can make, which fit into the expression.

[2] The UBIWARE platform requires all variables used in the expression to be bound to variables in statements which are used in conjunction before any expression is evaluated. The reason is that the way the UBIWARE *S-APL* engine is designed, does not allow for lazy evaluation.

$$query(c, st_{(qs,qp,qo)}) = \left\{ \begin{array}{l} \{(qs, qo\,(val_1, val_2, \ldots, val_m))\} \cup \\ \{(var_1, val_1)\,, (var_2, val_2)\,, \ldots (var_m, val_m)\} \mid \\ \forall i \in [1, m] : val_i \in \hat{\mathcal{N}} \text{ and} \\ qs \text{ is valid with each } var_i \text{ replaced by } val_i \end{array} \right\}$$

The size of the query result is potentially infinite and will in practical queries be limited by for instance filtering (see section 3.2.8) or using the expression statement in conjunction with other statements as described in section 3.2.5. In the UBIWARE platform version of *S-APL*, there is a big amount of expressions available for numerical calculations (see [58]) and the XML schema datatype of the expression object does not have to be "sapl:ExpressionLiteral", but can be any literal. One type of expressions does not fit into this definition. This are the expressions whose evaluation is not constant over time. Examples of this type of expressions are expressions generating random numbers and unique identifiers. They can, however, still be used when we define that the value of the query result can be different for different evaluations of the query function for the same document and query. This would, however, make the model much more complicated and is thus not supported. It is assumed that every *S-APL* engine at least supports these expressions which have a constant evaluation over time.

3.2.8 Filtering the results with filtering predicates

One more possible construct the *S-APL* query language is inspired by the SPARQL FILTER construct (see section 2.5.1). The goal of a filter, is the reduction of the number of bindingsets in the query result by stating conditions the variables have to fulfill. On can for instance require that the value of variable x must be equal to the value of variable y by using $st_{(x,sapl:eq,y)}$ in the query container. The version described here limits the one used in the UBIWARE platform by not evaluating possible expressions as the object of the test. This does, however, not limit expressiveness of the query function since it is possible to simulate the version from the UBIWARE platform with the version described here by using an extra expression statement (see 3.2.7) in conjunction as described in section 3.2.5. Another limitation, is that filters on statistics are not supported since statistics as such are not supported as described in section 3.3.1.

Despite the fact that expressions and filters are essentially different things, they

can be defined in a very similar way. Given a query $q_{\{qs_1\}} \in \mathcal{Q}$ and the container in which we query $c = c_{\{s_1,s_2,\ldots,s_n\}} \in \hat{\mathcal{C}}$ where $qs_1 = st_{(qs,qp,qo)}$ with qp a resource node with label

$$l \in \{sapl:gt, sapl:lt, sapl:gte, sapl:lte, sapl:neq, sapl:eq, sapl:regex\}.$$

Then, the query result is the set of all bindingsets which fulfill the filter expressed by the statement qs_1. We split the definition in three cases; if $qs, qo \in \hat{\mathcal{V}}$ then

$$query(c, q_{\{st_{(qs,qp,qo)}\}}) = \{(qs, qsv), (qo, qov) | qsv, qov \in \hat{\mathcal{N}} \text{ and 'qsv qp qov' is true } \}$$

else, if $qs \in \hat{\mathcal{V}}$ but $qo \notin \hat{\mathcal{V}}$ then

$$query(c, q_{\{st_{(qs,qp,qo)}\}}) = \{(qs, qsv) | qsv \in \hat{\mathcal{N}} \text{ and 'qsv qp qo' is true } \}$$

else, if $qs \notin \hat{\mathcal{V}}$ but $qo \in \hat{\mathcal{V}}$ then

$$query(c, q_{\{st_{(qs,qp,qo)}\}}) = \{(qo, qov) | qov \in \hat{\mathcal{N}} \text{ and 'qs qp qov' is true } \}$$

The queryresult will thus contain all possible minimal bindingsets which can pass the filter.[3] As a result, the result set will be infinitely big (except for $sapl : eq$ where the result can also contain only 1 bindingset). Analog to expressions, the amount of bindingsets in the queryresult should in practical queries be limited by using this type of query in conjunction with other queries as described in section 3.2.5.

3.2.9 Filtering the results with negation

Next to the use of filtering predicates it is possible to filter from the query result bindingsets using negation. A bindingset will be removed when a specified container, filled with the bindings of the bindingset forms a query which yields a result. Formally, the removal is defined as follows. Given a query $q = q_{\{qs_1,qs_2,\ldots,qs_m\}} \in \mathcal{Q}$ and a container $c = c_{\{s_1,s_2,\ldots,s_n\}} \in \hat{\mathcal{C}}$, i.e., a query with m statements and from a container with n statements where $qs_1 = st_{(qs,qp,qo)}$ and qs the resource node for "sapl:I" and qp the resource node for "sapl:doNotBelieve" and $qo \in \hat{\mathcal{C}}$. Then, we define the

[3] The *S-APL* language supported by UBIWARE does not support queries which contains filters in the same way. One can only use filters in conjunction with other statements which make the query result a finite set.

query as

$$query(c, q_{\{st_{(qs,qp,qo)}, qs_2, \ldots, qs_m\}}) = \left\{ \begin{array}{r} bindingset | bindingset \in query(q_{\{qs_2,\ldots,qs_m\}}, c), \\ query(fill(qo, bindingset), c) = \varnothing \end{array} \right\}$$

where the fill function was defined in section 3.2.2.

Informally, we first perform the query of all but the negation statement on the container. Then we see whether the query of the negation statement, filled with that each bindingset of the query result, yields the empty set. If it is not the case the bindingset is removed from the result.

3.2.10 Filter on whether something is a container

One more possible filter makes sure that a certain variable is a container. Let the query be $q_{\{qs_{(s,p,o)}\}} \in \mathcal{Q}$ where $s \in \hat{\mathcal{V}}$ and, p and o the resource nodes for "rdf:type" and "sapl:Container" respectively. Then the query function maps the query as follows:

$$query(c, q_{\{qs_{(s,p,o)}\}}) = \{(s, container) \,|container \in \hat{\mathcal{C}}\}$$

The query result is thus the set of all tuples with as first component the variable and as second component any container. When this statement is used in conjunction with other statements, only when the variable in the bindingset for these statements is mapping to a container, the bindingset will remain in the query result.

