Documentation

SemReasoner

Content

1 Introduction

Knowledge Models are used to provide central access to knowledge about certain application domains for different enterprise applications. Such a knowledge model consists of an ontology which defines the chosen application domain. An ontology determines the entities, properties of those entities and relationships between those entities. All those are defined for subject matter experts in understandable terms and structures. Thus they are easy to create and maintain by subject matter experts. In contrast to a database schema which has been created by an IT expert an ontology is created in a social process which ensures the acceptance of those social group.

A popular example for such an ontology is Schema.org which covers many different application domains. Schema.org is further developed by proposals for changes, refinement and extensions. There is a board which decides for such changes of the ontology. Schema.org has originally been developed for annotating web sides, but may also be used for other applications. The tool for storing and for handling such knowledge models together with concrete data (instances) is usually a triple store or a graph data base.

Such a knowledge model can be used for different purposes. For instance it can be used to create recommendations like next-best-offer or it can be used to create intelligent answers for a chatbot or similar. Such a model can also be used to perform an intelligent search (semantic search) for a set of documents or it can be used to integrate information from external sources or services (semantic information integration).

In the following we will first present the language OO-logic for describing ontologies and their instances. Then we will roughly describe the architecture of SemReasoner which is the graph store for ontologies and their instances.

2 Ontologies in OO-logic

Object oriented logic (abbreviated: OO-logic) combines the advantages of conceptual modelling that come from object-oriented frame-based languages with the declarative style, compact and simple syntax, and the welldefined semantics of a logic-based language. OO-logic supports typing, meta-reasoning, complex objects, properties, classes, inheritance, rules, queries, modularization, and scoped inference. We will describe the capabilities of knowledge representation systems based on OO-logic and illustrate the use of this logic for ontology specification. We give an overview of the syntax and semantics of the language in SemReasoner.

2.1 Introduction to OO-logic

A conceptual model (or an ontology) is an abstract, declarative description of the information for an application domain. It includes the relevant vocabulary, constraints on the valid states of the information, and the ways to draw inferences from that information.
Conceptual modelling has a long history in the area of database systems. It began with the seminal work on the Entity-Relationship (ER) model [Che76], which divided the world into entity types and relationship types. Entity types are sets of homogeneous entities specified via their attributes and ranges. Entities represent concrete objects that populate entity types and whose structure is compliant with the types they belong to. Relationship types are likewise sets of homogeneous relationships. They are specified by the roles and the entity types that are involved in the relationships. The ER model is thus a rudimentary language for specifying ontologies for the kinds of information that is natural to store in relational databases. ER was later extended to EER (Extended Entity Relationship) model [RES06,KBL05] by adding specialisation, generalization, grouping, and other features. Subsequent modelling languages, like UML, were greatly influenced by the works on ER and EER.
Declarative query languages, like SQL, were central to relational database systems from their inception. The most attractive aspect of database query languages is the fact that their queries say which things to find rather than how to find them. Clearly, such languages are much harder to implement than the traditional imperative languages, like C. In the early days, database query languages had to be severely emasculated in order to enable reasonably efficient implementations. However, as applications grew in sophistication, computing power increased, and our knowledge of algorithms for query processing enriched, the use of rule-based languages for processing information has become more and more attractive. A further push came from the Semantic Web, which increased the awareness of the need for logic-based languages for processing ontologies and other distributed knowledge on the Web. This awareness led W3C to create a working group that was chartered with creation of a Rule Interchange Format (RIF) for Web-enabled applications written in rule-based languages[1].
Datalog [AHV95] is the granddaddy of all database rule languages. It has a model-theoretic semantics, can be efficiently implemented, and is reasonably expressive. However, it does not support function symbols, which are important for representing objects, and it is a poor choice as a modelling language. To improve the modelling power of logic-based languages, a number of extensions were proposed. These include more powerful kinds of negation, function symbols, high-level modelling constructs, and frame-based syntax. F-logic (or Frame Logic) [KLW95] has emerged as a popular extension that provides all these features. As conceptual modelling goes, it faces little competition among rule-based logic languages. It accounts in a clean and declarative manner for most of the structural aspects of frame-based and object-oriented languages and, at the same time, is as powerful as any rule-based language for knowledge representation. An overview of logic-based languages can be found in [L99]. Related, but

[1] http://www.w3.org/2005/rules/wg.html

limited languages have also been proposed for semi-structured and XML databases (e.g. [GPQ97], [M01]).
There are several major implementations of F-logic, including FLORID [LHL98], OntoBrokerTM[DEFS99], and FLORA-2 [YKZ03]. Each implementation introduces a number of extensions to F-logic as well as restrictions to make their particular implementation methods more effective. OntoBrokerTM was a commercial system. Its main emphasis is on efficiency and integration with external tools and systems. SemReasoner is the successor of OntoBroker. It simplifies the syntax of F-logic to OO-logic and has its focus on rule based reasoning with huge sets of data (big data). The simplification of the language reflects the experience in various industrial projects. This simplified subset is what users really understood and what they really used in these projects.
This paper takes a view of OO-logic as an ontology modelling language as well as a language for building applications that use these ontologies. The ability to span both sides of the engineering process, ontologies and applications, is a particularly strong point of OO-logic.
In the following sections we give an overview of the syntax and the semantics of OO-logic by illustrating the main features through a number of simple examples. Then we describe the ways in which OO-logic can be implemented.

2.2 OO-logic by Example

In this paper we use the syntax for OO-logic. It incorporates experience gained in the course of a decade of using F-logic in real life applications.

2.2.1 A Simple Ontology-Based Application

OO-logic is an object-oriented language and ontologies are modelled in this language in an object-oriented style. One starts with class hierarchies, proceeds with type specifications, defines the relationships among classes and objects using rules, and finally populates the classes with concrete objects.

```
/* ontology */
Person::Class. // Person is a class
Woman::person. // class hierarchy
Man::Person.
Person[properties: father]. // signatures
Person[properties: mother].
person[properties: daughter].
Person[properties: son].

father[range:Man].                                          (1)
mother[range: Woman].
daughter[range: Woman].
son[range: Man].
```

```
/* rules consisting of a rule head and a rule body */
r1: ?X[son: ?Y] :- ?Y:Man, ?Y[father: ?X].
r2: ?X[son: ?Y] :- ?Y:Man, ?Y[mother: ?X].
r3: ?X[daughter: ?Y] :- ?Y:Woman, ?Y[father: ?X].
r4: ?X[daughter: ?Y] :- ?Y:Woman, ?Y[mother: ?X].

/* facts */
Abraham:Man.
Sarah:Woman.
Isaac:Man.
Isaac[father: Abraham, mother: Sarah].
Ishmael:Man.
Ishmael[father: Abraham, mother: Hagar].
Jacob:Man.
Jacob[father: Isaac, mother: Rebekah].
Esau:Man.
Esau[father: Isaac, mother: Rebekah].


/* query */
?-  ?X:Woman, ?X[son: ?Y], ?Y[father: Abraham].                    (2)
```

The first part of this example is a small ontology. It states that every woman and every man is a person. Persons also have attributes, like father, mother, daughter, son, and the attributes have ranges (the range of son is man, for example, and of mother is woman). The statements that specify the types of the attributes are called signatures.

Then follows a set of rules, which say what else can be derived from the ontology. The first rule says that if ?X is the father of ?Y and ?Y is a man, then ?Y is a son of ?X. A similar relationship holds for sons and mothers, and for daughters, fathers, and mothers. All variables in a rule are implicitly quantified outside of the rule. For instance, from the logical point of view the first rule above is really an abbreviation for

```
∀?X ∀?Y ( ?X[son: ?Y] ← ?Y:man[father: ?X])
```

Once the ontology is ready, we populate it with facts, some of which are specified at the end of the example. These facts tell us that Abraham is a man and Sarah is a woman. We also learn that Isaac is a man and his parents are Abraham and Sarah. Similar information is supplied on a number of other well-known individuals. These facts form the basis of an object base in our example. Other facts can be derived via deductive rules or other inference rules, such as inheritance. For instance, we can derive that Isaac is a son of Abraham even though this is not stated explicitly via a fact.

The final part of Example 1 contains a query (2) to the object base. It inquires about women and their sons by Abraham and will return the answer ?X = Sarah, ?Y = Isaac.:

Note that all properties are multivalued. For instance, from the rules we can derive

```
Abraham[son: Isaac].
Abraham[son: Ishmael].
```

2.2.2 Objects and their Properties

Example 1 shows that class hierarchies, signatures, rules, and objects are the main components of an OO-logic knowledge base. In this section we will take a closer look at each of these components in turn.

2.2.2.1 Object Names and Variable Names

Object names and variable names, also called id-terms, are the basic syntactical elements of OO-logic. To distinguish object names from variable names, the later are prefixed with a ?-sign. Examples of object names are Abraham, man, daughter, and of variable names are ?X, or ?method. There are two special types of object names that carry additional information: integers and strings. Any string can be an object name. Strings are enclosed in quotation marks '…', but alphanumeric strings do not require quotes. Complex id-terms are created from function symbols and other id-terms as usual in predicate logic: couple(Abraham, Sarah), f(?X). An id-term that contains no variable is called a ground id-term.

2.2.2.2 Properties

Application of a property to an object is specified using data-OO-atoms. A remarkable feature of OO-logic is that properties are also denoted by objects and can be handled like regular objects without any special language support. For instance, in Example 1 the property names father and son are object names just like Isaac and Abraham.

Note that variables in a query may either be bound to individual objects or to sets of objects. In our case the different values of the property son are not included in a set. Thus, the query: ?- Abraham[son: ?X]. binds ?X to the elements of all those values one by one. As a consequence, all the following queries return the answer true:

```
?- Abraham[son: Isaac].
?- Abraham[son: Ishmael].
```

Instead we can also define a set of such values:

```
Abraham[son: /Isaac,Ishmael/].
```

In this case the query: ?- Abraham[son: ?X]. binds the variable ?X to the whole set at once. If we want to have single values in this case we have to write the query like that: ?- Abraham[son: {?X}]. This handling of multiple values and sets in different to F-logic.

A property name can be an arbitrary term in OO-logic. For instance the following query returns all properties together with its values of Abraham:

```
?- Abraham[?PropertyName: ?PropertyValue].
```

2.2.2.3 Class Hierarchies

Class hierarchies are defined with the help of isa-F-atoms and subclass-F-atoms. An isa-F-atom of the form o:c states that an object o is a member of class c. The members of a class typically are called the instances of the class. A subclass-F-atom of the form sc::cl says that the class sc is a subclass of the class cl. In the following example the first three isa-F-atoms say that Abraham and Isaac are instances of the class man, whereas Sarah is an instance of the class woman. The two subclass-F-atoms that follow state that both man and woman are subclasses of the class person:

```
Abraham:Man.
Isaac:Man.
Sarah:Woman.
Woman::Person.
Man::Person.
```

In OO-logic, classes are also objects and thus are represented by id-terms. Hence, classes can have properties defined on them, and they can be instances of other classes. As mentioned earlier, properties are also objects and as such can belong to classes (or can themselves serve as classes). This can be used for meta modelling like annotating ontology elements with provenance information:

```
son:Relation.
son[authoredby: Hans].
```

Furthermore, variables are permitted at all positions in isa- and subclass-F-atoms, so objects, properties, and classes are represented and queried uniformly using the same language facilities. In this way, OO-logic naturally supports the meta-information facility. In contrast to other object-oriented languages where an object can be an instance of exactly one most specific class (e.g., ROL [L99]), OO-logic permits to be an instance in several, possibly incomparable, most specific classes. Likewise, a class can have several incomparable most specific superclasses. Thus, the class hierarchy is a directed acyclic graph.

2.2.2.4 Expressing Information about an Object: OO-Molecules

OO-molecules are used to make several different assertions about the same object in a compact way. For example, the following OO-molecule says that Isaac is a man, his father is Abraham, and Jacob is (one of) his sons.

```
Isaac:Man.
```

```
Isaac[father: Abraham, son: Jacob].                          (6)
```

This OO-molecule is equivalent to a conjunction of the following atomic statements:

```
Isaac:Man.
Isaac[father: Abraham].
Isaac[son: Jacob].
```

A property can have multiple values:

```
Isaac[son: Jacob].
Isaac[son: Esau].                                            (8)
```

(8) can be abbreviated by:

```
Isaac[son: /Jacob,Esau/].
```

2.2.2.5 Signatures

Signature-OO-atoms specify the schema of a class. They declare the properties that apply to the various classes, the types of the arguments used by those properties, and their ranges. In contrast to F-logic which provides a sophisticated schema modelling language in OO-logic meta modelling is used. This means that classes and properties itself are seen as objects and we can have properties for these objects as well.

In the following we want to express that class person has properties father,mother,daughter, and son. Property father has the range man.:

```
Person[properties: /father,mother,daughter,son/].
father[range: Man].
```

For simple valued properties the standard data types String,Integer, Float, Double, Boolean, Calendar, Duration, and URI are used:

```
firstName[range: String].
lastName[range: String].
birthday[range: Calendar].
```

Meta modelling allows to add various information to such properties. For instance we can express that a property is mandatory which means that every instance of the class must have a value for that property:

```
lastName[range: String, mandatory: true].
```

2.2.2.6 Inheritable Properties

OO-logic defines properties for a class, they are inherited by subclasses and instances, and they are invoked by applying them to instances of a class.

