An Active Domain Node Architecture for the Semantic Web

Dissertation

zur Erlangung des Doktorgrades
der Mathematisch-Naturwissenschaftlichen Fakultäten
der Georg-August-Universität zu Göttingen

vorgelegt von

Franz Schenk
aus Tegernsee

Göttingen
im Oktober 2008

Referent:	Prof. Dr. rer. nat. Wolfgang May, Universität Göttingen.
Korreferentin:	Prof. Dr. rer. nat. Nicola Henze, Universität Hannover.

Tag der mündlichen Prüfung: 21. November 2008

Abstract

The scope of this work is knowledge management in the Semantic Web. Its contribution is the design, development, and characterisation of an application node architecture for the Semantic Web, called SWAN. The core of the SWAN architecture consists of an OWL knowledge base which is supplemented by a hybrid reasoning mechanism. Hybrid reasoning in SWAN combines F-Logic and Description Logic reasoning. Update operations for the manipulation of the knowledge base are provided with a well-defined semantics. Intensional updates are possible and allow to define updates with respect to implicit knowledge. A unique trigger mechanism is used for the completion of intensional updates, but also for the maintenance of knowledge base integrity. By its ability to process abstract action definitions, the application node can be integrated directly into the event-driven Semantic Web architecture MARS. A protoype of this architecture has been implemented, which shows the flexibility and applicability of its concepts.

Keywords:
Semantic Web, Event-Driven Architecture, OWL, Active Knowledge Base, Intensional Updates, Hybrid Reasoning, Description Logic, F-Logic.

Contents

Part I

Introduction and Conceptual Background

Chapter 1

Introduction

Overview

The *World Wide Web* radically changed the ways that mankind can deal with information. Knowledge of any kind is easily accessible in vast abundance. On the one hand, the availability of data offers new possibilities. On the other hand, it poses severe problems: How to deal with all the data? How to find relevant data? How can related information be combined? Data integration and knowledge management are areas of research that started long before the beginning of the *World Wide Web* but become more and more important now with the growing amount of available data. How can different sources of information be integrated? Is it possible to define the meaning of knowledge such that also a computer can use that knowledge? How can one find information that is supposed to be somewhere? These are only some of the questions that we are challenged with right now.

Information is stored everywhere around the world, often it is globally accessible by web services. Such a service infrastructure normally has to be invoked in a specific manner, which is defined very strictly for each purpose. Often it is not known whether and where there exists an appropriate service for a certain task. In such a situation, one has to locate a service and find a way how to communicate with the newfound service. With the Web Service Description Language (WSDL), for example, there is a standardisation for the definition of some of the properties of a web service (interfaces, access mechanisms, and the like). Although these descriptions are machine-readable, they offer no way for the specification of the meaning of a service. Hence, the service calls have to be designed specifically for every service type. This is diametrically opposed to the expectation of the user who wants to be able to generically call a web service.

A simpler approach in this respect is to send an abstract description of what should be done to a broker. Everything else should be taken care of without further interaction. This is, more or less, what the Modular Active Rules for the Semantic Web framework (MARS) is about. It offers a rule-based architecture that is driven by events. The behaviour of the application domain is specified by rules following the well-known ECA paradigm: on the detection of an event E, given that a condition C is fulfilled, an action A will be executed. This is different from usual web services as the calling of the service does not lead to the execution of procedural code but to rule evaluation. The set of rules is not fixed, rather it can be altered or extended easily making the term *generic webservice* more appropriate.

ECA rules are triggered by events and result in actions, both of which are notions of the application domain and defined in a *domain ontology*. The domain ontology defines the vocabulary of an application domain. This is, with regard to the static aspects of a domain, comparable to the schema of a relational database, only that it can be augmented with rules (or axioms) for the derivation of further knowledge from what is known. Moreover, a domain ontology allows for the specification of dynamic aspects, for example, how actions and events can be correlated and what their pre- and postconditions are. It is important to stress that events and actions should not be confused with messages. They are *abstract* descriptions of what happened or should be done, using the notions of the domain ontology. For example, instead of booking a flight by explicitly contacting a web service http://travel.com/flightbooking?flightno=LH458&date=20081010 or giving an explicit update command to an SQL server (INSERT INTO flighttable VALUES('John Doe', 'LH458',20081010)) the following abstract action, given as an XML fragment, is sent to a domain broker: <book-flight passenger='John Doe' flightNo='LH458' date='20081010'/>. The broker ensures that an appropriate service for the action or event is found. The abstract action does not contain any references to procedural aspects at all, in fact it is completely left to the recipient how to react to the event or how to implement an abstract action. The advantage of abstract action definitions and event definitions is that ECA rules can be specified independently from the actual programming languages that are used in the domain.

With regard to abstract actions, this means that the recipient will not receive explicit specifications using an update language (for example SQL). Rather the application node applies further rules in order to realise actions (which are notions of the domain vocabulary) as explicit updates (expressed in the specific language of the underlying storage facilities).

It is one of the contributions of this thesis to show how a knowledge base can be integrated into an application domain. This integration is not a tight binding but realised by a mapping of abstract actions of the application domain to knowledge base updates. Hereby, the integration depends solely on the semantics of actions and not on their procedural aspects. Moreover, a knowledge base design is presented, which incorporates ideas from active databases. This knowledge base architecture is called Semantic Web Application Node (Swan). The behaviour of the application domain node is also defined and driven by rules that follow the ECA paradigm. There are two different types of rules: Firstly, there are ACA rules, which map abstract actions to knowledge base updates. Secondly, there are simple ECA rules in the form of knowledge base triggers. Both of these rules can be characterised formally such that it is possible to reason about the consequences of actions in the application domain.

Internally, the Swan architecture uses RDF as a data model and OWL DL for the description of the domain ontology. While OWL DL is a reasonably expressive language for describing concepts, it is quite weak when it comes to the modelling of properties. Furthermore, reasoning with datatypes is not yet fully supported by existing reasoning engines and non-monotonic features like defaults are not available. As a solution to these problems, the SWAN architecture internally uses a hybrid model which combines ontology-based reasoning with rule-based reasoning. The rule-based component of the hybrid reasoning process makes use of F-Logic as an additional deductive formalism.

Structure

This work is organised in three parts. Part I comprises Chapters 2 to 5 and gives an introduction and all necessary conceptual background for this thesis. Chapter 2 gives the necessary formal background and notational conventions for subsequent chapters. Chapter 3 offers a comprehensive description of the development in knowledge representation from the ancient times up to now. Besides some background to the relevant philosophical and logical questions an introduction to the conventional languages for knowledge representation is provided. This chapter is intended mostly for those readers who are not familiar with the concepts that are common in *Semantic Web* research. An abridgement of the MARS framework is presented in Chapter 4. This framework is the environment into which the SWAN (Semantic Web Application Node) architecture is integrated. The chapter includes a description of the concepts and a presentation of the infrastructural components that are used in the MARS architecture. Chapter 5 shows how domain ontologies are used in the specification of static and dynamic aspects of an application domain.

Part II contains the main contribution of this thesis and comprises the Chapters 6 to 10. Chapter 6 starts with an overview about the SWAN architecture and gives details about the update mechanism of the knowledge base. This is followed by a description of the concept and implementation of knowledge base triggers in Chapter 7. The ACA rule mapping component is explained in Chapter 8. By this component it is possible to combine the architectures of MARS and SWAN. Next, the hybrid reasoning component of SWAN is presented in Chapter 9. Both kinds of rules in SWAN, ACA rules and triggers, are logically characterised in Chapter 10, showing how the *behaviour* of the SWAN node can be specified.

Part III completes this work with Chapters 11 to 13. Chapter 11 gives a description of *travel booking* as a motivating example scenario. It demonstrates how the application domain behaviour can be specified by a set of rules and how actions and events interact hereby. In Chapter 12 a discussion, a look on related work, and an outlook to further work can be found. Finally, in Chapter 13 some conclusions are presented.

Chapter 2

Formal Preliminaries

This section gives an introduction to the logical formalisms that are used throughout this work. The semantics of first order logic is given in Section 2.1. This is followed by the formal definition of the semantics of the Description Logic $\mathcal{SHOIN}(\mathcal{D})$ in Section 2.2. This fragment corresponds to the fragment OWL DL of the Web Ontology Language which is used in this work. Section 2.4 contains the formal definition of F-Logic. Finally, classical default theory, default inheritance and the application of defaults in an OWL knowledge base are formally introduced in Section 2.5.

2.1 First-Order Logic

Each first-order language contains a set of distinguished symbols, consisting of parentheses "(" and ")", constants true, false representing the truth values, boolean connectives $\neg$, $\wedge$, $\vee$, $\rightarrow$, quantifiers $\forall$, $\exists$, and an infinite set of variables $x, y,\ x_1, x_2,\ \ldots$ which is denoted by Var. For first-order logic *with equality*, additionally the equality symbol "=" is part of the language.

An individual first-order language is then given by its *signature* Σ. Σ is partitioned into a functional part Σ_F of function symbols and a relational part Σ_R of predicate symbols, each of the symbols with a given arity which is denoted by $\mathsf{ord}(f)$ and $\mathsf{ord}(p)$, respectively. 0-ary functions are also called *constants*, 0-ary predicate symbols are called *propositional atoms*.
The set Term_Σ of *terms* over Σ is defined inductively as

- each variable is a term,
- for $f \in \Sigma_F$, $\mathsf{ord}(f) = n$ and terms $t_1, \ldots, t_n$, also $f(t_1, \ldots, t_n)$ is a term.

The set of *atomic formulas* over Σ is given as

$$\begin{aligned}\mathsf{At}_\Sigma := \quad & \{s = t \mid s, t \in \mathsf{Term}_\Sigma\} \cup \\ & \{p(t_1, \ldots, t_n) \mid p \in \Sigma_R,\ \mathsf{ord}(p) = n,\ t_1, \ldots, t_n \in \mathsf{Term}_\Sigma\}\ .\end{aligned}$$

The set of *first-order formulas*, $\mathtt{FO}_\Sigma$ over Σ is defined as the least set with the following properties:

- all atomic formulas are formulas,

- true and false are formulas,
- for formulas A and B, a variable x, $\neg A$, $A \vee B$, and $\exists x : A$ are formulas (additionally, $A \wedge B$ and $\forall x : A$ are derived as abbreviations for $\neg(\neg A \vee \neg B)$ and $\neg\exists x : \neg A$, respectively).

The notions of bound and free variables are defined in the usual way, $\mathsf{free}(\mathcal{F})$ denoting the set of variables occurring free in a set $\mathcal{F}$ of formulas.

A *substitution* (over a signature Σ) is a mapping $\sigma : \mathsf{Var} \to \mathsf{Term}_\Sigma$ where $\sigma(x) \neq x$ for only finitely many $x \in \mathsf{Var}$. $\sigma : \sigma(x) = t$ is written as $[x \leftarrow t]$. Substitutions are extended to terms and formulas as usual.
The semantics of first-order logic is given by *first-order structures* over a given signature:

Definition 2.1 (First-Order Structure)
A *first-order structure* $\boldsymbol{I} = (I, \boldsymbol{U})$ over a signature Σ consists of a nonempty set $\boldsymbol{U}$ (*universe*) and an *interpretation* I of the signature symbols over $\boldsymbol{U}$ which maps

- every constant c to an element $I(c) \in \boldsymbol{U}$,
- every n-ary function symbol f to an n-ary function $I(f) : \boldsymbol{U}^n \to \boldsymbol{U}$,
- every propositional atom A to a truth value $I(A) \in \{\boldsymbol{t}, \boldsymbol{f}\}$,
- every n-ary predicate symbol p to an n-ary relation function $I(p) : \boldsymbol{U}^n \to \{\boldsymbol{t}, \boldsymbol{f}\}$.

For short, I consists of two mappings $I_F : \Sigma_F \to (\boldsymbol{U}^\omega \to \boldsymbol{U})$ and $I_P : \Sigma_R \to (\boldsymbol{U}^\omega \to \{\boldsymbol{t}, \boldsymbol{f}\})$.[1]
For a given signature Σ, the set of all first-order structures over Σ is denoted by $\mathbb{S}_\Sigma$. The set of interpretations of a signature Σ with a given universe $\boldsymbol{U}$ is denoted by $\mathbb{I}_{\Sigma,\boldsymbol{U}}$. $\square$

A *variable assignment* over a universe $\boldsymbol{U}$ is a mapping

$$\chi : \mathsf{Var} \to \boldsymbol{U} \ .$$

The set of variable assignments is denoted by Ξ.
For a variable assignment χ, a variable x, and $\boldsymbol{d} \in \boldsymbol{U}$, the *modified* variable assignment χ_x^d is identical with χ except that it assigns d to the variable x:

$$\chi_x^{\boldsymbol{d}} : \mathsf{Var} \to \boldsymbol{U} : \begin{cases} y \mapsto \chi(y) & \text{if } y \neq x \ , \\ x \mapsto \boldsymbol{d} & \text{otherwise.} \end{cases}$$

Every structure $\boldsymbol{I}$ induces an evaluation $\boldsymbol{I}$ of terms

$$\boldsymbol{I} : \mathsf{Term}_\Sigma \times \Xi \to \boldsymbol{U}$$

and tuples of terms, $\boldsymbol{I} : \mathsf{Term}_\Sigma^n \times \Xi \to \boldsymbol{U}^n$, as follows:

$\boldsymbol{I}(x, \chi) := \chi(x)$ for a variable x ,
$\boldsymbol{I}((t_1, \ldots, t_n), \chi) := (\boldsymbol{I}(t_1, \chi), \ldots, \boldsymbol{I}(t_n, \chi))$ for terms $t_1, \ldots, t_n$,
$\boldsymbol{I}(f(t_1, \ldots, t_n), \chi) := (I(f))(\boldsymbol{I}((t_1, \ldots, t_n), \chi)) = (I(f))(\boldsymbol{I}(t_1, \chi), \ldots, \boldsymbol{I}(t_n, \chi))$
for a function symbol $f \in \Sigma$, $\mathsf{ord}(f) = n$ and terms $t_1, \ldots, t_n$.

[1] the equivalent definition of a predicate as a relation $I(p) \subseteq \boldsymbol{U}^n$ is not followed here since in the sequel also *partial* interpretations are needed.

To indicate the truth of a formula F in a structure $\boldsymbol{I}$ under a variable assignment χ, the standard notation $\models_{\mathrm{FO}}$ (or simply $\models$) is used: Let s, t be terms, p a predicate symbol, $\mathsf{ord}(p) = n$, $t_1, \dots, t_n$ terms, x a variable, A and B formulas. Then

$$\begin{array}{lll}
(\boldsymbol{I}, \chi) \models \mathsf{true} \ , & & \\
(\boldsymbol{I}, \chi) \models p(t_1, \dots, t_n) & :\Leftrightarrow & (\boldsymbol{I}(t_1, \chi), \dots, \boldsymbol{I}(t_n, \chi)) \in I(p) \ , \\
(\boldsymbol{I}, \chi) \models \neg A & :\Leftrightarrow & \text{not } (\boldsymbol{I}, \chi) \models A \ , \\
(\boldsymbol{I}, \chi) \models A \vee B & :\Leftrightarrow & (\boldsymbol{I}, \chi) \models A \text{ or } (\boldsymbol{I}, \chi) \models B \ , \\
(\boldsymbol{I}, \chi) \models \exists x : A & :\Leftrightarrow & \text{there is a } \boldsymbol{d} \in \boldsymbol{U} \text{ with } (\boldsymbol{I}, \chi_x^{\boldsymbol{d}}) \models A \ .
\end{array}$$

The symbols $A \wedge B := \neg(\neg A \vee \neg B)$, $A \rightarrow B := \neg A \vee B$ and $\forall x : F := \neg \exists x : \neg F$ are defined as usual.

2.2 Description Logics

Description logics (DL) are a family of logic-based knowledge representation formalisms [BCM+03]. Knowledge is *described* in terms of concepts, roles and individuals, hence the name *description* logics.

The most basic DL is $\mathcal{AL}$ ("attributive language"), from which other description languages differ in expressivity by allowing or disallowing parts of the language constructs. In $\mathcal{AL}$, concepts can be defined using concept conjunction ($\sqcap$), negation ($\neg$) of atomic concepts, limited existential quantification ($\exists R.\top$ with $\top$ as the only allowed filler) and universal restrictions ($\forall R.C$). Furthermore, the bottom concept ($\bot$) and the top concept ($\top$) are available.

Extensions to $\mathcal{AL}$ are indicated by the use of single letters, appended to the logics name, according to the following schema:

$\mathcal{U}$	concept union
$\mathcal{E}$	full existential quantification
$\mathcal{C}$	negation of arbirary concepts
$\mathcal{I}$	inverse roles
$\mathcal{N}$	unqualified role restrictions (number restrictions)
$\mathcal{Q}$	qualified cardinality restrictions
$\mathcal{H}$	role hierarchies
$\mathcal{R}^+$	transitive roles
$\mathcal{F}$	functional properties
$\mathcal{O}$	nominals

Note that $\mathcal{U}$ and $\mathcal{E}$ can be expressed by $\mathcal{C}$. $\mathcal{AL}$ without atomic negation is named $\mathcal{FL}^-$ ("frame language"), whereas $\mathcal{FL}_0$ denotes $\mathcal{AL}$ without existential quantification. Usually, $\mathcal{ALC}$ plus transitively closed primitive roles $\mathcal{R}^+$ is called $\mathcal{S}$.

The ontology description language OWL can be seen as a syntactic variant of the description logic $\mathcal{SHOIN}$(D). In the following, the syntax and semantics of $\mathcal{SHOIN}$(D) are given.

Concept Descriptions. The elementary descriptions in a DL knowledge base are *atomic concepts* and *atomic roles*. Concepts are the building blocks and can be seen as classes of individuals. Complex descriptions can be built from combinations of atomic concepts, roles, and the concept constructors that the logic offers. Individuals are objects that can belong to any number of concepts. Properties of individuals and their relationships can be modeled with roles. Rolefillers can be either objects (individuals) or literals. The latter are *concrete* datatypes, and it depends on the particular Description Logic, which kind of datatypes are supported. If a DL is parameterised with a set D of datatypes, this is indicated with an appended $'(D)'$ to the DL's name. Datatypes can be used in such logics via *concrete roles*.

Definition 2.2 (Syntax)
The set of $\mathcal{SHOIN}$(D) concepts is defined by the following syntactic rules, where A is an atomic concept, C and D are concept descriptions, R is an abstract role, S is an abstract simple role, T_i are concrete roles, d is a concrete domain predicate, a_i and c_i are abstract and concrete individuals, respectively, and n is a non-negative integer.

$$\begin{array}{lll} C & \rightarrow & \top \mid \bot \mid A \mid \neg C \mid C_1 \sqcap C_2 \mid C_1 \sqcup C_2 \mid \exists R.C \mid \forall R.C \mid \\ & & \geq nS \mid \leq nS \mid \{a_1, \ldots, a_n\} \mid \geq nT \mid \leq nT \mid \\ & & \exists T_1, \ldots, T_n.D \mid \forall T_1, \ldots, T_n.D \\ D & \rightarrow & d \mid \{c_1, \ldots, c_n\} \end{array}$$

If more than one type of constructor occurs, the precedence order $\{\neg\} > \{\exists, \forall\} > \{\sqcap, \sqcup\}$ applies. □

Definition 2.3 (Semantics)
Let N_{role} be the set of role names and N_{con} the set of concept names. Furthermore let N_{ind} be the set of individual names. N_{con}, N_{role} and N_{ind} are pairwise disjoint finite sets. The formal semantics of the basic DL concepts is defined by an interpretation $\mathcal{I}$ that consists of a non-empty set $\Delta^{\mathcal{I}}$ (the domain of interpretation) and an interpretation function $\cdot^{\mathcal{I}}$, which assigns to every concept $C \in N_{con}$ a set $C^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, to every role $R \in N_{role}$ a binary relation $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ and $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$ to every individual $a \in N_{ind}$.

The interpretation function is extended to concept descriptions according to the following inductive definition:

top concept	$\top^{\mathcal{I}}$	=	$\Delta^{\mathcal{I}}$
bottom concept	$\bot^{\mathcal{I}}$	=	$\varnothing$
concept negation	$(\neg C)^{\mathcal{I}}$	=	$\Delta^{\mathcal{I}} \backslash C^{\mathcal{I}}$
conjunction	$(C \sqcap D)^{\mathcal{I}}$	=	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
disjunction	$(C \sqcup D)^{\mathcal{I}}$	=	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
value restriction	$(\forall R.C)^{\mathcal{I}}$	=	$\{a \in \Delta^{\mathcal{I}} \mid \forall b.(a,b) \in R^{\mathcal{I}} \rightarrow b \in C^{\mathcal{I}}\}$
existential quant.	$(\exists R.\top)^{\mathcal{I}}$	=	$\{a \in \Delta^{\mathcal{I}} \mid \exists b.(a,b)\}$
full existent. quant.	$(\exists R.C)^{\mathcal{I}}$	=	$\{a \in \Delta^{\mathcal{I}} \mid \exists b.(a,b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\}$
at-most restriction	$(\leq nR)^{\mathcal{I}}$	=	$\{a \in \Delta^{\mathcal{I}} \mid \#\{b \mid (a,b) \in R^{\mathcal{I}}\} \leq n\}$
at-least restriction	$(\geq nR)^{\mathcal{I}}$	=	$\{a \in \Delta^{\mathcal{I}} \mid \#\{b \mid (a,b) \in R^{\mathcal{I}}\} \geq n\}$
nominals	$\{a_1, \ldots, a_n\}^{\mathcal{I}}$	=	$\{a_1^{\mathcal{I}}, \ldots, a_n^{\mathcal{I}}\}$

The cardinality of a set S is denoted by $\#S$.

The concepts and roles of a DL ontology are related by the use of axioms, namely equality ($\equiv$) and inclusion ($\sqsubseteq$). The *validity* of an axiom ϵ (either a concept or a role axiom) in an interpretation $\mathcal{I}$ is called a model of ϵ, which is denoted by $\mathcal{I} \models \epsilon$ and defined as follows: $\mathcal{I} \models (\mathcal{C} \sqsubseteq \mathcal{D})$ iff $(C^{\mathcal{I}} \subseteq D^{\mathcal{I}})$ and $\mathcal{I} \models (\mathcal{C} \equiv \mathcal{D})$ iff $(C^{\mathcal{I}} = D^{\mathcal{I}})$.

Equivalence of two concepts C and D is given if $C^{\mathcal{I}} = D^{\mathcal{I}}$ and is denoted with $C \equiv D$. A concept D includes another concept C if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. This is denoted with $C \sqsubseteq D$.

$\mathcal{I}$ satisfies a set of axioms E iff $\mathcal{I}$ satisfies each element in E, in that case $\mathcal{I}$ is called a model of E. Two sets of axioms E_1 and E_2 are equivalent if they have the same models. In the same way E_1 is entailed by E_2 (denoted by $E_2 \models E_1$ if all of the models of E_1 are also models of E_2. □

Definition 2.4 (Knowledge Base)
A $\mathcal{SHOIN}(\mathcal{D})$ knowledge base $\mathcal{K} = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$ consists of a TBox $\mathcal{T}$, an ABox $\mathcal{A}$ and an RBox $\mathcal{R}$. The notions of a TBox, an ABox and an RBox are explained in the following.

TBox. A *TBox* $\mathcal{T}$ consists of a finite set of *terminological axioms*, either *concept inclusion axioms* $C \sqsubseteq D$ or *equality axioms* $C \equiv D$ for concepts C and D. An equation where the left-hand side is an atomic concept is called a *definition* (of that concept): new concepts are defined in terms of previously defined concepts. The use of concept inclusion ($\sqsubseteq$) is called an incomplete definition or a specialisation.

Let N_{con} have partitions N_{con_b} and N_{con_d} being the sets of *base concept* names and *defined concept* names. Base concept names occur *only* on the right-hand side of axioms, whereas defined concept names occur on the left-hand side of some axioms. Hence, the terminology defines the *defined names* in terms of *base names.* $A \equiv C$ is a *definition* of A for $A \in N_{con_d}$ and C over N_{con}, N_{role}, and N_{nom}.

A terminology is unequivocal such that there is at most one definition for every atomic concept $A \in N_{con}$.

A terminology is *acyclic* iff $\mathcal{T}$ is of the form $\{A_i \equiv C_i \mid 1 \leq i \leq n\}$ such that for every $i \in \{1, \ldots, n\}$ only defined names from $\{A_1, \ldots, A_{i-1}\}$ occur in C_i.

RBox. An *RBox* $\mathcal{R}$ is a finite set of *role inclusion axioms* of the form $R \sqsubseteq S$ where R and S are abstract roles. An *abstract role* is an abstract role name or the inverse R^- of an abstract role name R. A set of abstract role names N_{role} consists of transitive role names R_+ and normal role names R_P where $R_+ \cap R_P = \emptyset$. The set of abstract roles can be defined as $N_{role} \cup \{R^- | R \in N_{role}\}$. A role hierarchy is a finite set of role inclusion axioms.

An interpretation $\mathcal{I}$ consists of a non-empty set $\Delta^{\mathcal{I}}$ (the domain of interpretation) and an interpretation function $\cdot^{\mathcal{I}}$, which maps every role to a subset of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ such that, for $P \in N_{role}$ and $R \in R_+$

$$\langle x, y \rangle \in P^{\mathcal{I}} \text{ iff } \langle y, x \rangle \in P^-, \text{and if } \langle x, y \rangle \in R^{\mathcal{I}} \text{ and } \langle y, z \rangle \in R^{\mathcal{I}}, \text{ then } \langle x, z \rangle \in R^{\mathcal{I}}.$$

Role composition '$\circ$' is interpreted as an associative binary operator with $r_1^{\mathcal{I}} \circ r_2^{\mathcal{I}} := \{(x, z) \mid (x, y) \in r_1^{\mathcal{I}} \wedge (x, z) \in r_2^{\mathcal{I}}\}$ for every $r_1, r_2 \in N_{role}$.

A role inclusion axiom is defined by $r_1 \circ \cdots \circ r_n \sqsubseteq r$ for $n \in \mathbb{N}^+$ and $r_1, \ldots, r_n, r \in N_{role}$. A *simple role* is a role that has no transitive sub-roles (where $n = 1$).

To avoid considering roles such as R^{--}, a function Inv is defined on roles such that $\mathsf{Inv}(R) = R^-$ if R is a role name, and $\mathsf{Inv}(R) = S$ if $R = S^-$. For an RBox $\mathcal{R}$ a role hierarchy $\mathcal{R}^+$ is defined as

$$\mathcal{R}^+ \quad := \quad (\mathcal{R} \cup \{\mathsf{Inv}(R) \sqsubseteq \mathsf{Inv}(S) \mid R \sqsubseteq S \in \mathcal{R}\}, \sqsubseteq^*)$$

where $\sqsubseteq^*$ is the transitive-reflexive closure of $\sqsubseteq$ over $\mathcal{R} \cup \{\mathsf{Inv}(R) \sqsubseteq \mathsf{Inv}(S) \mid R \sqsubseteq S \in \mathcal{R}\}$.

An interpretation $\mathcal{I}$ satisfies a role hierarchy $\mathcal{R}^+$ iff $R^{\mathcal{I}} \subseteq S^{\mathcal{I}}$ for each $R \sqsubseteq^* S \in \mathcal{R}^+$, which is denoted by $\mathcal{I} \models \mathcal{R}^+$. $\mathcal{I}$ is then a model of $\mathcal{R}^+$ (recall that $\mathcal{R}^+$ is a set of axioms).

ABox. An assertional box (*ABox*) $\mathcal{A}$ is a finite set of concept and role assertions $C(a)$, $R(a, b)$, $T(a, c)$ plus individual (in)equality relations $a \doteq b$ and $a \not\doteq b$.

The interpretation $\mathcal{I}$ maps each individual $a \in N_{ind}$ to an element $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$. It is assumed (following the *unique name assumption*) that distinct individual names denote distinct objects. Therefore, for distinct individuals $a, b \in N_{ind}$ holds $a^{\mathcal{I}} \neq b^{\mathcal{I}}$. Note that this is handled strictly different in OWL where no UNA can be applied, as will be shown later. Moreover, $\mathcal{I}$ satisfies the concept assertion $C(a)$ if $a^{\mathcal{I}} \in C^{\mathcal{I}}$ and the role assertion $R(a, b)$ if $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$. $\mathcal{I} \models \mathcal{A}$ iff $\mathcal{I}$ satisfies each assertion in $\mathcal{A}$ and is then called a model of the ABox $\mathcal{A}$.

If an interpretation $\mathcal{I}$ is a model of an ABox $\mathcal{A}$, an RBox $\mathcal{R}$ and a TBox $\mathcal{T}$, it *satisfies $\mathcal{A}$ with respect to $\mathcal{R}$ and $\mathcal{T}$*.

$\mathcal{I}$ is a model of a knowledge base $\mathcal{K}$ iff $\mathcal{I}$ is a model of each component $\mathcal{T}$,$\mathcal{R}$ and $\mathcal{A}$. □

Definition 2.5 (Concrete domain)
DL knowledge bases are not only capable of defining structural properties of concepts, moreover quantitative properties can be expressed. Accordingly, there exists the notion of *concrete domains* that express the availability of concrete data types (e.g. strings or integers) in role definitions.

Let N_{role_c} be the set of concrete role names cr. Every cr in N_{role_c} must be interpreted by a mapping $\Delta^{\mathcal{I}}_D \rightarrow 2^D$. Furthermore let a concrete domain D be a pair $\langle \Delta_D, \Phi_D \rangle$, where Δ_D is an interpretation domain and Φ_D a set of concrete domain predicates d over that domain with a predefined arity $n \in \mathbb{N} \backslash \{0\}$ and an interpretation $d^D \subseteq \Delta^n_D$. For every $d \in \Phi_D$ with arity n and $cr_1, \ldots, cr_n \in N_{role_c}, d(cr_1, \ldots, cr_n)$ is a *concrete domain restriction.* An interpretation $\mathcal{I}$ is a pair $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ consisting of a non empty set $\Delta^{\mathcal{I}}$ (the domain) that is disjoint from Δ_D and an interpretation function $\cdot^{\mathcal{I}}$, $d(cr_1, \ldots, cr_n)$ is interpreted as follows:

$$\begin{aligned} d(cr_1, \ldots, cr_n)^{\mathcal{I}} \quad := \quad \{x \in \Delta^{\mathcal{I}} \mid \exists y_1, \ldots, y_n \in \Delta_D :\\ cr^{\mathcal{I}}_i(x) = y_i \text{ for all } 1 \leq i \leq n\\ \wedge (y_1, \ldots, y_n) \in \Phi_D\}. \end{aligned}$$

For instance, consider the concrete domain $D = (\mathbb{N}_D, \Phi_D)$ over the set of natural numbers $\mathbb{N}$. Let $\Phi = \{\geq_{12}, \geq_{16}, \geq_{18}, \geq_{21}\}$ be the set of concrete domain predicates in this example. The interpretation of the predicate $\geq_{12}$ is $\geq^{\mathcal{D}}_{12}=$

$\{(x) \in \mathbb{N} \mid x \geq 12\} = \{12, 13, 14, 15, \ldots\}$, denoting the set of integers equal or greater than 12. Given that $\{hasAge\} \in \mathcal{P}$, the concept of *GrownUpPerson* can be defined as persons being at least 18 years old, similarly the concept *AdultPerson* requiring an age of at least 21.

$$\begin{aligned} GrownUpPerson &\sqsubseteq \text{Person} \sqcap \geq_{18} (\text{hasAge}) \\ AdultPerson &\sqsubseteq \text{Person} \sqcap \geq_{21} (\text{hasAge}) \end{aligned}$$

Note that this notation for concrete domain predicates should not be confused with role restrictions. For example,

$$\text{ExtendedFamilyPerson} \equiv \geq 6 \text{ hasChild} \sqcap \text{Person}$$

defines the concept of persons having at least 6 children.

Binary concrete predicates can be used likewise, e.g. $\leq$ which is interpreted by $\leq^{\mathcal{D}} = \{(x, y) \in \mathbb{N} \mid x < y\}$. For example, the concrete roles *shoesize* and *IQ* can thus be related: $\leq (\text{shoesize}, \text{IQ})$.

2.3 OWL and DL

Most of the examples in this work are not given in description logic directly. Rather the Web Ontology Language variant OWL-DL is used as the ontology description language. Whereas in OWL-Full (and RDFS) classes can be instances of both themselves and other classs, this is forbidden in OWL-DL. This is the most prominent reason that allows to relate OWL-DL to the description logic $\mathcal{SHOIN}(\mathcal{D})$ [HPS04] such that computing ontology entailment in OWL-DL has the same complexity as computing knowledge base satisfiability in $\mathcal{SHOIN}(\mathcal{D})$. For the extension from $\mathcal{SHOIN}(\mathcal{D})$ to $\mathcal{SHOIQ}(\mathcal{D})$ another decision procedure was found that (although $\mathcal{SHOIQ}(\mathcal{D})$ is NExpTime-complete) performs well in typical cases [HS07]. A recent extension to OWL is OWL1.1, which was a W3C working draft at the time of writing of this thesis and extends the expressiveness of the underlying DL to $\mathcal{SHROIQ}$(D).

Table 2.3 shows most of the common axioms in OWL (including RDF and RDFS axioms) and how they correspond to DL expressions. Reasoners for OWL like *Pellet*, *Racer* or *FaCT++* support full OWL-DL.

Entailment in OWL is defined as usual and is denoted by $\models_{OWL}$.

2.4 F-Logic: Language and Basic Concepts

The following section gives a formal definition of the deductive object-oriented database language F-Logic [KLW95].

Definition 2.6 (Syntax of F-Logic)
The syntax of F-Logic (without multivalued methods and schema reasoning) is defined as follows:

OWL : $x \in C$	DL Syntax
C	C
intersectionOf(C_1, C_2)	$C_1 \sqcap C_2$
unionOf(C_1, C_2)	$C_1 \sqcup C_2$
complementOf(C_1)	$\neg C_1$
oneOf$(x_1, \ldots, x_n)$	$\{x_1\} \sqcup \ldots \sqcup \{x_n\}$
OWL : $x \in C$, Restriction on P	**DL Syntax**
someValuesFrom(C')	$\exists P.C'$
allValuesFrom(C')	$\forall P.C'$
hasValue(y)	$\exists P.\{y\}$
maxCardinality(n)	$\leq n.P$
minCardinality(n)	$\geq n.P$
cardinality(n)	$n.P$
OWL Class Axioms for C	**DL Syntax**
rdfs:subClassOf(C_1)	$C \sqsubseteq C_1$
equivalentClass(C_1)	$C \equiv C_1$
disjointWith(C_1)	$C \sqsubseteq \neg C_1$
OWL Individual Axioms	**DL Syntax**
x_1 sameAs x_2	$\{x_1\} \equiv \{x_2\}$
x_1 differentFrom x_2	$\{x_1\} \sqsubseteq \neg\{x_2\}$
AllDifferent$(x_1, \ldots, x_n)$	$\bigwedge_{i \neq j} \{x_i\} \sqsubseteq \neg\{x_j\}$
OWL Properties	**DL Syntax**
P	P
OWL Property Axioms for P	**DL Syntax**
rdfs:range(C)	$\top \sqsubseteq \forall P.C$
rdfs:domain(C)	$C \sqsupseteq \exists P.\top$
subPropertyOf(P_2)	$P \sqsubseteq P_2$
equivalentProperty(P_2)	$P \equiv P_2$
inverseOf(P_2)	$P \equiv P_2^-$
TransitiveProperty	$P^+ \equiv P$
FunctionalProperty	$\top \sqsubseteq \leq 1P.\top$
InverseFunctionalProperty	$\top \sqsubseteq \leq 1P^-.\top$

Table 2.1: Translation from OWL to DL.

- The alphabet consists of a set $\mathcal{F}$ of *object constructors*, playing the role of function symbols, a set $\mathcal{V}$ of variables, and several auxiliary symbols. Object constructors are denoted by lowercase letters and variables by uppercase ones.

- *id-terms* are composed from object constructors and variables. They are interpreted by elements of the universe.

In the sequel, let O, $O_1, \ldots, O_n$, C, D, M, and V denote id-terms.

- An *is-a atom* is an expression of the form $O : C$ (object O is a member of class C), or $C :: D$ (class C is a subclass of class D).

- The following are *object atoms*:

– $O[M \rightarrow V]$: applying the *scalar* method M to O – as an object – results in V; in logical terms, scalar methods (without parameters) are unary functions over the domain of objects (also called *functional methods*).

– $O[M \bullet\!\!\rightarrow V]$: O – as a class – provides the *inheritable scalar* method M. For a member $o : O$, inheritance results in $o[M \rightarrow V]$; for a subclass $c :: O$, inheritance results in $c[M \bullet\!\!\rightarrow V]$.

– Analogously $O[M@(O_1, \ldots, O_n) \rightarrow V]$ and $O[M@(O_1, \ldots, O_n) \bullet\!\!\rightarrow V]$ with $n \in \mathbb{N}$ for parameterised methods.

- *Formulas* are built from atoms using first-order logic connectives.

- An F-Logic *rule* is a logic rule of the form head $\leftarrow$ body over F-Logic's atoms.

- An F-Logic *program* is a set of rules. $\square$

Note that F-Logic does not distinguish between classes, methods, and objects which uniformly are denoted by id-terms; also variables can occur at arbitrary positions of an atom.

The semantics of F-Logic extends the semantics of first-order predicate logic. Formulas are interpreted over a semantic structure. The discussion is restricted to Herbrand-interpretations where the universe consists of ground id-terms. An *H-structure* is a set of ground F-Logic atoms describing an object world, thus it has to satisfy several *closure axioms* related to general object-oriented properties:

Definition 2.7 (Closure Axioms)
A (possibly infinite) set $\boldsymbol{H}$ of ground atoms is an *H-structure* if the following conditions hold for arbitrary ground id-terms u, $u_0, \ldots, u_n$, and u_m occurring in $\boldsymbol{H}$:

- $u :: u \in \boldsymbol{H}$ (subclass reflexivity),

- if $u_1 :: u_2 \in \boldsymbol{H}$ and $u_2 :: u_3 \in \boldsymbol{H}$ then $u_1 :: u_3 \in \boldsymbol{H}$ (subclass transitivity),

- if $u_1 :: u_2 \in \boldsymbol{H}$ and $u_2 :: u_1 \in \boldsymbol{H}$ then $u_1 = u_2 \in \boldsymbol{H}$ (subclass acyclicity),

- if $u_1 : u_2 \in \boldsymbol{H}$ and $u_2 :: u_3 \in \boldsymbol{H}$ then $u_1 : u_3 \in \boldsymbol{H}$ (instance-subclass dependency),

$*$ there are no ground id-terms u and u' ($u \neq u'$) such that $u_0[u_m \rightsquigarrow u] \in \boldsymbol{H}$ and $u_0[u_m \rightsquigarrow u'] \in \boldsymbol{H}$, where $\rightsquigarrow$ stands for $\rightarrow$ or $\bullet\!\!\rightarrow$ (uniqueness of scalar methods; recall that from the logical point of view, they define functions).

For a set M of ground atoms, $\mathcal{C}\ell(M)$ denotes the closure of M wrt. the above axioms, $\mathcal{C}\ell(M) = \bot$ if the constraint ($*$) is violated in M.
$\mathrm{Th}_{\mathrm{FL}}(F)$ denotes the F-Logic theory of a set F of formulas which means the closure of F wrt. a complete set of axioms of first-order logic and the axioms

$$\frac{}{X :: X} \qquad \frac{X_1 :: X_2\ ,\ X_2 :: X_3}{X_1 :: X_3} \qquad \frac{X_1 :: X_2\ ,\ X_2 :: X_1}{X_1 = X_2}$$

$$\frac{X_1 : X_2\ ,\ X_2 :: X_3}{X_1 : X_3} \qquad \frac{O[M \rightsquigarrow V]\ ,\ O[M \rightsquigarrow V']\ ,\ V \neq V'}{\mathsf{false}}$$

(again, $\rightsquigarrow$ stands for $\rightarrow$ or $\bullet\!\!\rightarrow$.) □

For an H-structure, the truth of atoms and formulas is given in the usual way [KLW95]. Positive F-Logic programmes are evaluated bottom-up by a T_P-like operator including $\mathcal{C}\ell$, providing a minimal model semantics:

Definition 2.8 (Deductive Fixpoint)
For an F-Logic programme P and an H-structure $\boldsymbol{H}$,

$$\begin{aligned}
T_P(\boldsymbol{H}) &:= \boldsymbol{H} \cup \{h \mid (h \leftarrow b_1, \ldots, b_n) \text{ is a ground instance of some rule of } P \\
&\qquad\qquad \text{and } b_i \in \boldsymbol{H} \text{ for all } i = 1, \ldots, n\}\ , \\
T_P^0(\boldsymbol{H}) &:= \mathcal{C}\ell(\boldsymbol{H})\ , \\
T_P^{i+1}(\boldsymbol{H}) &:= \mathcal{C}\ell(T_P(T_P^i(\boldsymbol{H})))\ , \\
T_P^{\omega}(\boldsymbol{H}) &:= \begin{cases} \lim_{i\to\infty} T_P^i(\boldsymbol{H}) & \text{if the sequence } T_P^0(\boldsymbol{H}), T_P^1(\boldsymbol{H}), \ldots \text{ converges,} \\ \bot \text{ otherwise.} \end{cases}
\end{aligned}$$

Note that $\mathcal{C}\ell(\boldsymbol{H}) = \bot$ can also lead to the result $\bot$. □

The above T_P-operator does not deal with inheritance. In [KLW95], *inheritance-canonic* models are defined, based on *inheritance triggers* which extend the above fixpoint semantics with some procedural flavour. Note that although default inheritance is a standard feature of F-Logic that is provided by the FLORID implementation it is not used directly in the process of hybrid reasoning (see Section 9.4.3) in this work. Rather the default inheritance rules given in F-Logic are interpreted “outside” of FLORID.

2.5 Default Inheritance

Semantics of Default Logic

A default is given in the following way, following the definition of Reiter [Rei80, Poo94]:

$$d = \frac{\alpha : \beta_1, \ldots, \beta_n}{w}$$

where d consists of a *precondition* $p(d) = \alpha$, a *justification* $J(d) = \beta = \beta_1, \ldots, \beta_n$ and a *consequence* $\mathrm{c(d)} = w$, all given as first-order formulas. If α is fulfilled and all β can be assumed consistently, w can be concluded. When the justifications are consistent, the default is equivalent to a logic rule $w \leftarrow \alpha$.

A default theory E is typically given as a pair $\langle D, F\rangle$, where D is a set of defaults and F a set of formulas (background theory or world description). If a default rule can be applied safely to a theory, it's consequences are added to the theory.

Example 2.1
Two defaults define that birds fly and have feathers:

$$d1 = \left\{\frac{bird(X) : flies(X)}{flies(X)}\right\}, d2 = \left\{\frac{bird(X) : hasFeathers(X)}{hasFeathers(X)}\right\}$$

Let $D = \{d1, d2\}$ *and* $F = \{bird(penguin), \neg flies(penguin), bird(raven)\}$. *From the first rule and* $bird(penguin)$ *follows that the prerequisite is true, but the justification is inconsistent with what is known. Therefore the consequence cannot be assumed. On the other hand,* $flies(raven)$ *can be concluded, as there is no contradictory justification to it. Both ravens and penguins have feathers. Now the consequences from the second rule* $\{flies(raven),\ hasFeathers(raven),\ hasFeathers(penguin)\}$ *are added to the background theory* F. □

If no other default rule can be applied to the theory, it is called an *extension* of the default theory.

The following examples are intended to illustrate the semantical difficulties that are possible with the use of defaults and inheritance:

Example 2.2 (Nixon Diamond)
It is known that Nixon is a republican and a quaker. A typical republican's policy is being a hawk, the typical policy of a quaker is being a pacifist. Now, there is a direct conflict with Nixons policy.

$$d1 = \left\{\frac{quaker(X) : pacifist(X)}{pacifist(X)}\right\}, d2 = \left\{\frac{republican(X) : hawk(X)}{hawk(X)}\right\}$$

$F = \{quaker(nixon), republican(nixon)\}$

The following lines show the same example in F-Logic syntax:

$$\begin{aligned} P = \{&\mathsf{quaker[policy{\bullet\!\!\to}pacifist]},\ \mathsf{republican[policy{\bullet\!\!\to}hawk]},\\ &\mathsf{r\_nixon : quaker},\ \mathsf{r\_nixon : republican}\}. \end{aligned}$$

□

The first default's justification is that quakers are pacifists, whereas the justification of the second default is that republicans are hawks. As both defaults can be applied there are two possible extensions to the default theory: one that contains the conclusion $pacifist(nixon)$, the other the conclusion $hawk(nixon)$.

Considering that the rules might be applied in different order, a default theory might have several different extensions or none at all. Depending on the semantics there is a distinction between *credulous (brave)* and *sceptical (cautious)* reasoning. The latter means that a formula δ is a consequence of the default theory iff it is in all extensions, whereas the first means that it is a consequence iff it is in any of the extensions.

If defaults and rules are combined, the situation is even more complicated:

Example 2.3 (Nixon Family)

A single rule is added to the Nixon-Diamond example:

$$\mathsf{W[policy{\rightarrow}P]} \leftarrow \mathsf{W[husband{\rightarrow}O]} \land \mathsf{O[policy{\rightarrow}P]}$$

and the atoms mrs_nixon[husband→r_nixon] *and* mrs_nixon : quaker . □

Now, there are the following possibilities:

- r_nixon inherits r_nixon[policy→hawk] and from this, classical deduction derives mrs_nixon[policy→hawk]. In this case, mrs_nixon[policy→pacifist] must not be inherited – thus, she is an atypical quaker.

- r_nixon inherits r_nixon[policy→pacifist] – in which case classical deduction derives mrs_nixon[policy→pacifist] which is the same value as she would (have) inherit(ed) from being a quaker.