3.2.11 Construct for UNION

S-APL has support for something which is similar to the UNION from the SPARQL language. Concrete, it is possible to create a query which contains 2 sub-queries, where a bindingset in the query result of one of them is enough to be part of the query result of the whole query. The two queries are embedded in the bigger query by putting the queries as subject and object of a statement which has the resource node for "sapl:or" as a predicate. Formally, Let the query be $q_{\{qs_{(s,p,o)}\}} \in \mathcal{Q}$ where $s, o \in \hat{\mathcal{Q}}$ and, p the resource node for "sapl:or". Then the query function maps the query as follows:

$$query(c, q_{\{qs_{(s,p,o)}\}}) = query(c, s) \cup query(c, o)$$

3.2.12 The empty query

One possible query is the empty query, i.e., the query which consists of a container without any statements. The result of this query, with c an aribitrary element of $\hat{\mathcal{C}}$, is defined as follows:

$$query(c, q_{\{\}}) = \{\varnothing\}$$

Notice that the result is a set which contains the empty set. This is stricly different from a result which is just the empty set as described in section 3.2.1. The effect of this definition is that the empty query always has a result which is the empty bindingset.

3.3 Limitations and syntactic sugar for queries

In this section, I will give an overview of more query features which are available in the UBIWARE *S-APL* language. First, the feature is described and then explained how this can be emulated in the *S-APL* definition used in this thesis or the reason stated why this feature is not supported.

3.3.1 Statistics and filters on statistics

Up to now, we have not been talking about one possibility which the queries in UBIWARE *S-APL* have, namely the possibility to calculate statistics on the result set and bind the numbers to variables. The reason for not including statistics is that statistics are depending on the other statements in the query. It is thus not possible to determine the value of the statistics by solely looking at the statistics statement and the container which is being queried. Statistics can, however, be emulated by calculating them using expressions and rule features of the *S-APL* language which are defined in chapter 3.4. This is best explained with an example, the following query bind to ?count the number of unique matches for the variable ?x in the bindingset.

```
{
?x ex:a ex:b .
?count sapl:count ?x
} sapl:All ?x
```

This can be written without the use of the statistics by first adding the following rule which adds all statements to one container.

```
{
```

```
  ?x ex:a ex:b .
}
->
{
    ex:countContainer ex:temp {?x ex:a ex:b} .
}
```

And then performing the following query which will work inside the container.

```
{
  countContainer ex:temp ?container
  ?count sapl:expression "numberOfMembers(?container)"
}
```

Filters on statistics cannot be supported if statistics are not supported. However, one can use the normal filters in combination with the technique used in the previous example.

3.3.2 First match, sapl:All and sapl:Some

In the *S-APL* language used in the UBIWARE platform, a query will by default only result in one of the bindingsets of the query result as defined in this thesis. This result is the first bindingset which matches the query found by the engine. In order to get more bindingsets, the query must be what is called "wrapped in sapl:All". Concrete, this means that the query must be the subject of a statement in an encapsulating query of the form $(query, sapl : All, var)$, where *query* is the query, $sapl : All$ is the resource node for sapl:All and $var \in \hat{\mathcal{V}}$. When this is done, for each different mapping from *var* to a value which can be found from the bindingsets, one bindingset with that mapping is chosen. Another option is to use "wrap in sapl:Some" which will force the engine to find all solutions and then pseudo-randomly select the resulting bindingset(s).

In this thesis, the *S-APL* query is considered deterministic and results in all possible bindingsets of the queried document. Put another way, it works like a UBIWARE *S-APL* query where for each variable, the query is wrapped in sapl:All for that variable.

3.4 Rules and dynamics of S-APL

Up till now, we have seen what the structure of a *S-APL* document is and how a *S-APL* document can be queried. However, as mentioned above, a *S-APL* document is a self modifying graph. In this chapter, I will show and define what kind of dynamics the *S-APL* document has. First, we will look at the implies now rule which is the basic rule used for the self modification. Then I will show how these rules are applied to a *S-APL* document. Lastly I will show how other rule types available in UBIWARE *S-APL* can be emulated by a combination of basic rules.

3.4.1 Implies now rules

UBIWARE *S-APL* supports many types of rules. In this thesis I will, however, only define one type of rule, which I will call the implies now rule, which is the equivalent of the 'conditional action' as described in "Semantic Agent Programming Language (*S-APL*) Developer's Guide"[58]. Below, I will explain how combinations of one or multiple instances of this one type of rule can be used to imitate repeated implies rules, conditional commitments, behavioral rules, inference rules, meta-rules, and exists while conditions (see section 3.4.5).

This rule has much similarity with the CONSTRUCT query in SPARQL. Having the construct type of query available, one can imagine that the SELECT and ASK types are implicitly available as well. The SELECT type is available when the user binds the values to known variables in the user specified graph template. The ASK type is available because the user can see whether a graph is build which is equivalent to the question whether a match is possible.

Let us now look at how the implies now rule is defined. An implies now rule consists of:

- A query container which is executed against a container.
- A container which contains a pattern into which variables will be filled for each bindingset in the query result, the result of the rule is the set of all these filled containers.
- A container which will be the only member in the result set of the rule if the query result is the empty set, i.e., no results were found.