So if we have the statements:

```
Person[properties: /father,mother,daughter,son/].
Man::Person.
```

then man inherits the properties father, mother, daughter, and son.

2.2.2.7 Predicate Symbols

Experience shows that it is convenient to be able to use predicates alongside objects. In OO-logic, predicate symbols are used in the same way as in other deductive languages, e.g., in Datalog. A predicate formula is constructed out of a predicate symbol followed by one or more id-terms separated by commas and included in parentheses. Such a formula is called a P-atom. Example 13 shows some P-atoms. The last P-atom is a 0-ary predicate symbol, a proposition.

```
married(Isaac,Rebekah).
male(Jacob).                                                    (13)
```

Information expressed by binary P-atoms can usually be represented by OO-atoms, as shown below:

```
Isaac[marriedto: Rebekah].
Jacob:Man.
```

OO-logic supports built-ins which define procedural attachments that are seamlessly embedded into OO-Logic. Usually a built-in represents an infinite relation. Built-ins are always preceded by an “_”. E.g. adding two numbers to a third can be considered as an infinite relation between 3 numbers:

```
 _plus(?X,?Y,?Z)
```

Dependent on 2 of the three arguments an algorithm behind the built-in maths the third argument.
Appendix B shows a list of available built-ins.

2.2.2.8 Paths

Paths are a standard fixture in most object-oriented languages. In OO-logic, a path expression of the form obj.expr denotes the set of objects {a1,a2,...}, such that obj[expr: {a1,a2,...}] is true. The expression in the path expression can be a simple attribute or a method application. Method expressions can be further applied to the results of path expressions, and in this way longer path expressions can be constructed: obj.expr1.expr2.expr3. Here are some path expressions and the sets of objects they refer to in our example.

```
Isaac.son                    {Jacob,Esau}
Esau.father.father.son       {Isaac, Ishmael}
```

The second line shows a path expression where expressions are applied repeatedly. Since a method can yield a set of results, one might wonder about the meaning of an expression such as this:

```
Abraham.son.son
```

Since Abraham.son is a set, what does the second attribute, son, apply to? In OO-logic, the answer is that it applies to every object denoted by Abraham.son, and the results of all such applications are unioned. Thus, this path expression denotes the set of all Abraham's grandchildren. Path expressions can appear anywhere an object can.
In imperative object-oriented languages, path expressions provide the only way to navigate object relationships. In OO-logic, most general way to navigate through objects is to use OO-molecules and combine them with logical connectives and and or. However, the use of path expressions can simplify formulas in many cases. Since a path expression, such as obj.expr, denotes all the objects that can bind to the variable ?X in obj[expr: ?X], path expressions can help eliminate variables. For instance, Abraham's grandsons can be represented in either of the following ways:

```
?- Abraham.son.son=?X .
?- Abraham.son[son: ?X].
?- Abraham[son: ?Y], ?Y[son: ?X].
```

These queries have two answers, one per each of Abraham's grandsons. The first query is most concise. The second query combines path expression notation with frame-based notation. It is slightly longer, but both queries use just one variable. The third query does not use path expressions; it is bulkier than the first two, and requires two variables instead of one. It should be clear that by stacking more method applications in one path expression one can eliminate many variables and thus simplify some queries and rules.

2.2.2.9 Sets as values

A property can have a whole closed set as value. This is different to a property has multiple values:

```
Isaac[son: {Jacob, Esau}].
```

To access an element in the set with a variable we have to enclose the variable into curly brackets:

```
Isaac[son: {?X}].
```

If we want to bind the whole set to a variable we use the expression without brackets:

```
Isaac[son: ?X].
```

2.2.3 Rules

Rules are one of the best known technologies for building applications around ontologies. We have already seen examples of rules in Example 1. In general, a rule is an expression of the form head :- body, where the head of the rule is a Boolean combination of OO-molecule and the body is a Boolean combination of OO-molecules or negated OO-molecules. Conjunctions are represented using commas. Disjunction in the body is represented by semi colons. Molecules may contain variables, and all variables are implicitly ∀-quantified outside of the rule. Variables can be anonym like ?_. Using an anonym variable in a rule means that the value which instantiates the variable does not matter. Rules in OO-logic have a logical semantics based on the principles developed in the context of logic programming and deductive databases. Consider the following rule from Example 1:

```
r: ?X[son: ?Y] :- ?Y:Man, ?Y[father: ?X].
```

Its semantics can be informally explained as follows. Whenever we can find id-terms for the variables ?X and ?Y so that all molecules in the body become either existing or derived facts, then the head of the rule is derived after applying the same substitutions to ?X and ?Y in the head. In case of our rule above, its body is true for ?X=Abraham and ?Y=Isaac or ?Y=Ismael; or for ?X=Isaac and either ?Y=Jacob or ?Y=Esau. This is due to the facts that Isaac:man, Isaac[father: Abraham], Ishmael:man, Ishmael[father: Abraham], Jacob:man, Jacob[father: Isaac], Esau:man, Esau[father: Isaac] are explicitly given in Example 1. From these facts and the above rule we can derive Abraham[son: Isaac], Abraham[son: Ishmael], Isaac[son: Jacob], Isaac[son: Esau]. We can write these facts in a more concise way as Abraham[son: {Isaac,Ishmael}] and Isaac[son: {Jacob,Esau}]. Similarly, with the rule

```
r: ?X[grandson: ?Y] :- ?Y:Man, ?Y[father: ?Z], ?Z:Man, ?Z[father: ?X]].
```

we can derive Abraham[grandson: {Jacob, Esau}].
When molecules in a rule body are negated, a form of the well-founded semantics is typically used in F-logic systems [GRS91, YK06].

Rules can be recursive. For example, given a genealogy (a parenthood relationship), we may want to specify ancestry information as follows:

```
r1: ?X[ancestor: ?Y]  :-  ?X[parent: ?Y].
r2: ?X[ancestor: ?Y]  :-  ?X[ancestor: ?Z], ?Z[parent: ?Y].
```

Note that the generation property is, in general, multivalued, since there can be multiple genealogical lines connecting a pair of individuals.

The following rule shows a conjunction in the head and a disjunction in the body:

```
r: ?X:Person, ?X[parent: ?Y] :- ?X[father: ?Y]; ?X[mother: ?Y].
```

It states that if ?X has father ?Y or ?X has mother ?Y then ?X has parent ?Y and ?X is a person. In such expressions every disjunction in the rule body must ensure that to all variables in the rule head values are assigned, i.e. every variable in the head must occure in every disjunction of the body.

When molecules in a rule body are negated, a form of the well-founded semantics is typically used in F-logic like systems [GRS91, YK06].

A rule may call built-ins. The following rule maths the value of a mathematical expressions and assigns those value to the variable ?X (this can be expressed in a much nicer syntax, see 2.2.8):

```
r: ?X[a: ?V] :- ?X[b: ?Z], _plus(?Z,5,?Y), _mult(?Y,3,?V).
```

The last two built-in literals can be combined using the math built-in (this can be expressed in a much nicer syntax, see 2.2.8) :

```
r: ?X[a: ?V] :- ?X[b: ?Z], _math(mult(plus(?Z,5),3),?V).
```

Similar to that complex boolean expressions can be described using the istrue built-in:

```
r: ?X[a: ?V] :- ?X[b: ?Z], _istrue(and(less(?Z,5),greater(?Z,3)).
```

2.2.4 Constraints

Rules express additional facts which are not explicitly given but are deduced from given facts. In contrast to rules constraints express restrictions of facts. The following constraint expresses that an adult person must be older than 18 years:

```
adultConstraint: !- ?X:Adult, ?X.age > 18 .
```

As soon as we add an instance to class Adult whose age is less than 18 this constraint is violated.

2.2.5 Scoped Inference: Modularization and Integration

The concept of scoped inference [K05, KBBF05] is central to modularization and integration of knowledge. It was first proposed in TRIPLE [SD02] and FLORA-2 [YKZ03], and the F-logic Forum has adopted this concept as the main vehicle for modularization of F-logic ontologies.2

The concept of a module is well known in software engineering, and it is equally important in knowledge engineering. It is especially important for representing distributed knowledge, such as ontologies scattered over the Web, since rules and concepts that belong to different ontologies may interact in subtle and unintended ways.

The basic idea is that a knowledge base is a collection of scopes of inference or modules. Each module is a collection of rules and facts. The notion of a rule is extended as follows. As before, it is a statement of the form

```
Head :- Body.
```

Predicates and molecules in a rule can optionally be labeled with module references like this: pred-or-molecule@module-name. A subformula of the form L@N inside body or a rule is a query to module N, which asks whether L is implied by the knowledge base that resides in module N. For instance, some data source, gendata, may provide information about parents of various individuals. One may not be able to (or may not want to) insert new rules into that data source in order to preserve the integrity of that data. However, it is possible to create a different module, say mygenealogy, which reference the information in the aforesaid data source:

```
r1: ?X[ancestor: ?Y] :- ?X[parent: ?Y]@gendata.
r2: ?X[ancestor: ?Y] :- ?X[parent: ?Z]@gendata, ?Z[ancestor: ?Y].
```

Here the literals of the form …[parent:…]@gendata are queries to the module gendata. Some other module might query both of these modules, but the answers might be different. For instance, the queries

```
?-  ?X[parent: ?Y]@gendata.
?-  ?X[parent: ?Y]@mygenealogy.
```

will likely return different answers, since the parent attribute is not defined in mygenealogy. Thus, the first query will return all it knows about the parenthood relationship among individuals, while the second query will return nothing. Likewise,

2 http://projects.semwebcentral.org/projects/forum/forum-syntax.html

```
    ?-  ?X[ancestor: ?Y]@gendata.
    ?-  ?X[ancestor: ?Y]@mygenealogy.
```

will return different answers: the ancestor attribute is not defined in module gendata so the first query will return nothing, while the second will return the transitive closure of the parent attribute (which mygenealogy imports from gendata using the above rules). In the same way as rule bodies access certain modules the literals in the head may derive information for certain modules. For instance the rules above may derive data into a module called derivedgenealogy

```
r1:?X[ancestor: ?Y]@derivedgenealogy:-?X[parent: ?Y]@gendata.
r2:?X[ancestor:?Y]@derivedgenealogy:-?X[parent:?Z]@gendata,?Z[ancestor: ?Y].
```

Module names can be arbitrary terms. That module names can also be variables is an extension to F-logic.

Besides modularization, the concept of a module is a potent vehicle for integration of and reasoning about ontologies that reside at different sources. If one just unions the rules and the facts found at the sources of interest, as implied by the import mechanism of the OWL language, the rules may contradict each other or have subtle and unintended interactions. In contrast, if different sources are treated as separate modules, one can distinguish the information defined at the different sources and then specify the appropriate integration rules. These rules may give preference to some sources, partially or completely disregard information supplied by others, or clearly flag conflicting information.

2.2.6 Negation

Inside queries and rules literals can be negated. Using negation we can for instance easily express the set of men which are not fathers:

```
     r: nofather(?P) :- ?P:Man, ! ?P:father.
```

Using negation has some restrictions. In a negated literal all variables have to be bound which is the case in our example variable ?P is bound by the literal ?P:Man. While variables in a rule are all quantified in front of the rule, negation changes the quantification of a variable to an existential quantification. This means that in the following formula ?Y is all quantified. This means that for all possible values of ?Y body2 must not hold.

```
    ∀?X∀?Y ( head(?X,?Y) :- body1(?X), ! body2(?X,?Y))
    = ∀?X ( head(?X,?Y) :- body1(?X), ! ∀?Y body2(?X,?Y))
```

From the logical point of view all values means all values which are possible at all, e.g. all numbers which is infinite and thus cannot be processed. In contrast to that very often you want to express all possible values for ?Y which are available in the data for body2. This can be expressed in two rules only. This is one transformation of the Lloyd Topor transformation [LT84].

```
r1: head(?X,?Y) :- body1(?X), ! p(?X).
r2: p(?X) :- body2(?X,?Y).
```

In the area of deductive databases the semantics of negation differs from the semantics of negation in first order logic. In deductive databases we always consider one model only because we want to math one result set for our query only and not different alternative result sets. The semantics for negation in deductive logic is called stratified negation [GRS91].

2.2.7 Path Expressions in Graphs

Path expressions allow to run over edges in a graph. For example if we have nodes and edges like:

```
Node[properties: /name, edge/].
name[range: String].
edge[range: Node].
```

We can now express that we want to start at a node with name "peter" and get all dependent nodes:

```
// get all dependent nodes
?- ?N:Node[name:"peter", edge+:?D].

// skip two edges and get all dependent nodes
?- ?N:Node[name:"peter", edge{2,*}:?D].

// get dependent nodes with distance 2 to 5
?- ?N:Node[name:"peter", edge{2,5}:?D].

// get all dependent nodes with name "mary"
?- ?N:Node[name:"peter", edge+:?D], ?D[name:"mary"].
```

Path expressions may also be used inside paths:

```
?Y.edge{2,5}.name="mary"
```

Inside paths we can have brackets to cross sub paths which have a sequence of differently named edges. The following expression determines all classes related to class ?X with a max distance of 4:

```
?X::Class, ?Y := ?X(.properties.range){1,4}
```

2.2.8 Special Syntax

For convenience reasons OO-logic provides a special syntax for comparisons and for mathematical expressions:

```
r0: ?X: YoungPeople :- ?X[age: ?Z], ?Z < 20.
r1: ?X: Teenager :- ?X[age: ?Z], 13 <= ?Z < 20.
```

Besides '<' the comparison operators '>', '=', '!=', '<='. and '>=' supported in such comparisons.