- mrs_nixon[policy→pacifist] is assigned first. Although there is no *direct* conflict when inheriting r_nixon[policy•→hawk] the logical consequences require mrs_nixon[policy•→hawk] which is inconsistent with the already inherited facts. A correct semantics should not inherit in this situation and leave the *policy* property for r_nixon undefined.

In Default Logic a default only applies if its justification is consistent with the resulting structure whereas in inheritance nets such *indirect* conflicts are taken care of, called *mixed conflicsts* [Hor].

Default Inheritance in F-Logic

The implementation of default inheritance in the hybrid reasoning engine in SWAN (see Section 9.4) is, with regard to its semantics, quite similar to the one as described in [MK01] for FLORID. Whereas the implementation of FLORID uses *inheritance triggers*, this work implements the *cautious inflationary extensions* as proposed in [MK01].

In order to illustrate the combination of inheritance and defaults the *Tweety* example is given (here in F-Logic syntax):

Example 2.4

$P = \{$bird[fly•→true], bird[hasFeathers•→true], penguin[fly•→false], penguin :: bird, tweety : penguin$\}$.

With the above definition, $\mathcal{C}\ell(P) = P \cup \{$*tweety : bird*$\}$*. Here,* tweety *should inherit* tweety[fly→false] *from* penguin*, not* tweety[fly→true] *from* bird *since the potential inheritance of* tweety[fly→true] *from* bird *is* preempted *by the intermediate class* penguin[fly•→false]*.*

On the other hand, the property [hasFeathers•→true] *should be inherited from* bird *to* penguin[hasFeathers•→true] *and to* tweety[hasFeathers→true]*.* □

This example motivates one of the strategies which are applied in the sequel: properties are inherited *stepwise* downwards the class hierarchy.

The result of this section is that application of inheritance has to deal with two kinds of facts:

1. explicit: checking the superclass condition that inheritance is not preempted, and the requirement that the method to be inherited is not yet defined,

2. implicit: there can be facts which would be inconsistent with the inherited property, although they are not rejected by (1).

Inheritance in Default Logic

In an inheritance framework, the superclass condition belongs to (1); whereas the checks that inheritance is not preempted and that the inherited value must be consistent with the knowledge (wrt. the logical rules of the program) fall under (2).

For characterizing inheritance, only a specialised form of defaults is needed, called *semi-normal defaults*. Semi-normal defaults are of the form $\alpha(\bar{x})$:$\beta(\bar{x})/w(\bar{x})$ where the precondition $\alpha(\bar{x})$ is a conjunction of atoms, the consequence $w(\bar{x})$ is also an atomic formula, and $\forall \bar{x} : \beta(\bar{x}) \rightarrow w(\bar{x})$ holds. Translating the path-based concept of inheritance networks, inheritance in F-Logic syntax can be specified by defaults of the form

$$D'_{inh} := \frac{\begin{array}{c} O : C \ , \ C[M{\bullet\!\!\rightarrow}V] \ , \phi_{\text{path}}(O - C_1 - \ldots - C_n - C) \ : \\ \phi_{\text{not\_preempted}}(O - C_1 - \ldots - C_n - C, M{\bullet\!\!\rightarrow}V) \ , \ O[M{\rightarrow}V] \end{array}}{O[M{\rightarrow}V]}$$

(analogous for $C' :: C$)
where ϕ_{path} is a meta-predicate which states that $O - C_1 - \ldots - C_n - C$ is a path in the class hierarchy, and $\phi_{\text{not\_preempted}}(O - C_1 - \ldots - C_n - C, M{\bullet\!\!\rightarrow}V)$ is a meta-predicate which states that inheritance of $M{\bullet\!\!\rightarrow}V$ along the path $O - C_1 - \ldots - C_n - C$ is not preempted; i.e. that $c'[M{\bullet\!\!\rightarrow}V]$ is consistent for all intermediate classes c' on this path. Note, that for an H-structure $\boldsymbol{H}$, $o[m{\rightarrow}v]$ can only be assumed consistently if there is no $v' \neq v$ such that $o[m{\rightarrow}v] \in \boldsymbol{H}$. To avoid decoupling, inheritance along a path requires the inheritable property to be present in all intermediate classes:

$$D_{inh} = \frac{\begin{array}{l} O : C \ , \ C[M{\bullet\!\!\rightarrow}V] \ : \\ \forall C'((O : C' \wedge C' :: C) \rightarrow C'[M{\bullet\!\!\rightarrow}V]) \ , \ O[M{\rightarrow}V] \end{array}}{O[M{\rightarrow}V]} \ .$$

(analogous for $C' :: C$.)

Due to the fact that variables are also allowed at class and method positions, every instance of inheritance of an inheritable non-parameterised scalar method (which are denoted by ${\bullet\!\!\rightarrow}$) is an instance of the above default schema.

Extensions

The semantics of a default theory is defined in terms of *extensions*. In the following, for a set S of formulas, let $\text{Th}(S)$ denote the theory of S.[2]

[2] wrt. the respective framework, e.g. , propositional, first-order, F-Logic or DL.

Definition 2.9 (Extension; based on [Poo94])

Let $\Delta = (D, F)$ be a default theory. For sets S, T of formulas, let

$$\begin{aligned} GD(S,T,D) := \{d \mid\ & d \text{ is an instance of a default in } D, \\ & \mathrm{Th}(T) \models p(d)\ \text{, and} \\ & \mathrm{Th}(S \cup \{\beta\}) \text{ is consistent for every } \beta \in J(d)\} \end{aligned}$$

(GD stands for *generating defaults*). Then, for all sequences $S_0 = F, S_1, S_2, \ldots S_\eta$ of sets of formulas s.t. $S = (\bigcup_{i=0}^{\infty} S_i)$ and

$$S_{i+1} = S_i \cup C_i \quad \text{where } C_i = c(GD(S, S_i, D))\ ,$$

$\mathrm{Th}(S)$ is an *extension* of Δ. Since S is needed later on, it is called an *extension base* of Δ. □

Definition 2.10
Let D be a set of defaults and S a set of formulas. Then,

- $GD(S, D) := GD(S, S, D)$ is the set of applicable defaults in S,
- $GD^+(S, D) := \{d \in GD(S, D) \mid c(d) \notin \mathrm{Th}(S)\}$ is the set of applicable defaults which add knowledge not (yet) contained in S. □

Remark 2.1
Note that in Definition 2.9, $S_i = F \cup \bigcup_{j=0,\ldots,i-1} C_j$ and $S = F \cup \bigcup_{i=0}^{\infty} C_i = F \cup c(GD(S, D))$ and $GD^+(S, D) = \emptyset$, i.e., for all defaults d which are applicable in S, the consequence of d is in S. □

In [Mak94], this is termed a *quasi-inductive* definition: in the step $i \to i+1$, all β_j are required to be consistent with $\mathrm{Th}(S) = \mathrm{Th}(\bigcup_{i=0}^{\infty} S_i)$, thus, assumptions about *future* stages are made (note that in contrast, the evaluation of $p(d)$ does not use S). S must be guessed to prove that it is an extension, then it can be checked if S is the result of the fixpoint process. Note that, depending on which assumptions are made, there can be several *different* extensions (cf. Example 2.2).

Forward Chaining Evaluation

Motivated by the fixpoint semantics for positive logic programmes, the evaluation of logic programmes with inheritance should also be based on a forward-chaining approach, i.e., without having to guess S first. From Definition 2.9, a forward-chaining, *inflationary* strategy can be defined by replacing "$\mathrm{Th}(S \cup \{\beta\})$ is consistent" with "$\mathrm{Th}(S_i \cup \{\beta\})$ is consistent", i.e., evaluating defaults against the *current* belief set. In contrast to Definition 2.9, in every step the application of exactly one default is allowed. (May and Kandzia showed in [MK01] that this makes no difference as long as only positive programmes and defaults with positive preconditions are considered). The next section contains a review of the results from [MK01] which will then be applied applied to DL knowledge.

Definition 2.11 (Inflationary extension)

Let $\Delta = (D, F)$ be a default theory. Let AD be the set of applied defaults, $AD_0 = \emptyset$ and $S_0 = F, S_1, S_2, \ldots, S_\eta$ be a sequence of sets of formulas such that

$$d_i \in GD^+(S_i, D)\ ,\ AD_{i+1} = AD_i \cup \{d_i\}\ ,\ S_{i+1} = S_i \cup \{c(d_i)\}\ ,$$

and $GD^+(S_\eta, D) = \emptyset$ (for the definition of GD^+ see Def. 2.10). Then, with $S = (\bigcup_{i=0,\ldots,\eta} S_i)$, Th$(S)$ is called an *inflationary extension* of Δ; S is called an *inflationary extension base* of Δ. □

Remark 2.2
Note that again, $S_i = F \cup \bigcup_{j=0,\ldots,i-1}\{c(d_j)\}$ *and* $S = F \cup \bigcup_{i=0,\ldots,\eta}\{c(d_i)\}$ *and* $GD^+(S, D) = \emptyset$. *Nevertheless, it will be shown that in general there can be* d_j *such that* $d_j \notin GD(S, D)$. □

This approach is, e.g. , investigated in [MST93, Section 3.7, Def. 3.61]. As shown there, the above method is complete, but not sound: it generates theories which are no extensions. This problem can be solved in two steps.

Proposition 2.1 (Extensions vs. Inflationary Extensions)

Let $\Delta = (D, F)$ *be a Default theory.*

1. *Every extension* S *of* Δ *is also an inflationary extension of* Δ*, and*

2. *Let* S *be an inflationary extension computed by the algorithm given in Definition 2.11. If for every* $\beta \in J(AD_\eta)$, β *is consistent with* S*, then* S *is an extension of* Δ. □

Proof 1. *cf. [MST93, Cor. 3.68 and 3.71, Th. 3.73].*

2. *cf. [MST93, Th. 3.65].* □

The strategy is inflationary in the sense that a default which has been once applied is not undone (which would require to undo also all its logical consequences) if in a later step one of its *justifications* turns out to be wrong which is exactly the tested criterion in (2) of the above proposition.

Motivation. There are two alternatives how to deal with this problem: (i) forbid the application of defaults whose justifications will be falsified later, or (ii) forbid the application of a default whose logical consequences would falsify the justifications of another default which has been applied earlier.

The notion of extensions includes (i) whereas (ii) is much easier to implement. (i) leads to theories where no further default is applicable whereas (ii) can lead to structures where some defaults are still applicable. On the other hand (i) does not guarantee that such a structure exists, whereas a structure satisfying (ii) always exists.

It will be shown that (ii) is weaker than (i), but the difference can be controlled in case of inheritance.

Example 2.5

Consider a default theory $(\{d_1, d_2\}, F)$ *such that* $GD(F, \{d_1, d_2\}) = \{d_1, d_2\}$, $GD(F \cup c(d_1)) = d_2$, $GD(F \cup c(d_2)) = \emptyset$, *and* $c(d_2) \rightarrow \neg J(d_1)$.
Here, both $T_1 = Th(F \cup c(d_2))$ *and* $T_2 = Th(F \cup c(d_1) \cup c(d_2))$ *are inflationary extensions.* T_1 *is the only extension.* T_2 *is not an extension since* $T_2 \models \neg J(d_1)$, *thus, the justification of* d_1 *is falsified by application of* d_2.
$T_3 = Th(F \cup c(d_1))$ *is* not *an inflationary extension (and also not an extension) since* $GD^+(T_3, D) = d_2$.
The strategy (ii) above would result in T_1 *and* T_3 *as acceptable structures.* □

Cautious inflationary extensions are defined similar to Definition 2.11, following strategy (ii), i.e., avoiding the falsification of previous justifications:

Definition 2.12 (Cautious inflationary extension)

Let $\Delta = (D, F)$ be a default theory. For a set S of formulas and a set AD of ground instances of defaults, let

$$\begin{aligned} GD^+_{caut}(S, D, AD) := \{d \mid\ & d \text{ is an instance of a default in } D,\ \mathrm{Th}(S) \models p(d)\ ,\\ & c(d) \notin \mathrm{Th}(S),\ \text{and } \mathrm{Th}(S \cup c(d) \cup \beta) \text{ is consistent}\\ & \text{for every } \beta \in J(AD \cup \{d\})\}\ . \end{aligned}$$

Let $AD_0 = \emptyset$ and $S_0 = F, S_1, S_2, \ldots, S_\eta$ be a sequence of sets of formulas such that

$$d_i \in GD^+_{caut}(S_i, D, AD_i)\ ,\ S_{i+1} = S_i \cup \{c(d_i)\}\ ,\ AD_{i+1} = AD_i \cup \{d_i\}$$

and $GD^+_{caut}(S_\eta, D, AD_\eta) = \emptyset$. Then, with $S = (\bigcup_{i=0,\ldots,\eta} S_i)$, $\mathrm{Th}(S)$ is called a *cautious inflationary extension* of Δ and S is called a *cautious inflationary extension base* of Δ. □

Remark 2.3
Note that again, $S_i = F \cup \bigcup_{j=0,\ldots,i-1}\{c(d_j)\}$ *and* $S = F \cup \bigcup_{i=0,\ldots,\eta}\{c(d_i)\}$, $d_j \in GD^+(S, D)$ *for all* $j = 0, \ldots, i-1$, *but now* $GD^+(S, D) \neq \emptyset$ *is possible, i.e., there can be defaults* d *applicable in* S *such that* $c(d) \notin S$ *(then,* $c(d)$ *would lead to falsification of a justification of a previously applied default, thus,* $d \notin GD^+_{caut}(S, D, AD_\eta)$*).* □

Example 2.6
The above notions define strictly different notions of extensions. Consider the following default theory:

$$(D, \{p\}) \quad where \quad D = \left\{ \frac{p : \neg q}{r, s}\ ,\ \frac{p}{r, q} \right\}\ .$$

Here, $S = \{p, r, q\}$ *is the only extension, generated by* $GD(\{p, r, q\}, \{p\}, D) = \{p/r, q\}$. S *is also an inflationary extension and a cautious inflationary extension.*
But, $GD(\{p\}, D)$ *does not only contain* $p/r, q$ *since* $p : \neg q/r, s \in GD(\{p\}, D)$. *Applying* $p : \neg q/r, s$ *in* $\{p\}$ *leads to* $S' = \{p, r, s\}$ *which is not an extension*

since $GD^+(\{p, r, s\}, D) = \{p/r, q\}$. *Subsequent application of* $\{p/r, q\}$ *results in* $\{p, r, s, q\}$ *which is an inflationary extension, but the justification of the previously applied default* $p : \neg q/r, s$ *is invalidated. Thus,* S' *is a cautious inflationary extension – with* $GD^+(S', D) \neq \emptyset$.

There is no extension where $\neg q$ *is consistent, and the default* $p : \neg q/r, s$ *is not applied in the construction of any extension. Thus, when the inflationary strategy chooses to apply the default* $p : \neg q/r, s \in GD^+(\{p\}, D)$ *it runs into a* garden path *– it is not possible then to reach a valid extension.* □

As in the above example, the cautious strategy can run into garden paths, i.e., apply defaults such that it is not possible to reach an extension. Garden paths can only be cured by backtracking.

Proposition 2.2 (Cautious Inflationary vs. Inflationary Extensions)

Let $\Delta = (D, F)$ *be a default theory. Then,*

- *The computations of cautious inflationary extensions are the maximal prefixes of computations of inflationary extensions such that no justification of a previously applied default is falsified.*
- *A cautious inflationary extension* S *of* Δ *is an inflationary extension if* $GD^+(S, D) = \emptyset$.
- *If an inflationary extension* S *satisfies the criterion given in Proposition 2.1(2), then* S *is also a cautious inflationary extension.* □

Note that an inflationary extension not necessarily contains a cautious inflationary extension:

Example 2.7 (Cautious Inflationary vs. Inflationary Extensions)

Consider a default theory (D, F) *with* $D = \{d_1, d_2, d_3\}$ *such that* $GD(F, D) = \{d_1\}$, $GD(F \cup \{c(d_1)\}) = \{d_2, d_3\}$, $GD(F \cup \{c(d_1), c(d_2)\}) = GD(F \cup \{c(d_1), c(d_3)\}) = \emptyset$, *and* $F \cup \{c(d_1), c(d_2)\}$ *is consistent with* $\beta(d_1)$, *whereas* $F \cup \{c(d_1), c(d_3)\}$ *is inconsistent with* $\beta(d_1)$.

Then, $Th(F \cup \{c(d_1), c(d_2)\})$ *is an extension (and also a cautious inflationary extension), and* $Th(F \cup \{c(d_1), c(d_3)\})$ *is an inflationary extension which does not satisfy Proposition 2.1(2) and which does not contain a cautious inflationary extension.* □

Proposition 2.3 (Extensions vs. Cautious Inflationary Extensions)

Given a default theory $\Delta = \langle D, F\rangle$, *a cautious inflationary extension* S *of* Δ *is an extension of* Δ *if* $GD^+(S, D) = \emptyset$. □

Proof *By Prop. 2.2, every cautious inflationary extension* S *such that* $GD^+(S, D) = \emptyset$ *is an inflationary extension. Since every cautious inflationary extension satisfies the additional criterion stated in Proposition 2.1, it is then an extension of* Δ. □

Defaults in DL Knowledge Bases

A terminological default theory Δ is a pair $\langle D, K\rangle$ where K is a $\mathcal{SHOIN}$(D) knowledge base consisting of closed formulas and D is a set of semi-normal defaults[3].

As shown in [BH95] terminological default theories can be undecidable. Also did the authors show that decidability can be retained if defaults are applied to named individuals only. For this reason defaults are considered only for the individuals occuring in the assertional part of DL knowledge bases.

$\mathrm{Th}_{\mathrm{DL}}(F)$ denotes the description logic theory of a set of formulas F, which means the closure of F wrt. to the set of axioms of the particular description logic.

Definition 2.13
Given is a semi-normal default theory $\Delta = \langle D, K\rangle$. For a DL theory $\mathcal{M}$ and a set AD of ground instances of defaults, let

$$\begin{aligned} GD^{+}_{caut}(\mathcal{M}, D) := \{d \mid\ & d \text{ is a ground instance of a default in } D,\ p(d) \subseteq \mathcal{M}, \\ & \mathrm{Th}_{\mathrm{DL}}(\mathcal{M} \cup \{\beta\}) \text{ is consistent for every } \beta \in J(d), \\ & \text{and } c(d) \notin \mathcal{M}\}\ . \end{aligned}$$

Let $\mathcal{M}_0, \mathcal{M}_1, \ldots, \mathcal{M}_\eta$ be a sequence of DL theories such that $\mathcal{M}_0 = K$, $AD_0 = \emptyset$ and $d_i \in GD^{+}_{caut}(\mathcal{M}_i, D)$,$\mathcal{M}_{i+1} = \mathrm{Th}_{\mathrm{DL}}(\mathcal{M}_i \cup \{c(d_i)\})$, $AD_{i+1} = AD_i \cup \{d_i\}$, $S_i = \{c(d) \mid d \in AD_i\}$ and $GD^{+}_{caut}(\mathcal{M}_\eta, D) = \emptyset$.

If $\mathcal{M} := \bigcup_{i=0}^{\eta} \mathcal{M}_i \neq \bot$, then with $S = S_\eta$

$$\mathcal{M} := \mathrm{Th}_{\mathrm{DL}}(K \cup \bigcup_{j \in 0,\ldots,\eta} \{c(d_j)\}) = \mathrm{Th}_{\mathrm{DL}}(K \cup S)$$

is called the *DL-extension* of Δ to S (analogous for *inflationary DL-extensions* and *cautious inflationary DL-extensions*). $\square$

Application to Inheritance

For inheritance, only defaults of the form given in D_{inh} are used. For the forward-chaining strategy, the class hierarchy in S is not completely known when computing S_i. Instead, the fragment already known in S_{i-1} must be used for checking the consistency of the justifications. In D_{inh}, a justification can be annulled in later steps only when some path is chosen which is not preempted in S_i, but it turns out to be preempted in later steps. This can be due to one of the following effects:

(P1): for some class c', which is already known in S_i to be an intermediate class on the path, c'[m•→v] turns out to be inconsistent. In order to avoid such effects the inheritance rules are applied in an ordered way: The order of evaluation of a default d is defined such that $d = c[p\bullet\!\!\rightarrow v]$ is applied before any other default $d' = c'[p\bullet\!\!\rightarrow v']$ with $c' :: c$ and for any values v and v'.

[3]recall that for a semi-normal default d, $p(d)$ is a conjunction of atoms and $w(d)$ is an atomic formula.

(P2): in a later step, a new intermediate class-membership $\mathsf{o} : \mathsf{c}' :: \mathsf{c}$ on this path is derived for which $\mathsf{c'[m{\bullet}\!\!\to v]}$ is inconsistent. This effect is called *postemption*.

Consider the following example which inserts a postempting intermediate class-membership *after* inheritance has taken place:

Example 2.8

$P = \{\mathsf{cl1[m{\bullet}\!\!\to v1]},\ \mathsf{x : cl1},\ \mathsf{cl2 :: cl1},\ \mathsf{cl2[m{\bullet}\!\!\to v2]},\ \ \mathsf{x : cl2} \leftarrow \mathsf{x[m \to v1]}\}$.

The only computation sequence is

T_P^ω: $\{\mathsf{x : cl1},\ \mathsf{cl2 :: cl1},\ \mathsf{cl1[m{\bullet}\!\!\to v1]},\ \mathsf{cl2[m{\bullet}\!\!\to v2]}\}$
$\mathsf{Inh.}$:$\{\mathsf{x : cl1},\ \mathsf{cl2 :: cl1},\ \mathsf{cl1[m{\bullet}\!\!\to v1]},\ \mathsf{cl2[m{\bullet}\!\!\to v2]},\ \mathsf{x[m \to v1]}\}$
T_P^ω: $\{\mathsf{x : cl1},\ \mathsf{cl2 :: cl1},\ \mathsf{x : cl2},\ \mathsf{cl1[m{\bullet}\!\!\to v1]},\ \ \mathsf{cl2[m{\bullet}\!\!\to v2]},\ \mathsf{x[m \to v1]}\}$,

which yields an inflationary extension where postemption occurs: inheritance from cl1 *to* x *is postempted by the intermediate class* cl2 *although it has been justified (i.e., the trigger has been active). There is* no *"justified" model since inheritance is postempted exactly if it takes place. Note that this is not a* logical *inconsistency which would prohibit inheritance. Here, P has no extension; a similar* cyclic *inheritance network is given in [Hor, Sec. 2.3.1] as an example for a network which does not have a (credulous) extension.* □

In contrast to default inheritance in F-Logic this situation in (P2) is not relevant for DL knowledge bases. Here the class hierarchy is static, therefore no new classes will be generated during default inheritance. It is, however, conceivable that an F-Logic rule generates such classes during the hybrid reasoning process after the generation of a default. A default which has already been applied could become invalidated in such a situation. If default rules and deduction rules are likely to interfere because they make use of concept (class) terms which are related hierarchically among each other the rule sets have to be designed carefully.

The procedure for handling (P1) is different from the proposal in [MK01] where (P1) is avoided by fixing the inheritable property along the inheriting path. Fixing means that the property is passed on to each intermediate class down the class hierarchy, and an inheriting object only inherits from its direct superclass:

Definition 2.14

$$D^+_{inh} \ := \ \frac{O : C\ ,\ C[M{\bullet}\!\!\to V]\ :\ \neg\exists C'(O : C' \wedge C' :: C)\ ,\ O[M \to V]}{O[M \to V]}\ ,$$ □

The strategy of ordering of defaults comes to the same effect as long as only named individuals are considered for default inheritance. This is not a disadvantage, as default inheritance has to be limited to named individuals anyway. This was shown in [BH95] for default theories with underlying DL knowledge bases.

Rule ordering assures as well as path fixing that an instance inherits from its most specific superclass.

Example 2.9
Consider again Example 2.4. There are two defaults, penguin[fly•→false] and bird[fly•→true]. Both defaults share the same property. Rule ordering ensures that penguin[flies•→false] becomes evaluated before bird[fly•→true] as penguin is a subclass of bird. Therefore tweety *(and all other* penguins*) inherits* fly•→false*, whereas other birds than* penguins *will still inherit* fly•→true *in the next step.* □

Non-Determinism in Default Inheritance. There can be several different cautious inflationary extensions to a default theory, depending on the order in which the default inheritance rules are chosen for evaluation. Cautious inflationary extensions are therefore not deterministic, much the same as with normal defaults as has been shown in Example 2.2 with the Nixon-Diamon. See also Example 9.12 for another aspect of this limitation, which also motivates to use defaults in hybrid reasoning in the light of non-deterministic behaviour.

After having provided the formal introduction to this work, a comprehensive introduction to the basic concepts and terminologies of knowledge classification, knowledge bases, and the Semantic Web is following in the next chapter.

Chapter 3

Towards the Semantic Web

3.1 What is an Ontology?

Most scientific undertakings have one thing in common: they try to describe the world and find explanations for the observations. The term *ontology* (the science of what is) is relatively new and is first mentioned in the work *Ogdoas Scholastica* by Jacob Lorhard (Lorhardus) from 1606 and in the *Lexicon philosophicum* by Rudolf Göckel from 1613. But the practice of finding explanations for the world that we live in is much older, of course, and it was Aristotle, who gave the first scientific categorisation of the world that we know of. His *Categories* (or "Categoriae") enumerates all the possible kinds of things which can be the subject or the predicate of a proposition. He gave a very general categorisation of the world, dedicated very strongly to the idea of describing what there is. But it is a philosophical work, and it is doubtful whether it has any relevance to anybody outside the field of philosophy. Much later, in the late 17th and the early 18th century, we find efforts of a similar kind in the work of e.g. Gottfried Wilhelm Leibniz and Carl von Linné (Carolus Linnaeus). While the work of Leibniz, one of the last generalists, can be seen somehow in the tradition of Aristotle, Linnés work is a much more modern scientific work as far as categorisations are concerned. He did not try to describe the world in whole but just one particular section (here: botany and zoology). He used scientific classification based on attributes derived from morphology (e.g. characteristics of blossoms, leaf structures) instead of arbitrarily chosen domains (e.g. habitat of a life form).

But all these enterprises have one thing in common: their work reflects the urge of human thinking to find categories for what can be observed as well as explanations for why things are the way they are. Therefor the *primary task of ontology is to bridge the gap between what exists and the languages, both natural and artificial, for talking and reasoning about what exists* [Sow00].

Narrowing down the problem to one fundamental question "What is there?" allows the conclusion "To be is to be the value of a quantified variable." as it was given by W.V.Quine. But logic has no vocabulary for describing the things that exist. Here ontology comes into play: it is the study of existence, for example it gives the names and their relationships in biological taxonomies, it supplies the predicates of predicate calculus, it is even what people are doing when giving tags in social bookmarking systems. *What normally is known as an ontology can* [thus] *range from the simple notion of a taxonomy to a thesaurus to a conceptual model (with more complex knowledge), to a logical theory (with very rich, complex, consistent, meaningful knowledge). They differ in the strength of their semantics* [DOS03].

It was in 1980 that the usage of the term *ontology* was introduced into computational sciences by John McCarthy in the discipline of artificial intelligence. He argued the necessity of a list of things from common knowledge, to *add categories into our ontology (the things that exist)* [McC80] in order to be able to reason about them. The term has been used intensively since then, e.g. in the work of Pat Hayes [Hay79] and John Sowa [Sow], but the meaning has shifted significantly. The most common definition of ontology nowadays is that of a *specification of a conceptualisation (of knowledge about a domain)* [Gru93]. Although this definition by itself is not very likely to reduce the confusion about what an ontology really is, it is clearly a definition that aims at practical use. B. Smith pointed out [Smi] that *philosophical ontology does not seek predication or explanation, but rather taxonomy. It is a descriptive enterprise, it is (very largely) qualitative* whereas *science is (very largely) quantitative and starts with measurement and prediction.* In computational sciences the focus is largely on practical aspects: building a shared vocabulary that describes a domain, so that humans and, even more important, machines can be sure that, when talking about things, they talk about the same things. To put it more technically, an ontology can be built by giving a formal explicit description of concepts (classes) that have properties (slots, roles) describing various attributes of the concept. The concepts give the structure of an ontology whereas the *things* of the world are given as facts. These facts belong to classes, they have properties and they have relationships amongst each other.

For instance, a biological ontology for the classification of animals might describe groups of species as concepts (e.g. mamals), specify attributes which all members of these groups will exhibit (fur, vivipary, milk glands) and give instances of these concepts (bear, sloth, bat). Conceptual design tries to identify and reflect structures of a domain. Sometimes, however, there are irregularities that also have to be represented. For example, whales are mamals but are not covered with fur. Plathypus belongs to mamals but lays eggs. See Section 2.5 for an introduction to default inheritance which deals with this matter.

Another modelling challenge is about how to describe the dynamic aspects of a domain. The world is changing, therefore the data that describe the world will most likely become subject to change as well. Relationships and properties of instances might change, concepts be refined or deleted.

One important thing is the ability to reason about change. Such reasoning tasks could be [AF94]

- prediction (what will (or is most likely to) happen),
- planning (how to achieve a desired goal),
- explanation (find some explanation for a set of data).

To fulfill these tasks, it is necessary to analyse how changes occur, what formalisations exist in order to express the dynamic aspects of knowledge bases (and the world) and what mechanisms can be used to be able to reason about them.

Examples of Ontologies in Use

Nowadays, there are many ontologies in use. One of the oldest that still has major significance in science, has already been mentioned: it is Linné's taxonomy of botany, *Species Plantarum*, published in 1753. Here he first introduced a binary nomenclature for categorising species. Another example is the periodic table of the chemical elements, devised independently in the 1870s by Dimitri Mendeleev and Lothar Meyer. The periodic table lists all known chemical elements in an ordered way. The similarity to Linnae's taxonom is obvious, using species as atoms of biological classification.

The success of these ontologies is largely due to the fact that the domains that are being described are relatively small (not in terms of the number of affected individuals but of the concepts). Furthermore, there are strict formal categories that help to build the ontology.

It is relatively easy to perceive the idea that atoms should be ordered according to the mass of the nucleus (or the number of protons to be more precise). But what about the ordering and relationships of animals? A modern taxonomy tries to identify features that reflect the evolutionary relationship of species. This classification ranks the degree of similarity between whales and rodents much higher compared to the similarity between whales and sharks. Although in the latter comparison both groups of species live without exception in a similar habitat, it makes much more sense to put rodents and whales in a closer relationship. There is a lot of morphological evidence, which shows that whales and rodents share a common ancestor which is much more recent than one that could be found for whales and sharks. Hence, there can be different ontologies about the same fragment of the world, depending on the intentions and knowledge of the creator. For example, it is also possible to conceive ontologies which categorise animals by their habitat (saltwater-living, freshwater-living, land-living, airborne, ...) or maybe even by their taste.

But, with increasing size and complexity of a domain it is likely that arbitrary choices will be made during conceptual design. A good example to support this argument is a common classification scheme that is used in libraries known as the *Dewey Decimal System*. It was founded by Melvil Dewey in the 1870s and remains quite popular to this day. It is an ontology much in the classical tradition of Aristotle: it tries to find categories for all which is there (or in other words everything we have a book about in a library). But the distinctions that were made for classification are quite arbitrary, the top level categories concerning religion are one good example:

Example 3.1 (Library classification schema for religion)

210	*Natural theology*
220	*Bible*
230	*Christian theology*
240	*Christian moral & devotional theology*
250	*Christian orders & local church*
260	*Christian social theology*
270	*Christian church history*
280	*Christian sects & denominations*
290	*Other religions*

□

It can hardly be argued that this classification is biased with regard to religions of the world or the term religion in general. Rather, it reflects what the founder of the ontology thought was important to know about religion and, moreover, what he himself actually had knowledge about (books) to classify. Other, more modern ontologies like cyc, wordnet or dmoz also try to describe the world as a whole. But the more an ontology tries to describe of the world, the harder it is to agree on categories. An attempted solution is bringing together domain experts who know much about a certain field of knowledge together with experts in designing knowledge representations. It is hard work, and there is no guarantee that the outcome will be accepted.

It is interesting to observe that ontologies have recently evolved that are based solely on user contributions and are quite successful. In social bookmarking systems like del.icio.us as well as in *recommender systems* like last.fm or *people who ordered this book also showed interest in this* by amazon, the categories are not fixed. They are given by tagging: users provide categories that they think are appropriate. The more users contribute, the more precise and meaningful the ontology will become. This is interesting as the ontologies are not developed by a small number of experts but by a large number of untrained users, who most likely will not have any knowledge of ontology development.

But there are also some successful ontologies that are being developed in the traditional fashion by experts.

- http://www.geneontology.org/ Geneontology
- DSM-IV, the 4th version of the psychiatrists' Diagnostic and Statistical Manual, is a classic example of a classification scheme that works because of these characteristics. In theory DSM IV allows psychiatrists all over the United States to make the same judgment about a mental illness when presented with the same list of symptoms. Another widely used standard for the classification of diseases is ICD-1O which is issued by the World Health Organisation.

Ontologies and Computational Sciences

Ontologies in computational sciences have been emerging since the 1980s. They found application in different disciplines such as AI, Software Engineering (domain modelling, especially UML) and Databases (conceptual modelling). Although they are used quite frequently most users would probably not know they are using an ontology. That is because the term ontology is relatively new and uncommon. In each of these areas developers are faced with the problem of building an artifact that represents some portion of the world in a fashion that can be processed by a machine [Wel03]. Conceptual modelling has therefor become a discipline in itself. But not only the question of how to model the concepts of a domain is of importance. Moreover, what are the prerequisites for the use of such conceptualisations? The advent of the World Wide Web adds a new dimension to that problem. Data is now accessible in a completely new way from that of only two decades ago. And there are data for every purpose. Natural sciences are a very good example. Molecular biology unveils vast amounts of data every day, many of them are added to gene or protein databases that can be used by scientists all over the world. The problem is no longer how to get the data, but how to use it, how to find the *relevant* information. Because of the sheer number of data it is not possible to filter by hand, to let scientists search, select, and collect by themselves. It has to be a process done by machines. And it is a process that can be done by machines. But there has to be more than knowledge, there is a need for meta-knowledge. Meta-knowledge provides for the structure and the vocabulary that is needed to describe knowledge and define the relationships of the things of the world. That meta-knowledge also has to be provided by experts in their fields, but the technical preliminaries will be given by computational sciences.

The Semantic Web is expected to provide an infrastructure where both knowledge and meta-knowledge can be specified, stored, and used.

3.2 Knowledge in the Web

The idea of the Semantic Web was brought into play by Tim Berners-Lee. The World Wide Web provided mankind with the most remarkable new technology that is probably comparable in importance only to the invention of printing. Knowledge became widely distributed and accessible. The availability of such enormous amounts of data make it desirable to enable machines to process that knowledge, to combine, integrate and reason about data. To that end it is necessary to have a certain understanding of what the data is about, to have a semantic level. Until now, this level of semantic is available only in a small fragment of the Web: in recommender systems, social bookmarking, thesauri, yellow pages. But there is no *general* annotation of knowledge in the Web, no classification of content.

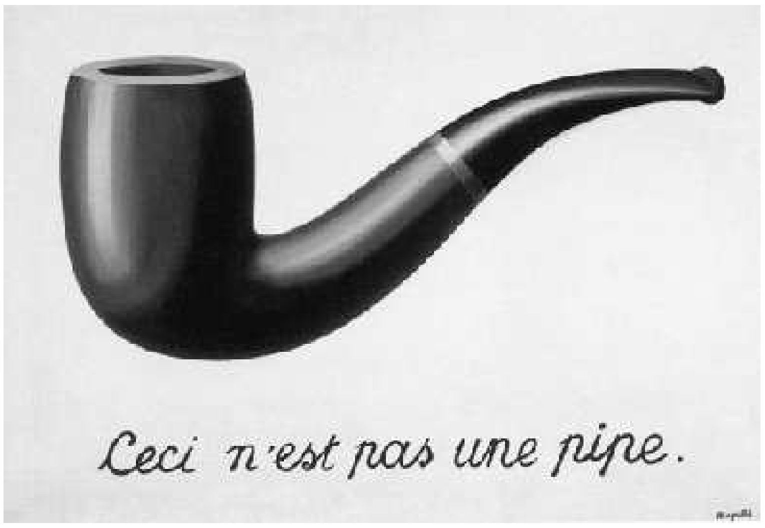


Figure 3.1: René Magritte: "Ceci n'est pas une pipe" [1]

The situation is quite similar to something that we all have experienced many

[1]© VG Bild-Kunst Bonn, 2009, used with kind permission.

times in our lifes. Without previous knowledge we are not able to get the meaning out of the things that surround us. The experience of art is a good example: when we first see the picture *Ceci n'est pas une pipe* (Figure 3.1) by René Magritte it is not obvious what's the artists intention. Reflecting might help to find out, but only if we are aware of the play on words that it contains. If we are already provided with a background on surrealism we might get the intention of the artist who tries to reflect about reality.

Whenever we are confronted with something that we know nothing about we have to find categories that we can use for interpretation. Either we adopt existing knowledge to the situation or we integrate new knowledge that helps us to understand. Learning is very much about building new categories, relating new knowledge to existing knowledge. With this analogy in mind it is probably easier to understand what computers cannot do but what we want them to be able to.

The vision is to have machine learning, knowledge representation (that enables machines to reason), and distributed, annotated knowledge. Many of the ingredients are already there: the World Wide Web and the different protocols and markup languages for data interchange resulting from decades of research in artificial intelligence, databases, conceptual modelling, even computer linguistics, so many fields of computational sciences seemed suitable to make their contribution. It is of course not as easy as just putting everything into a pot and stirring it up. But the overall concept is this: data will be (as it already is in the World Wide Web) distributed over the world at many physical locations. Giving access to the data is a matter with which the World Wide Web already deals, moreover, data can be addressed and uniquely identified (see Section 3.4). Ontologies are now able to define the metadata (the concepts and their relationships) (see Section 3.6). This is the semantic part in the Semantic Web, added by conceptualisation and annotation. The Semantic Web, however, is not static. Dynamic aspects of changes in knowledge bases have been dealt with already in database research (e.g. in active database research). But the different aspects of the Semantic Web add up to new problems, because knowledge is now distributed, probably from untrusted or unknown sources, and sometimes inconsistently annotated. There are lots of questions that have to be dealt with, some of them here in this work.

Before going into the detail of the problems at hand, a brief introduction to the development of knowledge management is provided.

3.3 Managing Knowledge

Databases

Knowledge managment has many different aspects, the importance of conceptualisation has already been discussed. One prerequisite for knowledge management is therefore the ability to store data in a structured way. The usual way in database design is to start with conceptual modelling, which normally means developing an *Entity-Relationship* diagram (ER diagram). The result is a conceptualisation of what kind of things there are (concepts of the domain of interest) and the relationships, dependencies and constraints that apply to them. This is an ontology. If the conceptual model is well defined, it is easy to proceed and transform the concept definitions into a database model.

One of the most successful database models is the relational model by Codd [Cod70]. It is based on set theory and defined in terms of predicate logic. Like in the ER model, the design of a relational database requires identification of entities of the world as well as their relationships among each other. That similarity makes it easy to transform an ER diagram into a set of relational tables. Entities and relationships may also have attributes. A relational schema is, by definition, a set of attributes, a relational signature the set of relational schemas. If we add some facts (=data) to the schemas, we have a relational structure: the state of a database. The semantics of the database is the current database state.

The advantages of the relational model are its simplicity and the well-known theoretical background from set theory. However, one disadvantage is the lack of expressiveness, but normally relational databases can be extended through procedural languages.

Besides storage of data a database has to give access to the stored data in a standardised way. For that purpose there exists *SQL*, the structured query language. SQL allows for the definition of queries to a relational database, also data manipulations can be expressed. Thanks to this uniform access mechanism it is possible to use the same queries independent from the database implementation.

From Databases to Knowledge Bases

Generally the term knowledge base refers to a collection of knowledge. This can be a compilation of articles or manuals, a cardbox, a library, but also a computer programme. A knowledge base has well-ordered content and offers sophisticated information retrieval. In the following knowledge bases are discussed in consideration of their relevance in computational sciences.

A knowledge base structures the knowledge of the domain of interest by means of an ontology. Additionally it uses inference mechanisms which allow for the derivation of additional knowledge. These derivations can be achieved by deduction rules or axioms. This is an aspect which is missing from typical databases.

Example 3.2
A knowledge base contains the information hasChild(Susan, Peter). *This reads as* Susan *is related to* Peter *by a hasChild relationship. It can be infered that* Susan *is a parent by the following rule:*

$$hasChild(X, Y) \rightarrow parent(X) \wedge child(Y)$$

The rule reads as: if X and Y are related by a hasChild *relationship then X is a* parent *and Y is a* child. □

In databases there is a strict separation of data and metadata. This is different with knowledge bases. Information can be given in form of new facts but also as new concept definitions or derivation rules. Whereas the structure of a database is static the structure of a knowledge base can change easily.

In order to encode the schema along with the data, a more sophisticated data model is required. If the data model allows arbitrary information to be encoded, then it can also be extended by axiomatic formulas or rules. This is the usual situation in knowledge bases.

Another important difference between databases and knowledge bases is reflected in the attitude towards the knowledge that is available about the world. In databases the Closed World Assumption (CWA) is applied. The database point of view is that everything that can be known is already known. Everything that cannot be answered, is false. Altough this could be regarded inappropriate it is a reasonable approach as long as this *closed world* contains all necessary information.

In knowledge bases, the Open World Assumption (OWA) is applied, which means that information is (always) incomplete. If something is not known (yet), it might exist in some part of the world, so it is not necessarily false.

Hereby, however, a knowledge base reflects a situation that is very much like the world that we live in: often it seems inappropriate to assume that the lack of information is equivalent to negative information. If nothing is known about Susan it would be a rather premature decision to say whether she is a parent or not. Rather, we assume that some additional knowledge might be available somewhere. This conception has many advantages, but there are situations, where it is eligible to treat an open world as a closed world and a decision has to be made upon the facts that are known. This can be done by an *epistemic queries* which is an interpretation such that everything that is not known has to be false.

Another difference between databases and knowledge bases adds to this. In databases, constraints are limitations. In knowledge bases, however, they are (type-) definitions. Together with the assumption that information might exist somewhere and is only not known by now, there might be unexpected effects or even (to a human) ridiculous conclusions.

Example 3.3
The ontology of our knowledge base states that every person has a father, hasFather is a functional property (there can be only one father). If hasFather(Peter, John) and hasFather(Peter,Jack) are added there is no contradiction although there should be only one father to a person. This can be explained by the fact that the knowledge base assumes that Jack and John are the same person[2]. The functional property is not used as a constraint but rather as an assertion. As both Jack and John are the father of Peter plus the fact that there can be only one father they have to be the same in order not to cause an inconsistency. □

So far, knowledge bases were presented but no formalism for knowledge representation has been introduced yet. In the Semantic Web, RDF is expected to become a standard in this respect.

3.4 RDF: A Data Model for the Semantic Web

The *R*esource *D*escription *F*ramework (RDF) [RDF00a]) is a very popular data model in knowledge representation. It is a simple, logical model that is used to

[2]This is only true as long as there is no unique name assumption.

represent things (either real or imaginary) as resources. Resources are connected by properties that again are resources. RDF has become a W3C recommendation in 2004. The data model of RDF is a directed graph structure, where every single node is a resource that is connected to other resources. The connecting edges are labelled and correspond to the properties of a resource. Every expression is encoded in (sets of) triples of the form *subject predicate object*. Such triples are called *statements*. Subject and object are nodes in the graph structure. It is easy to see that both data and metadata can be encoded using the triple structure: :Peter :hasMother :Susan is a simple statement about the relationship between two instances, from the point of view of the graph structure also the schema information :hasMother rdf:type rdf:Property are simply edges in a graph which are labeled with the resource identifier of the property.

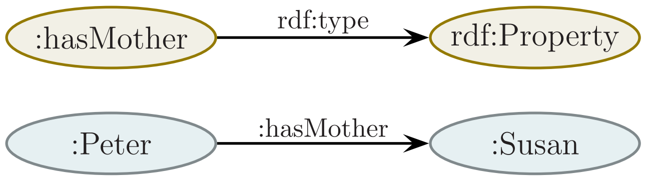


Figure 3.2: Graph Structure of (:Peter :hasMother :Susan)

One basic design element in RDF are *uniform resource identifiers*, abbreviated and more commonly known as URIs. The nodes and the labels of the edges can be identified by URIs, with the exception that the object of a statement can also be a literal. URIs are like addresses and in fact they are used as such in the Web: URLs (uniform resource locators) are a special kind of URIs, the notion *locator* indicates that URLs are meant to point to existing locations, reachable using the HTTP protocol. URIs are more general: for instance, the URI http://family.org#hasMother consists of a *protocol specification* (http://) followed by the *hierarchical part* (family.org) and finally the *fragment* (#hasMother). This is not a URL because it does not point to a valid address in the Web. In ontology design, the *fragment* usually identifies the local name of a resource (a concept, a property, or an individual name) whereas the rest of the URI is given as a *namespace prefix*. Nevertheless this is just a way to improve readability for human readers, the name of the resource is the full URI. This is also reflected in the unusual diction using colons in the example above, where the default namespace (:) and the RDF namespace (rdf:) have been used (but no definitions for the namespaces have been given). Although URIs need not point to an existing location, they often do. Consider the following example, where the resource identifiers are given in full length:

Example 3.4

```
(  http://example.org#Peter
   http://family.org#hasMother
   http://example.org#Susan  )
(  http://family.org#Susan
   http://www.w3.org/1999/02/22-rdf-syntax-ns#type
   http://family.org#Mother  )
```

The example uses the built-in predicate type from the RDF namespace *http://www.w3.org/1999/02/22-rdf-syntax-ns#*. This URI points to a real document on the web where the resource is defined:

```
<rdf:Property
     rdf:about="http://www.w3.org/1999/02/22-rdf-syntax-ns#type">
  <rdfs:isDefinedBy
     rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#"/>
  <rdfs:label>type</rdfs:label>
  <rdfs:comment>The subject is an instance of a class.</rdfs:comment>
  <rdfs:range
     rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
  <rdfs:domain
     rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource"/>
</rdf:Property>
```

The other URIs in Example 3.4 do not point to an existing document. As long as an URI is valid (wellformed) and unique, it is possible to use this resource in any ontology and all information about the same resource adds up to a whole if those ontologies are brought together. It is important to bring to mind that ontological data is not necessarily local, it can be distributed over many different locations.