In a *S-APL* document, the rule is written as two statements $st1_{(q,impliesNow,suc)}$ and $st2_{(q,impliesNow,fail)}$ where $q \in \hat{\mathcal{C}}$ is the query which gets performed, $impliesNow$

is the resource node for "sapl:impliesNow", $suc \in \hat{\mathcal{C}}$ is the container which will be filled for each bindingset to form a result, and $fail \in \hat{\mathcal{C}}$ is the container which will serve as the result in case the query result is empty. Let c be the container against which the rule is executed. I will define the result of the implies now rule formally as the result of the implies function which takes the four containers as parameters.

$$implies\colon (\hat{\mathcal{C}},\hat{\mathcal{C}},\hat{\mathcal{C}},\hat{\mathcal{C}}) \to \{\hat{\mathcal{C}}\}$$

$$implies(q,c,suc,fail) \mapsto \begin{cases} \{fill(suc,b) | b \in res\} & \text{if } res = query(c,q) \neq \varnothing \\ \{fail\} & otherwise \end{cases}$$

As can be seen from this definition, there is a separation between the execution of the rule and the addition of its result to the main container of a *S-APL* document as is always the case on the UBIWARE platform. Below, in the sections about dynamics in *S-APL* documents, I will define how the result of rules gets added to *S-APL* documents.

3.4.2 Removal of beliefs

It is possible that certain statements do become invalid. In order to reflect this change, the statement should be removed from the document. Removal of statements can be done by writing a "remove statement" in the $\mathcal{G}$ container of the *S-APL* document. The removal happens at the moment specified by the δ operator as defined in the next section. The actual statement is of the form $st_{(s,p,o)}$ where $s \in \hat{\mathcal{C}}$ is an empty container [4] and p is the resource nodes for "sapl:remove". The object $o \in \hat{\mathcal{C}}$ is the pattern which should be removed. Formally, the statements which will be removed are these statements which add to the bindingset of the part of the *query* function defined in section 3.2.3 and whose bindings do not get removed in the recursive definition of the *query* function. After all removal statements have been handled, the remove statements are themselves removed from the document (if they are still there).

3.4.3 Dynamics – definition of the delta operator

In this section, I will describe in which sense the *S-APL* document is dynamic. The dynamism is caused by two types of constructs, namely the appearance of implies

[4] This is a big difference from the UBIWARE *S-APL* language where the subject has to be the resource node for "sapl:I" and statements with empty containers are removed by the run time. See also section 3.5.7.

now rules, as described section 3.4.1, and removal of statements, as described in section 3.4.2, in the general context ($\mathcal{G}$) of a *S-APL* document. Dynamism is formalized as a function δ which maps the document on the document after making certain changes. The operator is repetitively applied on the document; we denote k repetitive applications of the operator as δ^k.

Now follows the definition of the δ operator:

$$\begin{aligned} \delta\colon \mathcal{SAPL} &\to \mathcal{SAPL} \\ \delta\colon (\mathcal{C},\mathcal{S},\mathcal{R},\mathcal{L},\mathcal{V},\mathcal{CS},\mathcal{SS},\mathcal{SP},\mathcal{SO},\mathcal{G}) &\mapsto (\tilde{\mathcal{C}},\tilde{\mathcal{S}},\tilde{\mathcal{R}},\tilde{\mathcal{L}},\tilde{\mathcal{V}},\tilde{\mathcal{CS}},\tilde{\mathcal{SS}},\tilde{\mathcal{SP}},\tilde{\mathcal{SO}},\tilde{\mathcal{G}}) \end{aligned}$$

where

- $\tilde{\mathcal{G}} = remove \circ applyRules(\mathcal{G})$,
- $\tilde{\mathcal{C}},\tilde{\mathcal{S}},\tilde{\mathcal{R}},\tilde{\mathcal{L}},\tilde{\mathcal{V}},\tilde{\mathcal{CS}},\tilde{\mathcal{SS}},\tilde{\mathcal{SP}}$ and $\tilde{\mathcal{SO}}$ are appropriate to make the result a valid *S-APL* document,
- remove is as specified in section 3.4.2,
- The function applyRules is defined as follows:

  $$\begin{aligned} applyRules\colon \hat{\mathcal{C}} &\to \hat{\mathcal{C}} \\ applyRules(\mathcal{G}) &\mapsto \mathcal{G} \Cup mergeContainers(\{implies(q,c,suc,fail)|rule\}) \end{aligned}$$

  where *rule* is any "applies now rule" in $\mathcal{G}$ with q, suc and $fail$ as described in section 3.4.1.

3.4.4 S-APL document classes

After having defined in which way a *S-APL* document is dynamic, it is possible to classify documents according to how the dynamism, i.e., the δ operator changes the document. A *S-APL* document D is said to be

stable if $\delta : D \mapsto D$, i.e., the document does not change when the operator is applied.

unstable if the document is not stable, i.e., $\delta : D \mapsto E$ and $D \neq E$.

converging if $\exists k, \forall n > k : \delta^n : D \mapsto E$, i.e., after applying the δ operator a sufficient amount of times, the document becomes stable.

diverging if the document is not converging, i.e., $\nexists k, \forall n > k : \delta^n : D \mapsto E$ where $D \neq E$.

revolving if $\exists k, \delta^k : D \mapsto D$, i.e., after applying the operator a fixed number of times, the document comes back to the same state.

3.4.5 Emulating other rules

In this chapter, I will give a description of how the "implies now rule" from section 3.4.1 and the dynamics of the document from section 3.4.3 are strong enough to emulate other rule types available in UBIWARE *S-APL*. The descriptions here are not formal proofs, but could be a starting point for those.

Repeated implies now rule This rule which is closely related to the implies now rule. In UBIWARE *S-APL*, this is an implies now rule which is member of the subject container of the statement {} sapl:is sapl:Rule in $\mathcal{G}$. The way the rule works is that is is executed and does not get removed as long as its query remains true. We thus need to find a way to prevent the *S-APL* document from removing the rule as long as the query has had a result in the previous iteration of the δ operator. The way to solve this is rather straightforward; the recursive inclusion of the rule statement itself in the *suc* container of the implies now rule gives the wanted behavior. [5]

Conditional commitment A conditional commitment rule is a rule which stays as long as its query is not matched. Once the query is matched it gets removed. This type of rules look similar to implies now rules, except that they have no *fail* container and a different predicate ("sapl:implies" or as shorthand '=>'). To emulate the behavior of this type of rule, it is sufficient to use an implies now rule with the same query q and *suc* container, and the addition of a *fail* container which contains as its only statement a recursive inclusion of the whole rule.