An assignment of the value a mathematical expression is given in the following way:

```
r: ?X[a: ?Y] :- ?X[b: ?Z], ?Y := (?Z+5)*3.
```

These mathematical expressions allow besides the basic arithmetical operations also functions like sin, cos, sqrt, etc. A list of available functions is given appendix B. In all places where numbers can occur mathematical expressions are allowed to be used.

```
r1: ?X: Teenager :- ?X[age: ?Z], 13 <= ?Z + 1 < 20.
```

Aggregations are special built-ins. In contrast to ordinary built-ins they have a whole set of values as input and generate output values dependent on this whole set of values. In newer logic languages [] a special syntax has been introduced for such aggregations:

```
?N := count{?X | ?X:Person}
```

This expression means: count all instances of class Person and assign this number to variable ?N. Such aggregation expressions can occur in rule bodies:

```
r: numberOfPersons(?N) :- ?N := count{?X | ?X:Person, ?X[hasAge: 20]}.
```

This rule determines the number of persons who are 20 years old.

Such aggregation expressions can also have a grouping variable. This means that the results are grouped according to the values of the grouping variable ?G:

```
r: numberOfPersons(?G,?N) :- ?N := count{?X [?G] | ?X:Person, ?X[hasAge: ?G]}.
```

This rule determines for all occurring ages the number of persons with that age. Assumed we have the following persons:

```
        peter[hasAge: 15].
        thomas[hasAge: 20].
        anja[hasAge: 20].
```

The rule will create the following results:

```
        15,1
        20,2
```

i.e. there is one person with age 15 and two persons with age 20.

In boolean expressions comparisons may be combined with *and* (&), *or* (|), *not* (!) and functions:

```
    r0: ?X: OldOrYoungPeople :- ?X[age: ?Z], (?Z < 20) | (?Z > 60).
    r1: ?X: MyPeople :-
          ?X[name: ?N],
          (_contains(?N,"Maier") | _contains(?N,"Schmidt")) & _length(?N) > 10).
```

2.3 Implementation of OO-logic Semantics

Implementations of OO-logic based systems can be roughly divided into two categories: those that are based on native object-oriented deductive engines and those that use relational deductive engines. The engines can be further divided into bottom-up and top-down engines.
FLORID implements F-logic using a dedicated bottom-up deductive engine, which handles objects directly through its object manager. In that sense, it is similar to object-oriented databases. In contrast, FLORA-2 and OntoBrokerTM use relational engines, which do not support objects directly. Instead, both systems translate F-logic formulas into statements that use predicates (relations) instead of F-logic molecules and then execute them using relational deductive engines. FLORA-2's target engine is XSB – a Prolog-like inference engine with numerous extensions, like tabling, which make it into a more declarative system than the regular prologs. XSB's inference mode is top-down with a number of bottom-up-like extensions (notably, the aforesaid tabling). SemReasoner uses its own relational deductive engine. SemReasoner's main inference mode is bottom-up, but it includes several enhancements inspired by top-down inference, such as optimized, embedded Magic Sets [BR91].
The main ideas of the actual translation from OO-logic into the relational syntax are as follows:

- First, molecular expressions are replaced by equivalent conjunctions of atomic molecules. We illustrated this process in earlier sections.
- Next, these atomic expressions are represented by first-order predicates.

- The resulting set of rules augmented with additional “closure rules” to capture the specific semantics of OO-logic. Some rules are needed to express statements such as the transitivity of the subclass relationship. Others implement inheritance, and so on.

Table 1 shows the second stage in the above process. Whenever an OO-logic specification is split into modules, the predicates type, subclass, member, etc., in table are disambiguated for different modules either by adding an additional attribute or by specializing predicate names for each module.

Table 1. Transformation of OO-logic atoms into predicate notation

OO-atom	**Predicate**
`A::B`	`subclass(A,B)`
`o:C`	`member(o,C)`
`o[A: b]`	`value(o,A,b)`
`o[A: {b,c}]`	`value(o,A,b). value(o,A,c).`
$p(b_1,.,b_n)$	$p(b_1,.,b_n)$
`A::B@M`	`subclass(A,B,M)`
`o:C@M`	`member(o,C,M)`
`o[A: b]`	`value(o,A,b,M)`

The following are examples of some of the closure rules added in stage 3 of the above process:

```
 // closure rules for ?X :: ?Y
a1: subclass(?X, ?Z) :-  subclass(?X, ?Y), subclass(?Y, ?Z).

// closure rules for ?X : ?C
 a2: member(?O, ?C) :-  subclass(?C1, ?C), member(?O, ?C1).

 // structural inheritance of signatures
 a3: property(?C1, ?A) :- subclass(?C1, ?C2), property(?C2, ?A).
```

The resulting sets of rules are then processed using the stratified semantics for rule-based languages [GRS91].

3 SemReasoner Concepts

3.1 Introduction

SemReasoner provides a comprehensive, scalable and high-performance deductive database. It provides Horn logic as well as OO-logic for defining rules. OO-logic is a successor of F-Logic [KLW95,AKL09].
Based on SemReasoner, ontology-based applications can be developed which offer the following advantages (amongst others):

- The shared meaning (semantics) of information in a knowledge model
- The capturing of complex relationships

Hence, the know-how and the business logic can be modeled separately from the execution logic. Users can flexibly adapt and extend the logic. When used systematically, ontologies form a conceptual semantic layer. This contains the relevant know-how for a particular business area or company, and can be accessed from all applications.
In the following paper we describe SemReasoners's architecture, and SemReasoners's features.

3.2 Architecture

SemReasoner is an ontology repository that combines a graph store with a high performance deductive reasoning engine.
SemReasoner has its own search index integrated for efficiently searching in ontologies, and for searching in documents. SemReasoner includes a Web-server for supporting Web-based applications and Web services.
SemReasoner has been designed to be used as a runtime system within semantic applications, or as a core part of ontology-based services in an Intranet set-up. Hence, it

is available with a lot of different well-documented interfaces and extension possibilities. In the following chapters we will give more details on all of the components mentioned (see fig.1).
The core of SemReasoner contains a storage layer where the ontologies and ontology instances are stored. It is possible to configure this storage option as a persistent layer or as being main memory-based. The persistent layer is implemented as a graph store which is able to load and store huge amounts of triples.
The deductive reasoning engine, (i.e. the inference kernel) processes normal logic programs (Horn logic extended by non-monotonic negation) [L87] using various reasoning methods. The reasoning does not integrate equality reasoning.
One of the major design goals for SemReasoner was a system which is easy to extend and easy to integrate into existing IT landscapes and applications. For this reason, a variety of interfaces are available for this purpose. At the bottom, a connector framework enables various data sources to be attached to an ontology. As the connector API is well-documented, customers and partners can easily develop their own connectors to relational databases, of for LOD (linked open data) [HB11]. SemReasoner can also be extended using built-ins which define procedural attachments that are seamlessly embedded into OO-Logic. Again it is easy for customers to develop their own built-ins. The green parts of fig. 1 show the option of:

- Extending the available built-ins
- Extending the given data types by new data types
- Adding additional rewriters
- Adding importers for additional formats

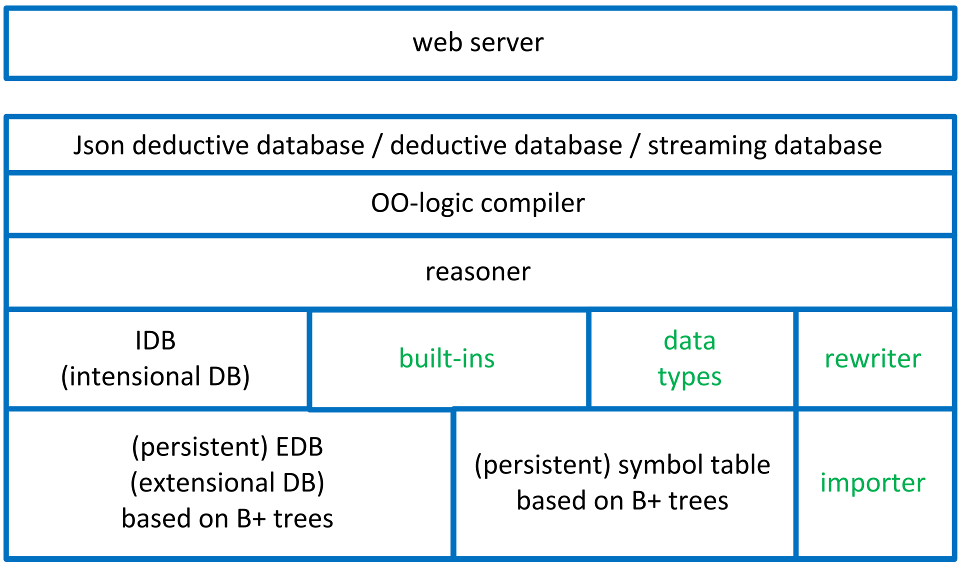

web server
Json deductive database / deductive database / streaming database
OO-logic compiler
reasoner
IDB (intensional DB)
built-ins
data types
rewriter
(persistent) EDB (extensional DB) based on B+ trees
(persistent) symbol table based on B+ trees
importer

Figure 1. SemReasoner's Architecture

The top of the graphic shows the standard web server API that allows the adding and removal of facts, plus the placing of queries in OO-logic. The Java APIs given for deductive database/cep database, for the reasoner, for the intensional database, and for the extensional database are well-documented powerful APIs for embedding the system into arbitrary applications.
SemReasoner is therefore the a very powerful graph store and rule-based reasoner for companies and fulfills most of the information requirements needed for applications on top.

3.2.1 Inference Algorithms

In the kernel of SemReasoner, there are two reasoning methods available: a bottom-up reasoner (also called semi-naive evaluation, or forward chaining reasoner) [U89], and a top-down reasoned based on magic-set technology [BMS86] (backward-chaining reasoner).
In the following we briefly describe bottom-up reasoning, and magic sets. Let's introduce a small example for this purpose. We describe facts and rules about relatives, like parent, child, and sibling in OO-logic (the question marks indicate variables):

```
Leon[hasParent: Anja].
Luca[hasParent: Anja].
Kevin[hasParent: Petra].
Jan[hasParent: Petra].

r: ?X[hasSibling: ?Y]
:-?X[hasParent: ?Z],?Y[hasParent: ?Z].

// ?X is sibling to ?Y if both have the same parent ?Z

?- Leon[hasSibling: ?Y].
// who are the siblings of Leon
```

3.2.1.1 Bottom-up Reasoning

A bottom up reasoner (forward-chaining reasoner) takes the given facts, applies the rules and hence creates derived facts. Once again the rules are applied, and so on. This continues until no new facts can be derived. In our example this looks as follows.
First of all the rule is applied to the facts by substituting the variables by the facts:

```
Luca[hasSibling: Luca] :-Luca[hasParent: Anja,Luca[hasParent: Anja].
Leon[hasSibling: Leon] :- Leon[hasParent: Anja],Leon[hasParent: Anja].
Luca[hasSibling: Leon] :- Luca[hasParent: Anja], Leon[hasParent: Anja].
Leon[hasSibling: Luca] :- Leon[hasParent: Anja],Luca[hasParent: Anja].
Kevin[hasSibling: Kevin] :- Kevin[hasParent: Anja],Kevin[hasParent: Anja].
Jan[hasSibling: Jan] :- Kevin[hasParent: Petra],Jan[hasParent: Petra].
Kevin[hasSibling: Jan] :- Kevin[hasParent: Petra],Jan[hasParent: Petra].
Jan[hasSibling: Kevin] :- jan[hasParent: Petra],Kevin[hasParent: Petra].
```

This leads to the following derived facts:

```
Luca[hasSibling: Luca].
Leon[hasSibling: Leon].
Luca[hasSibling: Leon].
Leon[hasSibling: Luca].
Kevin[hasSibling: Kevin].
Jan[hasSibling: Jan].
Kevin[hasSibling: Jan].
Jan[hasSibling: Kevin].
```

A query is simply a rule without a head. Therefore, the query is applied in the same way to facts by substituting the variables in the query. This leads to the following instantiations of the query:

```
?- Leon[hasSibling: Leon].
?- Leon[hasSibling: Luca].
```

Hence SemReasoner finally answers with the substitutions for the variables in the query:

```
?Y = Leon
?Y = Luca
```

We see that bottom-up reasoning is a simple reasoning method. It has the disadvantage that a lot of (intermediate) facts are generated which are generally not necessary for answering the query. In our case, all of the facts which were derived by the rule but do not instantiate the query were derived needlessly. On the other hand, top-down reasoning sometimes provides so much overhead that this simple reasoning strategy performs best of all.