RDF provides the user with a set of built-in predicates like rdf:type. This predicate states that the subject of a statement is a member of the class which is specified by the *object* of the statement. Furthermore there are constructs for collections like bags, sequences, and lists.

There are different ways in which RDF graphs can be serialised for output, storage and communication. For example, RDF can be serialised in XML which is especially useful for the interchange of data. More suitable in terms of readability is N3. The Example 3.4 is given as serialisations to N3 and RDF/XML:

Example 3.5 (Extending Example 3.4 using namespace prefixes)

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix family: <http://family.org#>.
@prefix : <http://example.org#>.
:Peter family:hasMother :Susan.
:Susan rdf:type family:Mother.
```

Example 3.6 (The same RDF data, serialised as RDF/XML)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<rdf:RDF
     xmlns="http://example.org#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:family="http://family.org#" >
  <rdf:Description rdf:about="http://example.org#Peter">
    <family:hasMother
      rdf:resource="http://example.org#Susan"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://example.org#Susan">
    <rdf:type rdf:resource="http://family.org#Mother"/>
  </rdf:Description>
</rdf:RDF>
```

In Example 3.5, there are two statements, but one seems to be contained in the first. If :Peter has a mother :Susan, :Susan is of course a mother. This derivation can be made easily by common sense knowledge, but it is not possible to draw that conclusion logically from these RDF statements alone. For that purpose, there exists a schema layer on top of RDF, named RDF Schema (or RDFS) [RDF00b] which adds further concepts for the *description* of knowledge, or, in other words, for the modelling of meta-data.

Example 3.7
RDFS specifies additional axioms that can be used to define assertions. (N3 notation allows to use semicolons to separate property-value pairs on the same subject).

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix family: <http://family.org#>.
family:hasMother   rdf:type      rdf:Property  ;
                   rdfs:range    family:Mother  ;
                   rdfs:domain   family:Child.
```

Now it is defined that every object of a hasMother relationship is of type family:Mother. The domain axiom from the RDFS namespace makes it possible to give an assertion with regard to the subject of a relationship: if any resource has a family:hasMother relationship, it is known to be a family:Child. Note that in contrast to databases these restrictions are assertions: they add knowledge to a resource instead of putting constraints on relationships. If, for example, the statement (:Susan family:hasMother :Peter) is added to the facts of the example this is no contradiction. Neither is there an irreflexive definition for the family:hasMother relationship, nor exists any information which leads to a contradiction when :Peter becomes a mother (although, in fact :Peter will never be a mother).

Example 3.7 implicitly defines the resources family:Child and family:Mother to be classes (concepts). This deduction can be made because both the rdfs:domain property and the rdfs:range property relate properties to classes.

Example 3.8 (Mothers and Children are Persons)

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix family: <http://family.org#>.
family:Mother   rdfs:subClassOf   family:Person.
family:Child    rdfs:subClassOf   family:Person.
```

Hereby instances of family:Mother or family:Child become also instances of family:Person.

RDF data can easily be put together, it is simply a merging of graph structures that connect at identical nodes. They can be split, separated, distributed, and supplemented in any fashion. Therefore, it makes no difference, if the facts are given in separated files (as above in Examples 3.5, 3.7 and 3.8) or in one file like in the following example. In the course of this work many of the examples will not display the whole contents of a knowledge base but just add some additional details to existing facts (previous examples). In such cases the dependencies will be mentioned.

Example 3.9 (Examples 3.5, 3.7 and 3.8 in one file)

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix family: <http://family.org#>.
@prefix : <http://example.org#>.
```

```
:Peter  family:hasMother  :Susan.
:Susan  rdf:type  family:Mother.
family:hasMother   rdf:type      rdf:Property  ;
                   rdfs:range    family:Mother  ;
                   rdfs:domain   family:Child.
family:Mother   rdfs:subClassOf   family:Person.
family:Child    rdfs:subClassOf   family:Person.
```

Note that the explicit rdf:type information for :Susan is redundant as it can be derived from (:Susan rdf:range family:Mother). □

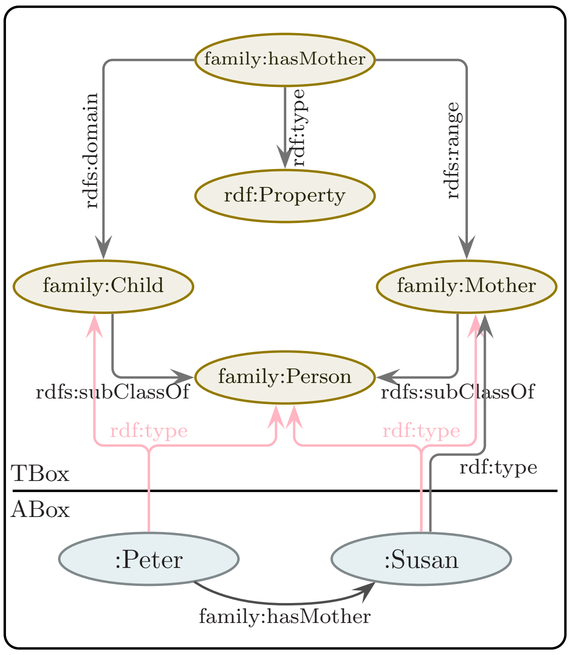


Figure 3.3: Graph Structure of an RDF/RDFS Knowledge Base

The data from Example 3.9 is presented as a graph in Figure 3.3. The upper part of the diagram shows the terminological knowledge, which is here called a TBox in analogy to Description Logic knowledge bases (see Section 2.2 for a formal introduction to Description Logics). The TBox is comparable to the schema information in a relational database. The lower part contains assertional knowledge (ABox) where all information about instantiations of concepts belong. The notion of an ABox is again a reference to Description Logics. Note that this separation of TBox and ABox is only theoretical. In most knowledge bases all information is contained in the same graph. The gray edges in the example above represent explicit, the red edges implicit knowledge. Implicit knowledge can be derived by a reasoning engine. It is not contained as real statements but it can be obtained by queries to the reasoning engine. The subject of queries to Semantic Web data is now investigated.

3.5 Querying Semantic Web Data: SPARQL

A query language for the Semantic Web has to be a query language for RDF data because this is the most prominent data format in the Semantic Web. There have been several proposals for Semantic Web query languages (see [FLB+06] for a comparison).

The most widely known and applied is SPARQL [SPQ06]. It is strictly a query language, which makes it much less powerful a language compared to SQL, which is, besides querying, also a data definition and manipulation language. Syntactically SPARQL and SQL are quite similar.

Essentially, the query is a conjunction of graph patterns (given with the *where clause(s)*) which define a subgraph. From that subgraph the answer is extracted according to the answer variables that are given in the *select-clause*. Furthermore, it is possible to specify in a *from-clause* what sources will add to the graph on which the query will be evaluated.

Example 3.10

```
SELECT ?person ?class
FROM <file:peterhasmother.n3>
FROM <file:hasmotherdomain.n3>
FROM <file:personsubclasses.n3>
WHERE {{ ?person
          <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
            ?class .
            ?person
           <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
           <http://family.org#Person> .
       }}
```

This example reads data from three files (containing the facts from examples 3.5, 3.7 and 3.8) and returns tuples containing the names of all resources that are found to be instances of the class http://family.org#Person *along with the names of all classes that they are instances of.* □

The answer to the query, using a reasoning engine, is:

```
| person                        | class                        |
================================================================
| <http://example.org#Susan>    | <http://family.org#Person>   |
| <http://example.org#Susan>    | <http://family.org#Mother>   |
| <http://example.org#Peter>    | <http://family.org#Person>   |
| <http://example.org#Peter>    | <http://family.org#Child>    |
```

There are some further features in SPARQL for more sophisticated result set operations. Using *UNION* inside the *where clause* builds a set union of the subgraphs as specified by the given triple patterns. Another combination of result patterns can be given using the *OPTIONAL*-keyword which allows to add further graph patterns. This is no conjunctive but an optional combination (corresponding to the left outer join in SQL).

Example 3.11

```
SELECT ?person ?mother
FROM <file:peterhasmother.n3>
FROM <file:hasmotherdomain.n3>
FROM <file:personsubclasses.n3>
WHERE { ?person
         <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
         <http://family.org#Person> .
         OPTIONAL{?person <http://family.org#hasMother>
                   ?mother.}}
```

The answer to that query shows that the variable *mother* does not need to be bound to a value in order to make the tuple appear in the result set:

```
| person                      | mother                      |
=============================================================
| <http://example.org#Susan>  |                             |
| <http://example.org#Peter>  | <http://example.org#Susan>  |
```

Another feature can be used with the *FILTER* keyword which allows to define conditional expressions inside the *where-clause*. The combination of *FILTER* and *OPTIONAL* enable negation in SPARQL queries which is not available otherwise. It is a form of negation-as-failure.

Example 3.12

```
SELECT ?person ?mother
FROM <file:peterhasmother.n3>
FROM <file:hasmotherdomain.n3>
FROM <file:personsubclasses.n3>
WHERE { ?person
         <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
         <http://family.org#Person> .
         OPTIONAL{?person <http://family.org#hasMother>
                   ?mother.}
         FILTER( !bound(?mother))}
```

This query selects all family:Person *instances and filters those that have no* family:hasMother *relationship. Hence, only* :Susan *will be returned in the result set.* □

SPARQL offers an intuitive way of querying RDF knowledge bases (or, more generally speaking, the Semantic Web). Though the query language design is intended to be similar to SQL, it is not as expressive. For example, it does not offer nested queries. It is possible to generate new RDF from the query using the *CONSTRUCT* keyword but these facts cannot be fed back into another query. Also, negation is only available in the form of the above mentioned workaround but not as a semantically clear language feature. But then SPARQL is still under development, both in theory and implementation, so some features might be available sooner or later.

It has already been shown that knowledge can either be given by explicit statements about what exists or by implicit definitions of what can be derived (see Example 3.9). In the Semantic Web, the most interesting language candidate for the specification of implicit knowledge is OWL, which is described in the following.

3.6 OWL and Friends: Reasoning in the Semantic Web

A short introduction to reasoning has already been given. Basically, it is the combination of data, metadata and a reasoning engine. To illustrate this, consider again the query from Example 3.10. Without inference support the query will yield no result. That is because there is no statement saying explicitly that :Peter is an instance of the class family:Child and there is no statement that allows to know that a family:Child is a family:Person. The information is given implicitly, therefore an inference engine is needed to make the additional deductions. With the appropriate inference mechanisms the results can easily be given as:

```
| person                        |
=================================
| <http://example.org#Susan>  |
| <http://example.org#Peter>  |
```

The language features of RDF together with RDFS allow for definitions of basic type information and constraints, but further aspects of metadata modelling like cardinality restrictions and datatype reasoning, enhanced object properties like reflexivity, symmetry, transitivity, or functionality are desirable. In the late 90s of the last century two independent projects emerged in order to develop the first ontology description language for the Semantic Web. One was named DAML *(Darpa Agent Markup Language)*, funded by the DARPA, while OIL *(Ontology Inference Layer)* was funded by the European Union under the On-To-Knowledge project.

OIL

OIL was based on Description Logics (DL) and the frames paradigm (see again Section 2.2 for a formal introduction to F-Logic). Frames help both designers and users of ontologies to read an ontology by grouping information by classes. Description Logics have complex class construction mechanisms. Furthermore, questions of decidability of the language components have been thoroughly investigated. It is possible to add distinct features, such as transitive roles or inverse roles while still having a decidable fragment. Also, the limits of decidability are well-known, e.g. with the usage of nominals the description logic theory is still decidable but reasoning is of worst-case non-deterministic exponential time (NExpTime). Another argument in favour of DL was that algorithms for reasoning were already at hand. Developers of OIL provided the users with a reasoner implementation, the FaCT reasoner [Hor98]. FaCT development is still ongoing work (now called FaCT++), but to date with focus on OWL reasoning. Although OIL had both XML and RDF syntaxes, the semantics of RDF has been neglected.

DAML-ONT

DAML was, much like OIL, an intermediate state in the development of an ontology description language. DAML-ONT was designed to extend RDF with language constructors from object-oriented and frame-based knowledge representation systems [HPSvH04], or, in the words of the developers, it is simply a vocabulary of properties and classes added to RDF and RDF Schema [MF02]. The main focus was therefore on web data integration which is the use of XML-serialisation for RDF and extensions to RDF Schema by more expressive descriptors. Though the language was more expressive, there was much criticism about the ambiguity and the lack of formal specification [MF02].

DAML+OIL

It became clear that the capabilities and goals of DAML-ONT and OIL were very similar. Therefore, already in 2000, joint work begun on DAML+OIL. Efforts from OIL added to the formal background. Research on Description Logic has shown that the fragment $\mathcal{SHIQ}$ is expressive enough (e.g. including the much desired role transitivity) whilst still being tractable. The basis is the $\mathcal{ALC}_{\mathcal{R}^+}$ Description Logic, which already has transitive roles and is also named $\mathcal{S}$. Adding a property hierarchy ($\mathcal{H}$), inverse roles ($\mathcal{I}$) and generalised cardinality restrictions ($\mathcal{Q}$) gives the $\mathcal{SHIQ}$ Description Logic. While the foundation on description logics can be seen as a contribution by OIL, the integration with RDF and therefore towards a Semantic Web based on RDF and RDFS, was backed by work on DAML-ONT.

OWL

Eventually a new language was designed: the Web Ontology Language (OWL) [OWL04]. Semantically, a fine-tuned design was chosen: in order to retain decidability, OWL is offered in three different versions with increasing expressivity: OWL Lite, OWL DL and OWL Full. The semantics of OWL Lite and OWL DL is based on Description Logics (DL). At the time when OWL was introduced, there was ten years of research on Description Logic, which mapped out in considerable detail the complexity-tractability landscape for a wide range of constructors and axioms and their various combinations [HPSvH04]. In Section 2.2 a (formal) introduction to DLs can be found. Although OWL Lite and OWL DL ontologies make use of the RDF Schema vocabulary they are not fully compatible (semantically) with RDF Schema. The relationship between OWL and DLs is described in Section 2.3. OWL Full is designed to support the full expressiveness of RDF Schema. This variant of OWL, however, is not decidable.

In DLs, there is a distinction between the so-called ABox (assertional) and TBox (terminological). The ABox contains facts (individuals) whereas the TBox contains the definitions of concepts and relationships as well as the vocabulary of the language itself. Analysing a typical OWL ontology, there is no such distinction. The vocabulary needs not be given (it is known to the knowledge base or the reasoner), and concept definitions and individuals are mixed. Nevertheless, the programmes used for OWL reasoning are in fact DL reasoners (e.g. FaCT++ [FaC], Pellet [Pel]) and do use such a distinction internally.

OWL Lite

OWL Lite is the most simple OWL dialect. The feature set of OWL Lite is equivalent to the Description Logic $\mathcal{SHIF}(D)$. Reasoning is possible in worst-case deterministic exponential time (ExpTime). Most of the available OWL reasoning engines cover OWL Lite. Neither reification is possible nor is it allowed for instances to also be classes (both of which are possible in RDF). These limitations guarantee decidability. The following features are offered by OWL Lite:

- restrictions on classes that are in fact assertions of class membership in case that an individual is assigned a certain (restricted) property (see Section 3.3).
- In the Semantic Web, two resources are not assumed to be different unless otherwise stated (no *unique name assumption*).
- It is possible to declare equality or inequality of resources (individuals).
- Inverse properties.
- Transitivity of properties can be defined with owl:TransitiveProperty.
- Symmetric relationships can be defined with owl:SymmetricProperty.
- Constraints on the range of properties to specific classes can be given with owl:allValuesFrom and owl:someValuesFrom. The following example illustrates these constraints: if a member of the class family:ParentWithDaughtersOnly has a family:hasChild relationship then the object of this relationship is a family:girl. Likewise the definition of the class family:ParentWithAtLeastOneDaughter: at least one family:hasChild relationship has to be related to a family:girl. Hence :Lisa has to be a family:Girl.

```
family:ParentWithDaughtersOnly owl:equivalentClass
[ a owl:Restriction;
  owl:onProperty family:hasChild;
  owl:allValuesFrom family:Girl
].
family:ParentWithAtLeastOneDaughter owl:equivalentClass
[ a owl:Restriction;
  owl:onProperty family:hasChild;
  owl:someValuesFrom family:Girl
].
family:Girl a owl:Class.
:Peter family:hasChild :Lisa;
 a family:ParentWithDaughtersOnly.
```

- Furthermore, it is possible to define cardinality restrictions on properties with owl:minCardinality or owl:maxCardinality, but in OWL Lite only to the value 1. Note that owl:maxCardinality 1 is equivalent to a owl:FunctionalProperty.
- *Simple XML Schema* datatypes can be used [OWL]. These datatypes comprises of xsd:string,xsd:boolean,xsd:decimal,xsd:float,xsd:double plus some further datatypes derived hereof. Moreover there are time-related datatypes.
- Properties can be defined to be either an owl:ObjectProperty or a owl:DatatypeProperty.

Not having the unique name assumption has significant consequences on the design of knowledge bases or in query design. Consider the following example:

Example 3.13 (Does Peter have another mother?)

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix family: <http://family.org#>.
@prefix : <http://example.org#>.
family:hasMother a owl:FunctionalProperty.
:Peter family:hasMother :Katrin.
```

If the knowledge base from example 3.5 is used together with the above mentioned example, the knowledge base is still consistent, although the family:hasMother relationship is defined as functional (only one family:hasMother relation is possible). The open world assumption allows us to deduce that :Katrin and :Susan have to be the same as long as no other knowledge disproves this conclusion explicitly. *owl:FunctionalProperty* is a subclass of *rdf:Property*.

With the additional knowledge from the next example it is known that :Susan and :Katrin are different, therefore the restriction on the property family:hasMother is violated and the knowledge base becomes inconsistent. The *owl:sameAs* axiom can be used to express the equality of classes.

Example 3.14 (Invalidate the son of two mothers)

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <http://example.org#>.
:Susan owl:differentFrom  :Katrin.
```

The use of inverse properties is straightforward as illustrated by the next example: The family:hasParent relationship is inverse to the family:hasChild relationship. The family:hasMother property is an rdfs:subProperty of family:hasParent. Therefore, every resource A that has a family:hasMother relationship to resource B also has a family:hasParent relationship to B. If resource A is related by family:hasParent to resource B then is implied that B is related by family:hasChild to A. Note that B family:hasChild A does not imply A family:hasMother B as it is not yet known whether B is a family:Mother or a family:Father.

Example 3.15 (Inverse Properties)

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix family: <http://family.org#>.
@prefix : <http://example.org#>.
family:hasChild a owl:ObjectProperty;
    owl:inverseOf family:hasParent.
family:hasMother rdfs:subPropertyOf
    family:hasParent.
```

Moreover, properties can be defined to be transitive. If A p B is true and also B p C is true it can be derived that also A p C is true. An intuitive example is given by the definition of the *hasAncestor* property:

Example 3.16 (Transitive Properties)

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix family: <http://family.org#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix :<http://example.org#>.
family:hasAncestor
         a owl:ObjectProperty;
         a owl:TransitiveProperty;
         rdfs:domain family:Person;
```

```
            rdfs:range family:Person.
family:hasMother rdfs:subPropertyOf family:hasAncestor.
:Susan family:hasMother :Katrin.
```

The following query shows that also :Peter is an ancestor of :Katrin, although it was not stated explicitly.

Example 3.17 (Query for transitive relationships)

```
SELECT ?person ?ancestor
FROM <file:peterhasmother.n3>
FROM <file:familytransitiverelationships.n3>
WHERE { ?person <http://family.org#hasAncestor>
          ?ancestor . }
```

This query allows to find the following results:

```
| person                      | ancestor                    |
==============================================================
| <http://example.org#Susan>  | <http://example.org#Katrin> |
| <http://example.org#Peter>  | <http://example.org#Katrin> |
| <http://example.org#Peter>  | <http://example.org#Susan>  |
```

Symmetric relationships can also be explained easily. If A p B is true then A q B is also true if p and q are symmetrical.

Example 3.18 (Symmetric relationships)

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix family: <http://family.org#>.
@prefix : <http://example.org#>.
family:hasRelative a owl:SymmetricProperty.
:Susan family:hasRelative :Peter.
```

OWL DL

OWL DL offers some additional features, which are:

- Arbitrary cardinality restrictions
- *owl:disjointWith* allows to express that two classes have no instances in common. *owl:unionOf*, *owl:intersectionOf* and *owl:complementOf* express set operations on the sets of instances of given classes.
- Nominals: A class can be defined by an enumeration of its individuals (*owl:oneOf*) or it can be defined based on the existence of particular property values.

The use of nominals as well as the set operators can limit the decidability of an ontology. The Description Logic fragment that is equivalent to OWL DL is $\mathcal{SHOIN}(\mathcal{D})$, which is decidable but reasoning is in NExpTime. There is another fragment $\mathcal{SHOIQ}(\mathcal{D})$, where again generalised cardinality restrictions ($\mathcal{Q}$) are allowed. Although reasoning is in NExpTime, recent findings [HS07] could show that reasoning with the given algorithms is only in the worst case in NExpTime but for common use cases it is in ExpTime.

OWL Full

OWL DL and OWL Lite are extensions of a *restricted* use of RDF and RDFS, because, unlike RDF and RDFS, they do not allow classes to be used as individuals, and the language constructors cannot be applied to the language itself [HPSvH04]. OWL Full is the union of RDF/RDFS and OWL DL, implementing all remaining features of RDFS that are not already part of OWL DL. The additional traits are

- Reification: adding statements about statements.
- Classes can now be both a class and an instance at the same time.
- In OWL DL, properties have to be either object or datatype properties. Datatype properties relate instances of classes to RDF literals or XML Schema datatypes; datatype properties cannot be inverse functional. These limitations do not hold any longer in OWL Full.

So far there are no reasoning engines that implement full OWL Full. The reason is that OWL Full is undecidable. For example it is possible to express paradoxical situations like Russel's paradox (the set of all sets that do not contain themselves as members) in OWL Full.

OWL Terminology

Often the notion of a *model* is connected with OWL ontologies. This is not just an RDF graph extended with further edges, it is a *theory* which consists of an RDF graph and a set of formulas (OWL axioms). Queries to that model retrieve atomic facts from the theory. A model has an intensional and an extensional part although this separation is typically invisible to the user. An OWL model has an operational representation by a set of statements that are all *ground facts*.

It has been described how OWL can be used in knowledge representation. All derivations are based on the interpretation of axioms that are used in the ontology. Rule-based derivations, however, are not possible. This deficit can be compensated by a combination of different reasoning formalisms.

3.7 Hybrid Reasoning

Right from the beginning, when the idea of the Semantic Web was formed, it was clear that there is the need, in addition to an ontology description language, for another "layer" of inference (referring here to the allegory of the Semantic Web Tower as visioned by Tim Berners-Lee [BLHL01]). This additional layer should provide access to formalisms of deduction that are common in logic programming.

Limitations of OWL

OWL has considerable expressive power as an ontology description language. But, in order to retain the decidability of key inference problems, there are limits to this expressiveness. While OWL offers a large set of class constructors, there are only a few language features that allow to relate properties. For example, it is not possible to express the *uncle* relationship as the composition of the *parent* and *brother* relationships.

Furthermore there is the open world assumption which might cause undesired effects in query answering. This is not so much a limitation but in fact a consequence of classical logic in general. And, with respect to the representation of highly distributed knowledge in an environment like the World Wide Web this paradigm is very well suited. Available information is (potentially) always incomplete, therefore it would be incorrect to assume that the lack of information is equivalent to negative information. For the retrieval of information, however, it might desirable to draw conclusions on the basis of what is known *de facto*.

Consider, for example, the following facts describing the resource mon:France by its mon:capital-relationships to the resources mon:Paris and mon:Berlin.

Example 3.19 (Limitations by OWA and UNA)

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix mon: <http://www.semwebtech.de/mondial/10/meta#> .
mon:France mon:capital mon:Paris .
mon:France mon:capital mon:Berlin .
mon:capital a owl:FunctionalProperty .
```

Without violating the functional property restriction of the mon:capital property, OWL reasoning allows to conclude that both mon:Paris and mon:Berlin are the mon:capital of mon:France because it is consistent to assume that these resources might be the same. In other words, there could be some additional knowledge *somewhere* expressing (mon:Paris owl:sameAs man:Berlin).

This limitation can be overcome easily in such a simple example. With the addition of the statement (mon:Paris owl:differentFrom mon:Berlin), the two resources are now known to be disjoint. However, in a more complex situation it is much more complicated and sometimes simply not possible to define the dissimilarity for every pair of individuals. In general, knowledge base design in OWL faces some specific challenges (for a more detailed analysis of the limitations in knowledge base design in OWL, see also [HPSvH04]).

Another constriction concerns a technical aspect of OWL reasoning engines, which has considerable consequences on reasoning with OWL ontologies. OWL reasoners are usually based on the tableau algorithm. While these reasoners show good performance in complex TBox reasoning (concept reasoning) they are weak in ABox reasoning (instance reasoning) as soon as large numbers of instance data are involved [MS06],[HLTB04]. Hence it is disputable whether the use of OWL reasoners together with real-world ontologies that often contain large numbers of individuals and only a limited set of concept definitions is advisable.

Combining Rules and Ontologies

Hybrid reasoning for the Semantic Web is commonly considered to be a combination of DL reasoning (open world) with logic programming (closed world) (see again Section 3.3). Logic programming adds features that cannot be expressed in FOL (which DL is a subset of), e.g. negation as failure or procedural attachments (the association of action-performing procedural invocations with the drawing of conclusions about particular predicates).

Depending on the degree of integration, combinations of rules and ontologies can be characterised as being either *hybrid* or *homogeneous* [ADG+05]. The *homogeneous* approach combines both parts to a single logic language. The ontology part is tightly integrated by a rule-based mapping that coexists with rule predicates (e.g. DL+log [Ros06]). Here, OWL reasoning is *simulated* by the rule engine. Although the *homogeneous* variant offers a seamless semantic integration of rules and ontologies, it suffers from problems concerning either limited expressiveness or undecidability, because of the interaction between rules and ontologies [EIST06]. In *hybrid* reasoning, different formalisms coexist: rules and ontologies are kept separate though rules typically make use of the same predicates that are also used in the ontology. Different reasoning engines can be combined in a modular way, one for ontology reasoning (e.g. Pellet as a DL reasoner) and one for rule evaluation (e.g. a Prolog engine in [DHM07] or a Datalog engine in [DLNS98]).

For an overview on different combinations of rule systems and ontologies, see the discussion in Section 12.5. Chapter 9 describes *hybrid* reasoning in Swan and presents an implementation in which F-Logic reasoning is combined with DL reasoning.

Rule formalisms are used in many different contexts. For instance, in active databases behaviour is specified by rules. These rules are called ECA rules and have gained a lot of attention in the Semantic Web research community.

3.8 ECA Rules: Formalising Behaviour

Data that is subject to change has different needs in terms of care than static data. Dynamic aspects in data storage have been dealt with for a long time in active database research. The most basic form of active behaviour is common to most of the present-day databases: triggers. Triggers are simple rules on the (database) programming language and data structure level. The underlying paradigm for such behaviour are ECA rules: On the occurence of an Event, when some Condition is fulfilled, do some Action. In databases, the pattern for trigger definitions is: **ON** *event* **IF** *condition* **THEN** *action*. With triggers, the conditions are given in the database query language and will evaluate to false or true, only in the latter case allowing the trigger to fire and the action part to be evaluated. The action component is given in a simple, operational programming language. The event is a request for the execution of a database operation, e.g. update the number of booked seats of a flight (see Example 4.3).

Triggers can be defined to initiate some action before or after the requested database operation is executed, and the action can be supplemental or substitutional to the requested operation. Since 1999, triggers are part of the SQL standard. Real-world trigger definitions in SQL are somewhat more complex than the abstract definition above, as they allow for complex, finetuned definition of behaviour that is integrated into the data definition and update language. The purposes of triggers are manifold: they can be used to maintain database integrity and exception handling, to implement business rules, or to fulfill maintenance and monitoring tasks within the database itself instead of programmes outside the database.

Trigger implementations differ depending on the database systems in use. Some make it possible to use embedded procedural languages in the action part of the trigger. That is a very useful feature, because relational algebra, which forms the basis of SQL, is not Turing-complete.

The following example is a typical *after*-trigger, following the SQL standard:

Example 3.20

```
CREATE TRIGGER flightavailablechecking
AFTER INSERT ON  Flightbookings
         REFERENCING NEW AS N
WHEN (select count(*) as bookedseats from Flightbookings
                          where flnumber = n.flight)
UPDATE bookingsperflight SET bookings = bookedseats
  WHERE flightNumber = n.flight
```

On the event of an insert on the table *Flightbookings* the condition of the trigger rule is evaluated (a query that selects the number of booked seats for that flight). Then another table is updated with the result from the query. This is a typical situation where a trigger is used for the update of a materialised view.

The ECA paradigm, which makes it possible to define the behaviour of the database system, can easily be extended. On the one hand, the action part can be, as mentioned before, a fragment of an arbitrary procedural language. On the other hand, the event part can be a composition of many single events combined by an event algebra. There has been a lot of research on event algebras for ECA rule systems in the (active) database community [CM94, GJS92] during the '90s. This branch of research was stimulated again with the advent of the Semantic Web, when again the usage of the ECA paradigm seemed fitting for the description of the dynamic aspects of change in knowledge bases [MAB04]. There have been several examinations on how to develop meta-models, encouraging the definition of a general semantics for ECA-Rules that covers most of the popular ECA-rule systems from previous research, e.g. [Hin03, ZU99].

The situation in a (distributed) knowledge base is up to a certain extent similar to the one in a database. But the distinction of what an action and what an event is can be different. In the database the point of view is determined by the perception of the process: the event (something changes) is directly connected to the update (which causes the change of state). In a database the action-performing agent and the event-detecting observer are always the same. In the Semantic Web there are many agents, and events can be about arbitrary changes performed by any agent. If an event is detected it is not necessary to know where it originated or which action was causing it. Without consideration of the agent, we talk about an event (following a common definition given by e.g. [Kem01]). An event is primarily something linguistic or cognitive in nature as the world does not really contain events. But how exactly do actions and events interact? A general representation of actions and events has to be found in order to support reasoning about the dynamic aspects of application domains. These questions will be examined in more detail in Chapters 4 and 5.

Chapter 4

MARS: Modular Active Rules in the Semantic Web

This chapter explains the design principles of Modular Active Rules (Mars) [BFMS06, BFK+07, BFMS08]. A description of the architecture and the components of the service infrastructure is provided.

4.1 Overview

As the name *active rules* indicates, the Mars rule system offers an infrastructure which can be used for the modelling of the *dynamic* aspects of an application domain. This means that the behaviour of a domain (e.g. a set of services) can be *specified* by rules which are *executed* directly by the rule evaluation system of Mars. Consider the following example:

Example 4.1 (Travel booking)
The application domain consists of services for travelling. In a flight booking process a travel agency and an airline company are involved. The plot is: a customer wants to book for flight LH458, the travel agency orders the booking at the airline company and the airline has to update its internal database in order to reflect that change on flight LH458. The intended behaviour can be modelled by the use of a simple rule: ***on*** *the occurence of a* flight-booking ***if*** *there are available seats for that flight* ***then do*** book-seat*. This rule is given to* Mars *and in case of a* flight-booking *the rule evaluation engine will conclude that* book-seat *has to be done.* □

The rule formalism that is used in Mars follows the ECA paradigm (see again Section 3.8). ECA rules consist of an event, a condition and an action part. Initially, an event from the application domain (e.g. book-flight) activates the evaluation of the rule. Eventually, if the condition (the query to the airline for available seats) is satisfied, the action book-seat is executed.

The execution of such a rule relies on a set of service components which will be described in Section 4.2. The components and their interaction are mostly invisible to the user. In this respect, the Mars architecture differs greatly from other architectures like Web Services. The latter demand from the user to address services directly in order to execute programme code *of* the service. In MARS, however, it is not necessary to know how the services are implemented, for example how a flight booking has to be done. Instead, the behaviour of the domain is modelled abstractly by rules which in turn can be executed directly. The only preconditions are knowledge about

- how to define rules using the MARS rule formalism and
- what vocabulary has to be used (e.g. flight-booking, book-seat) in order to achieve the desired behaviour.

Section 4.3 presents the concept of ECA rules and how it is used in MARS. This includes a description of the notions of actions and events and how they are conceived in the context of MARS. Domain ontologies provide the vocabulary which is used for the specification of events, actions, and other notions of the rules and the application domain. The concept of domain ontologies, however, is a subject of its own and will be discussed in Chapter 5.

4.2 MARS Components

This section gives a brief description of those components in MARS that are needed in order to understand the integration of SWAN into MARS.

Domain Broker. Much of the communication between the components in the MARS framework is handled and conveyed by a domain broker. Its main task is to distribute actions, queries and events to appropriate services. Hereby it serves as an interface between an application domain and the MARS service infrastructure. Services are known if they registered at the domain broker beforehand. Each application node announces its capabilities to the domain broker by giving a list of those actions that the application node supports. In a similar way other services register at the domain broker by giving a list of events that they are interested in. In case that the domain broker receives an event or an action, the appropriate recipients can be looked up in the registry of actions and events of the domain broker. Query brokering is more complicated. Here it is necessary that the domain broker has access to the ontologies of the application nodes. The concepts that are used in the query are compared to concepts of the available domain ontologies. Those application nodes that use the same concepts as those that are contained in the query will receive the query for evaluation.

ECA Rule Engine. This service component is responsible for rule evaluation. The duties of the rule engine comprise registration, deregistration, and evaluation of ECA rules. In case that a new rule becomes registered, the specification of the event component is passed on to an event processor. This service will in turn inform the rule engine as soon as one of the specified events is detected. The detection of an event causes the next part of the rule to be evaluated: the condition. The condition part can be any kind of query and will be given to the query handler. This service takes care of query evaluation. The results are returned to the rule engine which now processes the consequences of the rule: the action part which is given to an action engine.

Rule evaluation is mainly a task which consists of mediating the execution of the rule components and passing on contents from one of the rule components to the next, e.g. parameters of the event or results from a query in the condition.

Event Processing. Event processing involves detection and matching of events. At first, a client (e.g. the ECA rule engine) registers for an event. This event specification consists of an event name and attributes that have to be contained in the event. In the next step the event processor on its own part has to register at the domain broker. The domain broker keeps the event processor informed about all events that match the registration. If an event turns up at the event processor it is matched against all registered event specifications. For each match a notification is sent back to the client. The notification contains the extracted parameters of the event in form of variable bindings.

Query Handling. The condition part is evaluated by the query handler. This service analyses the condition with regard to the query language that is used. For example, SPARQL queries (see Section 3.5) can be distributed by the domain broker and are sent there. Other query languages are implemented by query services of their own (e.g. XQuery). It is the task of the query handler to distribute the queries to the appropriate services and return the results in form of variable bindings to the ECA rule engine.

Action and Process Execution. The action component is the last part in the chain of rule evaluation. Therefore, the action definition is simply passed on to the action engine which inserts the variables and sends the resulting action(s) to the domain broker which is expected to distribute the actions to appropriate services.

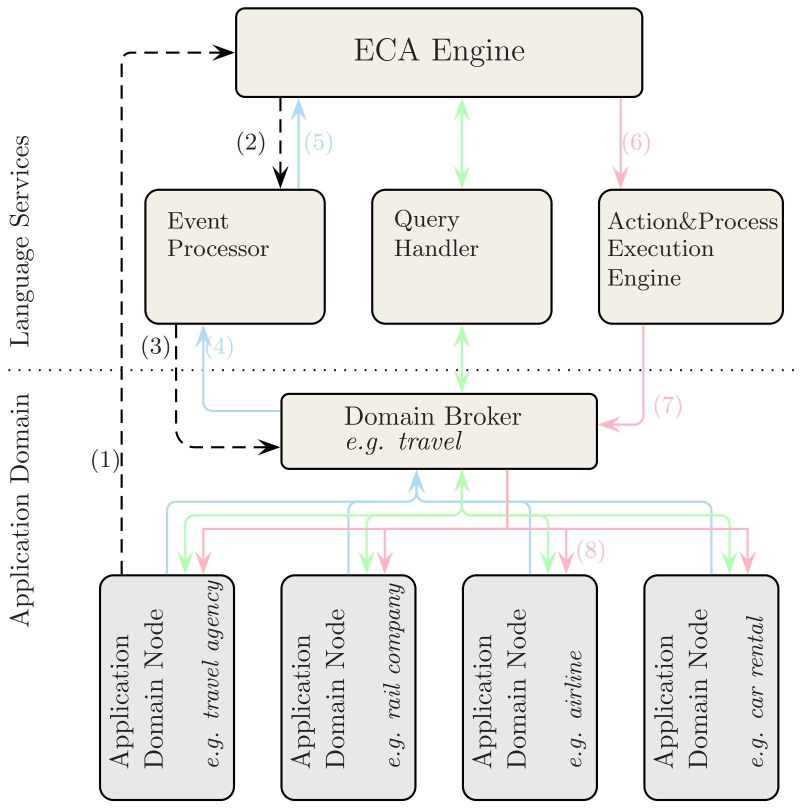
Domain Application Nodes. The domain nodes (also called domain application nodes or simply application nodes) operate on real data, which is stored locally, either conventionally in a database or, like in SWAN, in a knowledge base. This is different from other infrastructural components in MARS where only rules, registrations or state information have to be kept. Therefore much of the *state* of a network is located at the domain nodes[1]. Domain nodes are the leaves in the Semantic Web architecture as they offer services in an application domain (e.g. railway companies, airlines, travel agencies). These services usually include querying for and manipulation of data. Data manipulation can be performed by explicit updates but also by execution of abstract actions. These actions result from ECA rule evaluation.

For example, an airline which serves flights offers the possibility to book seats for these flights by the action book-a-seat. The internal manipulation of data is a consequence of that action.

The domain nodes use events to talk about what happened in the knowledge base, hereby making changes of the local state visible to the rest of the domain. These events are sent to the domain broker.

Figure 4.1 shows schematically, with reference to Example 4.1, how the different service components in MARS interact. Initially, a domain node (e.g. *travel agency*) sends an ECA rule to the ECA engine (1). This rule specifies what has to be done upon an *event* named flight-booking. The next step is to ensure that the ECA engine will receive events of this kind. Hence, it registers at the event processor for that event (2). This service has in turn to register for events at the domain broker (3). If a flight-booking event is received by the domain broker

[1]Chapters 6 - 9 give a description of SWAN, which is the architectural concept of such a domain application node.



ECA Engine
Event Processor
Query Handler
Action&Process Execution Engine
Domain Broker e.g. travel
Language Services
Application Domain
(1)
(2)
(3)
(4)
(5)
(6)
(7)
(8)
Application Domain Node e.g. travel agency
Application Domain Node e.g. rail company
Application Domain Node e.g. airline
Application Domain Node e.g. car rental

Figure 4.1: MARS Infrastructure. (→Events, →Queries, → Actions, --→ Registrations)

(e.g. sent there by a user) it is handed over to the event processor (4). The event processer compares the event to the event specification that it has received from the ECA engine. For every registration that matches the event, a notification is sent to the ECA engine (5). This triggers the rule that has been registered in (1). As there is no condition part, the consequence of the rule is processed: book-a-seat. This action is handed to the action engine (6) which sends it to the domain broker (7). The domain broker delivers the action to a service which is known to be able to execute that action (e.g. the airline company) (8).

All these components of the MARS architecture are needed for the evaluation of ECA rules. These rules are now given a closer examination.

4.3 ECA Rules in MARS

The behaviour of a domain is defined by its ontologies and is implemented by the use of ECA rules. In active databases (where the ECA paradigm originated) an event is simply a database update. In MARS, however, an event is anything that can be observed.

The general structure of an ECA rule in MARS is shown in Figure 4.2. A

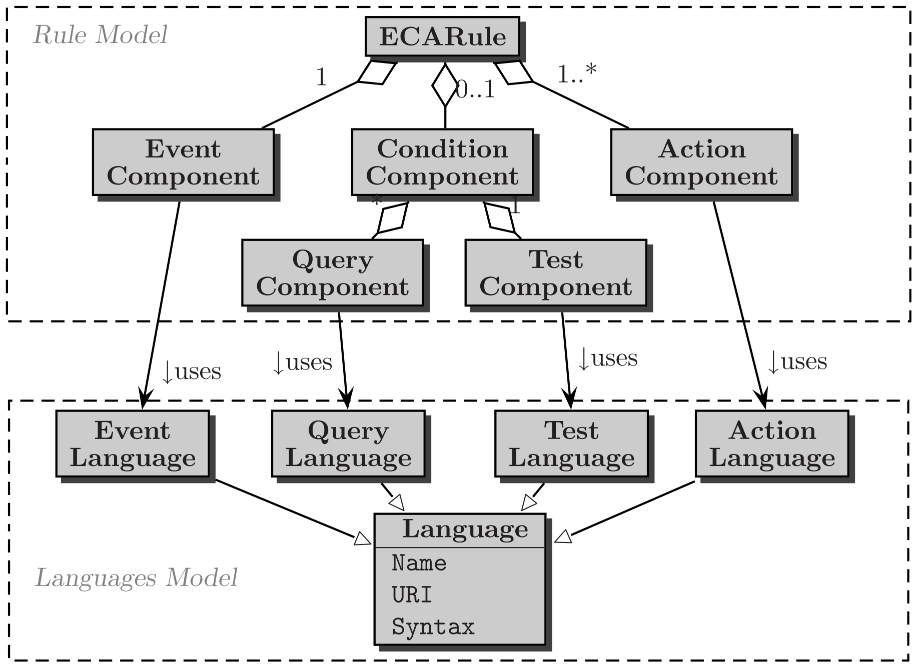


Figure 4.2: ECA Rule Components and Corresponding Languages

typical rule consists of one *event* component, an optional *condition* component and one or many *action* components. Each component uses for its specification an appropriate language. The event processor, the query handler and the action processor each have to identify the languages that are used in their rule components. Either these language can be interpreted by the component processor as a built-in language or a language processor has to be found. Languages and services can be identified using a *Language and Service Registry* (LSR).

Before the components of an ECA rule can be analysed in more detail, the syntactic aspects of rule definition have to be established. The specific markup that is used in MARS is described in the following.

4.3.1 ECA Rule Markup

MARS widely uses an XML markup language on the rule level, named ECA-ML. All elements of the markup language are defined in the ontology, which is used by the ECA rule engine. For example, the rule that was informally described in Example 4.1 is now given in proper ECA-ML markup:

Example 4.2

```
<eca:Rule xmlns:eca="http://www.semwebtech.org/languages/2006/eca-ml#">
  <eca:Event>
    <xqm:Event
      xmlns:xqm="http://www.semwebtech.org/languages/2006/xmlql#">
      <travel:flight-booking
        xmlns:travel="http://www.semwebtech.org/domains/2006/travel#"
        flight-number="{$flight-number}"
        date="{$date}"/>
    </xqm:Event>
  </eca:Event>
  <eca:Test>
    <eca:Opaque
        eca:url="http://airline.travel.example.com/queries">
      <eca:has-input-variable eca:name="flight-number"
          eca:use="$flight-number"/>
      ASK {
        $flight-number :booked-seats ?bs.
        $flight-number :person-capacity ?pc.
        FILTER (?bs &lt; ?pc).}
    </eca:Opaque>
  </eca:Test>
  <eca:Action>
    <xqm:Action
      xmlns:xqm="http://www.semwebtech.org/languages/2006/xmlql#">
      <travel:book-seat
        xmlns:travel="http://www.semwebtech.org/domains/2006/travel#"
        flight-number="$flight-number"
        date="$date"/>
    </xqm:Action>
  </eca:Action>
</eca:Rule>
```

□

Note that the condition part may consist of either a query or a test component or both of them. While the query is intended to bind query results to variables for later use in the action component, the test component is used to check constraints on the bound variables by a boolean query as shown in Example 4.2.

MARS does not have a fixed set of languages for the specification of the rule components. Rather, it provides for the possibility to use arbitrary languages presumed that there are appropriate services available for the handling of these languages.

For this purpose the rule engine needs to know, which languages are used in the rule components. This is realised by the use of language identifiers (e.g. XML namespaces or RDF resources) for the components that relate each component to a language specification. The language specification gives details about a language like obligatory or mandatory parameters or services that are available for the processing of the language. In the example above, the action and the event are both defined using XQM [DFF+99], which is specified by the resource identifier http://www.semwebtech.org/languages/2006/xmlql#. The LSR allows to find services that can be used for the execution of the rule components that use XQM, e.g. http://www.semwebtech.org/services/2007/aem-xmlql.

The different components of ECA rules have been introduced already. It will be shown now that these components can be defined either using terminology from the domain ontology or in form of *opaque* elements.

4.3.2 Opaque Rule Components

In Example 4.2 the event and the action part are defined in terms of the domain ontology: both travel:flight-booking and travel:book-seat are notions that are given by the domain ontology. The semantics of these components is fixed and no procedural aspects are needed to be specified in the rule. Opaque components, however, are given as programme code of some existing language that is usually not defined (semantically) in the domain ontology. For example, database updates in the action part or SPARQL queries in the condition component (as in Example 4.2) are typical. Compare the abstract action component of Example 4.2 to the following opaque fragment:

Example 4.3 (An opaque action definition)