Behavioral rule Behavior rules are like a conditional commitment, but defined in the same context as repeated implies now rules. The difference with conditional commitments is that they do not get removed after a successfully match. The way they can be emulated is a combination of the two previous ways. This

[5] Recursive inclusion of statements or containers is very poorly supported in the UBIWARE platform and might bring the agent into an infinite loop. For more information, see section 3.5.5.

time the recursive inclusion of the whole rule is needed in both the *suc* and the *fail* container.

Inference rule A way for emulating inference rules, i.e., rules using the arrow '==>' or "sapl:infers" as a predicate, is already described in the "Semantic Agent Programming Language (*S-APL*) Developer's Guide"[58]. The basic idea is to make sure that all bindingsets are used by wrapping in "sapl:All" (see section 3.3.2) for all variables used, which is done in any case in the *S-APL* version described here and adding a negation of the *suc* container to the query by means of sapl:doNotBelieve as described in section 3.2.9.

Meta rule Meta rules are just like normal rules, except that they are executed before and after the execution of other rules. This abstraction can also be made by temporarily removing a certain set of rules and enabling another set. It is even possible to introduce a hierarchy of meta rules, i.e., meta rules which manage meta-rules.

Exists while condition "Exists while conditions" are not actual rules, rather they assert that certain set of statements will be removed if a given query does not have any results any more. It is easy to see that they can be imitated with implies now rules, which will have the statements to be removed in a remove statement in the *fail* container and a recursive inclusion of the rule in the subject container.

3.5 Use of S-APL in agents.

The *S-APL* language as initially proposed was intended for use in software agents. This chapter will shortly look at how the model used up to now needs to be extended to be extended to be useful for agent programing.

3.5.1 Software agents

In general, the word agent has many meanings. It should be clear that I am talking here about an agent in the sense of something which has the power to act. More specifically, I want to discuss about software agents, i.e., software which is able to act itself. The concept of software agents has been defined in several ways. The Foundation for Intelligent Physical Agents (FIPA) gave a definition in "FIPA

Agent Management Specification" [59]. The most relevant parts of the definition for this discussion are that "An agent is a computational process that implements the autonomous, communicating functionality of an application." and that "an agent must support at least one notion of identity. " The FIPA definition does not make any statements about the agent having any specific knowledge, nor that the agent is able to act on and upon its environment.

Russel shows a different view on agents by saying that "An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators." [60] This definition is clearly much broader as the definition given by FIPA, since not only the communicative aspect is taken into account. It is not clear from Russels definition whether each agent needs its own notion of identity, since it could even be said that a group of agents can be seen as one.

The *S-APL* language was designed for agents which are more in the sense which Russel is defining. For the scope of this thesis, I will limit the concept of agent or software agent to a software entity which has a description about its own internal state and part of its environment. It is less relevant whether the agent is able to sense and react on its environment.

3.5.2 The roots of S-APL

The *S-APL* language was originally developed in the Smartresource [61] and the UBIWARE project [62]. One of the aims of the UBIWARE project was to make a semantic agent platform, i.e., an agent platform where the agents use semantic data. This platform uses ideas from the Smartresource project by make agents representatives of resources external to the platform. *S-APL* is the language used to give agents an internal state, make them communicate, start actions external to the agent, etc...

As was shown in chapter 2, there have been many attempts to define languages for use in the Semantic Web. The problem with all of these language is, however, that their scope is very limited and not usable for programming. Furthermore, a new language needed to be elaborated because the existing languages did not allow for explicit removal of existing information.

The aim of the language evolved to become a one-fits-all language for the Semantic Web. The concrete language developed was described in section 3.1.1, but is basically a combination of many of the concepts found in other languages used in the Semantic Web with the addition of removal of information.

3.5.3 External actions

One definition of agent quoted above in section 3.5.1 about software agents is that "An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators." [60]. In the definition of *S-APL* in this thesis, there is no interaction mechanism with the document whatsoever. The document is a graph which is dynamic, but the results of this dynamism can in no way be observed nor influenced. Therefore, we need to enable an agent which is using *S-APL* to represent its internal state to

- perceive its environment trough sensors and
- act upon that environment trough actuators.

The way this is solved in UBIWARE *S-APL* on the UBIWARE platform is described in "Semantic Agent Programming Language (*S-APL*): A Middleware Platform for the Semantic Web" [63]. In this document, there is a description of the Reusable atomic behavior (RAB) as an atomic function which is implemented in a programming language (in this case Java), which serves as actuators and sensors of the environment. Further, it is argued that these RABs should be parametrized. The way the external code is started has changed over time and more advanced construct have been added for controlling what happens when the behavior starts, stops, succeeds, fails and even if the behavior cannot be run due to policies of the platform. [64] Essentially, all of the methods which have emerged over time make the agent check in the *S-APL* document whether statements of a specific form are present and starts external code with parameters specified in the statements. In order to sense the environment, the code is able to add data back to the model with a procedure similar to the on described in section 3.1.7 about the merging of containers.

3.5.4 Agent time and embedded beliefs

The agents on the UBIWARE platform have certain statements which are always in the $\mathcal{G}$ container and which are kept up to date by the platform. These statements include information about time and the agent's own name. This way, it is possible to define queries which only give a result when the time has advanced beyond a certain time point. The reason why this is not included in the theoretical model, it that this would lead to a different evaluation of a query at different times. Furthermore, it would also render the definitions given in section 3.4.4 about document

types useless. Another problem with time is that a *S-APL* document will be time dependant making it impossible for any implementation to make hard guarantees about how the document will be run on the platform

3.5.5 Inability of implementations to support infinite loops

The UBIWARE implementation cannot support the version of *S-APL* which is described in this thesis because it is unable to handle infinite loops. If a document is revolving but not stable then the UBIWARE agent will get in an infinite loop changing the document between the states which are part of the revolution. Furthermore, as already described in sections 3.2.8 and 3.2.7, the UBIWARE platform is unable to support queries which yield an infinite set of results nor is it able to determine that a resultset will be limited in size. The only way this type of query is possible on the platform is if the variables used in filters and expressions are explicitly bound in statements of the query which are in conjunction with them.