3.2.1.2 Magic Set Evaluation

Magic set reasoning modifyies the rules and then processes them using a bottom-up reasoner. Let’s take another look at our example to motivate the rule transformation process.
In our query Leon is a ground term. The rule transformation process creates a magic fact and then creates a new rule from our rule:

```
magic1(Leon).
?X[hasSibling: ?Y] :- magic1(?X), ?X[hasParent: ?Z],?Y[hasParent: ?Z].
```

We immediately see that this transformation brings the restricting ground term Leon directly to our rule, hence restricting the intermediate results just at the bottom of the rule graph. Hence the results of the `and` operation of the first two rule literals is:

```
Leon[hasParent: Anja].
```

The result of the `and` operation with the last rule literal then leads to the following intermediate result (instantiations of the three rule variables):

```
Leon, Luca, Anja
Leon, Leon, Anja
```

Hence we get two different rule results, which are also the final results of the query:

```
Leon[hasSibling: Luca]
Leon[hasSibling: Leon]
```

We again see that transforming the rules in that way and processing the resulting rule set in a bottom-up way reduces the number of intermediate results which do not contribute to the answer. For magic set reasoning we observe a trade-off between this reduction effect and the additional performance loss by creating these additional rules. We have seen queries which are better evaluated in a purely bottom-up fashion and others which are better evaluated by magic sets.

3.2.2 The Reasoning Process

A query (an OO-logic query like we have seen in section 2.1) is processed in several successive steps (see Fig. 2). First of all, the query is parsed and compiled into an internal data structure. The IDB (intensional database) contains all of the rules. From this set of rules, our query selects all of the rules that could contribute to the query. The resulting set of rules is then optimized by so called rewriters. Rewriters modify the rules but ensure that the modified rules evaluate to the same answers. For example, our magic set rewriter is such a rewriter. There are much more of these, simple ones like eliminating duplicate literals in rules or more complex ones like rewriting rules into SQL statements when integrating SQL databases.

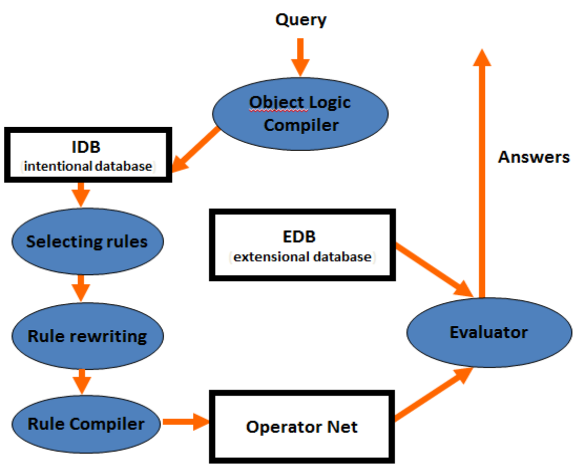


Figure 2. Processing of a query in SemReasoner

Then a rule compiler creates a so-called operator net. Basically an operator net is a low level representation of all of the operations needed for processing the set of rules. Such an operator net contains operations like join, match, access to the EDB, projection, operations for built-ins, operations for connectors, and so on. Fig. 3 shows such an operator net. It contains move operations (MV), join operators (&), collectors, distributors (V), and rule out operators (R). Move operations just move tuples to another node, collectors store intermediate results, and distributors distribute tuples to several other nodes.

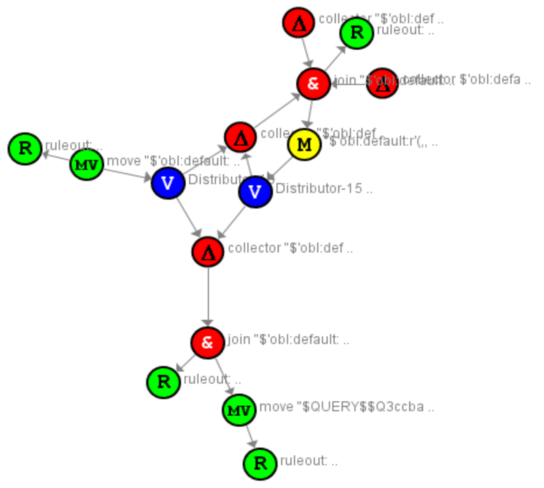


Figure 3. Operator net as a result of bottom-up rule compilation

Such an operator net is purely data flow-oriented. This means that the data flow from data sources (accessors for EDB facts) through the operator net. Every operator performs its operation and sends the results to the successor nodes. Every reasoning algorithm has a separate rule compiler. Our example operator net in fig. 3 has been produced by a bottom-up compiler. The operator net for the same set of rules, but for magic sets as a reasoning algorithm is shown in fig. 4.

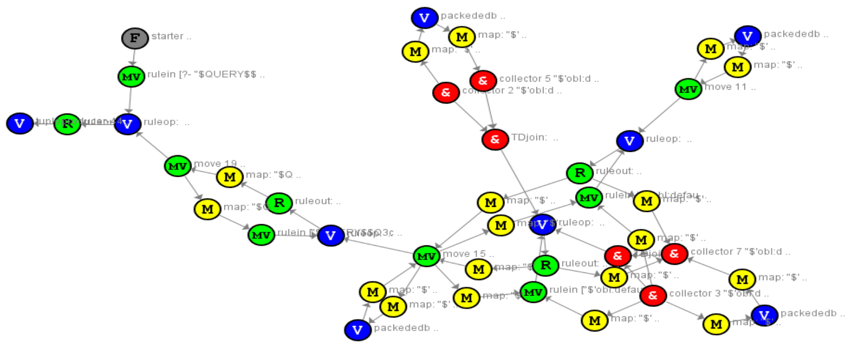


Figure 4. Operator net as a result of magic-sets rule compilation

We see that, on the one hand, the operator net becomes much more complex. On the other hand, in this case the reasoning time is drastically reduced because of the top down propagation which is expressed herein.

The whole query processing is multi user-capable. This means that multiple users can send queries to SemReasoner at the same time. All of those queries are processed in parallel.

Fig 5 shows a comparison of the answering times (smaller is better) for SemReasoner with ontobroker for some tests in the open rule benachmark [LFW09]. Ontobroker was the top-performer up to now in this benchmark.

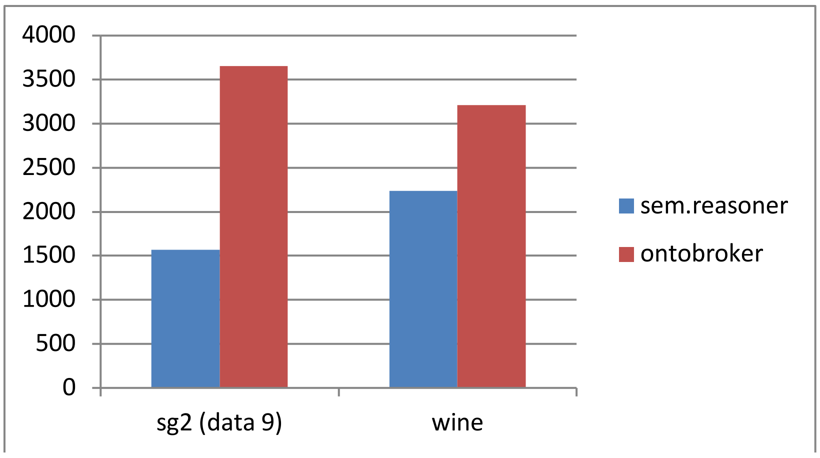


Figure 5. Answering times (smaller is better) in open rule benchmark comparing SemReasoner to ontobroker

3.2.3 Materializing Models

In the previous section we described how query evaluations take place inside SemReasoner. The same reasoning methods can also be used to materialize inferences. This means that rules are evaluated directly after loading and the results are stored in the internal ontology/ triple store which has the effect that, during query evaluation, these facts have to be accessed and retrieved only so that no evaluation takes place during query time. Clearly this may drastically improve response times but may also blow up the amount of data to be stored. SemReasoner even provides the functions for incrementally materializing models. This means that if a new triple is added to the model or deleted from the model, this materialization process only has to be repeated for a small subset of the model and not for the entire model.

3.2.4 The Storage Layer

All facts are separated from the rules and are stored in the extensional database (EDB). This EDB may either be configured to reside in the main memory or in its own graph store. It is clear that for performance reasons the first configuration is preferable.

The graph store is based on B+ trees [CLR01]. It has been tested for up to 22 billion triples. This layer is seamlessly integrated into our reasoning algorithms and thus has a dramatically increased performance compared to reasoners sitting on top of a relational database like e.g. ontobroker. This persistent layer ensures rapid loading, rollback, parallel snapshot transactions and backups. For instance loading a billion facts is done in roughly 3.5 hours on an ordinary PC.

3.2.5 Procedural Attachments (Built-ins)

Some aspects cannot be easily described using logic. For example, complex mathematical algorithms should be described in a procedural way instead. SemReasoner can easily be extended with such procedural algorithms. Within OO-logic, these procedural attachments are called built-ins. They may be used inside rules or queries using predicate logic literals. For example, all of the mathematical built-ins are internally given in the same way. If we want to multiply numbers we could use the multiply built-in:

```
?X[hasDollarPrice: ?Y] :- ?X:Product[hasEuroPrice: ?Z],_mult(?Z,1.2,?Y).
```

The last body literal is a built-in literal. The extension of such a built-in is not a given set of facts. Instead the extension is mathd by an algorithm.

Such a built-in may be easily developed by partners or customers. A well-documented API is available for such a built-in and the Java code must use this API. This Java code is compiled against the SemReasoner code and is the registered as new built-in in a class *BuiltinProvider*. This built-in is subsequently available within queries and rules.

SemReasoner supports action built-ins. Action built-ins occur in the head of a rule and perform some action like writing to a file, sending an e-mail etc. An example for such an action rule is

```
_send("juergen.angele@adesso.de","test mail",?Text,"192.168.178.10") :-
?X[hasTemperature:?T],?T > 35, ?Text := "it is hot".
```

This rule sends a message "it is hot" as soon as the temperature is higher than 35 °C. Especially in case of stream-reasoning (cf. 3.3) this is useful to trigger such actions.

3.3 SemReasoner as a stream-reasoning engine

The standard way to use SemReasoner is to fill it with ontologies, ontology instances and rules and to pose queries which are answered by the system. This is similar to a way a database is used. For complex event processing things are the other way round: the system is filled with ontologies, instances, rules, and queries. External events create new instances which are added to the set of instances and these add event s cause the stored queries to come up with new answers. SemReasoner does this in an incremental way. This means that adding a new instance does not mean that the full original query must be evaluated. Instead the query evaluation considers the previously evaluated partial results and hence also an incremental effort is necessary only to derive the new additional answers.

3.4 SemReasoner as a semantic integration engine

A very important application of SemReasoner is run-time ontology-based information integration. This means that information sources such as databases, Web services, linked open data (LOD) [HB11] sources, search engines, and so on, are attached to an ontology. The data in these data sources usually remain in these data sources and are not copied to the internal ontology store. At query-time, a query to the SemReasoner server is then translated into the query language for the attached data source, for example, SQL for relational data bases. These queries are then sent to the information sources, evaluated, and the results are integrated back into the reasoning process. As SemReasoner also provides rule materialization, i.e. executing all of the rules and storing the data in the internal ontology store, this also includes extraction of the data from the data sources and storing them locally in the own store. On the other hand, usually customers prefer query-time integration as they want to continue to maintain their own data stores.
Conceptually, information integration requires at least four different layers (cf. fig. 6):

- The bottom layer represents different data sources which contain or deliver the raw information which is semantically reinterpreted on an upper layer viz. ontologies.

- The second layer assigns a so called “data-source ontology” to each of the data sources. These “data-source ontologies” reflect only database or WSDL schemas of the data sources in terms of ontologies and can be created automatically. Hence, they are not real ontologies as they do not represent a shared conceptualization of a domain.
- The third layer represents the business ontology using terminology relevant to business users. This ontology is a real ontology, i.e. it describes the shared conceptualization of the domain at hand. It is a reinterpretation of the data described in the data-source ontologies and hence gives these data shared semantics. As a result, a mental effort is necessary for the reengineering of the data source contents, which cannot be done automatically.
- On a fourth layer, views to the business ontologies are defined. Basically, these views query the integration ontology for the information required. Exposed as Web services they can be consumed by portals, composite applications, business processes or other SOA components.

The elements of the different layers are connected via so-called mappings. The mappings between the data-sources and the source ontologies may be created automatically in an appropriate tool, the mappings between the ontologies are engineered manually and the views are manually defined queries. Mappings define how source structures are mapped to destination structures. Hence, mappings provide ways of restructuring information, renaming information or transforming values.
The arrangement of information on different layers and the conceptual representation in ontologies, plus the mediation between the different models via mappings bring the following advantages:

- The reengineered information in the business ontology is a value of its own. It represents a documentation of the data source contents. Representation as an ontology is a medium that can be easily discussed by non-IT experts. By aggregating data from multiple systems, this business ontology provides a single view of the relevant information in the user’s terminology. More than one business ontology enables different perspectives on the same information.
- It is easy to integrate a new data source with a new data schema into the system. It is sufficient to create a mapping between the corresponding source ontology and the integration ontology and hence does not require any programming know-how; pure modeling is sufficient.
- The mediation of information between data sources and applications, via ontologies, clearly separates both. In this way, changes in the data source schemas do not affect changes in the applications but only affect changes in the mediation layer, i.e. in the mappings.
- This conceptual structure strongly increases business agility. It makes it very easy to restructure information and hence react to changing requirements. It is not necessary to change either the data sources or the applications. It is only necessary to modify the business ontology and the mappings. Once again no programming skills are required, all the steps are made at model level. In this way it minimizes the impact of change, eases maintenance and allows for the rapid implementation of new strategies

- Ontologies have the powerful means to represent additional knowledge on an abstract level [FAM04]. So, for example, the business ontology may be extended by additional knowledge about the domain using rules. Hence, the business ontology is a reinterpretation of the data, plus a way of representing complex knowledge correlating this data. In this way, business rules are directly captured in the information model, they determine the optimal system access and bring every user to the same level of effectiveness and productivity.