```
<eca:Action>
  <eca:Opaque eca:url="http://airline.travel.example.com/sqlupdates">
    <eca:has-negative-variable name="flight-number" />
    UPDATE flighttable SET seats =
      ((SELECT seats
        FROM flighttable
        WHERE flight = $flight-number)-1)
    WHERE flight = $flight-number;
  </eca:Opaque>
</eca:Action>
```

□

The abstract action travel:book-seat in Example 4.2 is defined only semantically. It is left to the recipient of the action book-seat how to implement the *intention* of the action. In the opaque action in Example 4.3 it is stated explicitly what to do, and where. To this end, the actual update language of the recipient application node has to be known.

The use of opaque rule components has the disadvantage of a tight coupling of rules with service components. If a service is not available or changes its update procedure the opaque content will fail to be processed. Moreover, if another service with a different update language should be used instead of an existing implementation, the rule has to be changed.

With abstract event and action definitions that make use of the vocabulary of the application domain, the integration of services is highly modular. Any service which uses the vocabulary of the domain ontology can be used for action execution.

Up to this point, only *atomic* actions have been considered. The action part of an ECA rule may, however, also consist of the specification of a *complex* action.

4.3.3 Complex Actions

The action component enforces the consequence of an ECA rule. It can be

- an *atomic action* on the domain level,
- a *composite action*, expressed in a language for specifying complex actions,
- an opaque action which invokes specific web services,
- an action component raising an event in an application domain (implementing ECE rules, see Section 5.2.1)

Atomic actions have already been used in Example 4.2 (which is an abstract action in the travel domain) and Example 4.3 (which is an opaque action). Atomic actions can be executed at a domain node, whereas *composite* actions consist of an arbitrary number of actions and have to be *decomposed* first. Composite actions allow to combine actions in many different ways, e.g. as sequences or in concurrent execution. Composite actions need an additional formalism in order to specify the combination of actions that are to be executed. This can be achieved, for example, by the use of Process Algebras like *the Calculus of Communicating Systems* (CCS) [Mil83]. Because of the modularity of the MARS architecture it is easy to use complex action definitions in rule execution. The formalism, which is used in the definition of the composite action can be identified by the XML namespace that is used in the definition. Hereby, it is possible for the rule engine to pass the action component to an adequate service for execution. The following example, which makes use of CCS, illustrates how a sequence of actions can be specified:

Example 4.4 (Action Specification using basic CCS)

```
<eca:Action xmlns:ccs="http://www.semwebtech.org/languages/2006/ccs#">
  <ccs:Sequence>
    <ccs:AtomicAction>
      <travel:book-seat
        flight="{$flight-number}" date="{$date}"/>
    </ccs:AtomicAction>
    <ccs:AtomicAction>
      <travel:book-train
        from="{$start}" to="{$airport}" date="{$date}"/>
    </ccs:AtomicAction>
  </ccs:Sequence>
</eca:Action>
```

□

The process engine has to decompose the action component into several separate atomic actions, here combined sequentially: After a seat for a flight is booked, a train ticket is bought for a connection to the airport.

Not only the action component can be given as a composition of atomic actions. Also the event part may consist of a complex event definition.

4.3.4 Composite Events

The composition of events makes use of formalisms known as event algebras. They define what (algebraic) combinations of atomic events are allowed. A well-known event algebra is SNOOP [CM94], which offers the combinators *and*, *or*, *sequence*, *any n out of alternatives*, *periodic*, and *aperiodic*. The usage of complex events demands, of course, for complex event detection. Like other event processing services, this service has to register at the domain broker for every atomic event that occurs in the complex event definition. When the complex event is detected a notification is sent back to the client which registered the complex event definition.

Consider the following composite event, which specifies the sequence of two events, travel:DelayedFlight followed by travel:CancelledFlight (using the join variable id).

Example 4.5

```
<eca:Rule>
  <eca:Event
    xmlns:travelns="http://www.semwebtech.org/domains/2006/travel#">
    <snoopy:Sequence>
      <snoopy:Atomic>
        <xmq:Event
          xmlns:xmq="http://www.semwebtech.org/languages/2006/xmlql#">
          <travel:DelayedFlight travel:flight="{$id}"/>
        </xmq:Event>
      </snoopy:Atomic>
      <snoopy:Atomic>
        <xmq:Event
          xmlns:xmq="http://www.semwebtech.org/languages/2006/xmlql#">
          <travel:CancelledFlight travel:flight="{$id}"/>
        </xmq:Event>
      </snoopy:Atomic>
    </snoopy:Sequence>
  </eca:Event>
    :
</eca:Rule>
```

□

Because the flight has been at least delayed once, most of the passengers are expected to be waiting at the airport already. Hence, a rule with this composite event specification could be used to inform all passengers of the delayed-and-then-cancelled flight about alternative flights via SMS.

The rules and services that have been presented so far make use of notions from a fixed vocabulary. These notions are defined in the MARS ontologies.

4.4 MARS Ontologies

The MARS framework uses several ontologies. The *Language and Service Registry* is used to describe existing *service implementations* and their capabilites whereas the *Service Ontology* defines the *classes* of services that are part of Mars. Furthermore there is the Mars ontology which defines the notions that are used in rules, language descriptions, and ontologies. The following example shows parts of the Mars ontology.

Example 4.6 (Basic Mars Ontology)

```
@prefix owl:  <http://www.w3.org/2002/07/owl#> .
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix mars: <http://www.semwebtech.org/mars/2006/mars#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .


mars:Domain              a        owl:Class .
mars:DomainNotion        a        owl:Class .
mars:Class               a        owl:Class ;
      rdfs:subClassOf             mars:DomainNotion , owl:Class .
mars:Property
      a          owl:Class ;
      rdfs:subClassOf mars:DomainNotion , rdf:Property .
mars:Action              a        owl:Class ;
      rdfs:subClassOf             mars:DomainNotion .
mars:Event               a        owl:Class ;
      rdfs:subClassOf             mars:DomainNotion .

mars:has-domain-broker
      a          rdf:Property ;
      rdfs:domain mars:Domain ;
      rdfs:range mars:DomainBroker .
mars:DomainBroker
      a          owl:Class ;
      rdfs:subClassOf mars:Service .
mars:DomainService
      a          owl:Class ;
      rdfs:subClassOf mars:Service .
mars:uses-domain
      a          rdf:Property ;
      rdfs:domain mars:DomainService ;
      rdfs:range mars:Domain ;
      owl:inverseOf mars:has-service .
mars:supports
      a          rdf:Property ;
      rdfs:domain mars:DomainService ;
      rdfs:range owl:Class .
```

In the Mars ontology, *static* aspects of rules, services and languages are defined. The *dynamic* aspects are given in the domain ontologies. Domain ontologies are examined in the next chapter.

Chapter 5

Domain Ontologies

A Semantic Web application node uses at least one domain. For example, the airline company uses the travel domain. If this application also has to consider financial matters then maybe the banking domain will also be used.

While the ontologies of MARS define the concepts that are needed for rule execution and service integration, the ontologies of an application domain describe static and dynamic aspects of the domain. Events and actions are *static* notions in a domain ontology and give the vocabulary to talk about the domain. The *dynamic* aspects of the domain ontology are given in form of rules.

5.1 Events and Actions in Domain Ontologies

The notions of event and action have already been used in the components of ECA rules. Actions cause changes to the state of an application domain whereas events are descriptions of what has happened. On the *conceptual* level events are not communicated, they are simply visible. In the following these conceptual aspects of events and actions are described. The formal aspects of events are investigated in Section 10.1.1.

Consider the travel domain. The concepts in the previous examples made use of actions and events of that domain, for example the abstract atomic action travel:book-seat, and the events travel:flight-booking, travel:seat-booked, travel:DelayedFlight and travel:CancelledFlight. These notions are given in the travel domain ontology:

Example 5.1 (Travel ontology, using concepts from the MARS ontology)

```
@prefix owl:    <http://www.w3.org/2002/07/owl#> .
@prefix rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix mars:   <http://www.semwebtech.org/2006/mars#> .
@prefix rdfs:   <http://www.w3.org/2000/01/rdf-schema#> .
@prefix travel: <http://www.semwebtech.org/domains/2006/travel#>.
travel:Action
       a          owl:Class ;
       rdfs:subClassOf  mars:Action ;
       mars:belongs-to-domain      travel: .
travel:book-seat
       a          owl:Class ;
       rdfs:subClassOf  travel:Action ;
       mars:belongs-to-domain      travel: .
```

```
travel:flight-booking
        a           owl:Class  ;
        rdfs:subClassOf  travel:Event;
        mars:belongs-to-domain      travel:  .
travel:seat-booked
        a           owl:Class  ;
        rdfs:subClassOf  travel:Event;
        mars:belongs-to-domain      travel:  .
travel:DelayedFlight
        a           owl:Class  ;
        rdfs:subClassOf  travel:Event;
        mars:belongs-to-domain      travel:  .
travel:CancelledFlight
        a           owl:Class  ;
        rdfs:subClassOf  travel:Event;
        mars:belongs-to-domain      travel:  .
```

From the point of view of the agent (that performed an action) an event can be both a consequence or an interpretation of an action. For example, the action book-seat and the event seat-booked are the same to the agent that performed the action (the airline company). Also half-booked (half of the available seats are booked) is an event that could be raised by the application.

Events allow to propagate changes using a shared vocabulary, which is commonly known in the application domain. This is a convenient way to describe actions that took place. For example, the booking of a flight-ticket is better desecribed by seat-booked compared to the much more explicit but circumstantial *the number of avaible seats for flight x has been reduced by one*. Moreover, if the events are named in terms of an ontology it is possible to talk about these events. Hence, rules can be related and it becomes possible for other participants to contribute additional facets to an event. For example, if the meaning of the concept seat-booked is established the event ticket-sold can be defined as directly related to that event by the travel agency. Which update led to the event seat-booked is of no importance to other agents.

Further relationships between events can be found if the domain ontology defines such relationships. This is done with rule-based definitions.

5.2 Rule-Based Definitions

Rules which raise or derive events allow to define what changes become visible. It is, in common sense, a special point of view that can be different from that of the agent (that actually executed an action) and another agent to whom the event becomes visible. Events may correspond to actions, but at the same time they can be derived from other events.

Ontologies may contain, besides the static notions of actions and events, rule-based definitions. These definitions, which can be logical derivation rules, ECE rules or ACA rules, are explained in the following.

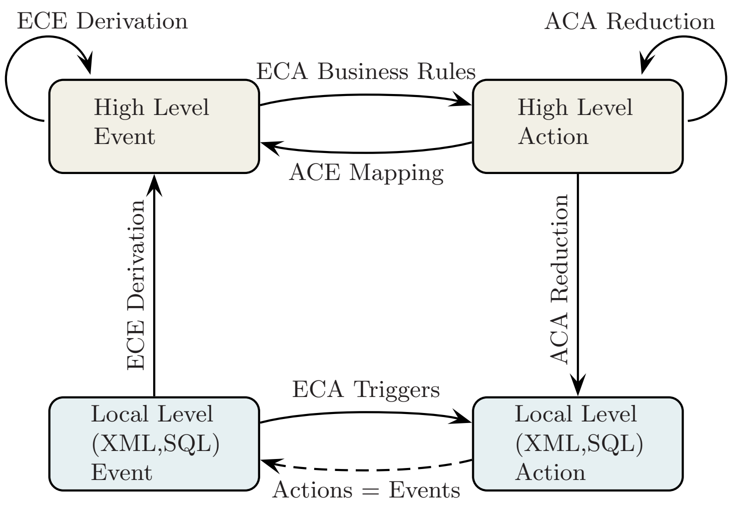


Figure 5.1: Types of Rules.

5.2.1 Derivation Rules

Derivation rules can be given in different forms. The easiest way with regard to the integration of rules into the domain ontology is provided with OWL. OWL axioms can be used to derive subclass relationships, classes can be defined as intersections, unions or restrictions of other classes. Furthermore, instances of properties can be derived.

For the integration of other, full first-order formalisms there are different proposals. The hybrid reasoning extension in Swan (see Chapter 9) is one example, other approaches are discussed in Section 12.5. While these derivations are based on *concepts* of a knowledge base, ECE rules allow for derivations based on the behavioural notions of an application domain.

5.2.2 ECE Rules

One way in which the relationship of events and actions can be specified has been shown with ECA rules. ECE rules are a variant of ECA rules which allow to derive high-level events from other events (see Figure 5.1). In order to understand the relevance of ECE rules the notions of explicit and derived events have to be explained:

- Explicit events have a direct relationship to actions. They do not add any new knowledge (or a different point of view) to the action but directly correspond to it. They usually are raised where the action occurs. For example, upon the execution of the database update

```
update flighttable set seats =
  ((select seats from flighttable
    where flight = 'LH458' and date='20081010')-1)
  where flight = 'LH458' and date='20081010';
```

  it would be possible to directly raise an event travel:seat-booked with parameters *flightnumber='LH458'* and *date='20081010'*. The event seat-booked and the update action are directly connected and, from the point of view of the database, they are the same.

- Implicit and derived events: An event can be derived from another event by the use of ECE rules. These rules reflect that there can be many different points of view on the same event. A travel:flight-booking event with destination New York can also be seen as a travel:flight-booking-to-USA event if New York is known to be a city in the United States and if there is an ECE rule **on** travel:flight-booking **if** ($destination cityIn USA) **do** raise-event travel:flight-booking-to-USA.

Consider another example of an ECE rule. This rule is defined to react upon a travel:seat-booked event and checks whether the number of available seats has reached a certain threshold (e.g. fifty percent), in which case the event travel:half-booked would be derived from the original event travel:flight-booking (needing the *flight-number* flight as an additional parameter).

Example 5.2 (ECE rule definition in the domain ontology)
The derived event travel:half-booked is specified by a definition in the domain ontology. This time, the airline company is not an SQL database but an XML database. Thus, the query language that is used is XPath instead of SQL.

```
<mars:Definition mars:syntax= "xml">
  <mars:defined>
    <!-- pattern of the event to be derived -->
    <travel:half-booked travel:flight="$flight" travel:date="$d"/>
  </mars:defined>
  <mars:defined-as>
    <!-- E/C components how to derive it -->
    <eca:Event>
      <travel:seat-booked travel:flight="$flight" travel:date="$d"/>
    </eca:Event>
    <eca:Test>
      <eca:Opaque eca:language="xpath">
        <!-- note: XML schema of the (local) database assumed to be known -->
        count($flight[@date="$d"]/booking) =
          $flight/id(@aircraft)/@number-of-seats div 2
      </eca:Opaque>
    </eca:Test>
  </mars:defined-as>
</mars:Definition >
```

□

The handling of ECE rules is quite similar to the handling of ECA rules, operationally. It is the task of the domain broker to take care of the handling of these aspects of the domain ontology. For this, ECE rule definitions are translated by the domain broker into the ECA-ML rule format. The action component of such an ECA rule contains an embedded event (e.g., travel:half-booked). Hereby, the (translated) ECE rules can be registered at the ECA rule engine for rule evaluation. The ECA rule engine in turn registers for all events (that were given in the mars:defined-as part of the rule definition) at the domain broker.

The translated ECE rule will eventually be evaluated and the resulting event handed to the domain broker.

ECE rules are used for the definition of the relationship between events. ACA rules can be used in a similar manner for actions.

5.2.3 ACA Rules

As discussed in Section 4.3.2 in the context of opaque rule components, a higher degree of abstraction allows for action definitions independent from whatever data manipulation language is used at the application domain nodes. There are two different ways for the handling of abstract actions (see also Figure 5.1):

a) Actions are the consequences of executed rules. The action execution engine delivers the action(s) to the domain broker. The action is then sent to all application domain nodes that support that kind of action. The abstract action still has to be mapped to an *application specific* update. Usually, domain nodes perform such mappings by the means of rules. These rules are called ACA rules because they define a mapping between different action specifications. The meaning of an action (as specified in terms of the domain ontology) becomes expressed by an update in the data manipulation language of the application node. Such a translation from abstract (higher level) to definite actions (local level) is called a vertical mapping because the degree of abstraction is reduced. ACA rule mappings offer an interface between the abstract rule level (MARS) and the data level of applications (e.g. SWAN). This will be discussed in more detail in Chapter 8 as a part of the SWAN architecture.

b) Horizontal mappings, also specified by ACA rules, map between abstract actions. As a consequence these mappings define an action either more precisely or by giving alternatives. An action specification in form of an ACA rule can be described as *in order to do A do B and then C*, for example map buy-flight-ticket to pay-ticket followed by book-seat). Moreover, ACA rules can be used like database *instead-of* triggers. Hereby, an abstract action is specified to be executed instead of the original action under certain conditions (e.g. execute buy-rail-ticket instead-of buy-flight-ticket if *seats are available* fails). ACA rules can be handled by the ECA rule engine much like ECA rules only that the rule engine reacts on actions instead of events.

Vertical ACA mappings can be defined as part of the domain ontology as long as both action components are defined in terms of the domain. ACA mappings which map abstract actions to application specific updates (*opaque* fragments) are defined and executed by the application. Example 8.4 in Section 8.4 shows how these mappings are realised in SWAN.

The next example from the banking domain illustrates how horizontal ACA mappings can be used:

Example 5.3 (ACA Rule Definition in the Domain Ontology)
The action transfer *€200 from bank account A to B is implemented by two actions, first* debit *€200 from account A, second* deposit *€200 on account B.*

```
<mars:Definition syntax="xml">
  <mars:defined>
    <banking:Money-Transfer amount="$amount"
          from="$from" to="$to"/>
  </mars:defined>
  <mars:defined-as>
    <eca:Action>
      <ccs:Concurrent
        xmlns:ccs="http://www.semwebtech.org/languages/2006/ccs#">
        <banking:debit amount="$amount" account="$from"/>
        <banking:deposit amount="$amount" account="$to"/>
      </ccs:Concurrent>
    </eca:Action>
  </mars:defined-as>
</mars:Definition>
```

□

This ACA rule is part of the domain ontology. The domain broker translates this ACA rule definition such that it can be registered as an ordinary ECA rule at the ECA rule engine. In this situation, no application node registered for supporting banking:Money-Transfer. Instead, the ECA rule engine registered tor banking:Money-Transfer as an event which will initiate the evaluation of the (translated) ACA rule. The execution of the action part will be performed by an action engine, e.g. a CCS engine.

It has been shown how actions and events are used in rule definitions. The dynamic relationship between actions and events still has to be investigated.

5.2.4 Dynamic Aspects of Actions and Events

The main component in Mars are ECA rules. Events and actions are the most important notions for the definition of the domain behaviour.

The Mars ontology (see Section 4.4) abstractly defines actions and events as domain notions. The domain ontologies give precise specifications of actions, events and rule definitions that are used in the application domain. The application domain nodes announce the range of actions that they support. By employing the same domain notions it is possible that the consequence of an ECA rule can eventually be executed at a domain node without specifying where. This is different from ordinary Web Service invocations where the Web Service has to be addressed explicitly. The abstract nature of an action in Mars implies that it does not specify how the action should be executed. It is completely left to the application domain node how this domain notion will be handled. The same holds true for abstract event definitions, which do not specify exactly what happened but how an incident is seen.

So far actions and events are merely declarations of domain notions. As a next step it is desirable to use them for *descriptions* of the dynamic notions of an application domain. Such a description needs a vocabulary which can be used to talk about actions and events in terms of their relationships among each other, their pre- and postconditions. An ontology that utilises such a vocabulary can be in turn applied in the reasoning about the effects of actions and events or, more generally speaking, about the dynamic aspects of an application domain. This is a subject of ongoing research in MARS.

The introduction of MARS and domain ontologies in the last chapters provided the basis for the main contribution of this thesis: A domain node for the Semantic Web. This domain node uses events for the communication and it translates abstract actions to knowledge base updates by the use of ACA rules. The components of this domain node architecture are described in the next part of this work.

Part II

SWAN: Semantic Web Application Node

Chapter 6

Introduction

The following Chapters 6 - 10 contain the main contribution of this thesis, giving a thorough examination of the features and characteristics of the SWAN architecture. The previous chapters about MARS and domain ontologies have prepared the ground for the description of the concept of an application domain node. It is the scope of this thesis to describe how an active knowledge base can be integrated into an event-driven environment and how its behaviour can be specified in form of rules.

This part is structured as follows: The present chapter explains the core features of the SWAN architecture. Furthermore, questions concerning data manipulation in a knowledge base are depicted. The specification of the behaviour of a domain node by reactive rules in form of triggers is presented in Chapter 7. Next, Chapter 8 gives a description of the ACA rule mapping mechanism, which provides an interface that allows to integrate SWAN into the event-driven environment MARS. Chapter 9 deals with the hybrid reasoning capabilities of SWAN and shows how F-Logic rules can be used in combination with OWL reasoning. A logical characterisation of the domain nodes' behaviour can be found in Chapter 10.

6.1 SWAN Architecture: Overview

6.1.1 RDF Storage

Whereas in relational databases there are only limited possibilities to express meta-data (in form of the database schema), the situation is completely different with knowledge bases (see Section 3.3). Here, the expressiveness of the meta information depends on the expressiveness of the ontology description language or the rule formalism that is used. Thus, one prerequisite to the data model of the knowledge base is how data and metadata can be represented. The most widely used formalism for the representation of data in the Semantic Web today is the Resource Description Framework (RDF), which has been described in Section 3.4. One of the central tasks in the SWAN architecture is to store RDF data, furthermore, to allow for querying and data manipulation. For the handling of RDF data in Java there are at least two major frameworks: JENA [Jen] and SESAME [Ses]. Both frameworks allow for the use of a relational

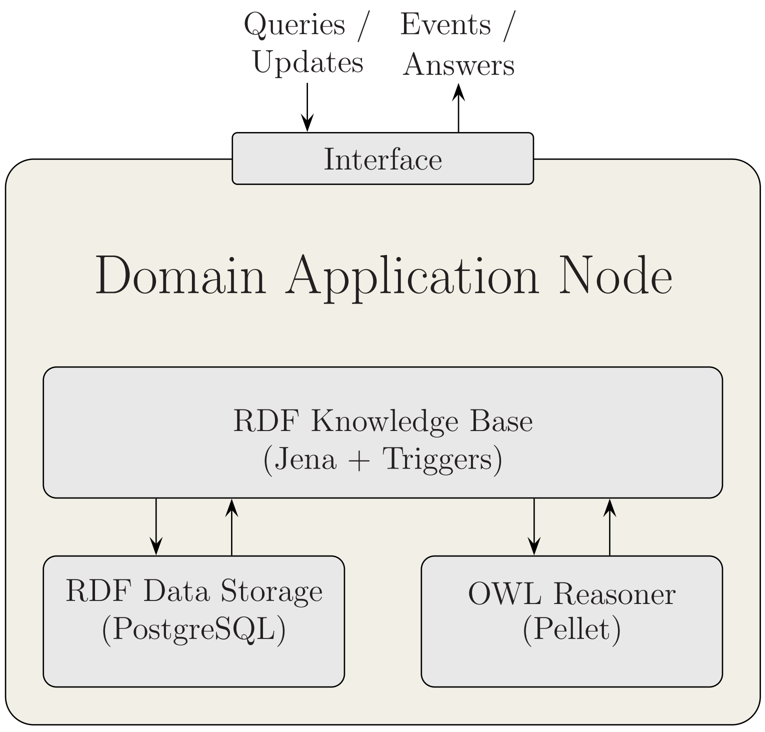


Figure 6.1: Architecture of the Domain Application Node

database as a backend for the persistent storage of RDF data. For the implementation of SWAN the decision was made in favour of JENA in combination with a PostgreSQL database. Figure 6.1 presents the basic architecture of the RDF knowledge base in SWAN.

Internally, manipulations of RDF data are additions and removals of RDF statements. These manipulations are realised by methods of the JENA API. This *storage layer*, however, is invisible to the user. Instead, SWAN offers a user interface for the manipulation and administration of the knowledge base.

6.1.2 User Interface

The user interface of SWAN allows to give update commands to the knowledge base. The following commands can be used for the specification of data manipulations:

- **insert**($sub,$pred,$obj): Insert a statement into the knowledge base.
- **delete**($sub,$pred,$obj): Delete a statement from the knowledge base.
- **delete-resource**($res): Delete all statements that contain the given resource. This actually results in a set of delete operations.
- **update-subject**($sub,$pred,$obj,$new-sub): Update the subject position of the given statement with a new value.

- **update-predicate**($sub,$pred,$obj,$new-pred): Update the property position of the given statement with a new value.
- **update-object**($sub,$pred,$obj,$new-obj): Update the object position of the given statement with a new value.

The control of the administrative functionalities of the domain node is covered by the following commands:

- **read-rdf**: Read a set of RDF statements from a file or URL into the knowledge base.
- **dump**: Dump the contents of the knowledge base into a file.
- **rename**($old-res,$new-res): Globally redefine a resource name in the knowledge base.
- **rename-property-of-class**($old-pred,$new-pred,$class): Redefine the name of the given property for all defined individuals of the given class.
- **actionSequence**: Execute a sequence of arbitrary commands.

The rename operations are different from the update operations: a resource name is changed globally in the knowledge base. The rename operation is not intended to be used for updates but only for internal revisions. Also, the read-rdf operation is not considered for updates. Rather it is designed to be used for the initialisation of the knowledge base.
Queries to the knowledge base can be given by the following commands:

- **sparql-query** Execute a given SPARQL on the knowledge base.
- **ask-query** Execute an ASK query, returning true or false.

These queries are executed by the query engine of JENA, which is an implementation of SPARQL with some additional features like aggregation and filter functions.

There are also commands that allow for high level communication with the application domain. On the rule level, SWAN communicates with the surrounding MARS framework or other domain nodes by the raising of events or by receiving actions. Event raising provides information about the internal state of affairs, action execution enforces the consequences of application domain behaviour as defined by ECA rules. The following commands can be used in this context:

- **raise-event**: Raise an event which will be sent to the domain broker.
- **raise-directed-event**: An event will be raised and sent to the address which is given as a parameter of the application node action.

XML Interface. The above commands can also be submitted to the application node via an XML interface. These commands are called application node actions in analogy to the notion of actions in MARS. The XML fragments may consist of update operations or sequences of actions as described before.

Knowledge base updates in XML markup use the rdfu namespace[1]. Updates can be given to the user interface of the Knowledge base as native update commands like in the following example:

[1]xmlns:rdfu="http://www.semwebtech.org/languages/2006/rdfupdate#"

```
insert( http://example.org#Susan,
        http://family.org#hasChild,
        http://example.org#Peter).
```

Usually, however, the commands are given as XML fragments. The following XML fragment causes the same insert operation as the native update command above:

```
<rdfu:insert xmlns:rdfu=
      "http://www.semwebtech.org/languages/2006/rdfupdate#">
   <rdf:subject rdf:about="http://example.org#Susan"/>
   <rdf:predicate rdf:about="http://family.org#hasChild"/>
   <rdf:object rdf:about="http://example.org#Peter"/>
</rdfu:insert>
```

The XML fragment will be translated by the XML interface and given as a native update to the knowledge base.

A SPARQL query can be given to the domain node in form of an applnode:query element, the query itself can be specified either with an sparql-query attribute node or with the content of that element like in the following example:

```
<applnode:query xmlns:applnode=
    "http://www.semwebtech.org/2006/application-node#">
    <![CDATA[ select ?x where {
      <http://example.org#Susan>
      <http://family.org#hasChild> ?x}
    ]]>
 </applnode:query>
```

If more than one action has to be executed at one time the actions can be given as an action sequence:

Example 6.1

```
<applnode:actionSequence
  xmlns:applnode="http://www.semwebtech.org/2006/application-node#"
  xmlns:rdfu="http://www.semwebtech.org/languages/2006/rdfupdate#">
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 <rdfu:insert
   <rdf:subject rdf:about="http://example.org#Susan"/>
   <rdf:predicate rdf:about="http://family.org#hasChild"/>
   <rdf:object rdf:about="http://example.org#Peter"/>
 </rdfu:insert>
 <applnode:query xmlns:applnode=
    "http://www.semwebtech.org/2006/application-node#">
    <![CDATA[ select ?x where {
      <http://example.org#Susan>
      <http://family.org#hasChild> ?x}
    ]]>
 </applnode:query>
</applnode:actionSequence>
```

Action sequences and queries are administrative actions (see Section 6.1.2) and make use of the application-node namespace[2].

Also the raising of events, adding or removal of knowledge base triggers or ACA rules can be achieved in this way by embedding the event or the rule definition in the application-node action element. The raising of directed events, which can also be used for the sending of messages, needs a target-url attribute for the specification of the recipient:

[2] xmlns:applnode="http://www.semwebtech.org/2006/application-node#"

```
<applnode:raise-directed-event
 xmlns:applnode="http://www.semwebtech.org/2006/application-node#"
 target-url="http://example.org#event-service/incomingevents">
<family:new-marriage xmlns:family="http://family.org#">
    <family:bride family:id="http://example.org#Alice_Miller"/>
    <family:groom family:id="http://example.org#John_Doe"/>
</family:new-marriage>
</applnode:raise-directed-event>
```

Basically, the knowledge base in Swan provides the storage of RDF data. Beyond that, also intensional data can be specified.

6.2 Intensional Data: The Reasoning Layer on Top of RDF

Swan supports the ontology description language OWL , which can be used for the specification of intensional data. In order to apply OWL reasoning, an inference engine has to be used in order to build the OWL knowledge base. Several projects exist implementing description logics reasoners. One that has been favoured for this work is *Pellet* [Pel], for it is freely available and implements the latest language features of OWL. Other popular reasoners are Racer [HM01] and Fact++ [Hor98].

The basic data structure in Jena is an RDF graph. Manipulations to the RDF data are performed by the *Model* interface. The *Model* is constructed from the *Graph* and can be enhanced by reasoning mechanisms, resulting in an *InfModel* (inference model) or in a further subclass, the *OntModel* (ontology model). These *models* contain additional facts that have been derived by an OWL reasoning engine. Note that the term *model* as it is used by the Jena framework is misleading as it does not denote the same thing as is known from *model theory*. The model as it is used in *Jena* is better described as a *(specialised) theory*. The theory consists of the given OWL axioms and the TBox data (concept and role definitions). Moreover, the theory is specialised with regard to the assertions in the ABox. As long as the theory is not closed there are many models in the sense of model theory. Hence, the knowledge base is better named a specialised theory. Nevertheless, the notion of a model is used in this work when considering implementation and architectural details of the Swan architecture as in Jena. Formal considerations, however, employ the model theoretic point of view.

Jena itself offers a variety of built-in reasoners, each of them covering different subsets of OWL, RDFS and (for historical reasons mainly) DAML. The capabilities of these reasoners are limited, though. Additionally, the integration of external reasoners is supported. One possibility is to use the DIG interface [DIG].

6.2.1 Limitations of the DIG Interface

The idea behind this design is to have a uniform access mechanism to arbitrary description logic reasoners. Many DL reasoners indeed implement the DIG interface protocol, which makes it easy to integrate them in an already existing set of services. The DIG interface is based on a *tell and ask* mechanism. First, the application *tells* the data contained in the knowledge base to the reasoner, then the interface can be *asked*. Changes to the knowledge base are told in the same way. In JENA, the coupling of the knowledge base and the reasoner is invisible to the application itself which means that the *InfModel* can be queried and updated in the usual fashion. The underlying programme logic translates the query into questions that are understood by DIG. However, the advantage of having a uniform access mechanism for arbitrary reasoners has a severe drawback: the translation via the *tell & ask* interface supports only a limited set of OWL axioms. Independent from both the set of OWL axioms that are used in the ontology and the reasoning capabilites of the reasoner that is called via DIG, the expressiveness of answers to the queries is limited to the expressiveness of the interface itself.

The following example shows that DIG is limited to entailments on hierarchical relationships. For instance, it is not possible to translate the OWL axiom owl:sameAs via DIG. The ontology defines four individuals: :Peter, :John, :Katrin and :Susan. :Katrin has a child :John, :John has two mothers :Katrin and :Susan. It is *possible* to derive that :Katrin and :Susan have to be the same person (see again Example 3.13).

Example 6.2 (Limitations of the DIG interface)

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix family: <http://family.org#>.
@prefix :<http://example.org#>.
family:hasMother a owl:FunctionalProperty;
    rdfs:subPropertyOf family:hasParent.
family:hasChild owl:inverseOf family:hasParent.
:Peter family:hasMother :Susan, :Katrin.
:Katrin family:hasChild :John.
```

Therefore, the following query, which selects all known family:hasChild relationships, should return both :John and :Peter being children of :Susan and :Katrin:

Example 6.3

```
SELECT ?person ?child
FROM <file:petersnotfoundbrother.n3>
WHERE { ?person <http://family.org#hasChild> ?child . }
```

However, using a DL reasoner via the DIG interface the only result is

```
| person                     | child                     |
==========================================================
| <http://example.org#Katrin> | <http://example.org#John> |
```

By the use of a sophisticated DL reasoner like PELLET, the following query results can be found:

```
| person                      | child                      |
============================================================
| <http://example.org#Susan>  | <http://example.org#John>  |
| <http://example.org#Susan>  | <http://example.org#Peter> |
| <http://example.org#Katrin> | <http://example.org#John>  |
| <http://example.org#Katrin> | <http://example.org#Peter> |
```

Considering that the additional family:hasChild relationships can be derived from the functional nature of the family:hasMother relationship it becomes obvious that it is not possible to derive the equivalence of :Katrin and :Susan by the use of the DIG interface.

Because of the severe restrictions of the DIG interface a different solution for the integration of a DL reasonser was chosen.

6.2.2 Pellet

A completely different approach is possible when using the OWL-DL reasoner PELLET. PELLET offers several interfaces that allow access to its reasoning capabilites. One interface enables a tight coupling of JENA with PELLET. The *OntModel* is built using PELLET in the same way (as far as it concerns the programmer of the application) as the "internal" reasoners that are shipped with JENA. PELLET is under active development and most of the features of the current OWL1.1 proposal are integrated into PELLET already.

Example 6.4
Pellet incorporates the OWL1.1 feature of property chains. The object property family:hasSibling *is introduced, consisting of the combination of properties* family:hasMother *and* family:hasChild*. Note that this example makes use of lists (a single linked list) and blank nodes (anonymous resources annotated with empty square brackets) that have not been mentioned before.*

```
@prefix family: <http://family.org#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix owl11: <http://www.w3.org/2006/12/owl11#>.
@prefix : <http://example.org#>.
:Peter family:hasMother :Susan.
:Peter family:hasMother :Katrin.
:John  family:hasMother :Katrin.

family:hasChild a owl:ObjectProperty;
                owl:inverseOf family:hasParent.

family:hasMother a owl:FunctionalProperty;
    rdfs:subPropertyOf family:hasParent.

family:hasMother   rdf:type    rdf:Property ;
                   rdfs:range  family:Mother ;
                   rdfs:domain family:Child.
family:hasSibling
   a    owl:ObjectProperty .

[]      a         rdf:List ;
        rdf:first family:hasMother ;
        rdf:rest (family:hasChild) ;
        rdfs:subPropertyOf family:hasSibling .
```

Example 6.5 (Query for siblings)

```
SELECT  ?child ?sibling
FROM <file:petersfoundbrother.n3>
WHERE { ?child <http://family.org#hasSibling> ?sibling. }
```

The query returns all pairs of resources that are related by the (derived) family:hasSibling property. The result contains :Peter being the sibling of :Peter and :John (and the same pairs for :John):

```
| child                          | sibling                        |
===================================================================
| <http://example.org#Peter>     | <http://example.org#John>      |
| <http://example.org#Peter>     | <http://example.org#Peter>     |
| <http://example.org#John>      | <http://example.org#John>      |
| <http://example.org#John>      | <http://example.org#Peter>     |
```

Note that it is not possible to refine the definition of the family:hasSibling relationship such that a person is no longer derived to be its own sibling. However, a FILTER clause could be used in the SPARQL query such that only distinct pairs of siblings are returned.

Now that the knowledge base is equipped with a reasoner it contains both explicit and implicit knowledge. This has severe consequences for updates to the knowledge base.

6.3 Updates to the Knowledge Base

Updates to knowledge bases is a topic which has been studied intensively for many years. Although these research activities stem from different communities (see Section 12.5 for a comparison and discussion) the problem statements of these works are the same. How should an update be performed if the update causes inconsistencies or if it is ambiguous? Consider the next example:

Example 6.6

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix family: <http://family.org#>.
@prefix : <http://example.org#>.

:Peter family:hasParent :Susan.
family:hasParent  owl:inverseOf family:hasChild.
```

It is explicitly known that :Susan is the mother of :Peter while it is implicitly known that (:Susan family:hasChild :Peter). What is expected to happen in a knowledge base if the statement (:Susan family:hasChild :Peter) has to be inserted? There can be different solutions:

a) The update operation is executed regardless of the intensional data of the knowledge base. As long as the statement is not already *explicitly* contained in the knowledge base, the statement is added. This may, as in the example above, cause redundancies.

b) The update operation is executed such that a desired state will be reached. Therefore, not only the extensional data but also the intensional knowledge has to be considered. With regard to the example above, for asserting the statement nothing has to be done because the statement is already (implicitly) known.

In order to be able to specify which kind of update is intended by the user, there are *intensional* update operations to the knowledge base in SWAN :

- **assert**($sub,$pred,$obj) Assure that the given statement will be entailed by the knowledge base after the update.
- **retract**($sub,$pred,$obj) Assure that the given statement will not be entailed by the knowledge base after the update.

All update operations in SWAN are, internally, operations on the graph structure which are realised by the use of the JENA framework. The commands of the user interface of SWAN provide the user the possibility to specify whether an update on the graph (an extensional update) or on the knowledge base (an intensional update) is desired. Hence, the first group of updates (as described in Section 6.1.1) comprises of the insert, delete, and update operations, all of which directly effect the RDF graph by adding, removing or updating statements. The second group comprises the assert and retract operations, which assure that the knowledge base is in a specified state after the update operation. While the former group needs no further explanation, the intensional updates are explained in detail in the following.

6.3.1 Retract

There are two possible update mechanisms for the removal of a statement: delete and retract (the delete-resource operation corresponds to a set of delete operations). A delete operation will finish successfully if the statement can be removed and the knowledge base is still valid. The effect of a delete operation, however, can be ambiguous: it might be executed successfully even in case that the statement is still entailed by the knowledge base afterwards. This can happen in situations like in the example above when the knowledge base contains redundancies. For this reason it is not necessarily possible to predict the state of the knowledge base after a delete operation. The semantics of the retract operation on a knowledge base, however, is well-defined. It is only considered to complete successfully if a statement is not entailed by the knowledge base after the update. Thus it can be regarded as a guarantee towards the state of the knowledge base after the completion of the retract operation. Hence, if the statement (:Susan family:hasChild :Peter) was added to Example 6.6 then the retraction of the information (:Peter family:hasParent :Susan) is only possible if also the statement (:Susan family:hasChild :Peter) is removed. So, a retract operation may consist of a set of updates. Note that retract is defined such that the update is also considered a success if the statement did not exist before.

6.3.2 Assert

The situation for the addition of a statement to the knowledge base is similar: even if a statement already exists (as an inferred statement), an insert operation would be performed. Moreover, adding a statement may cause the knowledge base to become inconsistent. Assert guarantees that a statement is entailed by the knowledge base afterwards. Hence, it is again a guarantee towards the state of the knowledge base after the update. In order to achieve that state, an assert operation may consist of an arbitrary number of updates. If, however, the knowledge base already entails the statement nothing has to be done. The assert fails, if the statement would invalidate the model.

These considerations about knowledge base updates are reflected in the following formal specification.

6.4 Formal Specification of Updates

The formal evaluation of updates in Swan in this section is given from a model-theoretic point of view. OWL entailment offers a model theory for a given RDF graph which consists of a Description Logic *ABox* and *TBox* (see Section 3.6). Furthermore, a distinction can be made between *base facts* and *derived facts*. The former are contained in the underlying graph whereas the knowledge base contains both base and derived facts. A knowledge base is not a simple RDF graph extended with further edges but it is a theory which consists of an RDF graph and a set of formulas (OWL axioms) that allows to draw further conclusions.

6.4.1 Graph Updates

A set of updates U on a graph G consists of either insertions of triples ins(x,y,z) or deletions of triples del(x,y,z).

$$\begin{array}{rcl} (\mathsf{ins}(x,y,z),G) & \mapsto & G \cup (x,y,z) \\ (\mathsf{del}(x,y,z),G) & \mapsto & G \backslash (x,y,z) \end{array}$$

A modification of a triple consists of a deletion followed by the insertion of a triple such that exactly one of the positions in the corresponding triple differs:

$$\begin{array}{rcl} (\mathsf{mod}((x,x'),y,z),G) & \mapsto & (G \backslash (x,y,z) \cup (x',y,z)) \\ (\mathsf{mod}(x,(y,y'),z),G) & \mapsto & (G \backslash (x,y,z) \cup (x,y',z)) \\ (\mathsf{mod}(x,y,(z,z')),G) & \mapsto & (G \backslash (x,y,z) \cup (x,y,z')) \end{array}$$

Equivalently, the notation $mod(t,t')$ can be used for modifications, where t is the original triple and t' is the resulting triple.

Applying updates. The effect of a set of updates U on a graph G is defined via the *Apply* operator. It is a mapping from a pair (U,G) to a result graph:

Definition 6.1 (Apply)

$$\begin{array}{rcl} \mathit{Apply} & : & 2^{\mathcal{U}} \times \mathcal{G} \rightarrow \mathcal{G} \\ & & (U,G) \mapsto G \ \backslash \ \{(x,y,z) : del(x,y,z) \in U\} \\ & & \qquad\qquad\quad \cup \ \{(x,y,z) : ins(x,y,z) \in U\} \end{array}$$

U is consistent if

$$\forall (x,y,z) : (ins(x,y,z) \in U \rightarrow del(x,y,z) \notin U)$$

Given that U is consistent, all $u \in U$ can be applied in arbitrary order:

$$\mathit{Apply}(\{u_1 \ldots u_n\},G) = (\mathit{Apply}(u_1,\mathit{Apply}(\ldots,\mathit{Apply}(u_n,G))))$$

□

Note that a set of updates $U = \{ins(x, r, y), del(y, r^{-1}, x)\}$ can be applied consistently to a *graph* because both $ins(x, r, y)$ and $del(y, r^{-1}, x)$ exist as explicit statements. The situation for updates to intensional knowledge (e.g. r^- is the inverse property of r) is different.

6.4.2 Updates to Intensional and Derived Knowledge

While insert, delete, and update only effect the graph (the ABox and TBox of the knowledge base), the definition of assert and retract has to consider changes to the knowledge base. In the following, updates to a knowledge base are examined with respect to the theory.

Definition 6.2 (Theory of a Graph G)
The relationship between a theory Th and a graph G is defined for closed formulae φ if

$$Th(G) = \{\varphi : Th \models_{OWL} \varphi\} \qquad \square$$

Definition 6.3 (Minimal Graph)
Let G be any graph and $Th := Th(G)$. If there is no G' such that $G' \subsetneqq G$ and $Th = Th(G)$, then G is a minimal graph. $\square$

Definition 6.4 (Inverse Properties)
If $Th(G \cup (x, r, y)) \models (y, r^-, x)$ then r^- is the inverse property of r . $\square$

For the definition of updates to the knowledge base the following restrictions are introduced:

a) Neither assert nor retract are allowed to perform changes on the TBox.

b) A theory update consists of minimal changes (according to a minimal changes semantics as described below).

No Changes to the TBox. Retract and assert operations are only defined for statements of the ABox. There are two reasons for this restriction.

- First, consider again Example 6.6 where the deletion of the statement (:Susan family:hasChild :Peter) was not possible. That statement is derived and does not exist as a base fact, hence, it cannot be *deleted*. If this statement has to be *retracted*, there are, theoretically, two possibilities: either delete the ABox statement (:Peter family:hasParent :Susan) or the TBox statement (family:hasMother owl:inverseOf family:hasChild). The deletion of the former is reasonable because the statements are closely related and the deletion reflects the intention of the update operation. A deletion of the other statement would also come to the desired result, but it would, at the same time, cause the removal of all other (derived) statements that make use of the inverse nature of that property definition.

- Intensional updates to the TBox can hardly be achieved. For an ABox, there are situations where a deleted statement can be reconstructed by derivation from another statement. There is no comparable situation for

TBoxes. This can be explained by the fact that there are no bidirectional axioms like owl:inverseOf as class constructors. Therefore, the delete operation should always be sufficient for the specification of an update of the TBox. The same is true for insert operations on the TBox.

Hence, assert and retract operations are defined for ABox updates only. Updates to the TBox have to be explicitly defined by insert or delete operations.

Minimal Updates. The restriction of effects of updates to minimal changes is proposed in order to prevent extensive changes to the TBox caused by theory updates. For example, consider a transitive family:hasAncestor relationship: (:Peter family:hasAncestor :Susan) (:Susan family:hasAncestor :Mary). It can be deduced that :Mary is :Peters ancestor. Retracting the inferred statement (:Peter family:hasAncestor :Mary) is only possible by removing both of the explicit statements followed by inserting the positive *and* the negative disjunction of the two statements (one does definitely exist, but not the two of them). This disjunction can be expressed in an OWL ontology, but it involves massive changes to the TBox: Two statements can be defined to be mutually exclusive only by constructing complex TBox assertions in form of disjoint classes from named individuals (nominals).

Definition 6.5 (Assert)
Consider a graph G and an ABox statement $s = (x, y, z)$. Now consider a set U of ABox updates to G such that $Th(U(G)) \models s$.

U is a *minimal set of updates* with respect to the assertion of s to G if for all U'

$$\text{if } U' \subsetneqq U \text{ then } Th(U'(G)) \not\models s.$$

Two sets of updates U_1 and U_2 are equivalent if

$$Th(U_1(G)) = Th(U_2(G))$$

A set $\mathcal{U} = \{U_1, U_2, \ldots, U_m\}$ of minimal sets of updates is *unambiguous* iff all U_i are equivalent.

Consider the set $\mathcal{U} = \{U_1, U_2, \ldots, U_n\}$ of all minimal sets of updates with respect to the assertion of s to G such that all $U_i \in \mathcal{U}$ are sets of ABox updates. If $\mathcal{U}$ is unambiguous then let $\mathcal{U}^{+m}_{G,s} := \mathcal{U}$.

If $\mathcal{U}^{+m}_{G,s}$ is defined for G, s and $\mathcal{U}^{+m}_{G,s} \neq \emptyset$ then s is an assertable statement to G. For the assertion of s to G any U from $\mathcal{U}^{+m}_{G,s}$ can be chosen. The signature and definition of *assert* are:

$$\begin{array}{rcl} assert & : & s \times \mathcal{G} \to \mathcal{G} \\ & & (s, G) \mapsto U(G) \text{ for some } U \in \mathcal{U}^{+m}_{G,s} \end{array}$$

□

Note that $\emptyset$ is the only minimal set of updates with regard to $assert(s, G)$ if $Th(G) \models s$.

Note also that despite the fact that sets of minimal updates can be equivalent there might be a *preferable* set of minimal updates. For example, $U_1 = \{\text{ins}(x, r, y)\}$ is equivalent to $U_2 = \{\text{ins}(y, r^{-1}, x)\}$. If, however, r is the predicate of the statement s that is to be asserted then U_1 is preferred over U_2 because it is closer to the intention of s.

Definition 6.6 (Retract)
Consider again a graph G and an ABox statement $s = (x, y, z)$. Now sets U of ABox updates to G are considered such that $Th(U(G)) \not\models s$.

U is a minimal set of updates with respect to the retraction of s from G if for all U'

$$\text{if } U' \subsetneqq U \text{ then } Th(U'(G)) \models s.$$

Consider the set $\mathcal{U} = \{U_1, U_2, \ldots, U_n\}$ of all minimal sets of updates with respect to the retraction of s from G such that all $U_i \in \mathcal{U}$ are sets of ABox updates. If $\mathcal{U}$ is unambiguous then let $\mathcal{U}_{G,s}^{-m} := \mathcal{U}$.

If $\mathcal{U}_{G,s}^{-m}$ is defined for G,s and $\mathcal{U}_{G,s}^{-m} \neq \emptyset$ then s is retractable from G. For the retraction of s from G, any U from $\mathcal{U}_{G,s}^{m}$ can be chosen. The signature and definition of *retract* are:

$$\begin{aligned} \textit{retract} \quad : \quad & s \times \mathcal{G} \to \mathcal{G} \\ & (s, G) \mapsto U(G) \text{ for some } U \in \mathcal{U}_{G,s}^{-m} \end{aligned}$$

□

The next chapter will deal with the actual realisation of these operations. This is achieved by the use of triggers which *complete* the intensional update.

Chapter 7

RDF-Triggers: An Active RDF-Database

7.1 Motivation

In Section 6.4.2 the problems of knowledge base updates have been analysed formally. Furthermore, the semantics of the theory update operations assert and retract were given. This chapter describes the concept of knowledge base triggers in SWAN, which allow to *realise* intensional updates as defined by assert and retract.

The presence of intensional data poses additional problems in comparison to simple graph updates. For example, consider a situation where an action demands for the deletion of facts from the knowledge base. What if that knowledge does not exist explicitly but only as derived facts? This situation is illustrated with the next example:

```
:Peter  family:hasParent  :Susan  .
family:hasChild  owl:inverseOf  family:hasParent.
```

The following statement is entailed by the knowledge base but it is not contained in the underlying graph:

```
:Susan  family:hasChild  :Peter  .
```

Hence, the statement cannot be removed from the graph by directly deleting it. This can only be achieved by the retract operation. The intensional update itself, however, does not specify what update on the graph level has actually to be carried out. Rather, it specifies the desired result. Therefore, intensional updates have to be *completed*. This means that update operations have to be performed instead of or in addition to the intensional update. For example, instead of retracting (:Susan family:hasChild :Peter) the statement (:Peter family:hasPartent :Susan) should be deleted. The completion of intensional updates can be realised by knowledge base triggers.

7.2 Classification of Triggers

7.2.1 Trigger Basics

The trigger mechanism for an RDF knowledge base as it is used in SWAN has been presented in [MSvL06]. Whereas triggers are common with databases there exists, to the best of the authors knowledge, no comparable trigger mechanism for RDF knowledge bases.

The basic facilities for storage, access, and manipulation of RDF data are provided by the JENA-framework (see again Figure 6.1). The trigger implementation in SWAN adds the possibility for the user to define how the knowledge base will react upon changes. Triggers can be added to the knowledge base by the command register-trigger, the removal of a trigger is possible with the command delete-trigger.

Triggers follow the ECA (*event-condition-action*) paradigm, the syntax of a knowledge base trigger in SWAN is

```
ON event WHEN condition DO BEGIN action END;
```

The relevant events on this level are the RDF update operations assert, retract, insert, delete, and modify. Optionally, the event definition can be refined such that only statements about individuals of a given class are considered. Quite similar to SQL triggers, events bind OLD and NEW variables that allow other trigger components access to information about subject, predicate, or object of the updated item. The condition of a trigger is optional and is specified by a SPARQL query, which is evaluated on the knowledge base. Additional variables can be bound by the condition. The action part of the trigger consists of a sequence of commands, either updates, the raising of *events* or the sending of messages.

Before going into the details of knowledge base triggers it is necessary to clarify the meaning of change.

7.2.2 Notions of Change

There are different notions which are used in the context of updates to a knowledge base.

Updates. An *update* is an operation which specifies either explicitly or implicitly what has to be done (see Section 6.3).

Changes. The execution of the *update* results in a *changed* knowledge base, i.e., the differences of the states of the knowledge base before and after the update. *Changes* are the visible effects of *updates.*

This distinction is reflected in different kinds of triggers that are available in SWAN. On the one hand, there are *pre-reasoning* triggers which react upon *updates* to a knowledge base. On the other hand there are triggers that react upon the *changes*. These triggers are called *post-reasoning* triggers. The characteristics of both kinds of triggers are explained in the following.

7.2.3 Pre-Reasoning Triggers

Pre-reasoning triggers react when the specified *update* occurs but *before* any *modifications* to the data have been made. Therefore, the specification of the event part of these triggers directly relates to the update operation name:

```
ON {INSERT|ASSERT|DELETE|RETRACT|UPDATE} OF property
    OF INSTANCE [OF class]
```

If an update is to be executed all pre-reasoning triggers are evaluated by comparing the event definition of the trigger to the update that is to be performed. Optionally, a condition has to be examined. Those triggers that eventually do fire upon that update add their consequences to a queue of additional updates. When all triggers have been evaluated then all updates in the queue are executed as a whole and the resulting knowledge base is checked for consistency. In case that the knowledge base becomes inconsistent, the update is rejected and all changes are rolled back.

Pre-reasoning triggers can be used for two different purposes. Firstly, they can ensure the integrity of the knowledge base. For example, consider an update which adds the statement (:Susan family:hasHusband :Jeff) to the following knowledge base:

Example 7.1

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix family: <http://family.org#>.
@prefix : <http://example.org#>.

family:hasHusband a owl:FunctionalProperty.
:Susan family:hasHusband :Jack.
:Jack owl:differentFrom :Jeff.
```

Here, family:hasHusband is defined as a functional property. Inserting the new statement violates the ontology with respect to this property. One way to prevent the knowledge base from becoming inconsistent is to reject the update. Another possibility is to use a pre-reasoning trigger which modifies the property of the existing family:hasHusband relationship of :Susan to family:divorcedWith so that the statement can be inserted. The next example shows the definition of a trigger, which can be used in this manner:

Example 7.2

```
ON INSERT OF http://family.org#hasHusband
WHEN select ?husband
     where {<$new.subject>
            <http://family.org#hasHusband>
            ?husband .}
DO BEGIN
update($new.subject, http://family.org#hasHusband, $husband)
    set predicate = http://family.org#divorcedWith;
END;
```

It is necessary for pre-reasoning triggers to react before the modifications occur and, even more important, before the integrity of the knowledge base is checked by the reasoner. For that reason the name *pre-reasoning* triggers was chosen.

A closely related purpose for pre-reasoning triggers is the completion of intensional updates. Here, the trigger specifies what updates should be performed on the occurence of an assert or retract operation. These additional updates have to be specified such that the desired state of the knowledge base is reached. Consider the following example:

Example 7.3

```
ON RETRACT OF http://family.org#hasChild OF INSTANCE
DO
BEGIN
 delete($old.object, http://family.org#hasParent, $old.subject);
END;
```

This example also demonstrates, how variable bindings can be used in the trigger definition. One possibility is to address *OLD* and *NEW* values, very much like with database triggers. For example, the subject of an inserted statement (from the event) can be used in the action part with *$new.subject*.

While pre-reasoning triggers react upon updates, the situation is different with post-reasoning triggers.

7.2.4 Post-Reasoning Triggers

Post-reasoning triggers do not react upon updates. Rather, the *changes* in the knowledge base *after* an update occured are the events which activate the post-reasoning triggers. Post-reasoning triggers are evaluated as soon as all modifications have been made. At that time, also the deductions by the reasoning process will be complete. Hence the name *post-reasoning* triggers. They can be divided into two classes. Those that perform external actions like sending messages or raising events. They are put into a queue for later execution whereas those post-reasoning triggers that cause update operations are executed immediately. In case that an update fails, the changes have to be reverted to the state before the initial update. If all updates are executed successfully, the queue of external actions can be processed. After that, the update process terminates.

The event part of a post-reasoning trigger specifies the changes that will activate the trigger:

- `ON {INSERTION|MODIFICATION|DELETION} OF` *property* `OF INSTANCE [OF` *class*`]` is raised, if a property is added to/updated/ deleted from a resource (optionally: of the specified class).
- `ON {CREATION|MODIFICATION|DELETION} OF INSTANCE OF` *class* is raised, if a resource of a given class is created, modified or deleted.
- `ON NEW PROPERTY OF INSTANCE [OF class]` is raised, if a new property is added to an instance (optionally: to a specified class). This extends `ON INSERTION OF property OF INSTANCE` to properties that cannot be named (are unknown) during the rule design.
- `ON NEW STATEMENT ABOUT INSTANCE [OF class]` is raised, if a new statement is added to an instance (optionally: of a specified class). This extends

`ON NEW PROPERTY` to the case that a new value for an already existing property is added that cannot be named (are unknown) during the rule design.

- `ON NEW CLASS` is raised, if a new class is introduced,
- `ON NEW PROPERTY [OF class]` is raised, if a new property (optionally: of a specified class) is introduced (in the metadata).

Post-reasoning triggers give specifications of the behaviour for the domain node. They allow to define *what else* has to be done when an event occurs. These specifications include either further updates or the raising of events. Example 7.4 presents an post-reasoning trigger which causes the raising of an event:

Example 7.4

```
ON INSERTION OF http://family.org#hasParent OF INSTANCE
WHEN select ?sibling
    where {<$new.object>
            <http://family.org#hasChild>
            ?sibling .
    FILTER (?sibling != <$new.subject> ). }
DO
BEGIN
 RAISE EVENT
 (
   <family:newChild name="$new.subject"
     xmlns:family="http://family.org#">
      <family:hasSibling name="$sibling"/>
   </family:newChild>
 );
END;
```

Furthermore, this example illustrates how variable bindings can be offered by the condition part of the rule (*WHEN* clause). The SPARQL query selects all children of the new object. Although the new child will already be part of the knowledge base at the time of that query, it is not returned as a query result because of the applied FILTER.

Triggers react on specified updates to resources in the knowledge base. If two or more resources are equivalent, the same trigger might fire more than just once upon a single update. In the following it is explained how triggers are evaluated in such situations.

7.2.5 Trigger Evaluation and Redundancy

Consider again the data from Example 6.4 and the trigger definition in Example 7.4. If the statement (:Gloria family:hasParent :Susan) is inserted, the trigger causes two events to be raised for every child of :Susan. This contradicts the expectation of only one event per child. For an explanation one has to consider that :Susan and :Katrin are defined to be the same. From the inserted statement the additional knowledge (:Gloria family:hasParent :Katrin) can be derived. The trigger will also fire upon this statement as it is part of the *changes* in the knowledge base. The question that arises in this context is whether triggers should be activated on every change or only on semantically distinct changes.

Here, the decision was made in favor of the first variant and only semantically distinct events are risen. It is assumed that the knowledge of the application domain is distributed properly such that it makes no difference for other nodes, which of the possible events is risen.

Technically, this is done by analysing the list of updated statements. If two statements are equivalent, one of them is filtered out. The remaining statements will be passed on for trigger evaluation. This does, of course, only effect post-reasoning triggers.

Until now, only the event part that causes the trigger to fire has been examined. Next it is shown, how the action part has to be specified.

7.2.6 Actions in Trigger Definitions

The action part of the trigger consists of a sequence of commands, either updates, the raising of *events* or the sending of messages. The difference between the raising of an event and the sending of a message is that events have to be well-formed XML fragments, which are sent to the domain broker. *Messages*, however, can be sent to arbitrary recipients with ordinary text content.

All updates that are consequences of a trigger are executed in the same way as any other knowledge base update. This will, in turn, cause the evaluation of pre- and post-reasoning triggers. The following actions are possible:

- `DO BEGIN`
  {`INSERT|ASSERT|MODIFY|DELETE|RETRACT`}(subject, predicate, object);
  `END;`
- `DO BEGIN RAISE EVENT` (xml-fragment);
  `END;`
- `DO BEGIN SEND` (recipients-http-address ; message-text) `END;`

The action part is executed for each tuple of variable bindings matched by the event part. In Example 7.4 the trigger reacts, *after* a new statement (x family:hasMother y) has been inserted. This means that an event will be risen for every other child of the mother y.

The different kinds of triggers in SWAN are now known. Next, a formal specification for these triggers is given.

7.3 Formal Specification of Triggers

7.3.1 Computing Changes

The *changes* in a knowledge base upon a set of updates U are calculated from the difference of two theories Th and Th' where $Th' = Th \cup U$. The difference can be computed as the difference of two sets of triples because the theories contain only positive ground atomic facts.

The *difference* between theories is a set of insert and delete operations which is computed by δ:

$$\begin{aligned} \delta: \quad \mathcal{M} \times \mathcal{M} &\to 2^{\mathcal{U}} \\ (M, M') &\mapsto Ins_0(M, M') \cup Del_0(M, M') \end{aligned}$$

where

$$\begin{array}{rcl} Ins_0(M,M') & = & \{(x,y,z) : (x,y,z) \in M' \wedge (x,y,z) \notin M\} \\ Del_0(M,M') & = & \{(x,y,z) : (x,y,z) \in M \wedge (x,y,z) \notin M'\} \end{array}$$

This *difference*, however, is only an approximation of the *changes* from M to M': If M' results from M by applying a set of updates U, and U contains any modifications, these are contained in $\delta(M,M')$ as pairs of insertions and deletions.

With respect to a given set of updates U, the *changes* between two specialised theories M and M' are computed by Δ:

$$\begin{array}{rll} \Delta : & (\mathcal{M} \times \mathcal{M} \times 2^{\mathcal{U}}) & \rightarrow 2^{\mathcal{U}} \\ & (M, M', \{U_1, \ldots, U_k\}) & \mapsto Del(M,M') \cup Ins(M,M') \cup Mod(M,M') \end{array}$$

Modifications in Δ are, with regard to an *update* in U, either explicit or implicit. Implicit modifications occur for updates involving a triple (s,p,o) where p is either an inverse, a symmetric or a transitive property.

- Let p^- be an inverse property of p. For all modifications $\mathsf{mod}(t_1,t_2)$ in U exists an implicit modification $\mathsf{mod}(t_1^-,t_2^-)$ in Δ where $t_1^- = (o,p^-,s)$ is an inverse triple of t_1 and either $t_2^- = (o',p^-,s)$ or $t_2^- = (o,p^-,s')$ is an inverse triple of the respective t_2.
- Let p be a symmetric property. For all modifications $\mathsf{mod}(t_1,t_2)$ in U there exists an implicit modification $\mathsf{mod}(t_1^s,t_2^s)$ in Δ where $t_1^s = (o,p,s)$ is the symmetric triple to t_1 and either $t_2^s = (o',p,s)$ or $t_2^- = (o,p,s')$ is the symmetric triple to the respective t_2.

Implicit modifications due to the use of transitive properties are not considered here. There are situations, especially with large sets of updates U, where the relationship between these modifications and the modifications in U cannot be reconstructed.

Del, Ins, and Mod are defined as follows:

Definition 7.1 (Updates)

$$\begin{array}{rcl} Del(M,M') & = & Del_0(M,M')\backslash \\ & & \{t_1 = (x,y,z) \in Del_0(M,M') : \\ & & \quad \text{there is a } t_2 \text{ such that } \mathrm{mod}(t_1,t_2) \in U \text{ or} \\ & & \qquad \mathrm{mod}(t_1^-,t_2^-) \in U \text{ or} \\ & & \qquad \mathrm{mod}(t_1^s,t_2^s)) \in U\} \\ Ins(M,M') & = & Ins_0(M,M')\backslash \\ & & \{t_2 = (x,y,z) \in Ins_0(M,M') : \\ & & \quad \text{there is a } t_1 \text{ such that } (\mathrm{mod}(t_1,t_2) \in U \text{ or} \\ & & \qquad \mathrm{mod}(t_1^-,t_2^-)) \in U \text{ or} \\ & & \qquad \mathrm{mod}(t_1^s,t_2^s)) \in U\} \\ Mod(M,M') & = & \{\ mod(t_1,t_2) : mod(t_1,t_2) \in U \text{ such that} \\ & & \qquad t_1 \in M \text{ and } t_2 \in M'\ \} \end{array}$$

□

7.3.2 Trigger Evaluation

The general structure of a trigger is

$$\textsf{on}\ u\ \ \textsf{when}\ \ \gamma\ \textsf{do}\ u'$$

where γ is a conditional expression, and u' is a set of updates. Note that u, γ may contain variables. A trigger is activated if a substitution σ exists such that $\sigma(u_i) = u'_i$ for a $u_i \in U$.

First, a general definition for the evaluation of triggers will be given. Later, this definition will be specified for pre-reasoning and post-reasoning triggers. The overall process of trigger evaluation is described schematically in Figure 7.1.

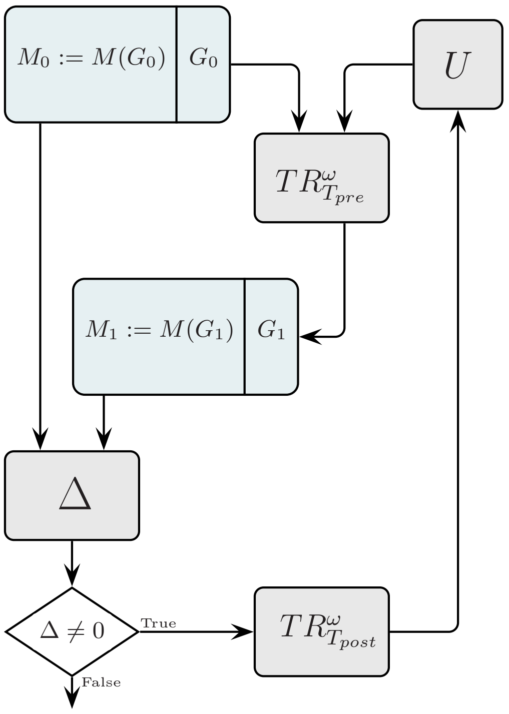


Figure 7.1: Evaluation of Triggers and Updates

Triggers are evaluated using the trigger evaluation operator TR. The signature of $TR_{T,G}$ is:

$$TR_{T,G} \quad : \quad 2^{\mathcal{U}} \rightarrow 2^{\mathcal{U}}$$

Definition 7.2 ($\boldsymbol{TR_{T,G}}$)
For a set T of triggers and a graph G, an update U is complemented by $TR_{T,G}$ such that

$$\begin{array}{rcll} TR_{T,G}(U) & = & U \cup \bigcup_{t\in T}(& \sigma(u'_t) \mid t = (\mathsf{on}\ u_t\ \mathsf{when}\ \gamma\ \mathsf{do}\ u'_t) \text{ and} \\ & & & \sigma \text{ is a substitution such that } \sigma(u_t) \in U \text{ and} \\ & & & u_t \in U \text{ and} \\ & & & M(G) \models_{OWL} \sigma(\gamma)) \end{array}$$

The fixpoint of $TR^{\omega}_{T,G}$ is defined as follows:

$$\begin{array}{rcl} TR^{0}_{T,G}(U) & = & TR_{T,G}(U) \\ TR^{i}_{T,G}(U) & = & TR_{T,G}(TR^{i-1}_{T,G}(U)) \text{ for } i \geq 1 \\ TR^{\omega}_{T,G}(U) & = & \lim_{i\to\infty} TR^{i}_{T,G}(U) \end{array}$$

$TR^{\omega}_{T,G}$ is well-defined if the iteration ends in a finite number of steps:

$$TR_{T,G}(TR^{imax}_{T,G}(U)) = TR^{imax}_{T,G}(U)$$ □

Pre-reasoning triggers. Pre-reasoning triggers T_{pre} are intended for the completion of an initial set of updates such that consistency of the specialised theory M is guaranteed after all updates are applied. It is necessary to compute a fixpoint, because the updates by the triggers again have to be completed . The operator $TR^{\omega}_{T_{pre},G}$ is used for that fixpoint computation, resulting in the completed set of updates U_{comp}.

$$U_{comp} \quad = \quad TR^{\omega}_{T_{pre},G}(U_{base})$$

Applying U_{comp} to G results in G' for which a specialiced theory M' exists. δ computes the set of *changes* Δ from M', M and U_{comp}.

$$\begin{array}{rcl} M' & = & M(\underbrace{Apply(U_{comp}, G)}_{=:G'}) \\ \Delta & = & \Delta(M, M', U_{comp}) \end{array}$$

Post-reasoning triggers. Post-reasoning triggers T_{post} are evaluated using the operator $TR_{T_{post},G}$, reacting on the changes Δ:

$$U_{base} \quad = \quad TR_{T_{post},G}(\Delta)$$

The resulting updates U_{base} are the consequences of the post-reasoning triggers.

Raising of Events. The consequence of a post-reasoning trigger can be either an update or the raising of an event. As events shall be raised only in case of a successful update, the raising of events has to be delayed until the update process TU is finished and no further updates are triggered. All events that have been accumulated up to this point are then risen simultaneously. The raising of events is realised by applying the TR operator for sets of event-raising triggers T_{evt}. The signature is the following:

$$\begin{aligned} TR_{T_{evt},G}: \quad 2^U \to & \quad (2^U, 2^E) \\ \mathcal{U} \mapsto & \quad (\mathcal{U}', \mathcal{E}) \end{aligned}$$

where E is the set of events that are delayed until the end of the update process. When $TR_{T_{post},G}$ is evaluated on a set of changes Δ also $TR_{T_{evt},G}$ is evaluated on the same Δ.

Note that this division of post-reasoning triggers into event raising triggers and others causing further updates is only theoretical. In the implementation, both kinds of post-reasoning triggers are evaluated at the same time. With regard to the formal definition, however, this assumption can be made without loss of generality as the event triggers have no side effect on the update process at all.

Definition 7.3 (Update execution *TU*)
The execution of updates is defined by the operator TU as follows:

$$\begin{aligned} TU \quad &: \quad M \times 2^U \to (M, \Delta) \\ TU(M,\Delta) \quad &= \quad (\underbrace{M(Apply(TR^{\omega}_{T_{Pre},G}(TR_{T_{post},G}(\Delta)), G))}_{=:M}, \underbrace{\delta(M, M(G), U)}_{=:\Delta}) \\ TU^0(M,\Delta) \quad &= \quad TU(M,\Delta) \\ TU^1(M,\Delta) \quad &= \quad TU(TU^0(M,\Delta) \downarrow 1 \ , \ TU^0(M,\Delta) \downarrow 2) \\ TU^{i+1}(M,\Delta) \quad &= \quad TU(TU^i(M,\Delta) \downarrow 1 \ , \ TU^i(M,\Delta) \downarrow 2) \\ TU^{\omega}(M,\Delta) \quad &= \quad \lim_{n \to \infty} TU^n(M,\Delta) \downarrow 1 \\ & \quad \text{is reached if } TU^i(M,\Delta) \downarrow 2 = \emptyset \end{aligned}$$

The application of a set of completed updates U (pre-reasoning triggers already fired once) is defined as:

$$TU^{\omega}(\underbrace{M(Apply(U,G))}_{=:M}, \underbrace{\delta(M(G), M(Apply(U,G)), U)}_{=:\Delta})$$

□

For an initial update u the set of induced updates U has to be calculated:

$$U := U_{comp} := TR^{\omega}_{T_{pre},G}(u)$$

TU^{ω} can be expressed with regard to the the initial update u as

$$\begin{aligned} \Delta \quad &= \quad \delta(\underbrace{M(G)}_{=:M}, \underbrace{M(Apply(TR^{\omega}_{T_{pre},G}(u), G))}_{=:M'}, \underbrace{TR^{\omega}_{T_{pre},G}(u)}_{=:U}) \\ M \quad &= \quad M(G) \end{aligned}$$

This formal specification completes the considerations about knowledge base triggers in Swan. In the following chapter, ACA rule processing is described as another central aspect of the Swan architecture.

Chapter 8

The ACA-Rule-Aware Application Node

The infrastructure of the MARS architecture is orchestrated by the exchange of events, which are distributed by a domain broker. The integration of the application domain nodes, however, depends on the actions that they receive from the domain broker. The abstract actions from the application domain are translated into knowledge base updates by the use of ACA rules (see Section 5.2.3). In the scope of this thesis an ACA rule mapping mechanism was integrated into the SWAN architecture. The following chapter gives a description of this component.

8.1 Wrapper Components

The translation of abstract actions is realised by a wrapper around the knowledge base. Abstract actions are translated by the help of XQuery scripts into knowledge base updates, given as XML fragments as presented in Section 6.1.2. These fragments are passed to the knowledge base for execution. Hence, the ACA rule mapper consists mainly of an XQuery engine.

Moreover, a set of built-in XQuery functions is available. These functions create proper XML fragments for update operations. For example, the following XQuery function returns an XML fragment for an rdfu:insert operation:

Example 8.1 (rdfu-Action insert)

```
declare function rdfu:insert($sub, $pre, $obj) {
   if (exists($obj)) then
   (
    <rdfu:insert>
      <rdf:subject rdf:about="{string($sub)}" />
      <rdf:predicate rdf:about="{string($pre)}" />
      {  if (rdfu:isLiteral($obj))
         then <rdf:object>{string($obj)}</rdf:object>
         else <rdf:object rdf:about="{string($obj)}" />
      }
    </rdfu:insert>
   ) else ()
};
```

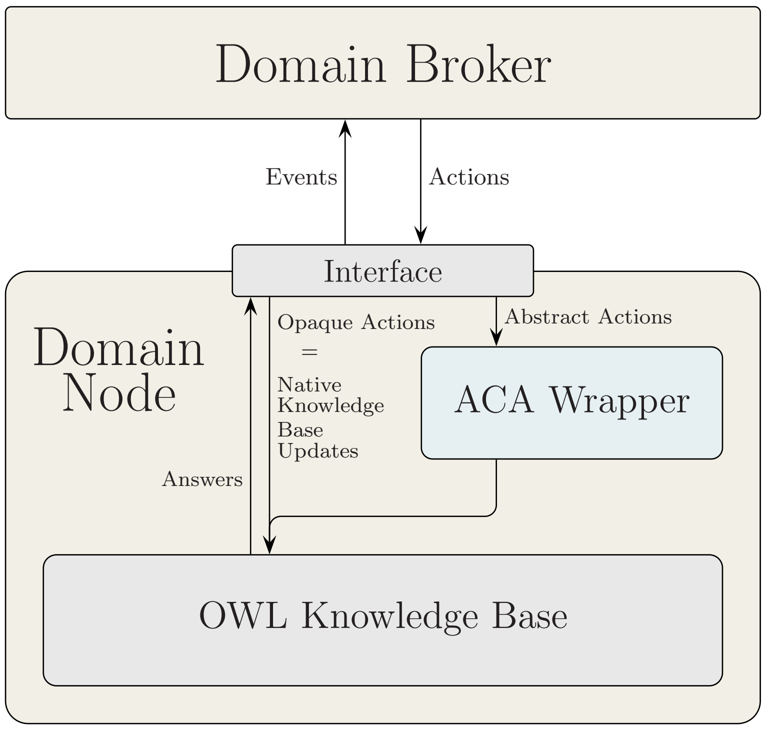


Figure 8.1: Architecture of the Domain Application Node

Instead of giving the whole XML fragment, the following XQuery function call is sufficient:

```
rdfu:insert(:Susan, family:hasChild,:Peter).
```

This is useful for better readability and also for the ease of writing of ACA rules.

ACA rules are added to or removed from the domain node by the use of the application-node actions applnode:register-aca-mapping and applnode:delete-aca-mapping.

In the following, an example of a simple application domain is given, which is used for illustrating the handling of abstract actions in Swan.

8.2 An Application Domain Example

Consider a city council that has its own application domain consisting of different services. If, for example, the civil registry office raises a new-marriage event, several other nodes will have to react upon that event. This is implemented by an ECA rule. The evaluation of the rule *ON new-marriage WHEN not already-married DO register-marriage* will cause the action family:register-marriage to be executed. The domain broker will send the action description to all appropriate application domain nodes, e.g. the tax office. The abstract action family:register-marriage has to be mapped to a knowledge base update.

First, consider the domain ontology which will be used in this chapter. It provides for a definition of the necessary vocabulary of the application domain.

Example 8.2 (Family Ontology)

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix family: <http://family.org#>.
@prefix : <http://example.org#>.
@prefix mars:     <http://www.semwebtech.org/2006/mars#> .
family:Action
        a          owl:Class ;
        rdfs:subClassOf mars:Action ;
        mars:belongs-to-domain      family: .
family:register-marriage
        a          owl:Class ;
        rdfs:subClassOf family:Action ;
        mars:belongs-to-domain      family: .
family:Female rdfs:subClassOf family:Person .
family:Male rdfs:subClassOf family:Person ;
        owl:disjointWith family:Female .
family:Groom rdfs:subClassOf family:Male.
family:Bride rdfs:subClassOf family:Female.
family:marriedTo
      a owl:FunctionalProperty ;
      a owl:SymmetricProperty ;
      rdfs:range family:Person ;
      rdfs:domain family:Person .
family:Husband owl:intersectionOf (
      family:Male
        [ a owl:Restriction ;
          owl:onProperty family:marriedTo;
          owl:cardinality 1 ;
          owl:someValuesFrom family:Female ]) .
family:Wife owl:intersectionOf (
      family:Female
        [ a owl:Restriction ;
          owl:onProperty family:marriedTo;
          owl:cardinality 1;
          owl:someValuesFrom family:Male    ]) .
```

The concepts family:Wife and family:Husband and the property family:marriedTo define aspects of the application domain, refining the concepts family:Person, family:Female, and family:Male. The fact base contains no individuals so far. Besides that, there are definitions for application node concepts: *family:Action* and its subclass *family:register-marriage.* Note that abstract action names have to be declared before they can be used in ACA mappings. Now consider the following *register-marriage* action:

Example 8.3 (Abstract Action Definition: Register New Marriage)

```
<family:register-marriage xmlns:family="http://family.org#">
      <family:Bride family:id="http://example.org#Alice_Miller"/>
      <family:Groom family:id="http://example.org#John_Doe"/>
</family:register-marriage>
```

The action is kept simple, the only elements are family:Bride and family:Groom which contain an attribute that provides an URI for identification. A real-world scenario would certainly deliver more information about the event, like dates of birth, date of marriage, maiden names, and so on. For the purpose of demonstration of the ACA rule mapping the action definition can be simplified without loss of generality.

The action family:register-marriage is no executable action like a knowledge base update, rather it is defined by the domain ontology to be an action of the application domain. The action has to be translated into an action description that is given in the domain nodes own update vocabulary.

8.3 Translating Actions into Updates

ACA rules can be given either as XQuery or XSLT scripts, which define the transformation from an abstract action (as specified in the application domain ontology) into one or a series of simple knowledge base actions.

Consider the following ACA mapping realised in form of an XQuery script. The script translates the action from Example 8.3 into a sequence of two knowledge base updates.

Example 8.4 (ACA rule for register-marriage actions)

```
declare namespace family="http://family.org#";
for $marriage in /family:register-marriage
let $bride := string($marriage/family:Bride/@family:id),
    $groom := string($marriage/family:Groom/@family:id)
return
applnode:actionSequence ((
 rdfu:insert({$bride}, "http://family.org#marriedTo",
              {$groom}),
 rdfu:insert({$bride},
              "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
              "http://family.org#Wife")
))
```

The XQuery is evaluated on the received XML fragment. The let-clause extracts the attribute values of the identifiers and binds them to the XQuery variables. The XQuery script returns an *applnode:actionSequence* which contains two simple actions. The update actions are given as XQuery function calls (rdf:insert). The *return* clause of the XQuery is equivalent to the following XML fragment:

Example 8.5 (Resulting Knowledge Base updates, given in XML)

```
<applnode:actionSequence
  xmlns:applnode="http://www.semwebtech.org/2006/application-node#"
  xmlns:rdfu="http://www.semwebtech.org/languages/2006/rdfupdate#">
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 <rdfu:insert
   <rdf:subject rdf:about="{$bride}"/>
   <rdf:predicate rdf:about="http://family.org#marriedTo"/>
   <rdf:object rdf:about="{$groom}"/>
 </rdfu:insert>
 <rdfu:insert
   <rdf:subject rdf:about="{$bride}"/>
   <rdf:predicate
    rdf:about="http://www.w3.org/1999/02/22-rdf-syntax-ns#type"/>
   <rdf:object rdf:about="http://family.org#Wife"/>
 </rdfu:insert>
</applnode:actionSequence>
```

As a result of the mapping of the action two statements will be inserted (using the namespace prefixes as defined in the ontology in Example 8.2): (:Alice_Miller family:marriedTo :John_Doe) and (:Alice_Miller rdf:type family:Wife). Because of the definition of family:Wife and family:Husband in the ontology the OWL reasoner will be able to conclude that :John_Doe is family:marriedTo :Alice_Miller and, moreover, that he is a family:Male and a family:Husband.

Note that these (internal) XML fragments do not need to contain any namespace declarations for rdfu or rdf elements, because the namespaces are added automatically by the wrapper.

In the previous examples, only the action parts in ACA rules have been considered. These rules may, however, also contain conditions.

8.4 Conditions in ACA rules

The condition component of an ACA rule is given as an rdfu:condition element in the return-clause where the resulting actions of the ACA rule have to be embedded as child elements. The condition element has to contain an ASK-attribute which specifies a SPARQL-ASK query and is evaluated on the knowledge base. If this query evaluates to *true* then all the elements which are contained within the condition element will be executed.

In the following example the ACA rule causes the insertion of a statement only in case that neither the family:Bride nor the family:Groom are already married. Note that the query uses negation by employing the *filter* clause (see also Example 3.11). The query will return *true* if neither the family:Bride nor the family:Groom already have a family:marriedTo relationship. Note also that this time the insert commands are given as XML elements.

Example 8.6 (ACA rule with a condition)

```
declare namespace  family ="http://family.org#";
for $marriage := //family:register-marriage
let $bride := string($marriage//Bride/@family:id)
let $groom := string($marriage/Groom/@family:id)
return
 <rdfu:condition xmlns:rdfu=
             "http://www.semwebtech.org/languages/2006/rdfupdate#"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:family="http://family.org#"
   rdfu:ask="OPTIONAL {{ &lt;{$bride}&gt;
                        &lt;http://family.org#married&gt; ?_X}}
             OPTIONAL {{ &lt;{$groom}&gt;
                        &lt;http://family.org#married&gt; ?_X}}
             FILTER (!bound(?_X))" >
    <rdfu:insert>
       <rdf:subject   rdf:about="{$bride}"/>
       <rdf:predicate rdf:about="http://family.org#marriedTo"/>
       <rdf:object    rdf:about="{$groom}"/>
    </rdfu:insert>
    <rdfu:insert
       <rdf:subject   rdf:about="{$bride}"/>
       <rdf:predicate rdf:about=
             "http://www.w3.org/1999/02/22-rdf-syntax-ns#type"/>
       <rdf:object    rdf:about="http://family.org#Wife"/>
    </rdfu:insert>
 </rdfu:condition>
```

Note that the curly brackets are doubled within the ASK query in order to escape them within the XQuery script. For the same reason "<" and ">" have to be escaped or given as character references (e.g. "&lt;"). □

Consider that the statement (:Alice_Miller family:marriedTo :John_Doe) is already contained in the knowledge base. Let the following action be evaluated by the ACA rule from Example 8.6:

```
<family:register-marriage>
   <family:bride family:id="http://example.org#Susan_Porter"/>
   <family:groom family:id="http://example.org#John_Doe"/>
</family:register-marriage>
```

As :John_Doe already has a family:marriedTo relationship to :Alice_Miller the rule condition is not fulfilled and no updates on the knowledge base will be made.

The examination of conditions in ACA rules completes the description of the ACA wrapper in SWAN. In the next chapter, the hybrid reasoning component of SWAN is presented.

Chapter 9

The F-Logic Reasoner Extension for Hybrid Reasoning

9.1 Introduction

The limits of the expressiveness of OWL are a well-known issue (see Section 3.7), which has led to many proposals for hybrid reasoning with conceptual languages like Description Logics. In search of a complementing formalism, F-Logic is an interesting candidate.

F-Logic is based on classical predicate calculus and adapts the concepts of classes, objects, and types from object-oriented programming. In this way, F-Logic integrates the paradigms of logic programming and deductive databases with the object-oriented programming paradigm. Unlike Description Logics, F-Logic is relationally complete[1]. It provides a simple and clear specification for many problems that are beyond the expressive power of any DL [ADG+05].

The general concept of hybrid reasoning has been presented already in Section 3.7. Hybrid reasoning in SWAN involves an OWL knowledge base and an F-Logic reasoner. The design principle of this reasoning architecture has been presented in [MSK07]. Both the OWL knowledge base and the F-Logic reasoner are self-contained. With respect to hybrid reasoning, however, the OWL knowledge base is the coordinating component. F-Logic reasoning is applied in order to *complete* the OWL knowledge base. The coupling of the hybrid reasoning components is realised by a translation of the OWL knowledge base into F-Logic expressions and, in the other direction, by queries from the hybrid reasoning core to the F-Logic reasoner (see Figure 9.1).

Before the hybrid reasoning mechanism in SWAN is explained in detail, the basic concepts of F-Logic are described.

[1]In fact, F-Logic is computationally complete as function symbols, and object creation are features of F-Logic.

9.2 F-Logic

The following considerations give an overview of F-Logic. For a formal introduction to F-Logic see Section 2.4.

Frame languages are deductive languages for knowledge representation. They are object-oriented where the objects are represented by frames. The attributes of a frame are called slots. The slots are named and their values are either literals or object references. Slots can be either single-valued or multi-valued. Frames and slots strongly correspond to instances and properties that can be identified in ontology description languages like OWL (and even more so in OIL). Because frame languages use mainly objects as modelling concepts, every object-oriented language can be regarded as a frame language. But usually the term *frame* is associated with knowledge representation and artificial intelligence.

There exist several frame languages in the area of knowledge representation, e.g. KL-ONE [BS85], OIL [HFB+00] (see again Section 3.6) and F-Logic [KL89] [KLW95].

9.2.1 Basic Concepts

Objects

The basic building blocks in F-Logic are objects. They have an object name (that can be used in order to address them) and an object identifier (for internal representation). Id-terms are terms composed of function symbols and constants and serve as names of classes, objects and attributes. Here, only function-free id-terms are considered. Following the classic convention in logic programming, constants are denoted by names beginning with a lowercase letter (whereas variable names always start with an upper-case letter). In F-Logic, entities, described via id-terms, act at the same time as *classes*, *objects*, and *methods*.

Following the object-oriented paradigm, objects can be organised by declaring them being members of classes, whereas the relationships between objects are realised by attributes.

Atoms and Molecules

In F-Logic, a basic expression consists of a *host object*, an *attribute name*, and a *result object*, all denoted by *id-terms*. Such expressions are called *data-F-atoms*.

Data-F-atoms provide information about objects. It is also possible to collect information about one object in an *F-Molecule*:

Example 9.1

```
susan : mother [ hasChild ->> { john , peter }].
```

This molecule defines that susan belongs to the class mother and gives a set of objects names (john and peter) that are result objects for the attribute hasChild. This *F-molecule* makes use of an isa-F-atom (susan : mother), that defines the relationship of an object (susan) to a class (mother). Besides that, there are *subclass-F-atoms* that define the hierarchical relationships of classes (e.g. mother :: person).

In addition to that there are also *P-molecules* that allow for the usage of predicate symbols in such a way as in predicate logic: a symbol followed by one or more *id-terms*. The expression peter[hasMother->susan] is equivalent to the *P-molecule* hasMother(peter,susan).

Attributes and Signatures

Attributes can be either single-valued (like in peter[hasMother->susan]), denoted by a single-headed arrow "->", or multi-valued, denoted by a double-headed arrow "->>" like in Example 9.1. In order to define which attributes are applicable to which classes (and their instances), *signature F-atoms* can be used. For signatures, the "->" and "=>" symbols are used (as with *data-F-atoms* a single or a double arrowhead indicates functional or multi-valued attributes). *Signature-F-atoms* are definitions, in contrast to DL, where restrictions on domain and range of a property actually are assertions. There are implementations of F-Logic, which give a semantics to the *signature-F-atoms* such that they can be used as constraints. But this is not the case for all F-Logic implementations.

Queries

An F-Logic knowledge base can be queried in a simple way by putting variables in appropriate syntactic positions of *F-atoms*. Both schema information associated with classes and the structure of individual objects can be queried in this way.

Queries are given by a conjunction of (possibly negated) *P-molecules* and *F-molecules*. The beginning of a query is denoted by "?-", any of the *id-terms* can be replaced by a variable. For example, consider the following queries:

```
?- X : mother
?- susan [hasChild ->> Y ].
?- susan[X->>Y].
```

The first query returns (given the programme from Example 9.1) the following result:

```
 X/susan
```

The second query returns all objects that are related to susan by the multi-valued attribute hasChild (john and peter). The last query returns tuples binding *attribute names* to *object names* for every *multi-valued* attribute that can be applied to the object susan:

```
 X/hasChild      Y/john
 X/hasChild      Y/peter
```

There is a special kind of variables that can be used in rules that are called *don't-care-variables* that begin with an underscore followed by an upper-case letter. They can be used as join variables in the query but are not considered in the answer to that query.

Rules

In contrast to DL, derivation of new facts is not based on assertions but on the use of derivation rules. They consist, like in Datalog, of a rule head and a rule body. The rule body is a query, the rule head (the conclusion) is a conjunction of *P-molecules* or *F-molecules*. If the query is true then also the head is true (head :- body). A rule may contain variables, it is then called a non-ground rule. Variables may occur in both the head and the body of a rule. If these variable bind tuples in the body of a rule, the same tuples are also bound in the head. Hereby, new attributes or additional values to an existing (multi-valued) attribute can be assigned to an object.

```
X : mother , Y : child
:-
Y [ hasMother -> X ].
```

For every *host object* Y that has a hasMother-*attribute* with a *result object* X, the rule derives, that X is a mother and Y is a child. For instance, if susan,peter are bound to the variables X,Y, then these variable bindings are available in the rule head, such that susan : mother and peter : child can be added to the fact base.

Similarly, the attribute hasChild of an object can be defined by the use of deductive rules instead of an explicit specification:

```
X [ hasChild ->> Y]
:-
Y [ hasMother -> X ].
```

This rule expresses that hasChild is the inverse relationship to hasMother, similar to the Example 3.7 where these relationships are defined in OWL.

9.2.2 Default Inheritance

Besides the logical derivation rules that have already been explained there are also default inheritance atoms, which can be used for the derivation of new facts.

In object-oriented systems there are two different kinds of inheritance: *structural* and *behavioural*. The first defines how signatures can be inherited, in other words, how methods are inherited by subclasses from their superclasses and made available to their instances. The latter (also called *value inheritance* or *result object inheritance*) allows to propagate results for methods from a class to its sub-classes. In F-Logic, multiple inheritance is possible.

Example 9.2 (Non-monotonic inheritance with penguins)
Consider a knowledge base that defines the class bird. All that is known about birds is that they have feathers and that they can fly. That knowledge is expressed via hasFeather *and* canFly *slots, both filled with the default value* true. *If we define* tweety *to be a bird, we automatically conclude that it has feathers and that it can fly. But what, if* tweety *is a penguin? The knowledge base defines the class* penguin *as a subclass of bird. It inherits (by structural inheritance) the property* hasFeathers *with the default value (by behavioural inheritance)* true *but overwrites the property* canFly *with the value* false. *Every instance of the class penguin inherits the default value* true *for the property* hasFeathers *and the default value* false *for the property* canFly. □

It is possible to make use of both *structural* and (non-monotonic) *behavioural* inheritance in F-Logic. Consider the following example:

Example 9.3 (Non-monotonic behavioural inheritance)

```
abraham : jew .
jew :: person .
peter:person.
```

```
4 person[believesIn*->something].
5 jew[believesIn *-> jehowa ].