3.5.6 Protection of removal of beliefs in an agent context

Next to the embedded beliefs, there are also certain statements or so called beliefs which are protected form removal. These include statements about certain types of rules, goals and currently active external actions. As a result, even if one tries to remove these beliefs, nothing will happen.

3.5.7 Exceptions for merging and empty containers

Some exceptions are made to the merging of containers when rules are executed and statements are added. I could not find any good argumentation on why these exceptions were made except that statements are used in such a way that merging them would lead to unexpected results. For instance, if the remove statements as described in section 3.4.2 would as in the UBIWARE platform have the resource node for "sapl:I" as a subject, then the queries of all remove statements would join into one big query which would have a different semantic meaning as the separate queries.

One more curiosity is the removal of any statement which has an empty container. This also applies recursively inside containers. The fact that this operation is performed has been the cause of many difficult to solve problems in program-

ming agents in *S-APL* for the UBIWARE platform which the author of this thesis has encountered.

3.5.8 Adding and Erasing of beliefs

UBIWARE *S-APL* also supports "sapl:erase" and "sapl:add" next to "sapl:remove". The reason for this is mainly historical and is mainly used for syntactic problems with the syntax. Further, these constructs show in my opinion too much of the internal implementation of the platform. The idea is that "sapl:erase" removes the actual container which it gets as an argument (the object of the statement) instead of using it as a query, while "sapl:add" just adds its argument to the $\mathcal{G}$ container when it appears there. Another reason for "sapl:add" is the addition of statements on the *suc* or $fail$ side of a rule as mentioned in the section 3.5.10.

3.5.9 Syntactic sugar for rules available in UBIWARE

As mentioned in the section on sapl:All (see section 3.3.2), the UBIWARE platform *S-APL* has a different way of performing queries with regard to how many binding sets are in the query result. The syntax use for denoting the fact that multiple solutions are needed can be as follows:

```
{{A B ?x } sapl:All ?x }
->
{use of ?x }
```

but also like this:

```
{A B ?x}
->
{{use of ?x} sapl:All ?x}
```

The later variant is against the concept that the query is completely stated in the container which is the subject of the rule statement, which is used in this thesis.

3.5.10 Referring to containers and statements in UBIWARE S-APL

The UBIWARE *S-APL* query language allows the programmer in concrete syntax to refer to a certain container or statements using IDs. This ID is local to the document while being loaded and will be removed when the engine loads the document.

Another thing which is made possible is to explicitly query for specific statements. The problem is, however, that once a statement node is bound to a variable,

it cannot be used in the fill operator discussed above. To overcome this problem, UBIWARE *S-APL* introduces yet another construct which will be replaced by the referred statement. I decided to leave these possibility out of the specification in this thesis, since it is possible to use other syntax constructs to emulate the same behavior and I did not want to introduce more irregularities in the definitions.

3.6 The problem of variables in higher order constructs

One problem of the *S-APL* language when used, is that variables are not unique enough. The problem might arise that two programmers use the same variable name for several things and therefore errors occur. One option to solve this problem is to use for the replacement of all variable names UUIDs before merging two *S-APL* documents. The problem is, however, that there is not possibility to determine upfront how variables will be used. The cause of this problem is that it is possible to write code which uses higher order variables, i.e., variables whose value is another variable. The following example shows the use of higher order variables, where the first document contains the data and the second one a rule. This is the first document, containing the data:

```
ex:a ex:hasLeftHandSide {?var ex:p ex:o}
ex:value ex:p ex:o
```

And this document containing the rule using an higher order variable.

```
//Rule
{
  ?example ex:hasLeftHandSide ?lhs
} =>
{
  {
    ?lhs sapl:is sapl:true
  } => {
    ?var ex:p2 ex:o2
  }
}
```

When merged, the rule will be matched and the right hand side added. The document will then look like this:

```
ex:a ex:hasLeftHandSide {?var ex:p ex:o}
ex:value ex:p ex:o
```

```
{
  ?var ex:p ex:o
} => {
  ?var ex:p2 ex:o2
}
```

Now, the variable ?var appears on both the left and right hand side of the rule and the rule can do some useful work which results in:

```
ex:a ex:hasLeftHandSide {?var ex:p ex:o}
ex:value ex:p ex:o
ex:value ex:p2 ex:o2
```

Notice that the second document counts on the fact that a certain variable in the first document has a certain name. We can thus not in general assume that we can replace the variable names in documents by unique names before merging them.

One way to overcome this problem is to treat variables in some sense as 'document global', i.e., a variables in a document must have only one meaning and do not have any meaning outside the document. This can, however, not be enforced by the *S-APL* engine and must thus be done by the programmer. Once this requirement is put, it is possible to use the aforementioned methods to make sure variables do not collide.

4 Use of theoretical model defined for S-APL

This chapter discusses tries to provide an answer to the first research question of this thesis, i.e, "Why is there a need for formalization?". The answer to this question is given by several use-cases for the formalization described in the sections of this chapter.

4.1 Data representation

S-APL can just like RDF be used to describe any possible data which is representable on computer systems. The proof that any data can be represented is rather simple. The trick is to realise that a literal can for instance contain data encoded in what is called Base64 encoding. Multiple possible encodings exist, one is described in [65]. The working of the encoding consists of a fixed mapping of binary data to a string in which 64 different characters can be used and also a reverse mapping is available. This character string can then be used as a literal in a *S-APL* or RDF document. A document could for instance look like this :

```
@prefix ex: <http://www.example.org/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

ex:a ex:a "Tm90aGluZyBzcGVjaWFsIGhlcmUh"^^xsd:base64Binary
```

Of course, it would not be of much benefit to use *S-APL* to represent data in such a way, since it loses the possibilities and semantics of the language.

4.2 Query language

As can be seen immediately from chapter 3.2, the language can be used as a query language for semantic data. This feature is already available on the UBIWARE platform, but requires the data to be inside the beliefs' structure of an agent. The formalization does, however, not require any agent as an intermediate and it is thus possible to use *S-APL* as a general purpose Semantic Web query language like SPARQL. Furthermore, thanks to the formalization it is possible to make statements about

what the exact results of a query will be.