Sometimes projects at customers integrate a fourth layer between the data-source layer and the business ontology. This is called the normalization layer. For example, different currencies may be normalized in this layer.

The logics such as renaming, restructuring, normalization, and filtering in mappings are represented by rules. For example, to map a database table to a class in the data source ontology we use a rule such as:

```
?O:Person[hasName: ?N] :-
       databaseAccess(database access info, database table name,
       C(column name, ?N)).
```

In a similar way, ontology elements of one layer are connected to the next layer via rules.

3.5 Summary

SemReasoner is a powerful graph store, semantic middleware and reasoning system that hosts ontologies and rule-based reasoning on top of these ontologies. The integrated rule engine is at the top of the open rule benchmark. Rule-based complex event processing is a central piece for industry 4.0 applications. The integration capabilities allow the dynamic integration of a variety of information sources which are queried during query-time.

4 JSON support in SemReasoner

JSON (JavaScript Object Notation) is a data format which has been developed for exchanging data between different applications. It is a readable text format. Though it has been developed for JavaScript it is independent of that programming language. In the meantime parsers exist for nearly every popular programming language.

SemReasoner integrates logic rules seamlessly with a triple store. SemReasoner allows to store and retrieve JSON objects and allows to use those data with logic rules and to query JSON objects and parts of them using OO-logic queries. SemReasoner supports JSON schemas for data validation purposes. SemReasoner extends the standardized use of JSON by some useful additional primitives. In this document these extensions are described. In addition SemReasoner supports GraphQL as query language.

4.1 Defined properties and types

There are a number of predefined properties for JSON object with special meanings. They base on JSON-LD defined properties and extend those:

`@id`	Uniquely identifies a JSON object. Must be given for every JSON object
`@type`	Classifies a JSON object. Must be given for every JSON object
`@subClassOf`	Refers to a super JSON schema. Properties from super schemas are inherited.
`@constraints`	Defines constraints in OO-logic for properties in a JSON schema (see OO-logic)
`Schema`	The predefined class for all JSON schemas.

4.2 Identifying and classifying a JSON object

JSON objects are uniquely identified by the value of the @id property. This means that every JSON object must have a different value for this property. If an @id property occurs in a nested object the properties of this nested object are skipped during loading. Only the reference given by the @id property value is retained. To insert this object it has to be inserted as a separate JSON object with the same value for the @id property in addition.

JSON objects are classified into different classes. The class an object belongs to is given by the value of the @type property. An object can be member of several classes.

An example for a simple JSON with an @id and a @type property and a nested JSON object is given as following:

```
{
        "@id":1,
        "@type": "Rectangle",
        "rectangle" : {
                "a" : -5,
                "b" : 5
        }
}
```

This JSON object has unique id 1 and is a member of class “Rectangle”.

4.3 JSON schemas

Schemas are used for *validating* JSON objects. The relation between a JSON schema and a JSON object is given by the class (@type) of the JSON object. The name of the class is the property value of the @id property of the schema. For instance given the following simple JSON schema:

```
{
        "title": "Person",
        "@id": "Person",
        "@type": Schema,
        "type":"object",

        "properties": {
                "name": {
                        "type": "string",
                }
        }
}
```

This schema is the schema for the class Person. JSON objects with value “Person” for their @type property are validated using this schema. For instance the following JSON object is validated using this schema:

```
{
        “@id”: “person0”,
        “@type”: “Person”,
        “name”: “Peter”
}
```

In SemReasoner schema objects are stored as normal JSON objects. As soon as a schema with id “Person” is stored in SemReasoner an object with type “Person” is added, this object is validated using this schema.

Schemas inherit their properties to sub-schemas. The relation of a sub-schema to its super schema is expressed using the property @super. This is illustrated by the following example:

```
{
        "@id": "Person",
        "@type": "Schema",
        "title": "Person",
        "type": "object",
        "properties": {
                "firstName": {"type": "string"},
                "lastName": {"type": "string"}
        },
        "required": ["firstName", "lastName"]
}
```

We now give an additional schema which inherits the properties of the person schema and defines an additional property “school”:

```
{
        "@id": "Scholar",
        "@type": "Schema",
        "@subClassOf": "Person",
        "title": "Scholar",
        "type": "object",
        "properties": {
                "school": {"type": "string"}
        }
}
```

This means that the schema “Scholar” now has three properties: “firstName”, “lastName”, and “school”.

To avoid multiple definitions of (partial) schemas as nested schemas we can define references to given schemas. For instance a length definition can be part of several schemas or even several length definitions can be part of a single schema:

```
{
        "@id": "Line",
        "@type" : "Schema",
        "type" : "object",
        "properties" : {
                "length" : {"type" : "number", "minimum" : 0}
        }
}
```

A rectangle consists of two lines a and b:

```
{
       "@id": "Rectangle",
       "type" : "object",
       "@type" : "Schema",

       "properties" : {
              "a" : {"$ref" : "#/Line" },
              "b" : {"$ref" : "#/Line" }
       }
}
```

The @constraints property allows to express local constraints for the properties of a JSON object. OO-logic constraints are used for that purpose:

```
{
       "@id": "Test",
       "@type": "Schema",
       "type": "object",
       "properties": {
              "start": {
                     "type": "integer"
              },
              "end": {
                     "type": "integer"
              }
},
       "@constraints":["!- ?X.start > ?X.end."]
}
```

The OO-logic constraint expresses that the value of the property "start" must not be greater than the value of the property "end". The variable ?X refers to the local JSON object.

The types in schemas are not restricted to the standard JSON types. There are arbitrary ones possible like "Person", "Car" indicating relationships to other classes.

```
{
       "@id": "Person",
       "@type": "Schema",
       "title": "Person",
       "type": "object",
       "properties": {
              "firstName": {"type": "string"},
              "lastName": {"type": "string"},
              “bestFriend”:{“type”: “Person”}
       },
       "required": ["firstName", "lastName"]
}
```

A concrete relation in a JSON object then refers to another JSON object with this property. So given two JSON objects describing two persons:

```
        {
                "@id": "person1",
                "@type": "Person",
                "firstName": "Hans",
                "lastName": "Schmid"
        }
        {
                "@id": "person2",
                "@type": "Person",
                "firstName": "Peter",
                "lastName": "Mueller"
                "bestFriend"": {"@id":"person1"}
        }
```

The bestFriend property in the second object refers to the first object.

4.4 GrapQL

JSON objects may be queried by GraphQL (https://graphql.org) queries. GraphQL is a query language to retrieve data from a data store. Usually GraphQL is used as an alternative to an ordinary REST call. It allows to retrieve the relevant information only instead of retrieving a sometimes extensive JSON object. As its syntax is closely related to JSON it fits very well to JSON. It allows simple queries to retrieve JSON object, but it is less powerful than OO-logic queries.
This GraphQL implementation supports GraphQL constructs in queries like aliases, unions, fragments, nested fragments, inline fragments, arguments, variables, default variables, directives (@include, @skip). It also supports introspection and mutations. In addition to standard GraphQL properties which begin with '@' may be used. In arguments OO-logic paths may be used.

Lets have a look at some simple examples for such queries:

```
{
        Person {
                firstName
                lastName
        }
}
```

The response is normal JSON for all persons with their properties firstName and lastName. So when we look at our examples above we get:

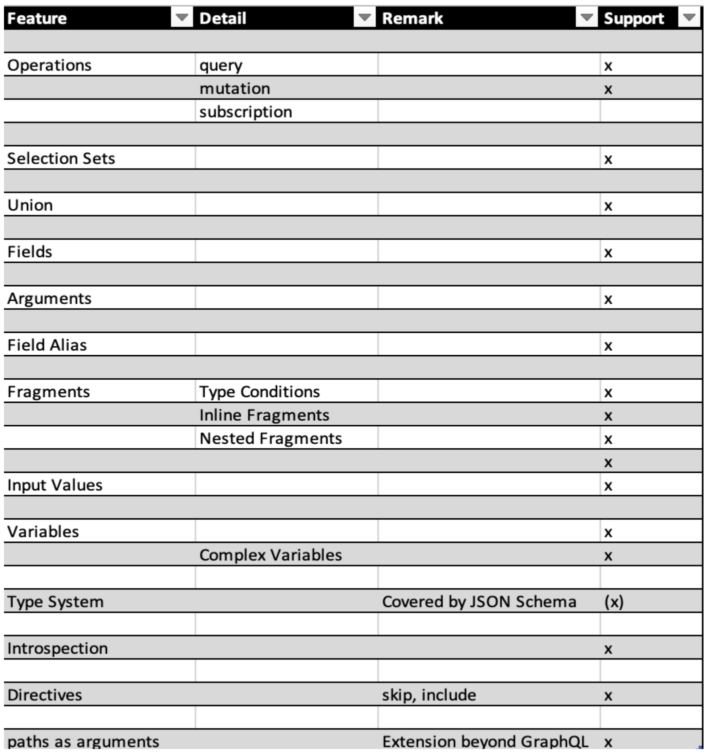
```
        {
                "firstName": "Hans",
                "lastName": "Schmid"
        },

        {
                "firstName": "Peter",
                "lastName": "Mueller"
        }
```

Each JSON object has the requested properties only. With the following query we are searching for all persons with first name “Hans” and get JSON object indicating the corresponding last names:

```
{
        Person(firstName:"Hans")   {
                lastName
        }
}
```

Table (fig. 6) shows the features from Graph QL supported by SemReasoner.



Feature	Detail	Remark	Support
Operations	query		x
	mutation		x
	subscription		
Selection Sets			x
Union			x
Fields			x
Arguments			x
Field Alias			x
Fragments	Type Conditions		x
	Inline Fragments		x
	Nested Fragments		x
			x
Input Values			x
Variables			x
	Complex Variables		x
Type System		Covered by JSON Schema	(x)
Introspection			x
Directives		skip, include	x
paths as arguments		Extension beyond GraphQL	x

Figure 6. Graph QL features supported

In Appendix C a set of GraphQL queries is shown.

5 Optimizing queries

Especially if there are huge data stored in SemReasoner optimizing queries and rules becomes crucial for practical use. For the evaluation performance the sequence of the conditions in a rule or query is very often crucial.
For instance given the following two queries:

```
?- q(?X,?Y),p(?U,?X),equals(?Y,2).
?- q(?X,?Y),equals(?Y,2),p(?U,?X).
```

For the first query the two predicates q and p at the beginning define a join operation over the variable?X. After the join operation all the instantiations for variable ?Y are checked for equality with 2. On the other hand the second query first checks variable ?Y for equality with 2, filters down to those tuples ?X,?Y and then joins the result with the tuples of predicate p. It is clear that in most cases the second query performs faster.

5.1 Selectivity and Weight

The selectivity of a predicate at argument *j* in a rule/query is given by:

selectivity(j) = number of different values (j) / number of tuples

So let us assume that in our upper example the second argument of predicate q is a key argument. This means that every value in argument 2 occurs only once for the relation of predicate q. In the same way the first argument of p is a key argument. So the selectivity for the second argument of *q* and the selectivity for the first argument of *p* is 1. This means that the filter 2 selects exactly one tuple. This single tuple is then joined with the relation of predicate *p*. So the general heuristics says: take the predicates with highest selections first.

5.2 Automatic body ordering

In the config file a heuristic for automatic body ordering of queries and rules can be switched on (Reasoning.BodyOrdering = on). This tries to reorder the rule bodies to generate a sequence of the rule bodies which takes the above mentioned heuristics into account. In relational databases this is called a query plan. As it is a heuristics this plan must not be necessarily optimal.

5.3 Manual body ordering

SemReasoner provides also means to adapt the body ordering manually. For this purpose a query directory can be given in the config file (Query.Directory=). As soon as such a query directory is given for every query a file is created in the query directory which stores the query and the rules. In this file the sequence of rules can be modified manually. The next time the same query (with same query name) is sent to SemReasoner this file is read and thus the modified rule body sequence is taken into account. The server caches the queries. So if a query file is manually modified, the server has to be restarted.

6 Data Safety and Recovery

SemReasoner provides different means to secure and recover data.

6.1 Backup/Restore

For the extensional database located in the extensional database directory a backup may be created. The deductive database interface offers method *backup* for that purpose. The backup process creates a zip file with all data needed to restore the full extensional database. The inverse operation *restore* restores the data to the original state in the extensional database directory. Before restoring data the extensional database directory should be deleted. Before executing the backup process please shut down the server.