7 ?- X[believesIn->Y].
```

The default in line 4 makes people typically believe in something and line 5 defines that jews normally believe in jehowa. Line 7 is a query for all host and result objects connected by a believesIn property. □

The answer to the query from the last example is:

```
X/abraham    Y/jehowa
X/peter      Y/something
```

The combination of deductive rules and default inheritance for object-oriented database systems has been evaluated in [MK01]. In Section 2.5 a formal introduction to default logic and its application to an OWL knowledge base has been given.

Some of the rich features of F-Logic have been presented above, many of them are not available in DLs. But there are also features in DLs that have no counterpart in F-Logic.

9.2.3 Comparison of F-Logic with DLs

One of the main differences between OWL and F-Logic concerns the representation of existential information. While it is hardly possible in F-Logic to express *if A is a parent there has to be an instance B that is a child* this can be specified easily in OWL:

Example 9.4 (Existential Quantifications in OWL)

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix family: <http://family.org#>.
@prefix : <http://example.org#>.
family:parent  owl:equivalentClass
[ a owl:Restriction;
  owl:onProperty family:hasChild;
  owl:minCardinality 1 ].
family:hasChild  a rdf:Property ;
  rdfs:range family:child.
:Susan a family:parent.
family:child owl:oneOf (:Peter).
```

It is known that every family:parent has a family:child, e.g. :Susan. In the example, the concept family:child is defined as an explicit enumeration of all possible objects that can be an instance of that concept. Hence :Peter is the only existing family:child. From that class definition and the existential quantification it can be deduced that (:Susan family:hasChild :Peter).

Moreover, disjunctive information can be expressed in OWL, for example: *a child is either a son or a daughter*:

Example 9.5 (Disjunctive information in OWL)

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix owl11: <http://www.w3.org/2006/12/owl11#>.
```

```
@prefix family: <http://family.org#>.
@prefix : <http://example.org#>.
family:son a owl:Class.
family:daughter a owl:Class; owl:oneOf (:Katrin :Susan).
[ a owl:AllDifferent;
  owl:distinctMembers ( :Peter :Katrin :Susan ) ].
family:child owl:equivalentClass
[ owl11:disjointUnionOf (family:Son family:Daughter)].
:Peter a family:child.
```

The example consists of the information that a family:child is either a family:Son or a family:Daughter. Furthermore :Susan and :Katrin are the only daughters in the world. This allows for the conclusion that :Peter cannot be a family:Daughter, and that he has to be a family:Son. Drawing conclusions from disjunctive information like this cannot be achieved in F-Logic.

So far, the features of F-Logic have been described. In the following a reasoning engine for F-Logic is presented.

9.3 Florid

Currently, there are several implementations of F-Logic available. For example, Flora-2 is an open-source system which was developed at Stony Brook and integrates F-Logic with HiLog [CKW93] and Transaction Logic [CS98]. OntoBroker [FAD+99] is a commercial product developed by Ontoprise. In this work, *F-Logic Reasoning in Databases* (FLORID) [FLO98] is used for F-Logic reasoning. FLORID is a C++ implementation of F-Logic and was developed by the Databases and Information Systems group at Freiburg University. It was released in version 1.0 in 1996, currently version 4.0 is available.

FLORID implements all essential features of F-Logic. Some of the features which are special to FLORID are explained in the following.

9.3.1 Handling of URIs

Normally in F-Logic, id-terms are represented by first-order variable-free terms. In FLORID, the built-in class uri is available. Instances of this class can be identified by strings, e.g.

```
"http://example.org#Susan" : uri.
```

Hereby, resource identifiers from RDF can be used directly (when enclosed in quotation marks) for object identification in F-Logic. This feature of FLORID allows for a direct translation of OWL statements to F-Logic expressions. Note that it is not possible in F-Logic programmes to use namespace abbreviations like in N3. Nevertheless, for better readability, these namespace prefixes are used in the describing texts of some of the following examples. For instance, the object name "http://example.org/family#Father" will be written as family:Father and "http://example.org#Peter" as :Peter.

9.3.2 Built-In Predicates and Object Creation

For both integers and strings there are a number of built-in methods available that allow for string manipulation (e.g. substring extraction, string matching), arithmetical operations, and aggregations (e.g. minimum, sum).

Object creation in Florid is possible by a combination of rule application and built-in functions. Newly generated terms can be used as an object name, thus creating a new object. Object *creation* is possible using the built-in functions for string manipulations. Consider the following F-Logic programme:

Example 9.6 (Creating new objects with rules)

```
 F : "http://family.org#Family",
 X["http://family.org#isFamilyMemberOf"->> F ],
 Y["http://family.org#isFamilyMemberOf"->> F ]
:- X["http://family.org#marriedTo"-> Y],
 X["http://family.org#isFamilyMemberOf"->> _F],
 Y["http://family.org#isFamilyMemberOf"->> _G],
 not _F = _G, not substr("-",_F), not substr("-",_G),
 strcat(_F,"-",TMP), strcat(TMP,_G,F)   .
```

The built-in function strcat returns a new string literal which is a concatenation of two input strings. In this example, a new family name is created from the family names of objects that are related by the http://family.org#marriedTo attribute. The *input* family names must neither be identical nor double-barelled names. The new family name is assigned to both objects that are related by http://family.org#marriedTo. Consider that the following lines are added to this programme:

Example 9.7

```
"http://example.org#John"[
    "http://family.org#marriedTo"->"http://example.org#Susan";
    "http://family.org#isFamilyMemberOf"->> "Mueller"].
"http://example.org#Susan"[
    "http://family.org#isFamilyMemberOf"->> "Luedenscheidt"].
```

Florid will draw the following conclusions from the resulting programme:

```
"http://example.org#John"[
   "http://family.org#isFamilyMemberOf"->>"Mueller-Luedenscheidt"].
"http://example.org#Susan"[
   "http://family.org#isFamilyMemberOf"->>"Mueller-Luedenscheidt"].
"Mueller-Luedenscheidt" : "http://family.org#Family".
```

9.3.3 Architecture

Storage. The actual (extensional) database is stored in the *ObjectManager*, the *ObjectManagerAccess* provides a wrapper for the *ObjectManager*. The Florid *ObjectManager* implements a *frame-based* storage component which contains a frame for every object. A frame contains slots for storing properties of an object, including its class memberships and references to other objects. The frames are extensible to additional types of properties.

Data Model. The *ObjectManagerAccess* implements the abstract data model based on the database which is stored in the *ObjectManager*, i.e., the navigation graph extended with intensional properties (transitivity of class hierarchy, downwards closure of signatures with regard to the class hierarchy , support for inheritance, object fusion, synonyms, built-in functionality for data conversion, string handling including matching regular expressions, arithmetics, aggregation operators, and annotated literals). It provides *iterator-based* declarative access to the database. The above intensional properties are not materialised, but implemented by the iterators.

Evaluation. The central FLORID *Evaluation* module (*LogicEvaluation*, *AlgebraicEvaluation*, and *AlgebraicInsert*) provides in fact a *generic* implementation of a deductive language over a data model with complex objects. *LogicEvaluation* implements seminaive bottom-up evaluation of rules. *AlgebraicEvaluation* translates rule bodies and heads into the underlying object algebra and evaluates the generated algebraic expressions using the querying interface of *OMAccess*. *AlgebraicInsert* instantiates the rule heads with the generated variable bindings and adds the corresponding facts into the database using again the *OMAccess* interface. The evaluation of algebraic expressions does not materialize any intermediate result, but is purely based on nested iterators.

Output. The *PrettyPrinter* outputs answers to queries in the variable bindings format known from Prolog or as an instantiation of the queries.

UserInterface. The *UserInterface* module allows to use FLORID from the command shell, including interactive queries, and system commands. *SystemCommands* can also be executed in programmes, mainly controlling programme execution (user-defined stratification), debugging, and output formatting.

Besides this shell-like command line interface a Web Service interface is availabe, which is explained in the following.

9.3.4 Florid Server

The reasoning capabilities of FLORID can also be used via its Web Service interface. Here, the reasoner is accessed in a client-server-mode (using HTTP). The capabilities of the FLORID server are the same compared to the stand-alone implementation. The client-server model, however, makes the reasoner available over a network infrastructure and can be used by not just one but many clients. A programme that is given to the server is evaluated in the usual way by a T_P-like fixpoint computation. After the evaluation of the F-Logic programme, the knowledge base consists of the original facts plus the derivations. Now the knowledge base can be queried. Hereby it is possible to either request a complete dump of the knowledge base or an answer to a regular F-Logic query. Furthermore, *SystemCommands* can be sent to the web service either by explicit function calls or by including them within an F-Logic programme.

The output format of the FloridServer is configured such that answers to queries are returned as query instantiations (?- sys.prn.style@("instance"))[2].

The FLORID server keeps all facts in memory for the duration of the session. When the session ends, the whole knowledge base is dropped. It is also possible to drop the contents of the knowledge base during the session and restart with a new programme.

Now that the principles of F-Logic reasoning with FLORID are known, the concept of hybrid reasoning in SWAN can be introduced.

[2]Normally, the answers are given in Prolog style instead.

9.4 Hybrid Reasoning in SWAN

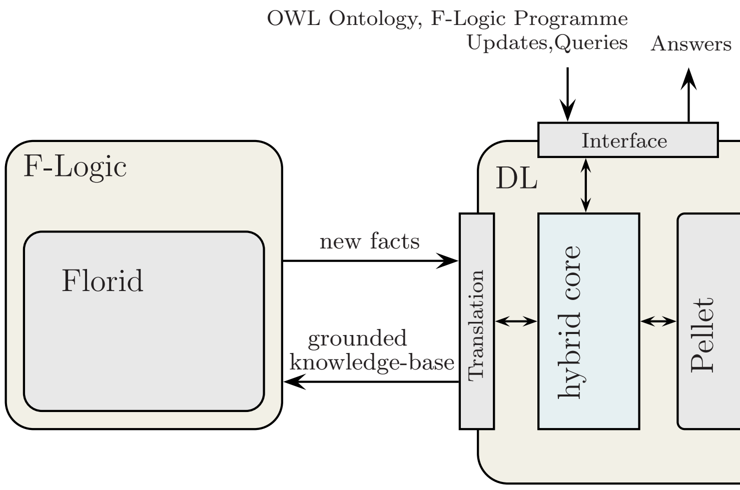


Figure 9.1: Hybrid Reasoning Architecture

In [Kat07] a prototype implementation for DL-F-Logic reasoning was realised. It uses the FLORID server and PELLET as reasoning engines. The hybrid reasoning core of that project was integrated into the domain node such that it enables the utilisation of the hybrid reasonig facilities as a native domain node action called *applnode:start-flogic-reasoning*. Furthermore, several extensions and optimisations were integrated into the hybrid reasoning engine. Central to the understanding of the hybrid reasoning process is the evaluation strategy.

9.4.1 Evaluation Strategy

Special about the hybrid reasoning combination in SWAN is the master-slave relationship between the two reasoning engines: An OWL knowledge base acts as a master, which uses an F-Logic reasoner as a slave for reasoning support.

The deductions of the OWL part depend on the definitions in the OWL ontology. The deductions of the F-Logic reasoner depend on the set of derivation rules. These rules are specified by an F-Logic programme which is given to the hybrid reasoning core in addition to the ontology. Default inheritance atoms are separated from the F-Logic programme, only the derivation rules are given to the FLORID server. The default inheritance atoms are evaluated by the hybrid reasoning core (the motivation for this strategy will be given later in Section 9.4.3).

The hybrid reasoning process consists of the following steps:

- The hybrid reasoning core translates a subset of the OWL knowledge base into an F-Logic programme. This translation will be explained in Section 9.4.2. Moreover, the F-Logic inference rules are added to this programme.
- The F-Logic programme is sent to the FLORID server, a new F-Logic knowledge base is created.

- The FLORID server evaluates the programme. Newly derived facts are added to the F-Logic knowledge base.
- The FLORID server is queried by the hybrid reasoning core.
- All newly found information is added as new base facts to the OWL knowledge base.
- The F-Logic knowledge base is dropped after it was queried by the hybrid reasoning core.

This is one iteration of the hybrid reasoning process. It will be re-iterated as long as any of the two reasoning engines is able to derive new facts. Eventually, after a fixpoint in alternating derivations is reached, the OWL knowledge base is complete with regard to the set of given F-Logic rules. In this way the OWL knowledge base evolves step-by-step by adding newly derived information from both the F-Logic reasoner and, in response to these additions, by the DL reasoner. The F-Logic knowledge base, however, is used as a disposable knowledge-base and is re-created during each iteration.

After reaching a stable state, default inheritance atoms are evaluated by the hybrid reasoning core on the OWL knowledge base. If any new facts are derived by default inheritance, another hybrid reasoning process is started.

The described process yields an evolving ontology. Each iteration adds new information, which can result in the inference of even more information in the next iteration. Note, however, that *usually* the OWL knowledge base will be completed by F-Logic derivations after the first iteration.

The definition of new rules or the addition of new facts to the OWL knowledge base may necessitate another hybrid reasoning process until again a stable state is reached. However, the OWL part of the knowledge base needs not to be continuously supplemented by F-Logic reasoning but rather on demand when the specific F-Logic capabilities are needed. This can be controlled by the use of triggers reacting on changes in the knowledge base, e.g.

```
ON insertion OF INSTANCE OF http://www.w3.org/2002/07/owl#Thing
DO BEGIN
 start-flogic-reasoning();
END;
```

The hybrid reasoning process depends on the translations of the exported subset of the OWL knowledge base into F-Logic as well as on the translations of the query results from the FLORID server into OWL expressions.

9.4.2 Translation

For the initialisation of the FLORID server, the following parts of the OWL factbase are translated to F-Logic expressions:

- class definitions,
- subclass-relationships between classes as *subclass-F-atoms*,
- instances of the classes along with their properties as *data-F-atoms* (distinguishing between functional and multi-valued properties),
- range-assertions of object properties as *F-signature atoms.*

Table 9.1 shows corresponding statements as OWL and F-Logic expressions, the DL equivalents are also given for a better understanding.

OWL	DL	F-Logic
A *owl:sameAs* B	$A \equiv B$	A = B
A *rdfs:subClassOf* B	$A \sqsubseteq B$	A :: B
x *rdf:type* A	A(x)	x : A
x p y	p(x,y)	x[p->>y]

Table 9.1: Translation between OWL and F-Logic. *A and B are classes (=concepts) in OWL, host or result objects in F-Logic, whereas p are properties (=predicates) in OWL and method names in F-Logic.* x and y are instances in OWL and again objects in F-Logic.

Many of the OWL axioms have no direct correspondence to F-Logic expressions. This is not a restriction, however, as only the base facts (the ABox) are needed for F-Logic reasoning. Therefore, only a fraction of the OWL knowledge base has to be exported.

All resources that are exported to F-Logic have to be declared to be instances of the class uri (e.g. "http://example.org#Peter" : url). Hereby, they can be used as object names in FLORID.

The resulting F-Logic programme is complemented with F-Logic rules which will be used for the derivation of new facts by the FLORID server. The F-Logic rules are separated into two sets of rules:

- Deduction rules which are added to the F-Logic programme.
- Default inheritance atoms which will later be evaluated by the hybrid reasoning core on the OWL knowledge base.

Rule evaluation in the FLORID server is applied until a local (F-Logic) fixpoint is reached. Now the hybrid reasoning core retrieves the contents of the F-Logic knowledge base by a series of queries:

Example 9.8 (Query returning the F-Logic fact base)

```
?- X : Y, X:url,
   Y:url, not Y: "http://www.w3.org/2002/07/owl#Thing" .
?- X [ Y -> Z], X:url, Y:url, Z:url.
?- X [ Y ->> Z], X:url, Y:url, Z:url.
?- X:url, X :: Y, Y:url.
?- _X : Y, Y:url .
?- _X [Y -> _Z], Y : url.
```

These queries retrieve information about class memberships, subclass relationships, functional, and multi-valued properties, method and class definitions. The queries are evaluated one by one by the FLORID server, each of the answers is translated accordingly into OWL statements. All OWL statements are added to the OWL knowledge base as a whole. For example, the first of the queries:

```
?- X : Y, X:url, Y:url .
```

retrieves tuples of individuals that are instances of a class. Again, the built-in class url is used. All these queries retrieve objects that are instances of the class uri, all other instances will be ignored. For example, intermediate resources that are created by F-Logic rules can be ignored. For the use of the class url compare Example 9.6 with the follwing programme which is extended by the use of the url class. This example also illustrates how FLORID can be used for object creation:

Example 9.9 (Creating new objects with rules)

```
 F:"http://family.org#Family", F:url,
 "http://family.org#Family":url,
 "http://family.org#isFamilyMemberOf":url,
 X["http://family.org#isFamilyMemberOf"->> F ],
 Y["http://family.org#isFamilyMemberOf"->> F ]
:- X["http://family.org#marriedTo"-> Y],
 X["http://family.org#isFamilyMemberOf"->> _F],
 Y["http://family.org#isFamilyMemberOf"->> _G],
 not _F = _G, not substr("-",_F), not substr("-",_G),
 strcat(_F,"-",TMP), strcat(TMP,_G,F) .
```

Now consider an OWL knowledge base which contains the following statements:

Example 9.10 (Family ontology extended)

```
@prefix family: <http://family.org#>.
@prefix :<http://example.org#>.
:John family:marriedTo :Susan.
:John family:isFamilyMemberOf "Mueller" .
:Susan family:isFamilyMemberOf "Luedenscheidt" .
```

After the hybrid reasoning process, the concept http://family.org#Family, its instance "Mueller-Luedenscheidt" and the "http://family.org#isFamilyMemberOf" relationships will be added to the OWL knowledge base[3]. A more sophisticated example can be found at the end of this chapter with Example 9.18.

The result of the queries to the FLORID server are translated to a list of OWL statements, e.g. the results of the query above are translated to:

```
X rdf:type Y
```

Translations are necessary in both directions, from OWL to F-Logic and the other way round. They are, however, not needed for the evaluation of default inheritance atoms because these are handled by the hybrid reasoning core.

9.4.3 Handling of Default Inheritance Atoms

The second part of the hybrid reasoning process consists of the evaluation of default inheritance atoms. Default inheritance and its application to Description Logics has been analysed in Section 2.5. Although FLORID is capable of evaluating default inheritance atoms by inheritance triggers, this implementation is not suitable here. The semantics of default inheritance states that a slot (property) is to be filled if it cannot be filled in any other way.

Also the evaluation of derivation rules may cause the addition of properties to individuals (objects). Therefore, the evaluation of default inheritance atoms

[3] Object creation is a feature which is not available in OWL. DL reasoning allows to derive new properties for existing instances, but only from an existing set of properties as defined in the TBox of the knowledge base. Moreover, it is not possible to *derive* new entities, neither in the TBox nor in the ABox.

has to be postponed until the alternating fixpoint between F-Logic and OWL is reached. At this point, there are two possibilities for the handling of the default inheritance atoms:

- Translate the OWL knowledge base to F-Logic as usual, with the only difference that the set of default inheritance atoms are now added to the resulting F-Logic programme instead of the derivation rules. If FLORID inferes any new facts, a new alternating fixpoint has to be computed.
- The default inheritance atoms are evaluated by the hybrid reasoning core in SWAN. The defaults are directly applied to the OWL knowledge base.

The disadvantage of the first method is that the whole knowledge base has to be translated and transmitted to FLORID. That is because there is no possibility to drop just parts of an F-Logic programme (e.g. the inheritance rules). This continuous re-shipping can become a severe restriction when dealing with large knowledge bases.

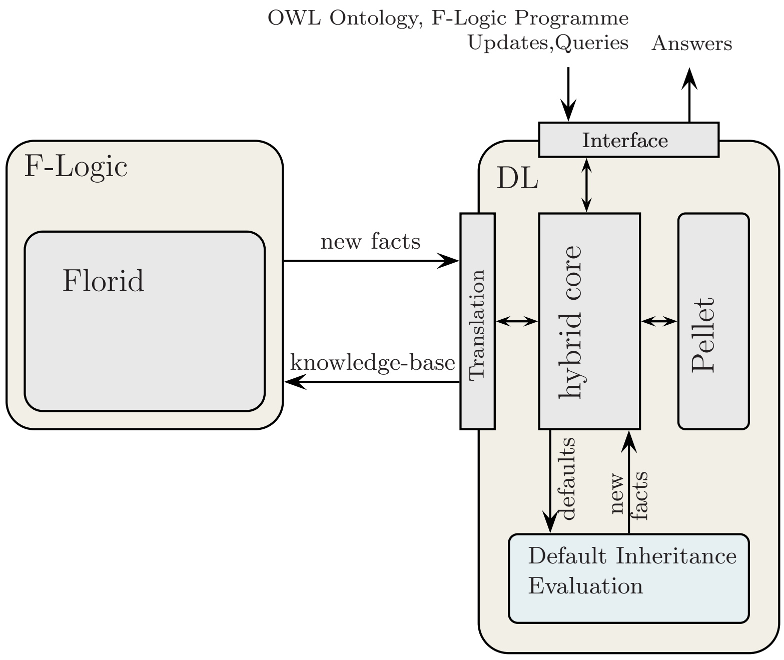


Figure 9.2: Handling of Default Inheritance Atoms

Therefore, the default inheritance atoms are evaluated by the hybrid reasoning core in this hybrid reasoning implementation (see Figure 9.2). The default inheritance atoms are still given in F-Logic syntax. Because of the strong similarities between the semantics of default inheritance for DL-knowledge bases and the semantics of default inheritance in F-Logic (see Section 2.5) these atoms can be evaluated easily on an OWL knowledge base.

Cautious default inheritance application. In Section 2.5 the notion of *cautious inflationary extensions* has been explained in detail. Default inheritance as implemented with the hybrid reasoning engine allows for the computation of exactly the kind of extensions which are described by this formalisation.

Default Reasoning Process. Default reasoning in the hybrid reasoning core consists of the following steps:

- Analyse the default inheritance atoms. They have to be evaluated in correct order so that individuals inherit a default value from their most specific super-class.
- For each default inheritance atom a set of inherited statements is calculated (by checking class memberships).
- The sets of inherited statements are added to the knowledge base one by one. Sets that cannot be added consistently to the knowledge base are reverted.
- The statements of a reverted set are added separately. Additions of statements that cannot be added consistently are reverted.

Sets of inherited statements might be reverted because they contain mutual exclusive statements. By adding the statements of reverted sets separately in the last step at least some of the statements can be added to the knowledge base given that a consistent subset of applicable statements exists.

Consider the following example where the knowledge base contains :Sarah, :Abraham, and :Isaac as instances of :Person. The F-Logic programme contains three default inheritance atoms, all of which define the inheritable property :believes_in for instances of :Person.

Example 9.11

The hybrid knowledge base consists of an OWL ontology...

```
@prefix : <http://example.org#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
:Sarah a :Woman.
:Abraham a :Man .
:Isaac a :Person ; :believes_in :Jehova .
:Baal a :OldGod .
:Woman rdfs:subClassOf :Person ; owl:disjointWith :Man .
:Man   rdfs:subClassOf :Person .
:OldGod owl:disjointWith :God .
:believes_in rdfs:domain :Man ; rdfs:range :God .
```

...and an F-Logic programme:

```
"http://example.org#Baal" : url .
"http://example.org#Jachwe" : url .
"http://example.org#Person" : url .
"http://example.org#believes_in" : url .
"http://example.org#Believer" : url .
X:"http://example.org#Believer" :-
 X["http://example.org#believes_in"->> Y ].
"http://example.org#Person"
 ["http://example.org#believes_in"*->>"http://example.org#Baal"].
"http://example.org#Person"
 ["http://example.org#believes_in"*->>
                                        "http://example.org#Jachwe"].
"http://example.org#Person"
 ["http://example.org#believes_in"*->>"http://example.org#Ra"].
```

*Every inherited statement (*x :believes_in :Baal*) violates the definition of the* believes_in *property because only instances of* :God *are in the defined range of that property.* :Baal*, however, is an instance of* :OldGod *which is disjoint with* :God*.* :Isaac *is not considered for default inheritance at all because this individual already has a* :believes_in *property.* :Sarah *cannot inherit the* :believes_in *property because every inherited statement (*:Sarah :believes_in y*) would be inconsistent with the definition of the domain of the* :believes_in *property. As a consequence, there exists no set of inherited statements that can be added as a whole to the knowledge base. However, applying the inherited statements (*:Sarah :believes_in :Jachwe*) and (*:Abraham :believes_in :Jachwe*) one by one, the former statement will be reverted for the known reasons whereas the latter statement can be added consistently.* □

Depending on the order in which inherited statements and the default inheritance atoms are evaluated there can be different extensions to an OWL knowledge base. Hence, this evaluation strategy is non-deterministic (as was already shown for defaults in general in Section 2.5). For example, :Abraham will believe in either :Jachwe or :Ra.

Despite of the non-determinism of default inheritance it can be utilised in a meaningful way. Consider the following example where the extension of the OWL knowledge base depends on the order in which the inherited statements are applied:

Example 9.12

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix :<http://example.org#>.
@prefix owl11: <http://www.w3.org/2006/12/owl11#>.
:peter a  :Human.
:paul a :Human; owl:differentFrom :peter.
:Pope  owl:equivalentClass
[ a owl:Restriction;
  owl:onProperty :hasAnulusPiscatoris;
  owl:minCardinality 1 ].
:hasAnulusPiscatoris a owl:InverseFunctionalProperty.
```

The F-Logic programme contains the following default inheritance atom:

```
"http://example.org#Human"[
    "http://example.org#hasAnulusPiscatoris" *-> true].
```

The property :hasAnulusPiscatoris is an inverse functional property, i.e. only one :Human can be subject to that property (and therefore be owner of the ring). If the extension is computed using GD^{+}_{caut} (see the definition on page 30) it is possible to apply the default to all instances separately, which leads to the situation where either :Peter or :Paul will be the ring bearer.

Translated into common-sense knowledge the default together with the ontology reads like *the person that first catches the fisherring, thrown into the air, will be the next pope*[4].

The main characteristics of the hybrid reasoning process in SWAN have been described. For better performance, this process is optimised in several ways.

[4]Of course this is not likely to be a new strategy for the Vatican, although it would be a considerably faster method compared to the current election procedure.

9.4.4 Optimisations

As the coupling of the reasoning engines involves data shipping via HTTP, the amount of transferred data has to be as small as possible. There are several optimisations for that purpose.

Optimised Fact Base Export. The first improvement concerns the export of the OWL knowledge base to the Florid server. It is necessary to recreate the F-Logic knowledge base at the beginning of each iteration of the hybrid reasoning process because Florid might have created temporary resources. In Florid, there is no possibility to remove any object from the object base, thus the temporary resources remain in the F-Logic knowledge base and potentially interfere with evaluations in subsequent iterations. In order to get rid of the temporary resources it is necessary to drop the whole F-Logic knowledge base. Consequently, the F-Logic knowledge base has to be built again in each iteration of the hybrid reasoning process.

However, this process is optimised such that not the whole OWL knowledge base has to transferred to the Florid server. Rather, only the additions in the OWL knowledge base after incorporating the F-Logic facts from the last iteration are sent.

For this, all F-Logic programmes sent to Florid consist of three parts: a prologue, the main part, and an epilogue. Consider the following programme structure:

Example 9.13 (F-Logic programme for optimised fact base shipping)

```
1  %%% ----  the programme prologue
2  ?- sys.consult@(
3            "/tmp/florid_{$sessionID}_{$floridCounter}.flp").
4  ?- sys.eval.
5  %%% ----  OWL facts, translated to F-Logic, are inserted here

7  %%% ----  the programme epilogue
8  ?- sys.eval.
9  ?- sys.prn.style@("instance").
10 ?- sys[output->sys.open@(
11            "/tmp/florid_{$sessionID}_{$floridCounter+1}.flp")].
12 ?- sys.answerChannel.setStream@(output)[].
13 ?- X : Y, X:url, Y:url .
14 ?- X [ Y -> Z], X:url, Y:url, Z:url.
15 ?- X [ Y ->> Z], X:url, Y:url, Z:url.
16 ?- X:url, X :: Y, Y:url.
17 ?- sys.prn.style@("bound").
18 ?- sys.answerChannel.setStream@(wout)[].
```

The variables sessionID *and* floridCounter *in lines 1 and 11 become substituted with the* http*-session id and the iteration counter, respectively. The first line of the programme tells* Florid *to read from the temporary file that contains a dump of the fact base stored after the last iteration. After that, the* new *facts from the DL fact base are given, already translated into F-Logic. Now the programme is evaluated and all facts are written to a new temporary file (lines 9,10,17,18 are commands to the* Florid *server in order to redirect the output stream and change the style of the output). Lines 13 to 16 correspond exactly to those queries that are used by the OWL part for the retrieval of the F-Logic facts from the* Florid *server.* □

In this respect, the hybrid reasoning process consists of the following steps:

- In the first iteration, the hybrid reasoning core sends the whole translated OWL knowledge base to FLORID.
- After FLORID evaluated the inference rules the contents of the F-Logic knowledge base are stored in a temporary file.
- The hybrid reasoning core queries the F-Logic fact base. The retrieved facts are added to the OWL knowledge base.
- If any new facts can be derived from these additions then a new iteration of the hybrid reasoning process begins. In this and all following iterations, only the new facts from the OWL knowledge base are sent to the FLORID server.
- FLORID creates a new F-Logic knowledge base from the received facts and the facts from the temporary file.

Optimised Property Export. Another optimisation concerns the individuals and their properties, which are exported from the OWL knowledge base to the FLORID server. Primarily, individuals with their class membership have to be exported. Beyond that it is sufficient to export only those statements where the property name can be matched with an attribute in the rule body of an F-Logic rule. For instance, the statement (:Peter family:hasMother :Susan) has to be exported only in case that there is any rule where the family:hasMother attribute is used in the rule body, like

```
Y["http://example.org/family#hasChild" ->> X] :-
X["http://example.org/family#hasMother" -> Y].
```

If there are no F-Logic rules that potentially target an OWL statement in this respect the statement can safely be ignored, thus reducing the costs for data transmissions between the reasoning engines. This optimisation is realised by building a list of method names which occur in the F-Logic programme. During the export of the OWL knowledge base all properties are tested whether they appear in that list. As the translation procedure has to iterate over all statements anyway, this causes only little extra costs.

Filtering statements about OWL axioms. While it is necessary to export statements containing OWL axioms to FLORID it is not desirable to re-import derived knowledge about OWL axioms. Derivations about OWL notions are trivial and can safely be ignored. Hence, statements where the subject belongs to the OWL namespace are filtered during re-import.

Despite the optimisation of the hybrid reasoning process, there are still some limitations to it. These will be discussed in the next section.

9.4.5 Limitations

As already pointed out in Section 3.7, there are limitations to the usage of hybrid reasoning.

DL expressions can be translated easily into FOL. Figure 9.4.5 shows how the semantics of some basic DL expressions is defined by a mapping to FOL.

$$\begin{array}{rcl}
\pi(C \sqsubseteq D) & = & (\forall x)(\pi_x(C) \rightarrow \pi_x(D)) \\
\pi(C \equiv D) & = & (\forall x)(\pi_x(C) \leftrightarrow \pi_x(D)) \\
\pi_x(C) & = & C(x) \\
\pi_x(\neg C) & = & \neg \pi_x(C) \\
\pi_x(C \sqcap D) & = & \pi_x(C) \wedge \pi_x(D) \\
\pi_x(C \sqcup D) & = & \pi_x(C) \vee \pi_x(D) \\
\pi_x(\forall R.C) & = & (\forall y)(R(x,y) \rightarrow \pi_y(C)) \\
\pi_x(\exists R.C) & = & (\exists y)(R(x,y) \wedge \pi_y(C)) \\
\pi_y(C) & = & C(y) \\
\pi_y(\neg C) & = & \neg \pi_y(C) \\
\pi_y(C \sqcap D) & = & \pi_y(C) \wedge \pi_y(D) \\
\pi_y(C \sqcup D) & = & \pi_y(C) \vee \pi_y(D) \\
\pi_y(\forall R.C) & = & (\forall x)(R(y,x) \rightarrow \pi_x(C)) \\
\pi_y(\exists R.C) & = & (\exists x)(R(y,x) \wedge \pi_x(C))
\end{array}$$

Figure 9.3: Mapping from DL to FOL

As such mappings exist for both F-Logic and DL, it is possible to map DL expressions to F-Logic and vice versa. However, not all DL expressions can also be expressed in F-Logic. For example, it is not possible in F-Logic to express existential quantification. Consider the following example:

Example 9.14
The TBox of a knowledge base contains an existential quantification

$$\exists \text{family:Parent.family:hasChild}$$

Moreover, the ABox contains the assertion that :Peter *is a* family:Parent *but it does not contain a* family:hasChild *relationship for* :Peter*. In DL such existential quantifications are actually* assertions*: there has to be child of* :Peter *but it might not be known yet. In a* closed world*, a quantification is a* restriction*: if* :Peter *is a family:Parent and does not have a family:hasChild relationship, the data is considered to be inconsistent.* □

In a situation when hybrid reasoning combines both paradigms of *open-* and *closed world* there is a semantic gap in the translation of the knowledge between the formalisms. As it is not possible to translate, for example, the assertion of the existence of a family:Child directly to F-Logic, this information has to be given indirectly. This can be achieved by a *temporary completion* of those individuals that are in the domain of existentially quantified properties (but have no such property yet). For instance, :Peter would have to be completed with respect to the family:hasChild relationship by a temporary resource (e.g. tmpres0815). These temporary resources are created by the hybrid reasoning core for the export to Florid. Neither will they be added to the OWL knowledge base nor be retrieved by the queries to the FLORID server[5]. Hence, the name temporary resource. With the temporarily completed fact base it is possible for the F-Logic rule engine to draw conclusions about not-existent resources, e.g. about parents, even if their children are not known yet.

Example 9.15
An F-Logic programme consists of the rule

```
X : "http://example.org/family#Father" }
:-
X["http://example.org/family#hasChild }-> _Y],
X : "http://example.org/family#Male".
```

Deducing that a resource is a family:Father *is only possible if a* family:hasChild *attribute is known. Hence,* :Peter *who is a* family:Parent *but has no* family:hasChild *relationship cannot be found to be a* family:Father. □

Amongst others, owl:allValuesFrom, owl:someValuesFrom, and owl:minCardinality can be used to express existential quantification in OWL. When the knowledge base becomes translated into an F-Logic programme, all individuals that belong to classes that are specified by such concept restrictions have to be identified. This can be achieved by the use of a simple SPARQL query:

Example 9.16 (SPARQL query for existentially quantified individuals)

```
prefix owl: <http://www.w3.org/2002/07/owl#>
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?X ?I ?P
WHERE { ?X a owl:Restriction.
        ?X owl:onProperty ?P.
        {?X owl:minCardinality []} UNION
        {?X owl:someValuesFrom []} UNION
        {?X owl:allValuesFrom []}
        ?I a ?X.
        OPTIONAL{?I ?P ?C.}
        FILTER(!bound(?C))
      }
```

All individuals that are returned by this query have to be *completed* with a temporary statement, e.g. (:Peter family:hasChild tmpres0815).

This approach, however, introduces a new problem. Consider the following (cyclic) concept definition:

Example 9.17 (Infinite relationships)

$$\text{family:Child} \equiv \exists\text{family:hasParent.family:Child}$$

If all individuals that are instances of family:Child *are completed such that they have at least one relationship with the* family:hasParent *property, there will be an infinite expansion of (newly created) individuals, because every newly created* family:Parent *is again a* family:Child. □

[5]These temporary resources are no instances of url in FLORID. Therefore they are not retrieved by the queries of the hybrid reasoning core.

Hence, reasoning about individuals in presence of cyclic TBoxes demands for blocking mechanisms that terminate the building of completion trees when cycles are detected. Blocking is implemented only to a limited degree in the hybrid reasoning engine. See Chapter 12.6 for an outlook about further work blocking mechanisms in SWAN.

The principles of hybrid reasoning in SWAN are now known. Next, an example is supplied which illustrates the interaction of F-Logic rules and OWL ontologies.

9.5 Application

In SWAN, hybrid reasoning is used only as a supplementary reasoning process that is activated *on demand* (in contrast to OWL reasoning which is applied continuously). Only when the additional deductive power of F-Logic is needed the additional inference mechanism is used.

Consider a railway company that uses a knowledge base containing a number of interconnected railway stations. Moreover, the fact base defines connections between neighbouring stations ("direct-connections" along with their "distances" and "durations"). The rules in the F-Logic programme calculate all possible "connections" between reachable rail stations.

Note that for ease of reading, the whole example is given in F-Logic. In SWAN, usually the facts are given as RDF statements and only the derivation rules are given in F-Logic.

Example 9.18 (Railway facts)

```
"Goettingen" : "station" [ "name" ->> "Goettingen" ].
"Hannover" : "station" [ "name" ->> "Hannover" ].
"Braunschweig" : "station" [ "name" ->> "Braunschweig" ].
"Hamburg_Hbf" : "station" [ "name" ->> "Hamburg_Hbf" ].
"Berlin_ZoologischerGarten":"station"
    [ "name" ->> "Berlin_ZoologischerGarten" ].
"Magdeburg" : "station" [ "name" ->> "Magdeburg" ].

"Goettingen-Hannover":"direct-connection".
"Goettingen-Hannover"["duration"->54;"distance"->100].
"Goettingen-Hannover"["from"->"Goettingen";"to"->"Hannover"].

"Goettingen-Braunschweig":"direct-connection".
"Goettingen-Braunschweig"["duration"->54;"distance"->110].
"Goettingen-Braunschweig"
    ["from"->"Goettingen";"to"->"Braunschweig"].

"Berlin_ZoologischerGarten-Hamburg_Hbf":"direct-connection".
"Berlin_ZoologischerGarten-Hamburg_Hbf"
    ["duration"->96;"distance"->175].
"Berlin_ZoologischerGarten-Hamburg_Hbf"
    ["from"->"Berlin_ZoologischerGarten";"to"->"Hamburg_Hbf"].

"Hannover-Hamburg_Hbf":"direct-connection".
"Hannover-Hamburg_Hbf"["duration"->68;"distance"->175].
"Hannover-Hamburg_Hbf"["from"->"Hannover";"to"->"Hamburg_Hbf"].

"Braunschweig-Berlin_ZoologischerGarten":"direct-connection".
"Braunschweig-Berlin_ZoologischerGarten"
```

```
    ["duration"->96;"distance"->240].
"Braunschweig-Berlin_ZoologischerGarten"
    ["from"->"Braunschweig";"to"->"Berlin_ZoologischerGarten" ].

"Braunschweig-Magdeburg":"direct-connection".
"Braunschweig-Magdeburg"["duration"->49;"distance"->85].
"Braunschweig-Magdeburg"
    ["from"->"Braunschweig";"to"->"Magdeburg"].

"Braunschweig-Hannover":"direct-connection".
"Braunschweig-Hannover"["duration"->33;"distance"->60].
"Braunschweig-Hannover"["from"->"Braunschweig";"to"->"Hannover"].

"Magdeburg-Berlin_ZoologischerGarten":"direct-connection".
"Magdeburg-Berlin_ZoologischerGarten"
    ["duration"->90;"distance"->147].
"Magdeburg-Berlin_ZoologischerGarten"
    ["from"->"Magdeburg";"to"->"Berlin_ZoologischerGarten"].
```

Example 9.19 (Computing new railway connections)

```
X:"connection" :- X:"direct-connection".
U:"connection"["from"->X ;"to"->Z ],U:url :-
C1:"connection"["from"->X ; "to"->Y ],
C2:"connection"["from"->Y ; "to"->Z ],
strcat(X,"-",TMP), strcat(TMP,Z,U) .
```

New connections can be found from the facts by computing the transitive closure using the F-Logic rules from Example 9.19. The new resource names are created by a concatenation of the names of the railway stations. Hereby, all newly created connection objects are added to the OWL knowledge base. With little extra effort the FLORID railway example can be extended such that only those connections for pairs of stations are added which have the shortest duration.

Example 9.20 (Computing fastest railway connections)

```
%% 1.  Symmetry of connections
NewU:"direct-connection"  ["distance"->D;"from"->T;"to"->F;
                            "distance"->DI;"duration"->DU]
:- U:"direct-connection"  ["distance"->D;"from"->F;"to"->T;
                            "distance"->DI;"duration"->DU],
    strcat(T,"-",PT),strcat(PT,F,NewU).

%% 2. create a new distinct term for pairs of F/T with
%%    tmpduration properties
c(F,T):connection[from->F; to->T; tmpduration->D ; path-> F],
U["path"->F]
:- U:"direct-connection"["duration"->D; "from"->F; "to"->T ].

%% 3. create new connections for terms of pairs
%%    PATH/DESTINATION (initialisation)
c(P,Z):connection[from->X; to->Z; tmpduration->D; path->P]
:- C1:connection[from->X; to->Y; tmpduration->D1; path->P1],
    C2:connection[from->Y; to->Z; tmpduration->D2; path->P2],
    not X=Z, D = D1 + D2 , strcat(P1,"-",PT),strcat(PT,P2,P),
    not _:connection[from->X; to->Z].

%% 4. add new term for existing connection X/Z if the new
%%    path P has shorter duration than existing ones
c(P,Z):connection[from->X; to->Z; tmpduration->D; path->P]
:- C1:connection [from->X; to->Y; tmpduration->D1; path->P1],
    C2:connection [from->Y; to->Z; tmpduration->D2; path->P2],
    not X=Z,
    D = D1 + D2, strcat(P1,"-",PT), strcat(PT,P2,P),
    D <= min{ S1[X,Z];
              _C:connection[from->X; to->Z; tmpduration->S1] }.

%% 5. Create a new connection selecting from all new
%%    connections that one with shortest duration
U:"connection"["from"->X; "to"->Y; "duration"->D; "path"->P ],
U:url :-
   C:connection[from->X; to->Y; tmpduration->D; path ->P],
   D = min{ DX[X,Y];
              _C:connection[from->X; to->Y; tmpduration->DX]},
   X["name"->>XN], Y["name"->>YN],
   strcat(XN,"-",TMP), strcat(TMP,YN,U).
```

These rules create temporary connection objects for all pairs (A, B) of railway stations where B is reachable from A and where the duration of the new connection is shorter than any existing one for the same pair of stations. The last rule creates "connection" objects for all pairs (A, B) choosing from the temporary connection objects the one with the shortest duration. Only these "connection" objects become instances of url, therefore they are the only resources that will be added to the OWL knowledge base.

The hybrid reasoning process, which calculates new connections along with their durations has to be performed only once when the knowledge base becomes initialised . The results from Florid are returned to the hybrid reasoning core where they are translated into RDF statements and added to the knowledge base. A new hybrid reasoning process is only necessary in case that there are changes in the knowledge base that would allow to derive new facts by the hybrid reasoning process (e.g. insertion of new stations). This behaviour can be defined and initiated by the use of triggers:

Example 9.21 (Hybrid reasoning activated by a trigger)

```
ON CREATION OF INSTANCE OF CLASS
   http://example.org/rail#RailStation
WHEN
   select ?_connected
   where {?connection ?_connected <$new.subject> .
          ?connection a <http://example.org/rail#Connection>.}
DO BEGIN
   start-flogic-reasoning();
END;
```

The trigger condition is not intended to bind any values to variables (actually, _connected is a *don't-care variable*). Rather it ensures that the trigger fires only in case of the insertion of a new station which is also part of a connection (connected either by the *from* or *to* property). The action part in the trigger definition (start-flogic-reasoning()) is a domain node action which initiates the hybrid reasoning process.

The last chapters described all features of the SWAN architecture. Knowledge base triggers, ACA rule wrapping, and hybrid reasoning have been presented. In the concluding chapter about Swan the behaviour of the domain node is given a logical characterisation.

Chapter 10

Logical Characterisation of Domain Node Behaviour

10.1 Integration into MARS

Rule formalisms play a central role in both the MARS and the SWAN architecture. Hereby, SWAN domain nodes can be integrated easily into the MARS architecture. ACA rules enable domain nodes to react on actions of the application domain (see Chapter 8) whereas knowledge base triggers allow to make changes within the domain node visible to other parts of the event-driven network (see Chapter 7). Basically, both of these formalisms are ECA rules and specify the behaviour not only of the domain node but also of global issues. In the event-driven architecture MARS where all behaviour is specified by rules, it is possible to reason about the dynamic aspects of a domain. Such reasoning can be, for example, the testing of the satisfiability of an action or proving of the validity of rules with regard to knowledge base updates. A prerequisite for reasoning tasks like this is the characterisation of the dynamic aspects of the domain node. This characterisation is given in the following for SWAN, covering ACA rules and triggers.

10.1.1 Characterisation of Events

The communication between the service components of the MARS framework and the domain nodes is mainly realised by events. Actions are discriminated from events with regard to operational aspects: actions actually are the activities that happen whereas events inform the rest of the application domain about these changes (see Section 4.3.4). In this respect it makes sense to speak of a "signaling of events".

On a conceptual level, however, the notion of an event is different. Basically, an event does not exist and is never communicated. An event occurs instantaneously and has no duration. It simply happens and becomes *visible*. The visibility depends on the agent. Every agent has different knowledge and therefore a different point of view. Events are not necessarily visible to all agents but can be distributed globally.

Definition 10.1 (Visibility)
Visibility of events is defined by a predicate $\mathsf{visible}(\mathcal{E},\mathcal{N})$ where $\mathcal{E}$ is a set of events, which is visible to a set of agents $\mathcal{N}$. $\square$

The set of agents defines the range of visibility. Local actions (e.g. updates to the knowledge base) are always visible locally. From the point of view of the agent (the domain node) they are considered at the same time as events, e.g. when a statement (x, y, z) becomes inserted into the knowledge base then the event $\mathsf{insertion}(x, y, z)$ becomes visible to the agent itself. The local node is the only agent that is able to see such an event unless it is explicitly made visible to the outside. Knowledge base triggers are ECA rules that are activated upon the detection of events, but as events directly correspond to actions (knowledge base updates) they actually fire upon actions. As a consequence it is possible to use triggers in order to make events visible (as realised by the domain-node action raise-event).

The operation of raising an event makes the event visible to other nodes, e.g. a domain broker, formally specified by

$$\frac{\mathrm{visible}(\{e\}, \{localhost\}), e \in \mathrm{DomainEvents}}{\mathrm{visible}(\{e\}, \{localhost, DomainBroker\})}$$

DomainEvents are events that are specified as the consequence of a trigger. If an event is visible locally and is a *DomainEvent* it becomes visible to the domain, too (here to the domain broker as an intermediate distributor).