4.3 Schemas

As discussed in section 2.4.1 and 2.4.2, RDFS and OWL can be used to make schemas for data in RDF. These two specifications make a description of the shape of the data by putting constraints on it. The way this is done is declarative, meaning that there is a static set of conditions to which the data has to comply. In *S-APL*one could, however, imagine a different way of defining schemas. Data could be for instance be according to a schema if the *S-APL* schema document merged with the data becomes a specific type of document or equal to a given document.

Let me illustrate the idea with an example. Imagine a set of data about students and supervisors with the condition that each supervisor must have more publications as the supervised student. For the sake of the example, I am assuming that all data about publications is available and that there is no other data in the document, which is a violation of the open world assumption. For the example I will use a *S-APL* notation similar to the one used in UBIWARE *S-APL*, but the semantics and dynamics should be interpreted as the *S-APL* described in this thesis.

Assume that the revolving document which must be reached is a document which has the statement (ex:schema ex:state ex:ok). The schema could then look like this:

```
ex:schema ex:state ex:ok .
{
  {
    ?A ex:supervises ?B .
    ?A ex:hasPublication ?APub
    ?B ex:hasPublication ?BPub
  }
  ->
  {
    ?A ex:hasPublicationContainer { ?APub rdf:type ex:Publication} .
    ?B ex:hasPublicationContainer { ?BPub rdf:type ex:Publication} .
    {
      ?A ex:hasPublicationContainer ?ACont .
      ?ACount sapl:expression
                "numberOfMembers(?ACont)"^^sapl:ExpressionLiteral .
      ?B ex:hasPublicationContainer ?BCont .
      ?BCount sapl:expression
```

```
                "numberOfMembers(?BCont)"^^sapl:ExpressionLiteral .
      ?ACount sapl:gt ?BCount
    }
    ->
    {  }
    ; sapl:else {
      ex:schema ex:state ex:violated .
      {} sapl:remove {ex:schema ex:state ex:ok}.
    }
  }
  ; sapl:else {
    __S1 sapl:is sapl:true
  }
} sapl:ID __S1 .
__S1 sapl:is sapl:true
```

Where the "sapl:ID" and "sapl:is sapl:true" are the way UBIWARE *S-APL* indicates that the same statement. This is thus a recursively defined rule which keeps on checking that everything is alright, but which will be revolving if there is, or no data or valid data.

4.4 Proof of correctness of implementation

When a formal model of a system is defined, it becomes possible to formally proof that a certain implementation is correct according to that definition. The benefit is that when a piece of code is run on a system which is proven to be correct, it cannot fail in the sense of working unexpectedly. If on top of that, the code is also proven to have certain properties, the properties will also be valid for the working system on which the code is deployed.

Concrete, if an implementation of a *S-APL* engine can be proven to be compliant with the version of *S-APL* described in this document, then any code run on this platform can be proven to have certain properties. For instance, if a piece of code, which is proven to solve a certain mathematical equation, is run on an implementation, which is proven to be compliant, then one can ensure that the code will give the answer to the equation.

4.5 Limit for space and time optimizations

When a formalization of a language is defined, it is possible to make statements about the complexity of evaluation of fragments of code. This way, it is possible to state maximum limits for space (in terms of memory) and time needed for evaluation of code.

4.6 Plans

A plan is a chain of atomic actions which can be performed in order to reach a certain goal. Analog to concepts described in the previous paragraphs, it is possible to prove that certain plans will work, even during run time. The system can, using knowledge about the input and output of the atomic actions, and knowledge about the formal model by which they are interacting, decide whether the chain of actions will lead to the desired goal.

5 Conclusion

This thesis tried to give an answer to the following research questions

1. Why is there a need for formalization?
2. How can one make a formalization of the *S-APL* language?

The first research question was answered in chapter 4. This chapter contains several use-cases for the *S-APL* language which are not all agent centric. Most of these use cases are centered around the fact that a formalization gives the possibility to make formal statements about the document.

The second research question was answered by giving a formalization in chapter 3. The formalization consists of a function δ which defines how a *S-APL* document, as defined formally in this thesis in section 3.1, changes its structure dynamically. The δ operator makes use of rules and remove statements which themselves depend on the query function. All of these where defined and elaborated in this thesis in sections 3.4, 3.4.2 and 3.2 respectively. The formalization was also compared to the version of *S-APL* which is used in the UBIWARE platform in section 3.5.

The content of this thesis can be used in further work on any of the topics described in the need for formalization chapter. For one, it is possible to formalize how a particular implementation of schemas based on *S-APL* will work and proof its correctness. Furthermore, it is possible to proof correctness of implementation and programs written in *S-APL*. Last but not least, it would be possible to extend this formalization to a formalization in which agents using this version of *S-APL* are interacting with each other in a multi-agent platform. Further research could also point out whether this formalization can be simplified without losing expressiveness.

6 References

[1] I. Herman, "W3c semantic web frequently asked questions," nov 2009. [Online; accessed 25-February-2012].

[2] J. J. Carroll and G. Klyne, "Resource description framework (RDF): Concepts and abstract syntax," W3C recommendation, W3C, Feb. 2004. http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/.

[3] Princeton University, "Wordnet - formalization." website, 2006. http://wordnetweb.princeton.edu/perl/webwn?s=formalization, last retrieved on 20 Feb 2012.

[4] J. Stewart, *Calculus*. Stewart's Calculus Series, Thomson Brooks/Cole, 2003.

[5] Wikipedia, "Digraph (mathematics) — wikipedia, the free encyclopedia," 2008. http://en.wikipedia.org/w/index.php?title=Digraph_(mathematics)&oldid=245066361 [Online; accessed 25-February-2012].

[6] Wikipedia, "Glossary of graph theory — wikipedia, the free encyclopedia," 2012. http://en.wikipedia.org/w/index.php?title=Glossary_of_graph_theory&oldid=478435016 [Online; accessed 25-February-2012].