6.2 Transaction Log

SemReasoner may be configured to maintain a transaction log. A transaction log contains for every predicate all additions and deletions. For every predicate *p|n* in the edb directory there exists a transaction log file *p_n.log*. These additions and deletions can be redone if the information for a predicate is corrupt. The replay of these actions must be done on top of a consistent state of the extensional database.

6.3 Detection of a corrupt extensional database state

An extensional database may get into a corrupt state when a writing transaction is processed while the server is killed for some reasons. Thus the writing transaction may not get completed which may result in an inconsistent state on disk. To detect this SemReasoner writes a shutdown file into the directory for the extensional database when it shuts down in a regular way. So if this shutdown file is not there this indicates that the server has not been shut down correctly. A writing transaction writes a file starting with “changeLog” at its beginning to the extensional database directory. The file content shows the predicates which are changed during this transaction. So if there is no shut down file these predicates are candidates for being corrupt.

6.4 Repairing a corrupt extensional database state

If a corrupt database state is detected start the following procedure. Look for possible corrupt predicates in the “changeLog” files. For these predicates *p* delete the index files (*p_n[i].iix, p_n[i].blk*). After that start the sever again and check whether the database state is correct now. If it is not correct the files *p_n.ix* for the predicates have to be restored. Restore the last backup file into a separate directory. Copy the predicate files *p_n.ix* from this separate directory to the extensional database directory. Replay the transaction log for these predicates (*p.log*) using the application *ReplayTransactionLog*.

7 Implementation Concepts

7.1 Transaction management

The reasoner supports snapshot transactions. This basically means:

- Multiple transactions can be executed at the same time.
- One user thread can have at most one open transaction at the same time.
- Facts and rules which are added during a transaction can only be seen by the thread running this transaction (the transaction is isolated).
- Queries executed within a transaction only see the database state of their transaction (committed database state plus transaction-local modifications).

If a user starts a transaction all read/write operations within the transaction operate only locally. All those read/write operations are store in main memory. When the transaction is committed the global database state is updated. At most one transaction is allowed to change the global database state at the same time.

For larger transactions which do not fit in main memory so called long transactions are provided. During a long transaction queries do not see local modifications. Only one long transaction can run at one time.

7.2 Lifecycle management

The DeductiveDatabase interface provides a shutdown() method which allows a clean shutdown of the database. The goal of this method is to shut down a database instance in a safe way (the persistent data structures should be fully written to disk after shutdown() was called).

This method works the following way:

- First the database will be set to “shutdown in progress” mode. This means no new transactions will be accepted.
- If there are running transactions the corresponding threads will be interrupted after some timeout.
- Then the system will wait a short time. After this time period the running transactions will be canceled.
- Then the shutdown waits for the write lock (it is not possible to shut down the database during a write operation).
- If the shutdown gets the write lock the database will be closed.

Due to the nature of the transaction implementation of the deductive database it is important to follow standard patterns:

- If you open a transaction, commit/rollback it in a finally block.

- If possible make sure the application code checks the interrupted flag.

7.3 Symbol table indices

We use Lucene as the backend for our symbol table indexes (text index and geo index). Both indexes can be configured individually.

7.3.1 Symbol table indices and transactions

The new symbols which have been created during a transaction are added to the main symbol table when transactionCommit() is called. After the symbol table has been committed the new symbols are also added to the symbol table indexes.

7.3.2 Extensional Database index

SemReasoner provides a text index for object properties. This index must be configured for this property in the configuration file (see 7.5.1):

```
?- ?X[name~:"xaver"].
```

This query delivers all objects with a property "name" which contains "xaver". We can also define a similarity measure between 0.0 and 1.0:

```
?- ?X[name~0.5:"xaver"].
```

7.3.3 Text index

The reasoner supports a full text index for all symbols. This allows to lookup the text index for specific strings with a simple query. The textSearch built-in is used in such rules and queries:

```
?- _textSearch("hello",?X).
```

This query results in all symbols containing the string "hello". The first argument could also be a complex search expression like "hello OR huhu". Look at the Lucene documentation for possible search expressions[3].

Query string queries also support wild card searches. E.g.

```
?- _textSearch("a*",?X).
```

[3] http://www.lucenetutorial.com/lucene-query-syntax.html

There is also a 3-argument built-in available which provides the score assigned by Lucene:

```
?- _textSearch("a*",?X, ?Score).
```

7.3.4 Geo index

The geo index indexes points. To query for such points geo shapes are available. E.g. we can query for all points within a circle. We currently support the following geo shapes:

- Point
- Circle
- Polygon

These datatypes follow the WKT syntax where possible (circle is not a WKT-supported data type). Below are some examples showing the syntax of the geo shape datatypes.

```
_'POINT(10 20)'.    // arguments are the x/y coordinates of the point
_'CIRCLE(10 20, 5)' // middle point and a radius
_'POLYGON(10 10, 20 20, 20 30, 10 10)' // sequence of points
```

A polygon consists of a sequence of points. The last point must equal to the first point (in order to close the polygon).

The geo index can be used to lookup explicit locations, but it is most useful if you want to e.g. retrieve a number of locations within a given area. The geoSearch/2 built-in is used to search for points:

```
?- _geoSearch(_'POLYGON ((0 0, 2 0, 2 2, 0 2, 0 0)',?X).
```

This query returns all points inside the mentioned polygon. If for instance for cities geo locations are given we could query for all cities inside such a polygon:

```
?- _geoSearch(_'POLYGON ((0 0, 2 0, 2 2, 0 2, 0 0)',?P),
?C:City, ?C[location: ?P].
```

7.3.4.1 Additional built-ins

For easier handling of geo shapes we added a number of built-ins for creating shapes.

```
?- _createPoint(1.1d, 2.2d, ?P).
?- _createPolygon(_'POINT (0.5 0.5)',                            (1)
             _'POLYGON ((0 0, 1 0, 1 1, 0 1, 0 0))', ?P).
?- _createPolygonFromPoints(_'POINT (10 10)',
             _'POINT (20 20)',_'POINT (20 30)', ?P).
?- _parsePoint('POINT (1.1 2.2)', ?P).
?- _pointX(_'POINT (1.1 2.2)', ?X).
?- _pointY(_'POINT (1.1 2.2)', ?Y).
?- _createCircle(_'POINT (0.1 0.2)', 0.3d, ?C).                  (2)
?- _createCircleWithDistanceUnit(_'POINT (0.1 0.2)', 1, 'm', ?C). (3)
```

The use of those built-ins should be obvious. (1) create a polygon out of a polygon and an additional point. (3) constructs a new circle from a given point and a given radius measured in a specified distance unit. In (2) you should be aware that 0.3 in SemReasoner is a float, while 0.3d is a double, what is needed here.

7.4 Built-ins

SemReasoner provides procedural attachments that provide computations which are not suitable for being described in logic. This includes comparisons, arithmetic or string operations, or access to external systems etc. These built-ins are also used like predicates.
As an example let's take string concatenation. For this purpose a built-in predicate *concat* with 3 arguments is provided. The first two arguments are the 2 strings to be concatenated, the last argument provides the result. So *concat* represents an infinite relation where the third string is the concatenation of the first and second string. Such a built-in predicate can be used in the usual way in a literal in a rule, e.g.:

```
?- name(?X), _concat(?X,"1",?Y).
```

This query concatenates a "1" to all elements of relation name. Relation name is a unary relation of strings.

There are six different kinds of built-ins in SemReasoner: filter built-ins, functional built-ins, relational built-ins, connector built-ins, aggregation built-ins and sensor built-ins. In the following we will describe these built-ins in more detail.

Appendix B show all available built-ins.

7.4.1 Filter Built-ins

Typical examples for filter built-ins are binary comparison operators like *less, greater, equals*. If such a filter built-in is used in a query or a rule all arguments of the built-in must be bound to values (either directly or by instantiation of variables). The filter built-in cannot math a new argument value. For instance in the following rule the two arguments of less are bound:

```
?- age1(?X), age2(?Y), _less(?X,?Y).
```

In the following query only one argument is bound. This query is NOT executable:

```
?- age(?X), _less(?X,?Y).
```

7.4.2 Functional Built-ins

Functional built-ins math one value out of a set of input values. Usually the output value is the last argument of the built-in. Arithmetical or string computations are typical examples for such built-ins. The *concat* built-in is an example for a functional built-in. Two strings are given (in the first two arguments) and the concatenated string is mathd as third argument. All the arguments which are input arguments have to be bound to values (either directly or by instantiation of variables).

The following query maths the result of 1 + 2 and assigns the result 3 to variable ?X:

```
?- _plus(1,2,?X).
```

Appendix B shows all functional built-ins in SemReasoner.

All built-ins from the Java packages Math and String are available. In all those built-ins the first argument is the object the built-in is applied to, then the Java arguments follow and finally the result is bound to the last argument. An example is given by the String built-in concat. In Java the signature of this built-in looks as following:

```
public String concat(String s)
```

If we apply it to a String name: name.concat(“hallo”) the result is the string name concatenated with “hallo”. So the object is name, the argument is “hallo” and the result is given by the concatenation. In OO-logic we use it as following:

```
?- ?X[name: ?Y], _concat(?Y,”hallo”,?Z).
```

7.4.3 Relational Built-ins

Relational built-ins math several values out of a set of input values. Computing the square root *sqrt* is an example for a relational built-in. A number has always two numbers as square roots. E.g *sqrt*(4) equals to {2,-2}.

The following query maths for each number in the number relation the square root which are always two results for each input number. The variable ?X binds to the numbers, the variable ?Y binds to the results.

```
?- number(?X), _sqrt(?X,?Y).
```

7.4.4 Connector Built-ins

While the previous built-ins get one set of input values and produce a set of output values, connector built-ins get a whole relation of input values and produce a whole relation as output values. Connector built-ins are used for accessing external data sources like databases, linked open data etc. To get a whole relation of input values allows them to optimize the access to the external data sources.

7.4.5 Aggregation Built-ins

All previous built-ins get input values and produce output values. This can happen several times during evaluation of a query. In contrast aggregations have to collect ALL input values and then produce output values out of that. For instance counting instances using the count aggregation must have all input values to count correctly and the produces the number of input values as one output value. So the reasoning process must assert that this built-in get all input values and after that the evaluation of the built-in is called.

The following query counts the elements of the unary relation p = {a,b,c} (p(a). p(b). p(c).). The answer is a binary relation {<a,3>, <b,3>, <c,3>}:

```
?- p(?X), _count(?X,?Y).
```

7.4.6 Sensor Built-ins

The evaluation of all previous built-ins is controlled by the reasoner. For instance the reasoner decides when all input values are available for an aggregation and then calls the evaluation of the aggregation built-in. Sensor built-ins deliver their values asynchroniously. They are triggered from outside the reasoner to produce new values and then send these values to the reasoner. They are for instance used for collecting sensor values or new values inserted into a database or similar.
These sensors turn the reasoning process upside down. The sensor values are fed into the system and these values trigger rules to fire and in turn dependent rules to fire until a query is fired. So in that case queries have to be registered in SemReasoner and we may get answers for these queries as soon as new sensor values have arrived. This is a kind of event driven process.

A well-known application area for complex event processing is the analysis of sensor data coming from complex machines like power plants, huge printing machines, gas turbines, jet turbines or similar. The analysis of this sensor data may detect defects or may predict certain maintenance actions or may be used to control the machine.

7.5 Configuring SemReasoner

7.5.1 Configuration File

Different options for SemReasoner can be configured via a property file. This property file looks as following:

```
############################################################
# extensional database settings
```

```
# different options for storing data: ram, persistent, mixed
EDB.Storage=ram
# directory of the persistent database
EDB.Directory=./sem/edb/
# temporary directory for intermediate results
EDB.TemporaryDirectory=./sem/temp/
# logging directory
EDB.LoggingDirectory=./sem/logging/
# backup directory
EDB.BackupDirectory=./sem/backup/
# import directory
EDB.ImportDirectory=./sem/import/
# which properties are indexed for text search
EDB.IndexedProperties = name,address,keywords
```

```
############################################################
# symbol table settings
############################################################
# turning text index off or on
SymbolTable.TextIndex = off
# turning geo index off or on
SymbolTable.GeoIndex = off

############################################################
# reasoning settings
############################################################
# resoning bottom-up or using magic sets
Reasoning.TopDown = on
# should rule bodies be sorted automatically
Reasoning.BodyOrdering = on
```

The storage property defines the way information is stored. An in-memory configuration is defined by the value *ram*. This means that all information in loaded into main memory, nothing is stored elsewhere. For this configuration memory must be large enough to hold all information. If this property has the value *persistent* all data are stored in B+ tree files in the edb directory. Rules are loaded into main memory. During reasoning nodes of the B+ trees are loaded and if main memory is exceeded those nodes are swapped out of main memory. Data are stored permanently in the edb directory which means, that if SemReasoner is finished and launched again the original data is available. The *mixed* mode is a mode in between both other modes. Data are sorted permanently in the edb directory in B+ tree files. During reasoning whole B+ trees are loaded into main memory and sometimes if main memory is exceeded whole B+ trees are swapped out. This mode should be used if there is enough memory available at least to hold all information used in one query execution. Otherwise the system begins to swap during the query execution which may result either in a very slow query execution or may lead to an out of memory exception.