Events and the Aspect of Time
Events have no duration, which makes it more difficult to describe *ongoing* changes. Depending on the domain of interest it might be necessary to continously talk about changes (e.g. a metereological station) or to have events which indicate only starting and ending points of periods of interest. In the latter case intermediate events might be necessary if the changes do not evolve monotonically.

Example 10.1
Again, a flight reservation system is considered. The availability of new flights is announced with new-flight *events. The end of the reservation period is naturally limited by either the departure of the plain or when it is fully booked. The latter needs to be made visible with a* fully-booked *event whereas the first can be derived from the flight schedule. Intermediate events can be used to indicate the booking status (e.g.* half-booked*).* $\square$

Events are well suited for the implementation of the notion of time in the MARS framework. Time passes synchronously in discrete steps at all nodes which can be realised by time-events that are globally visible.

$$\frac{\mathrm{visible}(\{\mathsf{time\text{-}passes\text{-}1sec}\}, \{timenode\})}{\mathrm{visible}(\{\mathsf{time\text{-}passes\text{-}1sec}\}, \{global\})}$$

There has to be a coordinating node (named *timenode* in the definition above) which originates such events. Like any other event, global events are distributed by the domain broker. The term *global* does not denote a specific node but the set of all known nodes.

10.1.2 Events and Rules

The relationship of events and actions can be expressed by ECA rules. Similarly, ACE rules define, from the point of view of the agent, how actions (that the agent is able to fulfill) correspond to one or more events.

Example 10.2
The booking of a seat for flight LH458 at the airline company is a local action. The action is, locally, also seen as an insertion event. This event, in turn, can be made visible by a trigger:

```
ON INSERTION OF
  http://www.semwebtech.org/domains/2006/travel#bookedFor
DO BEGIN
  raise event(
  <travel:flightBooked xmlns:travel=
        "http://www.semwebtech.org/domains/2006/travel#">
      <travel:flightNo>$flightNo</travel:flightNo >
      <travel:date>$date</travel:date>
      <travel:passenger>$pas</travel:passenger>
    </travel:flightBooked>
  );
END;
```

The following trigger raises, upon the same flight booking, an event in case that half of the seats of the airplane are booked:

```
ON INSERTION OF
  http://www.semwebtech.org/domains/2006/travel#bookedFor
WHEN SELECT ?flight WHERE {
  <$old.subject>
      <http://www.semwebtech.org/domains/2006/travel#hasBookings>
      ?bookings .
  <$old.subject>
      <http://www.semwebtech.org/domains/2006/travel#hasFlightNo>
      ?flight.
  <$old.subject>
      <http://www.semwebtech.org/domains/2006/travel#hasFlightNo> ?con.
  ?con <http://www.semwebtech.org/domains/2006/travel#withType>
      ?type .
  ?type
      <http://www.semwebtech.org/domains/2006/travel/iata/meta#capacity>
      ?cap .
  FILTER(?bookings = ?cap/2) }
DO BEGIN
  raise event(
    <travel:halfBooked
        xmlns:travel="http://www.semwebtech.org/domains/2006/travel#">
      <travel:flightNo>$old.subject</travel:flightNo>
    </travel:halfBooked>);
END;
```

□

In fact, local ECA rules can be used as if they were ACE rules for the raising of events to the application domain.

Whereas ECA rules express the relationships between events and actions, ECE rules allow for the definition of the relationship between events (see also Section 5.2.2). In this respect ECE rules can also be regarded as rewriting rules for events. Mostly, the correspondence of events expresses different points of view of agents. For example, the flight-booking event with destination New York can also be rewritten as a flight-booking-to-USA event.

More importantly, with regard to the domain application nodes, are ACA rules which have been presented in Section 8. The following section gives a logical characterisation for ACA rules.

10.2 Logical Characterisation of ACA Rules

ACA rules map a high-level action to a (sequence of) knowledge base update(s). High-level means that the action itself does not specify how to *execute* an action but what the action is about. The ACA rule closes the gap between abstract action specifications and concrete knowledge base updates.

The action is usually received by a domain node in form of an XML fragment, but could as well be, with regard to this logical characterisation, an RDF graph fragment. The parameters of the action are bound to variables and passed on to the corresponding (as specified by the ACA rule) knowledge base updates. Following the ECA paradigm, an ACA rule can be described by ON action a IF condition c DO knowledge base update u.

10.2.1 Axiomatising Knowledge Base Updates

The general characterisation of an ACA rule is an axiomatisation using a knowledge base $\mathcal{K}$ (see again Definition 2.4) and a set of knowledge base update operations $\mathcal{A}$ e.g. assert, insert, retract, delete, delete-resource, update-subject, update-property, update-object[1]:

$$\texttt{ON}\ a\ \texttt{IF}\ c\ \texttt{DO}\ u : \frac{u(r_1,\ldots,r_n) \in \mathcal{A}}{\mathcal{K} \models c \rightarrow (\text{assertions about } \mathcal{K}', \mathsf{visible}(\{u\},\{\text{local}\}))}$$

The effects of an update operation on a knowledge base $\mathcal{K}$ are described by the conclusions that can be drawn about $\mathcal{K}'$ (the knowledge base after the update operation) with regard to a set of resources $r_1,\ldots,r_n$. These conclusions include entailment ($\mathcal{K}' \models s$, $\mathcal{K}' \not\models s$) and containment ($s \in \mathcal{K}'$, $s \notin \mathcal{K}'$) of statements s[2].

In addition to the assertions given with regard to the state of the knowledge base $\mathcal{K}$, every update is also guaranteed to be a visible event locally.

The resources $r_1,\ldots,r_n$ are given as parameters of the action (that has to be mapped to update operations by the ACA rule) or they are defined in the ACA rule itself.

[1] Maintenance operations rename and rename-property-of-class are not characterised here because they are not meant to be called by external actions.

[2] Recall that a statement consists of three resources (subject, predicate, and object) and does not need to exist physically in the knowledge base ($(x,y,z) \notin \mathcal{K}$) but may hold in $\mathcal{K}$ through entailment ($\mathcal{K} \models (x,y,z)$), see again Definition 6.2 in Section 6.4.2.

The knowledge base $\mathcal{K}'$ results from $\mathcal{K}$ by minimal changes, which are characterised for all possible $u \in \mathcal{A}$ as follows:

Definition 10.2 (Axiomatisation of knowledge base update operations)

$$\textsf{assert} \Rightarrow \frac{\textsf{assert}(s,p,o)}{\mathcal{K}' \models (s,p,o)} \qquad \textsf{retract} \Rightarrow \frac{\textsf{retract}(s,p,o)}{\mathcal{K}' \not\models (s,p,o)}$$

$$\textsf{insert} \Rightarrow \frac{\textsf{insert}(s,p,o)}{(s,p,o) \in \mathcal{K}'} \qquad \textsf{delete} \Rightarrow \frac{\textsf{delete}(s,p,o)}{(s,p,o) \notin \mathcal{K}'}$$

$$\textsf{update-subject} \Rightarrow \frac{\textsf{update-subject}(s,p,o,ns), (s,p,o) \in \mathcal{K}}{(ns,p,o) \in \mathcal{K}', (s,p,o) \notin \mathcal{K}'}$$

$$\textsf{update-property} \Rightarrow \frac{\textsf{update-property}(s,p,o,np), (s,p,o)}{(s,np,o) \in \mathcal{K}', (s,p,o) \notin \mathcal{K}'}$$

$$\textsf{update-object} \Rightarrow \frac{\textsf{update-object}(s,p,o,no), (s,p,o)}{(s,p,no) \in \mathcal{K}', (s,p,o) \notin \mathcal{K}'}$$

$$\textsf{delete-resource} \Rightarrow \frac{\textsf{delete}(r)}{\mathcal{K}'|_r = \emptyset}$$

□

After a $\textsf{delete-resource}(r)$ operation the assertion $\mathcal{K}'|_r = \emptyset$ holds where $\mathcal{K}|_r$ denotes a restriction to a knowledge base $\mathcal{K}$ such that a resource r appears in all statements in $\mathcal{K}$.

Parallel Execution of Updates.

From a set of update operations $U = \{u_1, \ldots, u_n\}$ on a knowledge base $\mathcal{K}$ there follows a set of assertions about $\mathcal{K}'$:

$$\frac{u_1}{a_1(\mathcal{K}')}, \ldots, \frac{u_n}{a_n(\mathcal{K}')}$$

where $a_i(\mathcal{K})$ is an assertion about the knowledge base $\mathcal{K}$ (see Definition 10.2). If $u_1, \ldots, u_n$ can be executed consistently in parallel, resulting in $\mathcal{K}'$, obviously $\mathcal{K}'$ satisfies all $a_1, \ldots, a_n$. A set U of updates is consistent if the consequences are not contradictory. Therefore, if e.g. U contains $\textsf{delete-resource}(r)$ there must not be an $\textsf{insert}(r,s,o)$ operation in U at the same time whereas the operation $\textsf{retract}(r,p,o)$ could be part of U without contradicting the other operation[3]. Similarly, a $\textsf{delete}(s,p,o)$ cannot be consistently executed together with an $\textsf{update-subject}(s,p,o,ns)$ operation.

10.2.2 Reasoning About ACA Rules

Abstract actions of the application domain and update operations of the knowledge bases are correlated by the ACA rules at the domain nodes. These rules

[3]Recall that a retract operation ensures that a statement will not hold afterwards. Nothing has to be deleted if the statement did not hold.

combine actions and updates with an additional condition. If more than one ACA rule is defined to react upon an action, different update operations might be executed in the cause of a received abstract action specification, depending on the conditions:

Example 10.3
ON flight_booking IF

1) $\underbrace{(\neg \underbrace{\textsf{half-booked}}_{:=c_h} \text{ and } \neg \underbrace{\textsf{fully-booked}}_{:=c_f})}_{:=c_1}$ DO "sell ticket for price a"

2) $\underbrace{(\underbrace{\textsf{half-booked}}_{:=c_h} \text{ and } \neg \underbrace{\textsf{fully-booked}}_{:=c_f})}_{:=c_2}$ DO "sell ticket for price a*1.2"

3) $\underbrace{\textsf{fully-booked}}_{:=c_f=c_3}$ DO "notify failure-of-bookin"

For the sake of simplicity, the consequences of these rules are only given by a description of what should be done. The condition half-booked *is denoted by* c_h*, and* fully-booked *by* c_f*, furthermore the condition part of rules 1 to 3 are denoted by* c_1*,* c_2 *and* c_3*. Hereby, the conditions of all ACA rules that react upon the abstract action* flight_booking *can be given as*

$$c_1 = (\neg c_h \wedge \neg c_f), c_2 = (c_h \wedge \neg c_f), c_3 = c_f$$

At least one of the rule conditions will hold:

$$c_1 \vee c_2 \vee c_3 \leftrightarrow (\neg c_h \wedge \neg c_f) \vee (c_h \wedge \neg c_f) \vee c_f \leftrightarrow \textsf{true}$$

The following truth table proves, logically, that for every possible combination of conditions c_h *and* c_f *at least one rule condition is true.*

c_h	c_f	c_1	c_2	c_3	$c_1 \vee c_2 \vee c_3$
0	0	1	0	0	1
1	0	0	1	0	1
0	1	0	0	1	1
1	1	0	0	1	1

Note that the conjunction $\neg c_h \wedge c_f$ *will never occur because* c_h *follows from* c_f*. Nevertheless,* $c_3 \leftrightarrow \neg c_h \wedge c_f$*.*

Moreover, for every combination of c_h *and* c_f exactly one *of the rule conditions will hold:*

$$(c_1 \wedge \neg c_2 \wedge \neg c_3) \vee (\neg c_1 \wedge c_2 \wedge \neg c_3) \vee (\neg c_1 \wedge \neg c_2 \wedge c_3) \leftrightarrow \textit{true}$$

Hence, upon a flight_booking *action, exactly one of the three ACA rules will be executed.* □

If two ACA rules that have the same abstract action definition a_1 are given

$$\frac{a_1}{c_1 \rightarrow u_1}, \frac{a_1}{c_2 \rightarrow u_2}$$

and it can be derived that either c_1 or c_2 will hold, it can be derived, that either u_1 or u_2 follow from a_1:

$$\frac{\text{true}}{c_1 \vee c_2} \rightarrow \frac{a_1}{u_1 \vee u_2}$$

Discussion

The characterisation of updates from Definition 10.2 shows what assertions can be made about the knowledge base $\mathcal{K}'$ after the execution of an update with regard to the set of resources $r_1, \ldots, r_n$. Effects on other resources in $\mathcal{K}$ are not considered here, although the state of the knowledge base is not only affected by the update but also by further (intensional) updates by the reasoning engine. Whereas any *calculus* for dynamic worlds has to deal with the frame problem (that is how to specify what resources do *not* change, see e.g. [GL93]), the characterisation of ACA rules can be restricted to the direct effects of updates.

For proving properties of knowledge base updates, guarantees are of special interest. Such guarantees about effects of (combinations of) actions can be derived from pre- and postconditions as expressed in this characterisation. In a similar fashion such guarantess can be given in temporal logic with a formula like

$$\mathsf{insert}(s,p,o) \rightarrow \circ(s,p,o) \wedge (s,p,o)\ \mathcal{W}\ u$$

where $\mathcal{W}$ is the temporal logic *unless* operator (see e.g. [MA92]) and u is the conjunction of all update operations which change the state of $\mathcal{K}$ with regard to (s,p,o) (e.g. delete(s,p,o) or update-subject(s,p,o,ns)). The above formula expresses a frame axiom that defines that this particular resource does not change until u happens. This, however, is only possible as long as no implicit knowledge is used in the formula. Consider a temporal logic formula expressing an assert operation:

$$\mathsf{assert}(s,p,o) \rightarrow \circ(s,p,o)$$

Here it is not possible to give further guarantees about (s,p,o) after an assert because it is only known that $\mathcal{K} \models (s,p,o)$, whereas $(s,p,o) \in \mathcal{K}$ cannot be guaranteed. Given that there is no formalisation of the reasoning mechanism it is not possible to specify, which operations affect the asserted statement. The guarantees towards the implicit update operations assert and retract are much weaker compared to the insert and delete operations.

Nevertheless, all of the assertions can be used for the verification of updates (and hereby also of ACA rules) and reasoning about the effects of actions.

Similar to the characterisation of ACA rules, also trigger evaluation is logically characterised.

10.3 Logical Characterisation of Knowledge Base Triggers

Triggers can be described (like ACA rules) using the ECA formalism: `ON` event e `IF` condition c `DO` triggeraction a. The action part of a trigger is either a knowledge base update (see again Section 10.2.1) or the raising of events (which means sending the event to a domain-broker which handles the further distribution of the event). For the formal specification of the trigger evaluation process see again Section 7.3.

The general characterisation of a trigger definition (in the abstract form as an ECA rule) can be given as follows:

$$\texttt{ON}\ e\ \texttt{IF}\ t\ \texttt{DO}\ a : \frac{e}{(\mathcal{K} \models t) \rightarrow (\text{assertions about } \mathcal{K}', \mathsf{visible}(\{a\}, \{\text{local}\}))}$$

where t is a condition that is evaluated on the knowledge base and visible expresses that the action a (as executed by the trigger) is visible locally and therefore available for other triggers to react upon. e is the occurence of an update operation expressed by an event definition whereas a is an update operation which is executed in reaction to e. Both e and a are defined in $\mathcal{A}$ (the set of possible update operations in $\mathcal{K}$). Events, however, are not directly given in form of update operations but as update descriptions. Essentially, these event definitions are about insert, delete or modify operations (see Section 7.2.4) plus conditional expressions with regard to the concepts affected by the event. For example, `ON INSERTION OF` *property* p `OF INSTANCE OF` *class* c can be expressed in form of an ECA rule as:

$$\texttt{ON}\ \mathsf{insert}(x, y, z)\ \texttt{IF}\ (x : c \wedge y = p)\ \texttt{DO}\ \ldots$$

and similarly the trigger definition `ON NEW CLASS` c can be expressed as

$$\texttt{ON}\ \mathsf{insert}(c, y, z)\ \texttt{IF}\ (y = \mathsf{rdf{:}type} \wedge z = \mathsf{owl{:}Class})\ \texttt{DO}\ \ldots$$

In the same manner all trigger definitions can be expressed as ECA rules such that the event specifications of the triggers are expressed only in terms of update operations in $\mathcal{A}$. Hereby the set of update operations which cause a trigger to fire can explicitly be given.

A single update u can cause a set of updates $u, u_1, \ldots, u_n$, which are considered to be executed in parallel. Like in Section 10.2.1 such a set of updates can be expressed by a set of guarantees:

$$\frac{u}{a(\mathcal{K}')}, \frac{u_1}{a_1(\mathcal{K}')}, \ldots, \frac{u_n}{a_n(\mathcal{K}')}$$

There are some differences between ACA rules and trigger rules, though:

a) The action part of a trigger can also be the raising of an event which is not specified as an update operation in $\mathcal{A}$. But as the raising of an event has no side effects, the raise-event operation can be specified simply as

$$\mathsf{raise\text{-}event} \Rightarrow \frac{\mathsf{raise\text{-}event}(e)}{\mathcal{K}, \mathsf{visible}(\{e\}, \{\mathcal{N}\})}$$

where $\mathcal{N}$ is the set of agents to which the event e is made visible which normally includes the domain-broker and all other nodes that will receive e from the domain-broker. Although the raising of events has no side effects on the knowledge base it might well have consequences on the domain level.

b) Triggers may cause further updates (which may in turn cause further triggers to fire). Trigger evaluation therefore is a fixpoint computation process. Nevertheless, the update process reacting on a single initial action can be considered being atomic, or, in other words, a transaction resulting in a set of changes Δ (for the definition of Δ see again Section 7.3).

c) Trigger evaluation has two stages: before and after the actual update. This discrimination is only relevant in an operational sense. The operator TU^{ω} (see again Definition 7.3) computes from the initial update u (causing the event e) by an alternating application of pre- and post-reasoning triggers the final model M (or $\mathcal{K}^{'}$ respectively).

In order to give assertions about $\mathcal{K}^{'}$ after an update u it is not possible to simply use the set of changes Δ. Δ consists, on the one hand, of *explicit* knowledge which is the initial update u plus subsequent updates $u_1, \dots, u_n$ caused by further trigger activation. On the other hand, Δ may contain *implicit* knowledge added by the reasoning engine. The reasoning engine, however, is used as a black box. Without a formal specification of the reasoning process, guarantees can be given only for the *explicit* changes in the knowledge base, which makes Δ unsuitable for this characterisation.

Hence, the guarantees towards $\mathcal{K}$ after an update u in presence of triggers is the set of assertions about each update in $\Delta_{explicit} = \{u, u_1, \dots, u_n\}$. Consider

$$\frac{u}{a(\mathcal{K}')}, \frac{u_1}{a_1(\mathcal{K}')}, \dots, \frac{u_n}{a_n(\mathcal{K}')}$$

where the assertions $a, a_1, \dots, a_n$ (which correspond to the updates $u, u_1, \dots, u_n$) are described by the axiomatisation given in Definition 10.2.

The set $\{u, u_1, \dots, u_n\} = \Delta_{explicit}$ is specified by the operator $TR_{T,G}$ (see Definition 7.2), where T is the set of all triggers (both pre- and postreasoning triggers) defined on the knowledge base $\mathcal{K}$, and G is the underlying graph structure (the explicit knowledge in $\mathcal{K}$).

For that, the assertions about the consequences of an initial update u are the set of assertions about each single update in $\Delta_{explicit}$:

$$\frac{u}{a(\mathcal{K}'), a_1(\mathcal{K}'), \dots, a_n(\mathcal{K}'), \mathsf{visible}(\mathcal{E}, \{\mathcal{N}\})}$$

where $\mathcal{E}$ is the set of all events accumulated during trigger evaluation (see again Sections 7.3 and 10.1.1 for details on event raising) and $\mathcal{N}$ is the set of agents to whom $\mathcal{E}$ will be visible.

Example 10.4
Given are the trigger definitions t_1 =(`ON` u_1 `IF` c_1 `DO` u_2) and t_2 = (`ON` u_1 `IF` $\neg c_1$ `DO` u_3) with u_1 =$\mathsf{insert}(x, y, z)$, $u_2 = \mathsf{insert}(x', y', z')$ and $u_2 = \mathsf{insert}(x'', y'', z'')$.

For an initial update u_1, the following guarantees can be made with regard to $\mathcal{K}'$ after the knowledge base update is completed:

$$\frac{u_1}{((x,y,z) \in \mathcal{K}' \wedge ((x',y',z') \in \mathcal{K}' \vee (x'',y'',z'') \in \mathcal{K}'))} \qquad \square$$

Trigger evaluation as transactions. As already mentioned, all updates (here $\Delta_{explicit}$) are considered to be executed in parallel, starting with the initial update u plus all further updates $u_1, \ldots, u_n$ due to trigger evaluation. Therefore, the situation for $\Delta_{explicit}$ is the same as in Section 10.2.1 for U: the set of updates is consistent, if the consequences are not contradictory. In this respect $\Delta_{explicit}$ is treated as a uniform knowledge base update which can be regarded as a *transaction*. Such a transaction can be executed successfully only if all atomic updates can be applied consistently. Otherwise a roll-back has to be done, reverting all changes.

The conclusions, which follow from the characterisation of ACA rules and triggers, are stated in the following.

10.4 Conclusion

The differences between implicit and explicit updates have already been mentioned in Section 10.2. As the consequences of an insert operation are different from those of an assert operation the guarantees have to be different likewise. *Inserting* knowledge is an explicit operation which gives the guarantee that the knowledge base indeed contains the statement afterwards.

Consider the following insert operation:

```
insert( http://example.org#Katrin,
        http://family.org#marriedTo,
        http://example.org#John).
```

The knowledge base now contains the resources :Katrin and :John plus the predicate family:marriedTo which connects both other resources. Therefore it is possible to guarantee at the same time that, after the insert operation, the *deletion* of the same statement would succeed.

The situation is different if an assert operation with the same statement was executed instead of the insert operation. Although both operations ensure that the knowledge base will entail the given statement afterwards only the insert ensures that the statement is actually contained. Consider a situation where the knowledge base already contains the same statement through entailment:

```
:John family:marriedTo :Katrin.
family:marriedTo a owl:SymmetricProperty.
```

Because family:marriedTo is a symmetric property it can be derived that also :Katrin is married to :John. The insert operation from above would nevertheless add the (now redundant) statement and so it could also be deleted again.

In contrast, the assert operation would cause no changes in the knowledge base (though committing successfully) as the intention of the update is already fulfilled. A consecutive delete operation on the same statement, however, would fail because a non-existent statement cannot be deleted.

From this considerations follows that the consequences of an assert operation depend heavily on the existing data (both assertional and terminological) and the inference mechanism. As the reasoning process itself is treated as a black box it cannot be told alone from the success of an assert operation what actually happened within the knowledge base. Hence, the only guarantee towards the assert operation can be given with regard to the *entailment* of the statement by the model.

A similar situation can be found with the relationship of the delete and the retract operations. Both operations ensure that the statement, as specified with the operation, cannot be entailed afterwards. But it cannot be told whether the retract operation performed any changes to the knowledge base at all. Other than with insert and assert, here is no difference with regard to future operations because both delete and retract lead to a situation where the statement is not contained in the knowledge base.

The logical characterisations that have been presented, complete the main part of this work. In the next part, the results are discussed.

Part III

Results

Chapter 11

Applicability

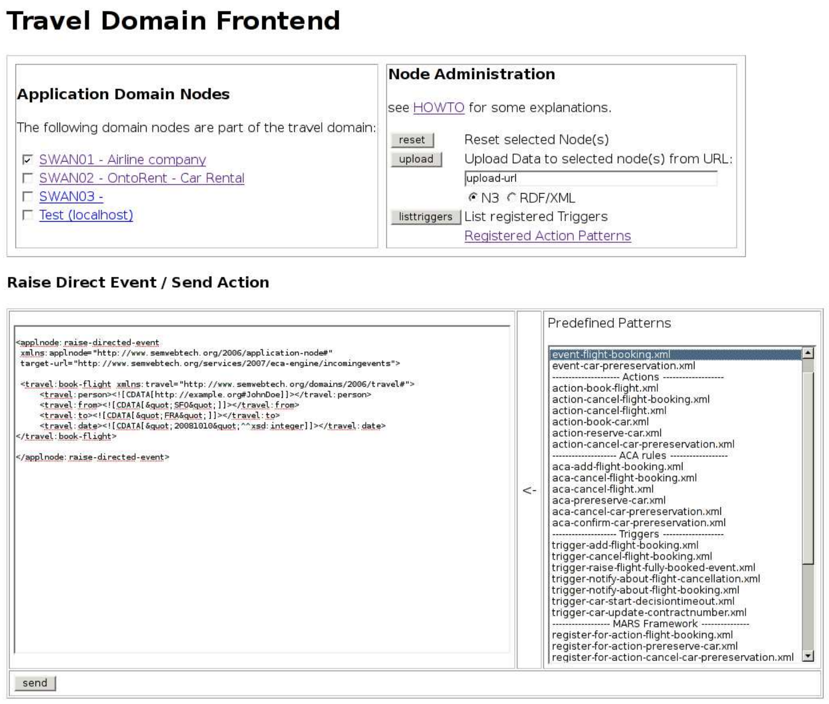
This chapter gives a motivating example scenario by which the capabilities of the SWAN architecture are demonstrated. The scenario consists of an application domain with two domain application nodes. These domain nodes use the components of an existing MARS implementation for ECA rule evaluation. The description of the scenario involves

- concepts of the application domain defined in the domain ontology,
- ECA rule definitions for event processing,
- ACA rules for a mapping of domain actions to knowledge base updates,
- local ECE rules in form of knowledge base triggers for the raising of events,
- local ECA rules, again as knowledge base triggers, for the completion of knowledge base updates,
- F-Logic programmes for hybrid reasoning,
- event definitions which initiate activities in the domain (simulating user interaction).

The chapter comprises a short description of the technical details, the rule specifications, and some conclusions.

11.1 Technical Details

Two domain nodes are integrated into a MARS infrastructure using virtual machines (VM). Each node runs in its own VM with the same technical setup. By the use of a VM for hosting a domain application node the scenario can be extended easily by an arbitrary number of additional nodes that are completely independent. The nodes differ only in the way that they become initialised by a startup script. This script loads all ontological data (the contents of the knowledge base), triggers, and ACA rules. Furthermore, the application domain node registers at the domain broker in order to receive actions. ECA rule definitions are sent to the ECA engine. Each node can be administrated seperately by its own frontend which is accessible via a web browser. Additionally, there is a domain frontend, which allows for the sending of commands to one or more nodes plus the sending of pre-defined domain actions (see the screenshot in Figure 11.1).


Travel Domain Frontend

Application Domain Nodes

The following domain nodes are part of the travel domain:

- [✓] SWAN01 - Airline company
- [ ] SWAN02 - OntoRent - Car Rental
- [ ] SWAN03 -
- [ ] Test (localhost)

Node Administration

see HOWTO for some explanations.

reset	Reset selected Node(s)
upload	Upload Data to selected node(s) from URL:
	upload-url
	[✓ N3] RDF/XML
listtriggers	List registered Triggers
	Registered Action Patterns

Raise Direct Event / Send Action

```
<applnode:raise-directed-event
 xmlns:applnode="http://www.semwebtech.org/2006/application-node#"
 target-url="http://www.semwebtech.org/services/2007/eca-engine/incomingevents">

 <travel:book-flight xmlns:travel="http://www.semwebtech.org/domains/2006/travel#">
     <travel:person><![CDATA[http://example.org#JohnDoe]]></travel:person>
     <travel:from><![CDATA[&quot;SFO&quot;]]></travel:from>
     <travel:to><![CDATA[&quot;FRA&quot;]]></travel:to>
     <travel:date><![CDATA[&quot;20081010&quot;^^xsd:integer]]></travel:date>
</travel:book-flight>

</applnode:raise-directed-event>
```

Predefined Patterns

```
event-flight-booking.xml
event-car-prereservation.xml
-------------------- Actions ------------------
action-book-flight.xml
action-cancel-flight-booking.xml
action-cancel-flight.xml
action-book-car.xml
action-reserve-car.xml
action-cancel-car-prereservation.xml
------------------- ACA rules ----------------
aca-add-flight-booking.xml
aca-cancel-flight-booking.xml
aca-cancel-flight.xml
aca-prereserve-car.xml
aca-cancel-car-prereservation.xml
aca-confirm-car-prereservation.xml
------------------- Triggers ------------------
trigger-add-flight-booking.xml
trigger-cancel-flight-booking.xml
trigger-raise-flight-fully-booked-event.xml
trigger-notify-about-flight-cancellation.xml
trigger-notify-about-flight-booking.xml
trigger-car-start-decisiontimeout.xml
trigger-car-update-contractnumber.xml
----------------- MARS Framework --------------
register-for-action-flight-booking.xml
register-for-action-prereserve-car.xml
register-for-action-cancel-car-prereservation.xml
```

Figure 11.1: Domain Frontend (Screenshot of the Browser Interface)

The application domain nodes make use of an implementation of the Mars framework, which resides on a separate machine (note that the place where the components are located is not important, all services can be distributed amongst many different servers). The Mars framework can be administrated via a browser interface that offers several options for managing the Mars infrastructure: rule (de-)registration, event raising (also by choosing from a set of pre-defined events), and framework logging facilities.

For the demonstration of the interaction of the components, an example scenario about travel planning and booking is presented.

11.2 Scenario Description

There are two different nodes that participate in the application domain: one is an airline company (*Onto-Flight*) which serves flights between airports worldwide. The flight plan of *Onto-Flight* is shown schematically in Figure 11.2. The other domain node represents a car rental company (*Onto-Rent*) which offers car rental services at certain cities (airports). Booking a flight from city A to city B consists of a booking for each flight that is part of the fastest connection. For instance, the fastest flight connection from San Francisco (SFO) to Frankfurt (FRA) consists of flights 9159 and 446.

Additionally, a car will be reserved at the destination city. That car will be the least expensive one chosen from a set of available cars all being in the same class, which the customer usually drives.

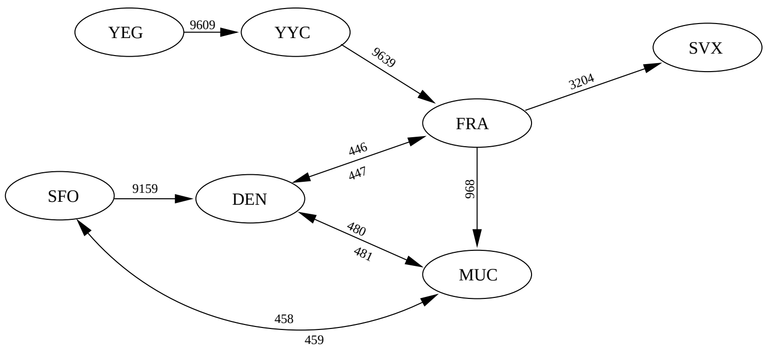


Figure 11.2: Flight plan of Onto-Flight. Airport names are given by the three-letter IATA codes, flight connections are identified by a flight number.

The travel domain nodes make use of the same domain ontology which has already been presented in Example 5.1. That ontology is extended by the domain nodes by their own concepts. For instance, the airline company adds the following concept definitions:

Example 11.1 (Airline Company Ontology, Concepts)

```
@prefix mars: <http://www.semwebtech.org/mars/2006/mars#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix : <http://www.semwebtech.org/nodes/2007/onto-flight>.
@prefix travel: <http://www.semwebtech.org/domains/2006/travel#>.
@prefix iata:
      <http://www.semwebtech.org/domains/2006/travel/iata/meta#>.

:      a         mars:ApplicationNode ;
       a         mars:DomainService ;
       mars:name "MARS Worldwide Airline Services" ;
       mars:supports travel:from ,
                     travel:to ,
                     travel:duration ,
                     travel:arrival ,
                     travel:departure ,
                     travel:ConnectedFlight ,
                     travel:served_flight ,
                     travel:flight-booking ,
                     travel:cancel-flight-booking ,
                     travel:cancel-flight ,
                     iata:Airport ;
       mars:uses-domain travel: , iata: .

 travel:flight-booking
       a         owl:Class ;
       rdfs:subClassOf travel:Action ;
       mars:belongs-to-domain      travel: .

 travel:cancel-flight-booking
```

```
       a          owl:Class  ;
       rdfs:subClassOf  travel:Action  ;
       mars:belongs-to-domain      travel:  .

travel:cancel-flight
       a          owl:Class  ;
       rdfs:subClassOf  travel:Action  ;
       mars:belongs-to-domain      travel:  .
```

The knowledge base of the airline company contains facts about airports, airplanes, and flight connections. The flight plan, which is used for the scenario consists of seven airports and eleven connecting flights. Although there is a flight plan available with many more airports and flight connections the amount of data is reduced for this scenario to a small fraction in order to achieve faster computations. As the intention is to demonstrate the behaviour of the application domain node this simplification can be applied without loss of generality. The example scenario for the application domain nodes could be used with much larger datasets, but then the limitations of DL reasoning will have severe effects on the performance of the application (see the discussion in Section 12.1).

In the next example a small excerpt is presented which contains all resources that are involved in a flight booking between the airports "Denver" and "Frankfurt".

Example 11.2 (Airline Company Ontology, Facts)

```
@prefix iata: <http://www.semwebtech.org/domains/\
                  2006/travel/iata/meta#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix owl:  <http://www.w3.org/2002/07/owl#> .
@prefix travel: <http://www.semwebtech.org/\
                    domains/2006/travel#> .

<http://www.semwebtech.org/domains/2006/\
 travel/iata/crafts/Airbus/A340-300>
  a          iata:Craft  ;
  iata:code  "343"^^<http://www.w3.org/2001/XMLSchema#string> ;
  iata:hasManufacturer <http://www.semwebtech.org/domains/2006/\
                          travel/iata/manufacturers/Airbus>;
  iata:person_capacity   247  .

<http://www.semwebtech.org/domains/2006/travel/iata/airports/\
 Stapleton_Airport/United_States/Denver/\
 Denver_International_Airport>
  iata:name  "Denver"  ;
  a          iata:Airport  ;
  travel:timezone  -0700  ;
  iata:nearestCity
  <http://www.semwebtech.org/mondial/10/\
    countries/USA/provinces/ Colorado/cities/Denver/> ;
  iata:code  "DEN"  .

<http://www.semwebtech.org/domains/2006/travel/iata/\
 airports/Germany/Frankfurt/Frankfurt_International_Airport>
  iata:name  "Frankfurt"  ;
  a          iata:Airport  ;
  iata:nearestCity
  <http://www.semwebtech.org/mondial/10/countries/D/provinces/\
    Hessen/cities/Wiesbaden/> ;
```

```
   travel:timezone 0100 ;
   iata:code "FRA" .

<http://www.semwebtech.org/domains/2006/travel/iata/airlines/\
  Lufthansa/Flight#LH447>
   a          travel:Flight ;
   travel:arrival "11.00+1" ;
   travel:departure
         "17.25"^^<http://www.w3.org/2001/XMLSchema#decimal> ;
   travel:duration 625;
   travel:flightNo "LH447" ;
   travel:from
     <http://www.semwebtech.org/domains/2006/travel/\
       iata/airports/Stapleton_Airport/United_States/Denver/\
       Denver_International_Airport> ;
   travel:operatedBy  <http://www.semwebtech.org/domains/\
                        2006/travel/iata/airlines/Lufthansa> ;
   travel:to
     <http://www.semwebtech.org/domains/2006/travel/iata/\
       airports/Germany/Frankfurt/Frankfurt_International_Airport>;
   travel:withType
     <http://www.semwebtech.org/domains/2006/travel/iata/\
       crafts/Airbus/A340-300> .

<http://www.semwebtech.org/domains/2006/travel/iata/airlines/\
  Lufthansa/Flight#LH447-20081010> a
  <http://www.semwebtech.org/domains/2006/travel#served_flight>;
   travel:hasDate 20081010;
   travel:hasFlightNo
     <http://www.semwebtech.org/domains/2006/travel/iata/\
       airlines/Lufthansa/Flight#LH447>;
   travel:hasBookings
      "0"^^<http://www.w3.org/2001/XMLSchema#integer> .
```

A fragment of the ontology of the car rental domain node is shown in the next example. It also contains information about its customer "John Doe".

Example 11.3 (Car Rental Company)

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix cars:
  <http://www.semwebtech.org/domains/2006/industry/cars#>.
@prefix vw:
  <http://www.semwebtech.org/domains/2006/industry/cars/vw/>.
@prefix audi:
  <http://www.semwebtech.org/domains/2006/industry/cars/audi/>.
@prefix foaf: <http://xmlns.com/foaf/0.1/>.
@prefix : <foo://bla/>.

:Branch a owl:Class.
:Offer a owl:Class.
:has-branch rdfs:range :Branch.
:has-offer rdfs:domain :Branch; rdfs:range :Offer.
:currentContractNumber
  a          owl:DatatypeProperty ;
  rdfs:range <http://www.w3.org/2001/XMLSchema#integer> .

:model rdfs:range cars:Model.
[ a travel:car-rental;
  :has-branch :frankfurt, :munich, :paris, :london, :lisbon].
```

```
:wiesbaden
  travel:located
 <http://www.semwebtech.org/mondial/10/countries/D/provinces/\
   Hessen/cities/Wiesbaden/> ;
  :has-offer
     [ :model vw:golf; :price 60],
     [ :model audi:A4; :price 80],
     [ :model audi:A6; :price 100].

vw:golf cars:class cars:B.
vw:passat cars:class cars:C.
audi:A4 cars:class cars:C.
audi:A6 cars:class cars:D.

[ a foaf:Person; foaf:name "John Doe" ;
  foaf:mbox <mailto:john@doe.nop>;
  :name <http://example.org#JohnDoe> ;
  :rental-car-max-price 60;
  :owns-car vw:golf , vw:passat ].
```

Besides the ontologies, which have been presented, the rule definitions are the most important component in the specification of the scenario.

11.3 Rule Specifications

From the existing direct flight connections (which are given as facts to the knowledge base) all possible connected flights between the airports are calculated in a hybrid reasoning process using an F-Logic programme (see Example 11.15). Now the node can be queried for connections between arbitrary cities which are served by the airline company. The connection which is returned is always the fastest (the calculation of the duration of a flight includes the waiting time between two flights at an airport). In case that a flight is fully booked, information about alternative flights is also available.

In the following the rules of the application domain are described. Figure 11.3 gives an overview on how these rules interact by use of actions and events.

The booking of a flight can be achieved by sending an action (travel:flight-booking) to the domain node which becomes mapped by the following ACA rule to a knowledge base update:

Example 11.4 (ACA rule: flight-booking)

```
import module namespace
   travel = "http://www.semwebtech.org/domains/2006/travel#"
   at "http://localhost:8080/domain-node/aca/modules/travel";
for $booking in //travel:flight-booking
let $flight := $booking/travel:flight
let $person := $booking/travel:passenger
return
<rdfu:condition
 xmlns:rdfu="http://www.semwebtech.org/languages/2006/rdfupdate#"
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:travel="http://www.semwebtech.org/domains/2006/travel#"
 rdfu:ask="&lt;{ $flight}&gt; a travel:served_flight.
   &lt;{ $flight}&gt; travel:hasBookings ?booking.
   &lt;{ $flight}&gt; travel:hasFlightNo ?con.
```

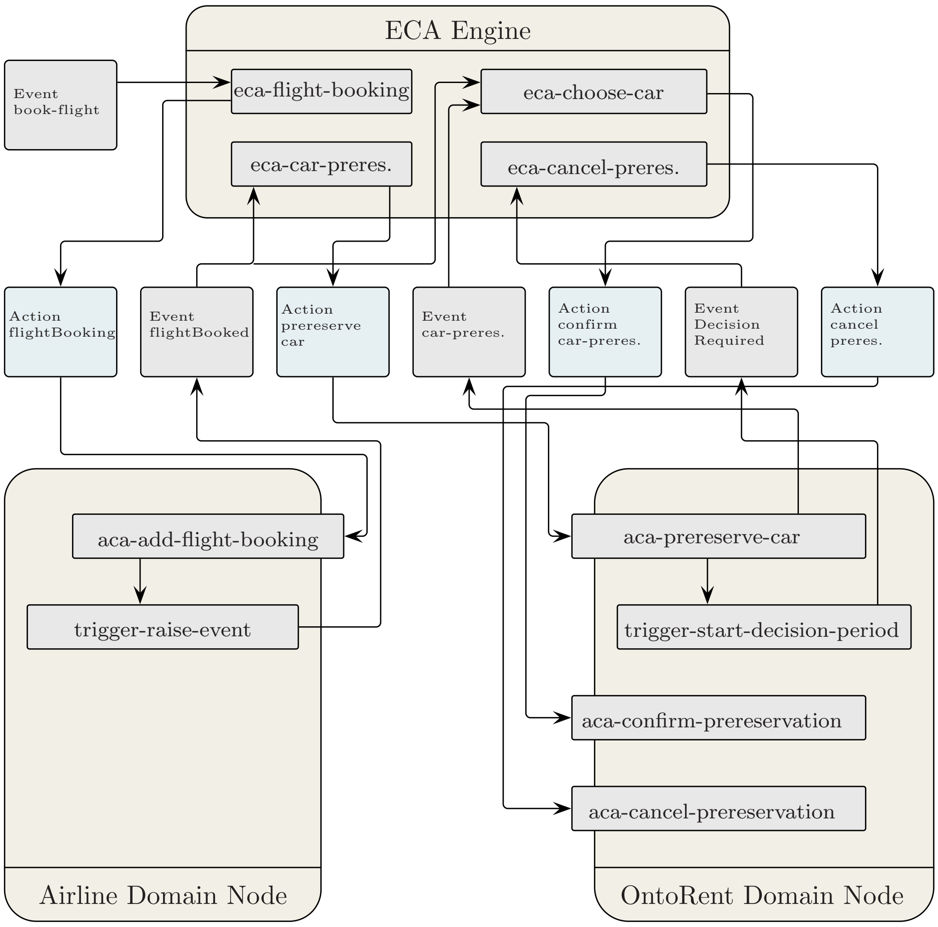


Figure 11.3: Interaction of Rules, Actions and Events in the Travel Scenario

```
    ?con rdf:type travel:Flight .
    ?con travel:withType ?typ.
    ?typ iata:person_capacity ?cap.
    FILTER (?booking &lt; ?cap).
  ">
<rdfu:insert
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:rdfu="http://www.semwebtech.org/languages/2006/rdfupdate#">
   <rdf:subject rdf:about="{$flight}"/>
   <rdf:predicate rdf:about=
      "http://www.semwebtech.org/domains/2006/travel#hasBooking"/>
   <rdf:object rdf:about="{$person}"/>
</rdfu:insert>
</rdfu:condition>
```

The condition of this ACA rule evaluates a test whether the requested flight still has available seats (by comparing the amount of bookings so far to the capacity of the airplane that is used for that flight). If there are any spare seats a travel:hasBooking statement is inserted for the combination of that flight and passenger.

In addition to that ACA rule there are two triggers which react on insertions of travel:hasBooking (the insert operation is seen as an event on the level of the application). The first trigger (Example 11.5) refreshes the travel:hasBookings counter for each flight that received a new booking:

Example 11.5 (Trigger: add-booking)

```
CREATE TRIGGER add-flight-booking
ON INSERTION OF
    http://www.semwebtech.org/domains/2006/travel#hasBooking
WHEN SELECT (count($b) as $bookings) $oldbookings
  WHERE {
    <$new.subject>
    <http://www.semwebtech.org/domains/2006/travel#hasBooking> $b.
    <$new.subject>
    <http://www.semwebtech.org/domains/2006/travel#hasBookings>
     $oldbookings .
 } group by $oldbookings
DO
BEGIN
update($new.subject,
         http://www.semwebtech.org/domains/2006/travel#hasBookings,
         $oldbookings)
         set object = $bookings;
END;
```

The second trigger (Example 11.6) raises an event for every single booked flight:

Example 11.6 (Trigger: raise booked-seat event)

```
CREATE TRIGGER raise-booked-seat-event
ON INSERTION OF
http://www.semwebtech.org/domains/2006/travel#hasBooking
WHEN SELECT ?flightNo ?date
WHERE {
<$new.subject> a
<http://www.semwebtech.org/domains/2006/travel#served_flight>.
<$new.subject>
<http://www.semwebtech.org/domains/2006/travel#hasFlightNo> ?con.
<$new.subject>
<http://www.semwebtech.org/domains/2006/travel#hasDate> ?date.
?con <http://www.semwebtech.org/domains/2006/travel#flightNo>
?flightNo. }
DO
BEGIN
raise event(
<travel:flightBooked xmlns:travel=
        "http://www.semwebtech.org/domains/2006/travel#">
<travel:flightNo>?flightNo</travel:flightNo>
  <travel:passenger>$new.object</travel:passenger>
  <travel:date>?date</travel:date>
</travel:flightBooked>
);
END;
```

In order to allow for the booking of a flight between two airports an ECA rule is registered at the ECA engine:

Example 11.7 (ECA-rule: book flight)