[7] J. Postel, "DoD standard Internet Protocol," RFC 0760, Internet Engineering Task Force, Jan. 1980.

[8] P. Mockapetris, "Domain names: Concepts and facilities," RFC 0882, Internet Engineering Task Force, Nov. 1983.

[9] P. Mockapetris, "Domain names: Implementation specification," RFC 0883, Internet Engineering Task Force, Nov. 1983.

[10] T. Berners-Lee, L. Masinter, and M. McCahill, "Uniform Resource Locators (URL)," RFC 1738, Internet Engineering Task Force, Dec. 1994.

[11] T. Berners-Lee, “Universal Resource Identifiers in WWW: A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as used in the World-Wide Web,” RFC 1630, Internet Engineering Task Force, June 1994.

[12] K. Sollins and L. Masinter, “Functional Requirements for Uniform Resource Names,” RFC 1737, Internet Engineering Task Force, Dec. 1994.

[13] R. Moats, “URN Syntax,” RFC 2141, Internet Engineering Task Force, May 1997.

[14] M. Duerst and M. Suignard, “Internationalized Resource Identifiers (IRIs),” RFC 3987, Internet Engineering Task Force, Jan. 2005.

[15] “Information technology, "Universal Coded Character Set (UCS)",” 2000.

[16] J. Hakala and H. Walravens, “Using International Standard Book Numbers as Uniform Resource Names,” RFC 3187, Internet Engineering Task Force, Oct. 2001.

[17] P. Hoffman, L. Masinter, and J. Zawinski, “The mailto URL scheme,” RFC 2368, Internet Engineering Task Force, July 1998.

[18] D. M. e. a. Drummond Reed, “Extensible resource identifier (xri) syntax v2.0,” OASIS committee specification, OASIS, Nov. 2005. http://docs.oasis-open.org/xri/xri-syntax/2.0/specs/cs01/xri-syntax-V2.0-cs.html.

[19] M. McRae, “Failed oasis standard ballot of xri syntax v2.0.” Mailing list on http://lists.oasis-open.org/archives/xri/200806/msg00001.html, jun 2008. Last retrieved 29 january 2012.

[20] “Information Technology, "Procedures for the operation of OSI Registration Authorities: Generation and registration of Universally Unique Identifiers (UUIDs) and their use as ASN.1 Object Identifier components",” 2004.

[21] P. Leach, M. Mealling, and R. Salz, “A Universally Unique IDentifier (UUID) URN Namespace,” RFC 4122, Internet Engineering Task Force, July 2005.

[22] O. Lassila, “Resource description framework (rdf) model and syntax specification,” W3C recommendation, W3C, Feb. 1999. http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/.

[23] E. Miller and F. Manola, "RDF primer," W3C recommendation, W3C, Feb. 2004. http://www.w3.org/TR/2004/REC-rdf-primer-20040210/.

[24] D. Beckett, "RDF/xml syntax specification (revised)," W3C recommendation, W3C, Feb. 2004. http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/.

[25] R. V. Guha and D. Brickley, "RDF vocabulary description language 1.0: RDF schema," W3C recommendation, W3C, Feb. 2004. http://www.w3.org/TR/2004/REC-rdf-schema-20040210/.

[26] P. Hayes, "RDF semantics," W3C recommendation, W3C, Feb. 2004. http://www.w3.org/TR/2004/REC-rdf-mt-20040210/.

[27] D. Beckett and J. Grant, "RDF test cases," W3C recommendation, W3C, Feb. 2004. http://www.w3.org/TR/2004/REC-rdf-testcases-20040210/.

[28] P. V. Biron and A. Malhotra, "XML schema part 2: Datatypes," first edition of a recommendation, W3C, May 2001. http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/.

[29] A. Malhotra and P. V. Biron, "XML schema part 2: Datatypes second edition," W3C recommendation, W3C, Oct. 2004. http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/.

[30] T. Bray, J. Paoli, E. Maler, F. Yergeau, and C. M. Sperberg-McQueen, "Extensible markup language (XML) 1.0 (fifth edition)," W3C recommendation, W3C, Nov. 2008. http://www.w3.org/TR/2008/REC-xml-20081126/.

[31] E. Ray, *Learning XML, Second Edition*. O'Reilly Media, Inc., Sept. 2003.

[32] T. Berners-Lee and D. Connolly, "Notation3 (n3): A readable rdf syntax," W3C team submission, W3C, Mar. 2011. http://www.w3.org/TeamSubmission/2011/SUBM-n3-20110328/.

[33] D. B. T. Berners-Lee, "Turtle - terse rdf triple language," W3C team submission, W3C, Mar. 2011. http://www.w3.org/TeamSubmission/2011/SUBM-turtle-20110328/.

[34] S. Pemberton, B. Adida, S. McCarron, and M. Birbeck, "RDFa in XHTML: Syntax and processing," W3C recommendation, W3C, Oct. 2008. http://www.w3.org/TR/2008/REC-rdfa-syntax-20081014.

[35] S. Pemberton, "XHTML^{TM} 1.0 the extensible hypertext markup language (second edition)," W3C recommendation, W3C, Aug. 2002. http://www.w3.org/TR/2002/REC-xhtml1-20020801.

[36] A. Perego, P. Archer, and K. Smith, "Protocol for web description resources (POWDER): Description resources," W3C recommendation, W3C, Sept. 2009. http://www.w3.org/TR/2009/REC-powder-dr-20090901/.

[37] A. Perego, P. Archer, and K. Smith, "Protocol for web description resources (POWDER): Grouping of resources," W3C recommendation, W3C, Sept. 2009. http://www.w3.org/TR/2009/REC-powder-grouping-20090901/.

[38] P. Archer and S. Konstantopoulos, "Protocol for web description resources (POWDER): Formal semantics," W3C recommendation, W3C, Sept. 2009. http://www.w3.org/TR/2009/REC-powder-formal-20090901/.

[39] K. Scheppe, "Protocol for web description resources (POWDER): Primer," W3C note, W3C, Sept. 2009. http://www.w3.org/TR/2009/NOTE-powder-primer-20090901/.