7.5.2 File formats

SemReasoner is able to import different input files located in the import folder. They are differentiated by their file endings:

- .json: files containing a set of jsons embraced in square brackets
- .fct: files containing OO-logic facts
- .rls: files containing OO-logic rules
- .raw: files containing triples, every element of a triple in a different line. The arity is given by the file name, e.g. member_2.raw
- .mat: files containing names of rules which are materialized after loading
- .csv: csv files for Excel. There may be companion files for csv files which define data types for the colums and which change the name of the headers
- .ttl: RDF turtle
- .hdt: RDF hdt
- .ttls: RDF turtle star

- .n3: RDF n3
- .nq: RDF quads, nquads
- .nt: RDF nt,
- .rdf: RDF xml
- .trix: RDF Trix,
- .trig: RDF Trig
- .trdf: RDF Thrift
- Json files, raw files, and csv files may be zipped. They are recognized by the zip ending, e.g. member_2.raw.zip. Also a set of raw, csv, and json files can be in one zip file.

7.5.2.1 CSV import

For special imports so called companion files contain additional information which is necessary to import the data in a proper way. Up to now the *csv* importer uses such a companion file. The format of a *csv* file for import contains in its first line headers for the different columns. A small example is shown in the following:

```
title1;title2;title3
abc;4,5;05.06.20
bcde;5,2;04.12.59
```

Unfortunately the *csv* format does no longer show the data types of the different cells (the original Excel format contains this information). So this is an information which is still needed for an import. If such a companion file does not exist the *csv* import estimates the data types of the columns.

The format of a companion file for the above *csv* file looks like:

```
title1;title2;title3
String;Float;Calendar
NewTitle1;NewTitle2;NewTitle3
TestClass
UTF-8
```

The first line contains the original headers. The next line contains data types for the colums. *String, Float, Double, Calendar, Integer, Boolean* are available data types. If for a column no data type is given, the column is not imported. The next line may contain new names for the headers. Then the class the data are instances of is defined. Finally the encoding of the file can be defined.

A companion file is recognized by the importer by the same name and the ending *.cmp*. So in our case the name of the *csv* file may be *test.csv* and the companion file is named *test.cmp*.

8 Appendix A: ANTLR-4 description of OO-Syntax

```
grammar OOLogicGrammar;


facts_and_rules: (fact | reasoningRule | query | constraint)+ ;



partof: '|'number;



module: '@' (groundTerm | variable | EXTERNALDATATYPE);
groundTerm:  '-'? number | SIMPLE_STRING | EXTERNALDATATYPE | quotedString;
set: '{' arguments '}' | '{' '}';
enumeration : '/' arguments '/' | '/' '/';
term: groundTerm | variable | '_'? function | path | set | enumeration;
path: (groundTerm | variable | function) way;
edge: '.' groundTerm;
way: ((edge pathExpression?)+ | (subway pathExpression)) way?;
subway:  '('edge+')';
argument: expression;
arguments:  argument |  argument ',' arguments;
function: SIMPLE_STRING '(' arguments ')';
variable: '?' SIMPLE_STRING | '?' '_';
variables: variable | variable ',' variables;
builtinAtom: '_' SIMPLE_STRING '(' arguments ')' partof? | '_' SIMPLE_STRING
'('  ')';
predicateAtom: SIMPLE_STRING '(' arguments ')' partof? | SIMPLE_STRING '('
')';
atomWOModule: builtinAtom | predicateAtom | ooMemberAtom | ooSubClassAtom |
ooPropertyAtom;
atom: atomWOModule | atomWOModule module;
ooMemberAtom:  term ':' term ;
ooSubClassAtom:  term SCLASS term;
ooPropertyAtom: term (':'term)?'[' ooProperties ']';
ooProperties: ooProperty | ooProperty ',' ooProperties;
ooPropertyName : SIMPLE_STRING | '@'SIMPLE_STRING | REFERENCE | term | term
pathExpression |  SIMPLE_STRING'~'FLOAT | SIMPLE_STRING'~' | SIM-
PLE_STRING'~'variable;
ooProperty: ooPropertyName ':' ooPropertyValue;
pathExpression: '+' |  '{'INTEGER','(INTEGER|'*')'}';
ooPropertyValue:  expression | '[' arguments ']' | '/' arguments '/' ;
inputVariables: variables;
aggregationWithoutGrouping: SIMPLE_STRING '{' inputVariables '|' orList '}';
aggregationWithGrouping: SIMPLE_STRING '{' inputVariables '[' variables ']'
'|' orList '}';
aggregation: aggregationWithoutGrouping | aggregationWithGrouping;


fact: atom '.';

expression: expression '+' expression | expression '*' expression | expres-
sion '/' expression | expression '-' expression | expression '%' expression
| simpleExpression | '(' expression ')';
simpleExpression: term;

boolExpression : expression COMPAREOPERATOR expression (COMPAREOPERATOR
term)? | boolExpression BOOLEANOPERATOR boolExpression | simpleBoolExpres-
sion | '(' boolExpression ')';
simpleBoolExpression: term;
```

```
compareAtom: boolExpression;


assignmentAtom: variable ASSIGNMENT expression | variable ASSIGNMENT aggre-
gation;
ruleAtom: atom | compareAtom;
literal:  '!'? ruleAtom | assignmentAtom;
literals: literal | literal ',' literals ;
headatoms: atom | atom ',' headatoms;
orList: literals | literals semicolon orList;
reasoningRule: '?'? SIMPLE_STRING ':' headatoms implies orList '.' ;
query: queries orList '.' | SIMPLE_STRING ':' queries orList '.';
constraint: constraints literals '.';


implies: ':-';
queries: '?-';
constraints: '!-';

quotedString: SINGLE_Q | DOUBLE_Q;

SINGLE_Q: SINGLE_Q_FRAGMENT;
DOUBLE_Q: DQ ( ~('\\'|'"') | ('\\' '"') | ('\\' ESCAPED_CHAR))*? DQ;

fragment ESCAPED_CHAR: 'u' | 'e' | 't' | 'b' | 'n' | 'r' | 'f' | '\\' | '.'
;

fragment SQ: '\'';
fragment DQ: '"';
fragment SINGLE_Q_FRAGMENT: SQ ( ~('\\'|'\'') | ('\\' '\'') | ('\\' ES-
CAPED_CHAR))*? SQ ;


COMPAREOPERATOR: '<' | '>' | '=' |'!=' | '<=' | '>=';
BOOLEANOPERATOR: '|' | '&' | '!';



ASSIGNMENT: ':=';
semicolon:';';
fragment NUMBER: [0-9]+;
number: INTEGER | LONG | FLOAT | DOUBLE;
INTEGER: NUMBER;
LONG:  NUMBER [l|L];
FLOAT: NUMBER '.' NUMBER;
DOUBLE: FLOAT [d|D] | FLOAT 'E' '-'? NUMBER [d|D]?;
SCLASS: '::';
EXTERNALDATATYPE: '_' SINGLE_Q_FRAGMENT;
SIMPLE_STRING: [A-Za-z][A-Za-z0-9_]* ;
REFERENCE: '$'[A-Za-z][A-Za-z0-9_]* ;

SINGLELINE_COMMENT
 : '//' ~('\r' | '\n')* -> skip
 ;

COMMENT: '/*' .*? '*/' -> skip;

WS
 : (' '|'\r'|'\t'|'\u000C'|'\n')+  -> skip;
```

9 Appendix B: built-ins

In the pdf file “builtins.pdf” all available built-ins may be found.

The first column (Name) shows the name and the arity (number of arguments) of the built-in. If X is given for the arity, the arity can be arbitrary large. The second column (Function) describes the function of the built-in. The third column (Arguments) provides a short description of the arguments. Finally the fourth column shows the possible bindings of the arguments.

A lot of built-ins are automatically generated from the Java package the built-in comes from. If it comes from a Java package the arguments are similar to the arguments of the appropriate Java method: the first argument is the object, the next arguments are the arguments of the Java method, and the last argument is the result. In this case the function description refers to the appropriate Java documentation

For the aggregation built-ins it is referred to 7.4.5. for the geo built-ins it is referred to 7.3.4. For the textSearch built-in it is referred to 7.3.3. The functionality of the comparison built-ins and the logical built-ins should be obvious.

In the following we will briefly describe the functionality of some more interesting built-ins.
builtins|3
Provides a list of all built-ins together with a description and a description of the arguments.

equals|X
All arguments are equal.

different|X
All arguments are different.

oneOf|X
First argument is equal to one of the other arguments.

startsWithOneOf|X
First argument starts with one of the other arguments.

minus|3
Mathematical minus operation. One of the three arguments can be unbound. This unbound argument is mathd.

plus|3

Mathematical plus operation. One of the three arguments can be unbound. This unbound argument is mathd.

day|2
Get the day of a calendar date.
```
        ?- _day(_'2014-02-10T11:15:00';?D).
```
results in
```
        ?D = 10
```

week|2
Get the week of a calendar date.
```
        ?- _week(_'2014-02-10T11:15:00.000+01:00',?W).
```
results in
```
        ?W = 7
```

month|2
Get the month of a calendar date.
```
        ?- _month(_'2014-02-10T11:15:00.000+01:00',?M).
```
results in
```
        ?M = 2
```

year|2
Get the year of a calendar date.
```
        ?- _year(_'2014-02-10T11:15:00.000+01:00',?Y).
```
results in
```
        ?Y = 2014
```

now|1
Get the calendar date for now.

levenshtein|3
Get the levensthein distance for two strings.
```
        ?- _levenshtein(\"hallo\",\"halo\",?X).
```
results in
```
        ?X = 1.0.
```

string2Number
Create a string out of a number or create a number out of a string.
```
        _string2Number("5",?X).    // binds ?X to 5
        _string2Number(?X,5).      // binds ?X to "5"
```

toFile|X
Write second to last argument to file. File path is given in the first argument.

sendMail|4
Send an e-mail. First argument is the e-mail address, second is the sender, third is the subject and last is the text.

print|X
Print all arguments to console. Often used for debugging rules.

isType|2

is first argument of data type (Double,Float,Long,Integer,Boolean,String,Calendar,Duration)

```
        _isType(5,String).
```

Evaluates to false because 5 is an integer and not a string.

toType|3
transforms first argument to data type (Double,Float,Long,Integer,Boolean,String,Calendar,Duration)

```
        _toType(5,String,?Y).
```

Transforms integer 5 to string “5”

toCSV|X
prints to a csv file.

```
        _toCSV('./test.txt',';','title1','title2',?X,?Y) :- p(?X,?Y).
```

First argument is the file path, second the separator, the next arguments are the titles and the remaining are the variables for the data for each line. The number of titles and variables must be the same.

seek|5
Searches for values of a property. In addition to 7.3.2 it provides also a score as result.

```
        _seek(“xaver”,0.5,name,?Object,?Score).?
```

In all values of property *name* it is searched for a value “xaver” with a similarity of *0.5*. Result is the object identifier and a score.

randomInt|4
Creates a random number between lower limit and upper limit for every identifier:

```
        _randomInt(?Id,?Lower,?Upper,?Random).
```

randomBool|2
Creates randomly true or false for every identifier:

```
        _randomBool(?Id,?Random).
```

randomDuration|2
Creates a random period smaller that given period for every identifier:

```
        _randomDuration(?Id,?Period,?Random).
```

randomEnum|X
Selects one of the arguments randomly for every identifier:

```
        _randomEnum(?Id,?Argument1,..,?ArgumentN,?Random).
```

E.g. one of *Berlin*,*Tokyo*, *Heidelberg* is randomly selected for each identifier

```
        _randomEnum(?Id,Berlin,TokyoHeidelberg,?Random).
```

jsonId2String|2
Creates a json id out of a string and vice versa

subJsonId|2
Detects whether a json id is a json id of a nested json object

toString|2
Creates a string representation of an object

10 Appendix C: GraphQL Queries

In the following a small collection of GraphQL queries is shown.

query with argument
```
query {
	Person (firstName: "Hans"){
		firstName
		lastName
	}
}
```

// query with path as argument
```
query {
	Person (address.city : "Stuttgart"){
		firstName
		lastName
	}
}
```

alias
```
query {
	me:Person {
		givenName:firstName
		familyName:lastName
	}
}
```

fragment in query
```
fragment personFields on Person {
	firstName
	lastName
}
query {
	Person (@id: "person1") {
		...personFields
	}
}
```

sub object with argument
```
query {
	Person{
		firstName
		lastName
		address (city : "Stuttgart") {
			city
		}
	}
}
```

variable
```
query ($city:Address) {
	Person {
		firstName
		lastName
		address(city : $city) {
```

```
                        city
                }
        }
}
```

default variable

```
query ($city:Address = Stuttgart) {
        Person {
                firstName
                lastName
                address(city: $city) {
                        city
                }
        }
}
```

directive @include

```
query (
        $city:Address = Stuttgart, $withAddress:Boolean = false) {
                Person {
                firstName
                lastName
                address  (city : $city)  @include(if:$withAddress) {
                        city
                }
        }
}
```

nested fragments

```
fragment addressFields on Person {
        city
}
fragment personFields on Person {
        firstName
        ...addressFields
}
query {
        Person {
                ...personFields
        }
}
```

inline fragment

```
query {
        Person {
                @type
                ... on Scholar {
                        firstName
                }
        }
}
```

introspection

```
query IntrospectionQuery {
        __schema {
                types {
                        name
                        fields {
                                name
                                type_ {
                                        ofType
                                }
                        }
                }
```

```
      }
}
```

mutation

```
mutation {
      Person(@id:"leon", firstName : "Leon", lastName: "Angele"){
            firstName
            lastName
      }
}
```

References

[AES07] J. Angele, M. Erdmann, H.P. Schnurr and D. Wenke: Ontology Based Knowledge Management in Automotive Engineering Scenarios. In: Proceedings of ESTC 2007, 1st European Semantic Technology Conference, 31.05.-01.06.2007, Vienna, Austria

[AHV95] S. Abiteboul, R. Hull, V. Vianu: Foundations of Databases. Addison-Wesley 1995.

[ALUW93] Serge Abiteboul, Georg Lausen, Heinz Uphoff, Emmanuel Waller: Methods and Rules. *Proceedings SIGMOD Conference 1993*, pages 32-41

[ASS00] J. Angele, H.-P. Schnurr, Steffen Staab, Rudi Studer. The Times They Are A-Changin' - The Corporate History Analyzer. In: *D. Mahling & U. Reimer.* In *Proceedings of the Third International Conference on Practical Aspects of Knowledge Management*. Basel, Switzerland, October 30-31, 2000.

[AS05] J. Angele, Hans-Peter Schnurr: Do not use this gear with a switching lever! Automotive industry. In: Proceedings of the industrial track of the Fourth International Semantic Web Conference (ISWC2005), Galway, Ireland, November 6-10, 2005.

[AMO06] J. Angele, E. Mönch, H. Oppermann, H. Rudat, H. Schnurr: Customer Service accelerated by Semantics. In: Proceedings of the industrial track of the Fifth International Semantic Web Conference (ISWC2006), Athens, USA, November 7-9, 2005.

[BH07] J. de Bruijn and S. Heymans. RDF and logic: Reasoning and extension. In *Proceedings of the 6th International Workshop on Web Semantics (WebS 2007),* Regensburg, Germany. IEEE Mathr Society Press 2007.

[Che76] P.P. Chen: The entity relationship model. Toward a unified view of data. In *ACM Transactions on Database Systems*, Vol. 1, 1976, 9-36.

[BCM03] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, P. Patel-Schneider (eds.). The Description Logic Handbook, Cambridge, 2003

[BR91] C. Beeri and R. Ramakrishnan. On the Power of Magic. Journal of Logic Programming, 10, April 1991, pp. 255—300.

[DBS98] S. Decker, D. Brickley, J. Saarela und J. Angele: A Query and Inference Service for RDF. In *Proceedings of the W3C Query Language Workshop (QL-98)*, Boston, MA, 3.-4. Dezember, 1998.

[DEFS99] S. Decker, M. Erdmann, D. Fensel, and R. Studer. OntoBroker™: Ontology based access to distributed and semi-structured information. In R. Meersman et al., editor, *Database Semantics: Semantic Issues in Multimedia Systems*. Kluwer Academic, 1999.

[FLU94] J. Frohn, G. Lausen, H. Uphoff: Access to Objects by Path Expressions and Rules. In **Proceedings VLDB 1994**: pages 273-284

[FHK97] J. Frohn, R. Himmeröder, P.-T. Kandzia, C. Schlepphorst: How to Write F-logic Programs in FLORID. Available from fpt://ftp.informatik.uni-freiburg.de/pub/florid/tutorial.ps.gz, 1997.

[GPQ97] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J.D. Ullman, V. Vassalos, Jennifer Widom: The TSIMMIS Approach to Mediation: Data Models and Languages. JIIS 8(2): 117-132 (1997)

[GRS91] A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, July 1991.

[K05] M. Kifer. “Nonmonotonic Reasoning in FLORA-2.” Proceedings of Logic Programming and Nonmonotonic Reasoning. Lecture Notes in Mathr Science 3662, pages 1-12, Springer Verlag, 2005.

[KBL05] M. Kifer, A. Bernstein and P.M. Lewis. “Database Systems: An Application Oriented Approach,” 2nd edition. Addison Wesley, 2005.

[KBBF05] M. Kifer, J. de Bruijn, H. Boley and D. Fensel. “A Realistic Architecture for the Semantic Web.” Proceedings of Rules and Rule Markup Languages for the Semantic Web, Lecture Notes in Mathr Science 3791, pages 17-29, Springer Verlag, 2005.

[KL86] M. Kifer, E.L. Lozinskii, “A framework for an efficient implementation of deductive databases,” Proc. of the 6-th Advanced Database Symposium, Aug. 1986, Tokyo Japan, pp. 109--116.

[KLW95] M. Kifer, G. Lausen, and J.Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42:741–843, 1995.

[L99] M. Liu: Deductive Database Languages: Problems and Solutions. ACM Computing Surveys 31(1): 27-62 (1999)

[LHL98] B. Ludäscher, R. Himmeröder, G. Lausen, W. May, Christian Schlepphorst. Managing Semistructured Data with FLORID: A Deductive Object-Oriented Perspective. Information Systems 23(8): 589-613 (1998)

[M01] Wolfgang May: A Rule-Based Querying and Updating Language for XML. In *Proceedings of DBPL 2001*, LNCS 2397, pages 165-181

[MK01] W. May, P.-T. Kandzia. Nonmonotonic Inheritance in Object-Oriented Deductive Database Languages, *Journal of Logic and Computation 11(4)*, 2001.

[OWL02] M. Dean ,D. Connolly ,F. van Harmelen ,J. Hendler ,I. Horrocks , D. L. McGuinness, P.F. Patel-Schneider, L.A. Stein. *OWL Web Ontology Language 1.0 Reference*. WWW Consortium, November 2002.

[RES06] R. Elmasri, S.B. Navathe: Fundamentals of Database Systems, 5th Edition. Addison Wesley, 2006.

[SD02] M. Sintek, S. Decker, “TRIPLE – A Query, Inference, and Transformation Language for the Semantic Web. International Semantic Web Conference (ISWC), June 2002.

[SEM01] S. Staab, M. Erdmann, A. Mädche, S. Decker. An extensible approach for modeling ontologies in RDF(S). In *Knowledge Media in Healthcare: Opportunities and Challenges*. Rolf Grütter (ed.). Idea Group Publishing, Hershey USA / London, UK. December 2001.

[SSA02] Y. Sure, S. Staab, J. Angele. OntoEdit: Guiding Ontology Development by Methodology and Inferencing. In: R. Meersman, Z. Tari et al. (eds.). *Proceedings of the Confederated International Conferences CoopIS, DOA and ODBASE 2002*, October 28th - November 1st, 2002, University of California, Irvine, USA, Springer, LNCS 2519 , pages 1205-1222.

[StM01] Staab, S. and Maedche, A.: Knowledge Portals - Ontologies at Work. In: AI Magazine, 21(2), Summer 2001.

[YK02] G. Yang, M. Kifer: On the Semantics of Anonymous Identity and Reification. Proceedings CoopIS/DOA/ODBASE 2002. pages 1047-1066.

[YK06] G. Yang and M. Kifer, Inheritance in Rule-Based Frame Systems: Semantics and Inference," Journal on Data Semantics, 2006.

[YKZ03] G. Yang, M. Kifer, C. Zhao. FLORA-2: A Rule-Based Knowledge Representation and Inference Infrastructure for the Semantic Web. Proceedings of International Conference on Ontologies, Databases and Applications of Semantics (ODBASE-2003), Springer Verlag, LNCS 2888, pages 671-688.

[AEW07] J. Angele, M. Erdmann, D. Wenke: Ontology-based Knowledge Management in Automotive Engineering Scenarios. In (Martin Hepp, Pieter De Leenheer, Aldo de Moor, York Sure. Eds.): Ontology Management: Semantic Web, Semantic Web Services, and Business Applications, ISBN 978-0-387-69899-1, Springer, 2007.

[AG06] J. Angele, M. Gesmann: Semantic Information Integration with Software AGs Information Integrator. In (Thomas Eiter; Knowledge Web.; et al Eds.): Proceedings of the 2nd RuleML conference, Las Alamitos, Calif. : IEEE Mathr Society, 2006.

[AKL09] J. Angele, M. Kifer, G. Lausen: Ontologies in F-Logic. In (S. Staab, R. Studer eds.): Handbook on Ontologies. International Handbooks on Information Systems, Springer Verlag, Second Edition, 2009, 45 –

[AMO05] J. Angele, E. Mönch, H. Oppermann, H. Rudat, H.-P. Schnurr: Customer Service accelerated by Semantics. In: Proceedings of the industrial track of the Fifth International Semantic Web Conference (ISWC2006), Athens, USA, November 7-9, 2005.

[BMS86]F. Bancilhon, D. Maier, Y. Sagiv, J. Ullman: Magic sets and other strange ways to implement logic programs. In Proceedings Fifth ACM Symposium on Principles of Database Systems, 1986, 1-15.

[BR91] Beeri C., Ramaksishnan R. On the power of magic. The Journal of Logic Programming, 10:255-300, January 1991.

[CLR01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Second Edition. MITPress and McGraw-Hill, 2001. ISBN 0-262-03293-7. Chapter 18:B-Trees, 434–454.

[BLK09] Bizer, Christian; Lehmann, Jens; Kobilarov, Georgi; Auer, Soren; Becker, Christian; Cyganiak, Richard; Hellmann, Sebastian (September 2009). DBPedia – A crystallization point for the Web of Data, Web Semantics: Science, Services and Agents on the World Wide Web 7 (3): 154–165. ISSN 1570-8268

[FAM04] Friedland N., Allen P., Matthews G., Witbrock M., Baxter D., Curtis J., Shepard B., Miraglia P., Angele J., Staab S., Moench E., Oppermann H., Wenke D., Israel D., Chaudhri V., Porter B., Barker K., Fan J., Chaw

S., Yeh P., Tecuci D., Clark P.. Project Halo: Towards a Digital Aristotle. In AI Magazine, vol. 25, no. 4, winter 2004, 29-4.

[SET09] Segaran T., Evans C., Taylor J.. Programming the Semantic Web, O'Reilly, 2009, ISBN:0596153813 9780596153816

[GRS91] A. Van Gelder, K.A. Ross and J.S. Schlipf. The Well-Founded Semantics for General Logic Programs. Journal of the ACM 38(3) pp. 620—650, 1991.

[KLW95] Kifer M., Lausen, and Wu (1995). Logical foundations of object-oriented and framebased languages. Journal of the ACM, 42; (1995), 741–843.

[KL86] Kifer and Lozinskii (1986). A framework for an efficient implementation of deductive databases. In Proceedings of the 6th Advanced Database Symposium, Tokyo, August (1986), 109–116.

[L87] J.W. Lloyd: Foundations of Logic Programming, 2nd Editon. Springer-Verlag, Berlin, 1987.

[LT84] J. W. Lloyd and R. W. Topor. Making Prolog more expressive. Journal of Logic Programming, 1(3):225–240, 1984.

[HB11] Tom Heath and Christian Bizer (2011) Linked Data: Evolving the Web into a Global Data Space (1st edition). Synthesis Lectures on the Semantic Web: Theory and Technology, 1:1, 1-136. Morgan & Claypool.

[HKR09] Hitzler P., Krötzsch M., Rudolph S. Foundations of Semantic Web Technologies, Chapman & Hall, 2009, ISBN: 9781420090505

[RB01] Rahm E., Bernstein P. (2001). A survey of approaches to automatic schema matching, VLDB Journal 10(4):334-350, 2001

[U89]J. D. Ullman: Principles of Database and Knowledge-Base Systems, vol II. Mathr Sciences Press, Rockville, Maryland, 1989.

[V86]L. Vieille: Recursive Axioms in Deductive Databases: The Query/Subquery Approach. In Proceedingsof First International Conference on Expert Database Systems, Charleston, 1986, 179-194.

[LFW09] Senlin Liang, Paul Fodor, Hui Wan, Michael Kifer: OpenRuleBench: An Analysis of the Performance of Rule Engines. In Juan Quemada, Gonzalo León, Yoëlle S. Maarek, Wolfgang Nejdl (Eds.): Proceedings of the 18th International Conference on World Wide Web, WWW 2009, Madrid, Spain, April 20-24, 2009. ACM 2009, ISBN 978-1-60558-487-4.

[FHL98] J. Frohn, R. Himmer, G. Lausen, W. May, C. Schlepphorst: Managing Semistructured Data with FLORID: A Deductive Object-Oriented Perspective. In Information Systems, vol. 23, no. 8, 1998, p. 589-613.

[BBG05]D. Berardi, H. Boley, B. Grosof, M. Gruninger, R. Hull, M. Kifer, D. Martin, S. McIlraith, J. Su, S. Tabet: SWSL: Semantic Web Services Language. Semantic Web Services Initiative, April 2005.

[YKZ03]G. Yang, M. Kifer, C. Zhao: FLORA-2: A Rule-Based Knowledge Representation and Inference Infrastructure for the Semantic Web. In International Conference on Ontologies, Databases and Applications of Semantics (ODBASE-2003)", Springer LNCS, vol. 2888, p. 671-688.