[40] D. Brickley and L. Miller, "Foaf vocabulary specification 0.98," specification, Aug. 2010. http://xmlns.com/foaf/spec/20100809.html.

[41] The Dublin Core Metadata Initiative, "Dublin core." website. http://dublincore.org last retrieved on 5 Feb 2012.

[42] R. V. Guha and D. Brickley, "RDF vocabulary description language 1.0: RDF schema," W3C candidate recommendation, W3C, Mar. 2000. http://www.w3.org/TR/2000/CR-rdf-schema-20000327/.

[43] P. Hayes, P. F. Patel-Schneider, and I. Horrocks, "OWL web ontology language semantics and abstract syntax," W3C recommendation, W3C, Feb. 2004. http://www.w3.org/TR/2004/REC-owl-semantics-20040210/.

[44] W3C OWL Working Group, "OWL 2 web ontology language document overview," tech. rep., W3C, Oct. 2009. http://www.w3.org/TR/2009/REC-owl2-overview-20091027/.

[45] P. F. Patel-Schneider and B. Motik, "OWL 2 web ontology language mapping to RDF graphs," W3C recommendation, W3C, Oct. 2009. http://www.w3.org/TR/2009/REC-owl2-mapping-to-rdf-20091027/.

[46] M. Krötzsch, P. F. Patel-Schneider, S. Rudolph, P. Hitzler, and B. Parsia, "OWL 2 web ontology language primer," W3C recommendation, W3C, Oct. 2009. http://www.w3.org/TR/2009/REC-owl2-primer-20091027/.

[47] P. F. Patel-Schneider, B. Motik, and B. C. Grau, "OWL 2 web ontology language direct semantics," W3C recommendation, W3C, Oct. 2009. http://www.w3.org/TR/2009/REC-owl2-direct-semantics-20091027/.

[48] M. Schneider, "OWL 2 web ontology language RDF-based semantics," W3C recommendation, W3C, Oct. 2009. http://www.w3.org/TR/2009/REC-owl2-rdf-based-semantics-20091027/.

[49] B. Motik, A. Fokoue, I. Horrocks, Z. Wu, C. Lutz, and B. C. Grau, "OWL 2 web ontology language profiles," W3C recommendation, W3C, Oct. 2009. http://www.w3.org/TR/2009/REC-owl2-profiles-20091027/.

[50] N. Eisinger and J. Maluszynski, eds., *Reasoning Web, First International Summer School 2005, Msida, Malta, July 25-29, 2005, Tutorial Lectures*, vol. 3564 of *Lecture Notes in Computer Science*, Springer, 2005.

[51] E. Prud'hommeaux and A. Seaborne, "SPARQL query language for RDF," W3C recommendation, W3C, Jan. 2008. http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/.

[52] P. Gearon, A. Passant, and A. Polleres, "SPARQL 1.1 update," W3C working draft, W3C, Jan. 2012. http://www.w3.org/TR/2012/WD-sparql11-update-20120105/.

[53] S. Harris and A. Seaborne, "SPARQL 1.1 query language," W3C working draft, W3C, Jan. 2012. http://www.w3.org/TR/2012/WD-sparql11-query-20120105/.

[54] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean, "Swrl: A semantic web rule language - combining owl and ruleml," W3C member submission, W3C, May 2004. http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/.

[55] A. Khandelwal, J. Bao, L. Kagal, I. Jacobi, L. Ding, and J. Hendler, "Analyzing the air language: A semantic web (production) rule language," in *Web Reasoning and Rule Systems* (P. Hitzler and T. Lukasiewicz, eds.), vol. 6333 of *Lecture Notes in Computer Science*, pp. 58–72, Springer Berlin / Heidelberg, 2010.

[56] A. Katasonov and V. Terziyan, "Semantic Agent Programming Language (S-APL): A Middleware Platform for the Semantic Web," in *ICSC '08: Proceedings of the 2008 IEEE International Conference on Semantic Computing*, (Washington, DC, USA), pp. 504–511, IEEE Computer Society, 2008.

[57] V. Terziyan, A. Katasonov, O. Kaykova, O. Khriyenko, O. Loboda, A. Naumenko, and S. Nikitin", "Deliverable D1.1 the central principles and tools of ubiware," Deliverable D1.1, Industrial Ontologies Group - Agora Center, University of Jyväskylä, Jyväskylä, Finland, nov 2007.

[58] A. Katasonov, "Semantic agent programming language (s-apl) developer's guide," technical report, Jyväskylän Yliopisto, Apr. 2010. `http://users.jyu.fi/~akataso/SAPLguide.pdf`, retreived on 6 april 2010.

[59] F. for Intelligent Physical Agents, "Fipa agent management specification," *Available online at* `http://www.fipa.org/specs/fipa00023/`, no. 23, 2004.

[60] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.

[61] V. Terziyan and et al.", "Smartresource project final report," deliverable, SmartResource Tekes Project - Agora Center, University of Jyväskylä, Jyväskylä, Finland, 2007.

[62] O. Khriyenko, V. Terziyan, and et al.", "Ubiware final project report," deliverable, UBIWARE Tekes Project - Agora Center, University of Jyväskylä, Jyväskylä, Finland, 2010.

[63] P. Petta and J. Müller, *Multiagent system technologies: 5th German conference, MATES 2007, Leipzig, Germany, September 24-26, 2007 : proceedings*. Lecture notes in artificial intelligence, Springer, 2007. SmartResource Platform and Semantic Agent Programming Language (S-APL) by Artem Katasonov and Vagan Terziyan.

[64] V. Terziyan, M. Nagy, M. Cochez, V. Pilli-Sihvola, J. Kesäniemi, and O. Khriyenko", "Deliverable D3.4 ubiware platform prototype v. 3.1," Deliverable D3.4, Industrial Ontologies Group - Agora Center, University of Jyväskylä, Jyväskylä, Finland, nov 2010.

[65] J. Linn, "Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures," RFC 1421, Internet Engineering Task Force, Feb. 1993.