```
<eca:Rule xmlns:eca=
    "http://www.semwebtech.org/languages/2006/eca-ml#">
  <eca:Event bind-to-variable="booking">
    <xqm:Event xmlns:xqm=
      "http://www.semwebtech.org/languages/2006/xmlql#">
      <travel:book-flight xmlns:travel
         "http://www.semwebtech.org/domains/2006/travel#">
        <travel:person>{$Person}</travel:person>
        <travel:from>{$From}</travel:from>
        <travel:to>{$To}</travel:to>
        <travel:date>{$Date}</travel:date>
      </travel:book-flight>
    </xqm:Event>
  </eca:Event>
  <eca:Query>
    <eca:Opaque
      uri="http://swan01.informatik.uni-goettingen.de:8080/domain\
           -node/rdfserver/actions" eca:method="post">
      <eca:has-input-variable name="From" use="$From"/>
      <eca:has-input-variable name="To" use="$To"/>
      <eca:has-input-variable name="Date" use="$Date"/>
       <![CDATA[
      <applnode:query xmlns:applnode=
        "http://www.semwebtech.org/2006/application-node#"
        sparql-query=
        "PREFIX xsd: &lt;http://www.w3.org/2001/XMLSchema#&gt;
         SELECT ?connection ?fdep ?farr ?ffrom ?fto
                ?dep ?arr ?duration
         WHERE {
           ?flight   a                 travel:ConnectedFlight.
           ?flight   travel:from       ?from.
           ?flight   travel:from       ?fromuri .
           ?fromuri  iata:code         $From .
           ?flight   travel:to         ?touri.
           ?touri    iata:code         $To .
           ?flight   travel:departure  ?dep.
           ?flight   travel:arrival    ?arr.
           ?flight   travel:hasFlight  ?flightNo.
           ?flight   travel:duration   ?duration.
           ?flightNo  travel:departure  ?fdep.
           ?flightNo  travel:arrival    ?farr .
           ?flightNo  travel:from       ?ffrom .
           ?flightNo  travel:to         ?fto .
           ?connection  a travel:served_flight.
           ?connection  travel:hasFlightNo ?flightNo.
           ?connection  travel:hasDate    $Date.
         }"/>]]>
    </eca:Opaque>
  </eca:Query>
  <eca:Action>
    <xqm:Action xmlns:xqm=
        "http://www.semwebtech.org/languages/2006/xmlql#">
      <eca:has-input-variable name="Person"/>
      <eca:has-input-variable name="Date"/>
      <eca:has-input-variable name="connection"/>
      <eca:has-input-variable name="fdep"/>
      <eca:has-input-variable name="farr"/>
      <eca:has-input-variable name="ffrom"/>
      <eca:has-input-variable name="fto"/>
      <travel:flight-booking xmlns:travel=
          "http://www.semwebtech.org/domains/2006/travel#">
        <travel:passenger>{$Person}</travel:passenger>
        <travel:date>{$Date}</travel:date>
        <travel:flight>{$connection}</travel:flight>
        <travel:departure>{$fdep}</travel:departure>
        <travel:arrival>{$farr}</travel:arrival>
        <travel:from>{$ffrom}</travel:from>
        <travel:to>{$fto}</travel:to>
      </travel:flight-booking>
    </xqm:Action>
  </eca:Action>
</eca:Rule>
```

This rule fires upon receiving an event of the following kind:

Example 11.8 (book-flight event)

```
 <travel:book-flight
   xmlns:travel="http://www.semwebtech.org/domains/2006/travel#">
      <travel:person><![CDATA[http://example.org#JohnDoe]]>
      </travel:person>
      <travel:from><![CDATA[&quot;SFO&quot;]]></travel:from>
      <travel:to><![CDATA[&quot;FRA&quot;]]></travel:to>
      <travel:date><![CDATA[&quot;20081010&quot;^^xsd:integer]]>
      </travel:date>
</travel:book-flight>
```

The condition part of the ECA rule sends a query to the airline domain node asking for a travel:ConnectedFlight between the two airports that are given in the event. If there is such a flight connection, the domain node will return a result tuple for each necessary flight (from San Francisco to Frankfurt, there are two flights necessary: San Francisco-Denver, Denver-Frankfurt). The action part of the ECA rule sends a travel:flight-booking action to the airline domain node for every result tuple.

The ACA rule in Example 11.4 translates the travel:flight-booking action into a knowledge base update: a seat for the flight is booked, causing the trigger from Example 11.6 to raise a travel:flightBooked event.

There is another ECA rule which serves the purpose of a car pre-reservation:

Example 11.9 (ECA-rule: car pre-reservation)

```
<eca:Rule
  xmlns:eca="http://www.semwebtech.org/languages/2006/eca-ml#">
 <eca:Event bind-to-variable="booking">
  <xqm:Event
    xmlns:xqm="http://www.semwebtech.org/languages/2006/xmlql#">
   <travel:flightBooked xmlns:travel=
               "http://www.semwebtech.org/domains/2006/travel#">
    <travel:passenger>{$Person}</travel:passenger>
    <travel:to>{$To}</travel:to>
    <travel:date>{$Date}</travel:date>
   </travel:flightBooked>
  </xqm:Event>
 </eca:Event>

 <eca:Query>
  <eca:Opaque eca:method="post"
     uri="http://swan02.informatik.uni-goettingen.de:8080/\
          domain-node/rdfserver/actions">
   <eca:has-input-variable name="To" use="$To"/>
   <eca:has-input-variable name="Person" use="$Person"/>
    <![CDATA[
     <applnode:query xmlns:applnode=
             "http://www.semwebtech.org/2006/application-node#"
       sparql-query="
         PREFIX ontorent:
          &lt;http://www.semwebtech.org/domains/2006/travel/\
               onto-rent/&gt;
         select (?model as ?AvailableCar) ?price ?tocity ?cn
           where {
             ?p a foaf:Person .
             ?p &lt;foo://bla/name&gt; &lt;$Person&gt; .
             ?p &lt;foo://bla/owns-car&gt; ?car .
             ?car cars:class ?class .
             ?x a ontorent:Branch .
             ?x ontorent:has-offer _:o .
             ?x travel:located ?tocity .
             &lt;$To&gt; iata:nearestCity ?tocity .
             _:o ontorent:model ?model .
             ?model cars:class ?class .
             _:o ontorent:price ?price .
             ontorent:statistics ontorent:currentContractNumber
               ?cn .}" />
     ]]>
  </eca:Opaque>
 </eca:Query>

 <eca:Query eca:bind-to-variable="newcn">
  <eca:Opaque eca:language="http://www.w3.org/XQuery">
   <eca:has-input-variable eca:name="cn"/>
     let $c := fn:sum(($cn,1))
     return $c cast as xs:int
   </eca:Opaque>
 </eca:Query>

 <eca:Action>
  <xqm:Action xmlns:xqm="http://www.semwebtech.org/languages/\
                            2006/xmlql#">
   <eca:has-input-variable name="Person"/>
   <eca:has-input-variable name="AvailableCar"/>
   <eca:has-input-variable name="cn"/>
   <eca:has-input-variable name="Date"/>
   <travel:prereserve-car xmlns:travel=
             "http://www.semwebtech.org/domains/2006/travel#">
     <travel:person>{$Person}</travel:person>
     <travel:car>{$AvailableCar}</travel:car>
     <travel:city>{$tocity}</travel:city>
     <travel:price>{$price}</travel:price>
     <travel:date>{$Date}</travel:date>
     <travel:cn>{$newcn}</travel:cn>
   </travel:prereserve-car>
  </xqm:Action>
 </eca:Action>
</eca:Rule>
```

As soon as the ECA engine receives a travel:flightBooked event the second domain node is needed: the car-rental domain node. This domain node represents a car-rental company with several branches at selected airports or nearby cities. Each branch has a set of available cars at specific prices. The purpose of this node is, similar as with the airline domain node, to demonstrate the capabilities of the rule driven architecture of Swan and Mars rather than to implement a real-world business application (although it can easily be extended for such a purpose). The domain node does not store information about single cars that are being booked or pre-reserved. Instead, all reservations are realised by adding a statement for the *types* of cars being reserved for a certain contract number. As there is no limit to such statements about a car type, there is also no limit to the availability of cars. In a more sophisticated application, there would be a booking object with information about contract, car-number, price, and date.

The car-rental company shares part of the domain knowledge with the airline domain node as both make use of the travel ontology. The domain node also knows about existing airports and where they are located at. Hereby corresponding branch offices of the OntoRent company and airports can be matched (e.g. the airport in Frankfurt and the car-rental branch in Wiesbaden).

Once the ECA engine receives a travel:flightBooked event, the car-rental domain node is queried for suitable cars. In this scenario the car rental domain node has access to details about registered customers with regard to their preferences. It is known, for example that *John Doe* owns a vw:Golf and a vw:Passat. It is assumed that a customer wants to rent a car of the same kind that he owns. Therefore, the query asks for cars belonging to the same class as the one that the customer owns, available in the car-rental branch of the city that is next to the destination airport[1]. If *John Doe* travels to Frankfurt there would be two types of cars suitable for him: a VW Golf and an Audi A4. For both of them a *pre-reservation* would be made from which he can choose. These pre-reservations are, again, realised by an ACA rule:

Example 11.10 (ACA-rule: Car pre-reservation)

```
import module namespace
 travel = "http://www.semwebtech.org/domains/2006/travel#"
        at "http://localhost:8080/domain-node/aca/modules/travel";
for $reservation in //travel:prereserve-car
let $car := $reservation/travel:car
let $person := $reservation/travel:person
let $from := $reservation/travel:city
let $price := $reservation/travel:price
let $cn := $reservation/travel:cn
let $date := $reservation/travel:date
return
<rdfu:condition xmlns:rdfu=
  "http://www.semwebtech.org/languages/2006/rdfupdate#"
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:travel="http://www.semwebtech.org/domains/2006/travel#"
 xmlns:ontorent="http://www.semwebtech.org/domains/2006/travel/\
                  onto-rent#"
 rdfu:ask=
  "prefix ontorent:
   &lt;http://www.semwebtech.org/domains/2006/travel/\
        onto-rent/&gt;
   ASK{{
       ?x travel:located &lt;{$from}&gt; .
       ?x ontorent:has-offer _:o.
       _:o ontorent:model &lt;{$car}&gt;
```

[1]Actually, the car *pre-reservation* is done for every intermediate airport on a connected flight, but only at the destination airport there will be a confirmation of a pre-reservation.

```
      }}">
  <rdfu:insert
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:rdfu=
       "http://www.semwebtech.org/languages/2006/rdfupdate#">
    <rdf:subject rdf:about="{$car/text()}"/>
    <rdf:predicate
     rdf:about="http://www.semwebtech.org/domains/\
                 2006/travel/onto-rent/prereserved_for"/>
    <rdf:object rdf:about="{$cn/text()}"/>
  </rdfu:insert>
  <applnode:raise-event>
    <travel:car_prereservation>
      <travel:person>{$person/text()}</travel:person>
      <travel:location>{$from/text()}</travel:location>
      <travel:car_type>{$car/text()}</travel:car_type>
      <travel:price>{$price/text()}</travel:price>
      <travel:date>{$date/text()}</travel:date>
      <travel:contractNumber>{$cn/text()}</travel:contractNumber>
    </travel:car_prereservation>
  </applnode:raise-event>
</rdfu:condition>
```

This ACA rule has two actions as consequences: firstly, the pre-reservation is inserted into the knowledge base. Secondly, a travel:car_prereservation event is raised. Besides the raising of the event by the ACA rule there is another reaction in the knowledge base: a trigger fires upon the event of an insertion of a statement with the predicate ontorent:prereserved_for:

Example 11.11 (Trigger: Car pre-reservation)

```
CREATE TRIGGER testtrigger007
ON INSERTION OF http://www.semwebtech.org/domains/\
                2006/travel/onto-rent/prereserved_for
DO
BEGIN
raise event (
<travel:DecisionRequired
 xmlns:travel="http://www.semwebtech.org/domains/2006/travel#"
 car_type="$new.subject"
 contractNumber="$new.object" />
);
END;
```

As it was shown with the rules above, a car-pre-reservation has, as a consequence, two events: the travel:car_prereservation event and the travel:decisionRequired event. This ECA rule causes all pre-reservations to be cancelled after a specified time (here 24h):

Example 11.12 (ECA-rule: Cancel car-pre-reservation)

```
<eca:Event>
 <xqm:Event xmlns:xqm=
      "http://www.semwebtech.org/languages/2006/xmlql#"
   xmlns:travel="http://www.semwebtech.org/domains/2006/travel#">
    <travel:DecisionRequired
      contractNumber="{$Cn}" car_type="{$Car}" />
  </xqm:Event>
 </eca:Event>
 <eca:Action xmlns:xqm=
      "http://www.semwebtech.org/languages/2006/xmlql#">
  <eca:has-input-variable name="Cn" />
  <eca:has-input-variable name="Car" />
  <xqm:Action xmlns:travel=
     "http://www.semwebtech.org/domains/2006/travel#">
  <ccs:Sequence xmlns:ccs=
     "http://www.semwebtech.org/languages/2006/ccs#">
   <ccs:Sleep ccs:hours="24"/>
    <travel:cancel-car-prereservation
     contractNumber="{$Cn}" car="{$Car}"/>
   </ccs:Sequence>
  </xqm:Action>
 </eca:Action>
</eca:Rule>
```

All pre-reservations that have not been confirmed before the timeout will be cancelled by a travel:cancel-car-prereservation action that is sent to the domain node. This ECA rule is registered at the ECA engine by the car-rental domain-node itself. Additionally, there is another ECA rule, which becomes registered by the customer:

Example 11.13 (ECA-rule: Choose car)

```
<eca:Rule xmlns:eca=
    "http://www.semwebtech.org/languages/2006/eca-ml#"
 xmlns:travel="http://www.semwebtech.org/domains/2006/travel#">
 <eca:initialize-variable eca:name="destination">
  http://www.semwebtech.org/domains/2006/travel/iata/airports/\
  Germany/Frankfurt/Frankfurt_International_Airport
 </eca:initialize-variable>
 <eca:initialize-variable eca:name="name">
  http://example.org#JohnDoe
 </eca:initialize-variable>
 <eca:Event>
  <xqm:Event xmlns:xqm=
   "http://www.semwebtech.org/languages/2006/xmlql#">
   <travel:flightBooked>
    <travel:flightNo>{$flightNo}</travel:flightNo>
    <travel:passenger>{$name}</travel:passenger>
    <travel:date>{$date}</travel:date>
    <travel:to>{$destination}</travel:to>
   </travel:flightBooked>
  </xqm:Event>
 </eca:Event>
 <eca:Action>
  <ccs:Sequence xmlns:ccs=
        "http://www.semwebtech.org/languages/2006/ccs#">
    <ccs:TopK ccs:topk="1" ccs:threshold="1000"
      ccs:continue="false" ccs:wait="3600000"
      ccs:type="xsd:integer" ccs:order="asc">
     <ccs:mapfunction>
      <ccs:Query>
       <eca:Opaque eca:language="http://www.w3.org/XPath">
        <eca:has-input-variable eca:name="price"/>
         <![CDATA[ $price ]]>
       </eca:Opaque>
      </ccs:Query>
     </ccs:mapfunction>
     <ccs:Event ccs:continuous="true">
      <xqm:Event xmlns:xqm=
         "http://www.semwebtech.org/languages/2006/xmlql#">
       <travel:car_prereservation>
        <travel:person>{$name}</travel:person>
```

```
          <travel:location>{$city}</travel:location>
          <travel:car_type>{$car}</travel:car_type>
          <travel:price>{$price}</travel:price>
          <travel:contractNumber>{$contractNumber}
          </travel:contractNumber>
          <travel:date>{$date}</travel:date>
        </travel:car_prereservation>
      </xqm:Event>
    </ccs:Event>
   </ccs:TopK>
   <xqm:Action xmlns:xqm=
        "http://www.semwebtech.org/languages/2006/xmlql#">
    <ccs:has-input-variable ccs:name="car"/>
    <ccs:has-input-variable ccs:name="contractNumber"/>
    <travel:confirm-car-prereservation
      contractNumber="{$contractNumber}" car_type="{$car}"/>
   </xqm:Action>
  </ccs:Sequence>
 </eca:Action>
</eca:Rule>
```

This rule automises the decision procedure and should be registered by the customer before sending the travel:book-flight event at the beginning of the booking process. The rule is triggered by a travel:flightBooking event where name and destination match the values given in the initialisation variables of the rule. The action part consists of a CCS process which collects, after the detection of the travel:flightBooked event, for the duration of one hour (specified with the ccs:wait attribute) all travel:car_prereservation events and chooses that one with the lowest price. Finally a travel:confirm-car-prereservation action is sent to the car-rental domain-node, which is mapped by the following ACA rule to a knowledge base update:

Example 11.14 (ACA: Confirm car pre-reservation)

```
import module namespace travel =
  "http://www.semwebtech.org/domains/2006/travel#"
   at "http://localhost:8080/domain-node/aca/modules/travel";
for $confirm in //travel:confirm-car-prereservation
let $contractNumber := $confirm/@contractNumber
let $car := $confirm/@car_type
return
<rdfu:update
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:rdfu=
        "http://www.semwebtech.org/languages/2006/rdfupdate#">
   <rdf:subject rdf:about="{$car}"/>
   <rdf:predicate rdf:about="http://www.semwebtech.org/domains/\
       2006/travel/onto-rent/prereserved_for"/>
   <rdf:object rdf:about="{$contractNumber}"/>
   <rdfu:set>
    <rdf:predicate rdf:about="http://www.semwebtech.org/domains/\
       2006/travel/onto-rent/reserved_for"/>
   </rdfu:set>
</rdfu:update>
```

This ACA rule simply updates the predicate of the pre-reservation-statement such that it is now a reservation statement.

F-Logic Programme

The domain node representing the airline company uses a flight plan in which direct flight connections between airports are defined, e.g. *San Francisco → Denver*. By use of hybrid reasoning all possible connections between pairs of airports can be calculated, e.g. *San Francisco → Denver → Frankfurt*. The F-Logic programme creates connected flight objects by concatenating the names of the departure and destination airports of two flights (either simple or connected flights). By appending further simple flights to a connected flight, longer connected flights are created until eventually all possible connections between airports are constructed. If two airports are to be connected by more than one connected flight only the one with the shorter flight duration is chosen. The duration of a connected flight is calculated by adding the time of the constituting flights plus the waiting time at the intermediate airports.

Note that this construction of new flights (or new objects in general) would not be possible in OWL or by the use of the built-in rule engines of Jena. Although the owl:transitiveProperty allows to deduce that *San Francisco* is connected with *Frankfurt* it is not possible to assign properties to such connections. Furthermore, F-Logic reasoning allows for stratification. For operations on cyclic graphs a fixpoint computation is necessary. This, however, is not possible with the in-built rule engines that come with the programming framework Jena. Hence, the built-in reasoning engines of Jena would fail because of the cycles in the flight plan (see Figure 11.2).

Example 11.15 (F-Logic: connected flights)

```
%% -----------------------------------------------------------------
%% Rule 1 : connect simple flights
%% -----------------------------------------------------------------

Z:connectedFlight [
"http://www.semwebtech.org/domains/2006/travel#from"->> A ;
"http://www.semwebtech.org/domains/2006/travel#to"->> C ;
"http://www.semwebtech.org/domains/2006/travel#hasFlight"->> X ;
"http://www.semwebtech.org/domains/2006/travel#hasFlight"->> Y ;
"http://www.semwebtech.org/domains/2006/travel#departure"->> S ;
 "http://www.semwebtech.org/domains/2006/travel#arrival" ->> T ;
"http://www.semwebtech.org/domains/2006/travel#duration" ->> DZ ;
isFastest->>"true" ],
"http://www.semwebtech.org/domains/2006/travel#hasFlight" : url,
Z:url
:-
X:"http://www.semwebtech.org/domains/2006/travel#Flight",
Y:"http://www.semwebtech.org/domains/2006/travel#Flight",
X["http://www.semwebtech.org/domains/2006/travel#from"->>A],
X["http://www.semwebtech.org/domains/2006/travel#to"->>B],
X["http://www.semwebtech.org/domains/2006/travel#departure"->>S],
Y["http://www.semwebtech.org/domains/2006/travel#from"->>B],
Y["http://www.semwebtech.org/domains/2006/travel#to"->>C],
Y["http://www.semwebtech.org/domains/2006/travel#arrival"->>T],
A["http://www.semwebtech.org/domains/2006/travel/iata/meta#name"
   ->>AN],
B["http://www.semwebtech.org/domains/2006/travel/iata/meta#name"
   ->>BN],
C["http://www.semwebtech.org/domains/2006/travel/iata/meta#name"
   ->>CN],
not A = C,
X["http://www.semwebtech.org/domains/2006/travel#duration"->>DX],
Y["http://www.semwebtech.org/domains/2006/travel#duration"->>DY],
pause(X,Y,DP),
DZ = DX + DY + DP,
strcat("http://www.semwebtech.org/domains/2006/travel#",AN,Z1),
strcat(Z1,"/",Z2),strcat(Z2,BN,Z3),
strcat(Z3,"/",Z4),strcat(Z4,CN,Z).

%% ----------------------------------------------------------------
%% Rule 2 : append simple flights to existing connected flights
%% ----------------------------------------------------------------

Z:connectedFlight
[
"http://www.semwebtech.org/domains/2006/travel#from"->> A ;
"http://www.semwebtech.org/domains/2006/travel#to"->> C ;
"http://www.semwebtech.org/domains/2006/travel#hasFlight"->> XF ;
"http://www.semwebtech.org/domains/2006/travel#hasFlight"->> Y ;
"http://www.semwebtech.org/domains/2006/travel#departure"->> S ;
"http://www.semwebtech.org/domains/2006/travel#duration" ->> DZ ;
"http://www.semwebtech.org/domains/2006/travel#arrival" ->> T ;
isFastest->>"true"
],
"http://www.semwebtech.org/domains/2006/travel#hasFlight" : url,
Z:url
:-
X:connectedFlight ,
not X[isFastest->>"false"],
Y:"http://www.semwebtech.org/domains/2006/travel#Flight",
X["http://www.semwebtech.org/domains/2006/travel#from"->>A],
X["http://www.semwebtech.org/domains/2006/travel#hasFlight"
   ->> XF],
X["http://www.semwebtech.org/domains/2006/travel#to"->>B],
X["http://www.semwebtech.org/domains/2006/travel#departure"->>S],
Y["http://www.semwebtech.org/domains/2006/travel#from"->>B],
Y["http://www.semwebtech.org/domains/2006/travel#to"->>C],
Y["http://www.semwebtech.org/domains/2006/travel#arrival"->>T],
C["http://www.semwebtech.org/domains/2006/travel/iata/meta#name"
   ->>CN],
not substr(CN,X),
X["http://www.semwebtech.org/domains/2006/travel#duration"->>DX],
Y["http://www.semwebtech.org/domains/2006/travel#duration"->>DY],
pause(X,Y,DP),
DZ = DX + DY + DP,
strcat(X,"/",Z1),strcat(Z1,CN,Z).

%% ----------------------------------------------------------------
%% Rule 3 : all connected flights with longer durations are
%%          marked as not being fastest
%% ----------------------------------------------------------------

A[isFastest->>"false"]
:-
A:connectedFlight
[
"http://www.semwebtech.org/domains/2006/travel#duration"->>_DA;
"http://www.semwebtech.org/domains/2006/travel#from"->>F;
"http://www.semwebtech.org/domains/2006/travel#to"->>T
]
,
_B:connectedFlight
[
"http://www.semwebtech.org/domains/2006/travel#duration"->>_DB;
"http://www.semwebtech.org/domains/2006/travel#from"->>F;
```

```
"http://www.semwebtech.org/domains/2006/travel#to"->>T
],
_DA > _DB.

%% ---------------------------------------------------------------
%% Rules 4-8 : get the hour and minute parts of  time values
%%             (which are given as decimals as one of
%%              #HH.MM, #HH.M or "HH.MM+1"
%% ---------------------------------------------------------------

hour(X,Y,H):-
X[Y->>D],
not string(D),
strcat("\"",D,D1),strcat(D1,"\"",DD),
pmatch(DD, "/([0-9]*)\./"  , "$1", DH),
string2integer(DH,H).

hour(X,Y,H):-
X[Y->> _D],
string(_D),
Y="http://www.semwebtech.org/domains/2006/travel#arrival",
pmatch(_D, "/([0-9]*)\./"  , "$1", DH),
string2integer(DH,H).

minute(X,Y,M):-
X[Y->>D],
not string(D),
strcat("\"",D,D1),strcat(D1,"\"",DD),
pmatch(DD, "/\.([0-9]{2})/"  , "$1", DH),
string2integer(DH,M).

minute(X,Y,M):-
X[Y->>D],
not string(D),
strcat("\"",D,D1),strcat(D1,"\"",DD),
pmatch(DD, "/\.([0-9]{1}\D)/"  , "$1", DH),
string2integer(DH,MT),
M = MT * 10.

minute(X,Y,M):-
X[Y->> _D],
string(_D),
Y="http://www.semwebtech.org/domains/2006/travel#arrival",
pmatch(_D, "/\.([0-9]*)/"  , "$1", DH),
string2integer(DH,M).

%% ---------------------------------------------------------------
%% Rules 9+10 : Calculate the amount of time between arrival of
%%              flight X and departure of flight Y
%% ---------------------------------------------------------------

pause(X,Y,HH) :-
hour(X,
  "http://www.semwebtech.org/domains/2006/travel#arrival",XAH),
hour(Y,
  "http://www.semwebtech.org/domains/2006/travel#departure",YDH),
minute(X,
  "http://www.semwebtech.org/domains/2006/travel#arrival",XAM),
minute(Y,
  "http://www.semwebtech.org/domains/2006/travel#departure",YDM),
YDH < XAH,
MM = YDM - XAM,
HH = ( YDH - XAH + 24 ) * 60 + MM .

pause(X,Y,HH) :-
hour(X,
   "http://www.semwebtech.org/domains/2006/travel#arrival",XAH),
hour(Y,
   "http://www.semwebtech.org/domains/2006/travel#departure",YDH),
minute(X,
   "http://www.semwebtech.org/domains/2006/travel#arrival",XAM),
minute(Y,
   "http://www.semwebtech.org/domains/2006/travel#departure",YDM),
YDH >= XAH,
MM = YDM - XAM,
HH = (YDH - XAH ) * 60 + MM.

%% ---------------------------------------------------------------------
?- sys.eval.
?- sys.strat.doIt.
%% ---------------------------------------------------------------------

%% ---------------------------------------------------------------------
%% Rule 11: Export only those connections that are fastest
%% ---------------------------------------------------------------------

X:"http://www.semwebtech.org/domains/2006/travel#ConnectedFlight"
,"http://www.semwebtech.org/domains/2006/travel#ConnectedFlight"
 : url
:-
X:connectedFlight ,
not X[isFastest->>"false"].

%% ---------------------------------------------------------------------
?- sys.eval.
%% ---------------------------------------------------------------------
```

The hybrid reasoning process has to be run only once during the initialisation of the domain node. Florid then delivers instances of the newly introduced concept travel:ConnectedFlight (which is also created and added to the concept definitions in the DL TBox). Only in case that, at a later stage, new flight connections are added to the knowledge base (which can be assumed to happen very seldom) the hybrid reasoning process has to be started again. This can be realised by use of a trigger like

```
ON CREATION OF INSTANCE OF CLASS travel:Flight
DO BEGIN start-flogic-reasoning();
END;
```

11.4 Summary

The scenario shows how the Mars and Swan architectures interact by the use of actions and events. Different types of rules are presented: ECA rules, ACA rules and triggers. The overall concept is the event driven architecture. Interacting with the framework is strictly different from usual web services. Instead of calling a remote procedure of a web service, abstract definitions can be given. Once the rules are defined and registered, the abstract action definitions can be sent to the domain nodes. The vocabulary for these actions is defined by the ontology of the application domain. For the user it is not necessary to know how these actions are realised. Using the abstract vocabulary of the common domain ontology it is possible to say what should be done (for example sending an event travel:book-flight to the domain broker).

The application domain nodes as presented in the example have very limited capabilities, as it was not the intention to demonstrate a fully-fledged application node. Rather the idea is to show how the communication between MARS and SWAN using events and actions works and how abstract actions translate into knowledge base updates. This approach can be extended to more complex situations very easily, e.g. by defining behaviour for flight cancellations. This ease of extensibility is one of the strong points of this architecture. Furthermore, all process logic is put into rules. Along with a logical characterisation of the rules it is possible to reason about the effects of events in the application domain. This is a broad field for further investigations.

In the following chapter, the concepts that were presented in this work are discussed and put into the context of related work.

Chapter 12

Discussion

12.1 Limitations of DL Reasoning

Some of the limits of Description Logics have already been discussed in the context of OWL in Section 3.7. In the following the limits of DL reasoning in real world applications are investigated.

The application domain that Description Logics were originally intended for was *ontology engineering*, that is to design, construct, and maintain large conceptual schemas [Hor98]. Later, with the advent of the Semantic Web, Description Logics were chosen for the logical foundation of the ontology description language OWL DL. OWL provides the designer of an ontology with axioms for the modelling of concepts, roles, and the relationships of individuals. The constructors and axioms of DLs were widely explored at that time. With the application of DL reasoning for the Semantic Web the power of that formalism could be tested with very heterogeneous applications. With regard to ontologies consisting of large TBoxes, DL reasoners indeed seem to be well suited [HO01]. But often applications have to handle queries over knowledge bases that are built from small and simple TBoxes, but also include large ABoxes. This is where DL reasoners reach their limits. The problem with ABox reasoning seems to arise not so much from the computational complexity of ABox reasoning, but from the fact that the number of individuals might be extremely large [HLTB04].

Many of the proposals that try to deal with the scalability problem of DL ontologies are (re-)using techniques from database research. Here, the problem of very large data sets was already explored intensively. But these findings cannot be reused in the world of DL without certain restrictions. It must not be forgotten that there is a gap between deductive databases and classical logical inference. Furthermore, the handling of individuals is strictly different: Equality of individuals can be inferred easily in DLs whereas the unique name assumption in databases prevents it. As individuals with multiple given names are a typical situation in the World Wide Web this is an important feature in DLs.

In [HLTB04] the authors use a relational database for the storage of ABox data in combination with a DL reasoner for TBox data. However, they had to put restrictions on the ABox such that only *role-free* instance data can be stored (e.g. A isa B). Hereby, the system is capable of dealing with very large ABoxes and allows for sound and complete answers to instance retrieval queries.

Another solution has been proposed with the KAON2 project [MS06]. The authors refer to research in the field of deductive databases. They present an algorithm that reduces a $\mathcal{SHIQ}$ knowledge base KB to a disjunctive datalog programme DD(KB). This system turned out to perform well on knowledge bases with simple TBoxes and large ABoxes whereas the performance for complex TBox reasoning was in general worse compared to the performance of sophisticated DL reasoners like PELLET, RACER, or FACT++. Also the presence of equality expressions significantly influences the performance of KAON2 badly.

In [MB08] a promising method for improved ABox reasoning was presented. The authors completely separate ABox and TBox, the DL reasoner is used on the TBox exclusively. OWL axioms expressing knowledge about ABox data are translated into rules that are given, together with the ABox instances, to a rule engine. The authors claim that the rule engine is able to achieve the same deductions as the DL reasoner in considerably less time. Although the application of rule systems as a supplement to DL reasoning is not novel at all, the strict separation of DL reasoning for TBox and rule-based reasoning for the ABox is new.

There are no ABox optimisations so far in SWAN. Therefore the aforementioned restrictions have to be considered when ontologies with very large ABoxes are used. See also the discussion in Section 12.4.

12.2 Application of Hybrid Reasoning

The SWAN architecture integrates F-Logic reasoning in order to add some features that are missing from DL reasoning. Some remarks towards these restrictions have already been made in Section 3.7 in reference to OWL. The integration of F-Logic accomplishes to create new objects, to perform algebraic calculations, to evaluate rules with fixpoint computation, and to apply default inheritance. Although these supplementary deductions are very useful they do not come for free. The computation of a fixpoint in alternating DL and F-Logic reasoning can take some time, depending on the size of the ontology and the complexity of the F-Logic programme. This is a tolerable restriction in a static environment, where the hybrid ontology has to be computed only once. As long as no further updates are to be expected and the (hybrid) knowledge base is only used for answering queries, the duration of one run of the hybrid reasoning process is not critical. The situation is different, though, if the knowledge base is integrated dynamically and is subject to change. Depending on the frequency of the updates, a hybrid reasoning process can become a bottleneck.

On this account, hybrid reasoning in SWAN is not intended to be applied continuously. Rather, it is designed to be run once in a while, e.g. during initialisation or upon critical updates. The hybrid reasoning engine integrates F-Logic reasoning by translating the OWL knowledge base into an F-Logic programme followed by a translation of the F-Logic programme (now including all F-Logic deductions) back to OWL. A *continuous* hybrid reasoning process would have to apply such a procedure once after each update to the knowledge base. As long as updates are not expected frequently, this is no problem. But, the more frequent updates are, the more likely it will be that the hybrid reasoning process will limit the performance of the whole knowledge base.

Often, however, it is sufficient to apply hybrid reasoning only from time to time, depending on what parts of the knowledge base become updated. In some situations it is possible to divide the assertional knowledge into *dynamic facts* and *static facts*. This distinction can be arbitrary and depends strongly on the ontology. *Static facts* are expected to change only scarcely whereas *dynamic facts* change more often. Typically, *static facts* express knowledge about *dynamic facts*. The ontology of the airline company, for example, consists of terminological knowledge (concept and role definitions like airplane, flight-connection) and instances to these concepts (e.g. simple flights, flight bookings). F-Logic rules define how to derive further knowledge from existing facts (connected flights from simple flights). Here, simple flights are considered *static facts*: they change only once in a while, for example when the airline company introduces a new flight connection between two cities. Flight bookings, however, are *dynamic facts* and are updated more frequently.

Initially, a hybrid reasoning process is necessary. Unless no further updates to the knowledge base perform any changes to the *meta-facts* (e.g. to the time table or the flight base) there is no need for further hybrid reasoning. Updates to normal facts (e.g. booking of a flight) are regular knowledge base updates that do not need hybrid reasoning capabilities. Hybrid-reasoning *on demand* can be realised with triggers as proposed with Example 9.21.

Hybrid reasoning in Swan offers no solution to the problems that have been discussed in Section 12.1. It does not accelerate the DL reasoning process, rather it is intended to be a supplement to DL reasoning.

As the F-Logic reasoner is only loosely integrated and every hybrid reasoning process is a fixpoint computation on the whole knowledge base it is very likely that this approach will be outperformed by many of the other hybrid reasoning systems. Nevertheless, Swan offers a unique integration of rule-based reasoning into a DL knowledge base. The concept of on-demand-reasoning depends on the rules (triggers) that are loaded into the application domain node. Hereby it is possible to specify precisely when the additional features of F-Logic shall be applied.

12.3 Problems with Datatypes

The reasoning engines for OWL and F-Logic support a different range of datatypes. In Florid integers, strings, and decimals are supported. In OWL, there are numerous propositions for datatypes, often the use of XSD datatypes is suggested. In a hybrid reasoning system like Swan this has to be taken into account when typed literals are translated from one formalism into the other. Especially when the built-in predicates for arithmetic operations in F-Logic are intended to be used, the literals have to be typed properly. In case that an ontology originates from an external source some literals have to be cast to datatypes that are supported in Swan. Here, the range of supported datatypes depends on the Jena framework which allows for the representation of all main XSD datatypes. Reasoning with datatypes, however, is limited by the capabilities of the OWL reasoning engine. For example, datatype reasoning in Pellet is still incomplete (though the developers claim to support all built-in dataypes of XSD).

This problem is aggravated by the fact that XSD datatypes do not seem to be related in an intuitive subsumption hierarchy. Some datatypes can be related easily, for example xsd:int is subsumed by xsd:integer, xsd:integer is subsumed by xsd:decimal. On the other hand, xsd:decimal and xsd:float are not related in a subsumption relationship, the same is true (but even less intuitive) for xsd:double and xsd:float. For instance, it is not possible, when reasoning with XSD datatypes, to equate 13.1^^xsd:decimal with 13.1^^xsd:float, for none of both values entails the other.

Furthermore, the handling of datatypes depends on the way how the RDF data is stored. In the Swan architecture, PostgreSQL is used for the persistent storage of RDF data. All database operations for storing and retrieving data are carried out by Jena automatically. But storing typed literals in this way can cause the loss of type information. The reason for this is that datatype information is interpreted by the intermediate JDBC driver instead of storing the typed literal natively as a string. This has inconvenient concequences. For example, it is not possible to equate the values 13^^xsd:int and 13^^xsd:integer when they are stored as described in a database. When the model is kept in main memory, however, this equation can be easily done.

In real world applications the limitations concerning typed literals become a severe restriction. Ontologies have to be checked carefully in consideration of the datatypes in use, problematic datatypes should be cast to a safe choice. As a consequence, the import of ontologies from external sources becomes more difficult. Instead of simply being added to the knowledge base (which is one of the catching concepts of RDF) the data has to be examined and modified beforehand.

12.4 Practicability

Taking the aforementioned limitations into consideration one might ask the question whether it is possible at all to use the Swan architecture in real world scenarios.

For instance, it is very likely that a knowledge base contains large numbers of individuals. Also the use of nominals is a feature of OWL DL which is supported by Pellet and is frequently used in ontologies. But both large numbers of individuals and even more so the use of nominals slow down the performance of the reasoning engine considerably. Furthermore, the unsatisfactory situation with regard to datatypes is a real handicap. If data from different sources have to be integrated, chances are very high that different datatypes are used. Often literal values have to be compared, sorted, added or manipulated. Also, triggers or ACA-rules may rely on datatype properties (see Example 11.4). If the condition part of a rule relies on the comparison of two literals it must be ensured that this operation is supported for all types of literal values in the knowledge base.

To date, there are no ABox optimisations in Swan. For this reason, ontologies have to be chosen carefully, especially nominals have a considerable impact on the overall performance of an application domain node. The use case as presented in Chapter 11 clearly showed that computational time and system memory become a limiting factor. For the mere demonstration of the capabilities of the Swan architecture, the data set of the travel booking scenario was reduced to a minimum of only seven airports, eleven simple flight connections, a car rental company with three branches and four different kinds of cars. By using this small data set it was possbile to achieve very fast response times when updating or querying the application domain nodes.

But the use of larger data sets (full flight plan with hundreds of airports and flight connections, a car rental company with dozens of different types of cars) made the limitations of this architecture obvious. For example, one flight booking consists of the insertion of a booking statement plus the update of a booking counter for the respective flight. Each of these operations requires the calculation of the difference of two theories which consumes a lot of resources (system memory). In the test scenario each domain node was hosted in a separate virtual machine with 512MB main memory available. If a *connected flight* is to be booked there are at least two such single flight bookings. Using the described setup, such an update takes more than a minute.

In a realistic airline application, updates will occur very frequently. Other application nodes might even have to deal with continuous updates.

Hence, it has to be considered in what way the domain node should be used. Is it necessary to give fast responses and if yes, how many updates are to be expected at a time? If a large number of single updates have to be performed with short response times, the knowledge base updates are likely to become a performance bottleneck for the whole network. Otherwise, these limitations have only little significance. If, for example, the services of an application domain node are needed only once in a while (planning of a time table or scheduling of processes) and the results are not needed at once, this domain node architecture is perfectly suitable.

It is now examined how the concepts that are central to the Swan architecture can be related to other work.

12.5 Related Work

Event-Driven Architectures

The Mars framework is an event-driven architecture where ECA rules are used to define the reactive behaviour of an application domain. These rules are conceptually on an abstract level where no implementation details of the application domain have to be provided. This is different from conventional web services, which are invoked by procedure calls or application-specific commands for data storage or data manipulation. In Mars, the concepts of domain ontologies are used in order to specify the *meaning* of an action or an event. It is left to the application how to translate and execute these specifications.

A similar level of abstraction can be found in the object-oriented programming language Smalltalk [GR89]. A central concept in Smalltalk is the *message*. Messages are sent to target objects instead of directly calling functions. The receiving object decides what to do with the message by comparing the *selector* (identifier) of the message with the methods in its own *method namespace* (a very simple ontology in form of a dictionary). This binding of message and method is done at runtime. The idea behind the message passing system is quite similar to the way that abstract actions and events are used in Mars: the message only specifies the *logical* function that should be computed by the object whereas the object itself decides for the best way to *physically* achieve the desired result.

A further example with a similar level of abstraction can be found with the Unified Modeling Language (UML) [OMG]. One of the purposes of UML is to describe programmes or workflows without giving details about the implementation in a programming language. Static concepts like objects, attributes and relationships can be defined, but also dynamic aspects can be modelled using UML state charts, object collaboration or message sequence diagrams.

MARS, however, gives not only the necessary model for specifying the reactive behaviour of a domain but also a run-time environment for the execution of these rules. Actually there are various approaches to the same aim for UML [MEMS, RFBLO01, MB02], being called *virtual machine for UML* or *executable UML*. All of them try to achieve the direct execution of UML models without an intermediate compilation step. Therefore a proper semantics is needed for UML models, which comprises UML class diagrams for the specifications of concepts, UML statechart diagrams for the specification of behaviour, and an action language for the specification of actions. Hereby, programme development can be reduced to conceptual design, all implementation details are left to the underlying mapping mechanism.

Another form of event-based application-independent infrastructures can be found with Event-Notification-Services (ENS) and Publish-Subscribe systems [CRW01, HV02]. Generators of events publish event notifications to the infrastructure whereas consumers of events subscribe with the infrastructure to receive relevant notifications.

These approaches try to tackle the technical problems of event processing which is also a matter of interest in MARS. In MARS there exist two classes of infrastructural components (AEM and CED, see Section 4.2) that deal with event detection and event processing.

But this situation is transcended in MARS insofar as the conceptual model is not limited to (syntactic) event *handling* but is a semantic treatment of the reactive behaviour of an application domain. Such abstract concepts are seldomly found in research on ENS. In [JH04], for example, the authors propose a meta-service for event notification. Instead of subscribing at many different ENSs which each use different event specification languages, a meta-language for event specification is suggested.

The meta-service employs transformation rules which translate the abstract event specification into the languages that are used by the ENSs. Hereby, the events can be specified completely independent from the application domain services that will eventually process the subscriptions. This can be compared to the use of ACA rules in MARS where abstract action specifications are transformed into knowledge base updates of the application. The mapping, though, is realised by the application domain node and not by a meta-service.

Knowledge Base Updates

RDF is expected to be the standard data model for the Semantic Web. The RDF model is more than a simple relational structure. The built-in vocabulary of RDF Schema adds transitivity of predicates and inheritance axioms, thus making this data model resemble a fragment of binary first order logic (i.e. only binary predicates). Due to the increasing popularity of RDF there has been a lot of interest in the questions of RDF data management and processing. There are many resemblances between RDF and graph databases. That is why the survey on graph databases in [AG08] is also a valuable contribution to the research on RDF databases.

There is quite a number of works that have addressed the important problem of updates to RDF data. However, some of these efforts merely propose update languages for RDF data [ACK+01, MSCK05] which completely neglect the semantic problems resulting from the presence of blank nodes and built-in semantics of RDFS.

The semantic problems in RDF are similar to the problems that have been examined in research about belief revision. One of the classical examples can be given in RDF:

```
:a rdfs:subClassOf :b,
:b rdfs:subClassOf :c,
:A rdf:type :a.
```

If the knowledge ($: A$ rdf:type $: c$) should be deleted, there is no unambiguous way to achieve this: the removal of any one of the three statements could lead to the desired result.

The theoretical background for updates to RDF data in this respect was prepared by a large body of research on updates in knowledge bases. A standard approach in knowledge bases is to ensure that, after the deletion of a statement t from a RDF graph G, t should not be derivable from G, and that the deletion should be minimal. The concept of minimal changes demands for a *measure of closeness*.

A number of different proposals to that end are classified in [EG92]. One approach rests upon the distinction between *updates* and *revisions* as different kinds of modifications to a knowledge base [KM91]. An update brings a knowledge base up to date when the world, as decribed by it, changes (for example the action of some agent), whereas a revision incorporates new, more precise, or more reliable information obtained about a static world. The choice of which of these types of *change* is more suitable depends on the application at hand. They define a model-theoretic point of view for an update: for each model M of the theory to be changed, a set of models closest to M has to be found which incorporate the changes. This can be compared to the notion of minimal-changes semantics for intensional updates in Swan (see Section 6.4.2).

With regard to belief *revision* the AGM postulates (named after the authors Alchourron, Gärdenfors, and Makinson) [AGM85] have gained a prominent position. They propose three main operations of change for belief sets: *contraction* (retraction of a belief), *expansion* (expanding a belief set without a guarantee to consistency), and *revision* (expansion with guaranteed consistency). A *belief base* is considered to contain the basic beliefs from which the *belief set* results by deduction of additional beliefs. The AGM postulates cannot be applied without drastic modifications to *updates* [KM91].

In [GHV06] the claim is made that *revisions* are trivial in the context of the RDF data model. On the other hand, *updates* pose new problems when treating an RDF database as a knowledge base. For example, a solution to the deletion of $(:A \text{ rdf:type } :c)$ is

$$\begin{array}{l}(:a \text{ rdfs:subClassOf } :b \wedge \neg :b \text{ rdfs:subClassOf } :c) \quad \vee \\ (\neg :a \text{ rdfs:subClassOf } :b \wedge :b \text{ rdfs:subClassOf } :c)\end{array}$$

This cannot be expressed in RDF as neither negation nor disjunction are available. This is the motivation for the authors to propose an approximation of the Katsuno-Mendelzon postulates [KM91] tailored to RDF plus an algorithm for calculating the update and erase operations.

Although it can be assumed that knowledge bases for the Semantic Web will use RDF as the data model, it is arguable whether the semantics will depend on RDF Schema. Rather, it is commonly anticipated that the reasoning layer in the Semantic Web will be based on OWL and hereby on Description Logics. It is a bit surprising in this respect that research on updates to Description Logics emerged only in recent years.

The problem of updates is first brought out for Description Logics in [RSS02] and [LLMW06]. The first relates the semantic problems of updates in DLs to view management in relational databases. The latter proposes a formal semantics for updates and shows for *unrestricted* updates that a Description Logic $\mathcal{L}$ is not closed in the sense that the set of models corresponding to an update applied to a knowledge base in a DL $\mathcal{L}$ may not be expressible by ABoxes in $\mathcal{L}$. In [LPR07] these findings are extended such that the authors could show that DLs are also not closed with respect to erasure operations also following the Katsuno-Mendelzon approach. Moreover, this work provides for a best-approximation for update and erasure operations and gives a polynomial algorithm for computing these best-approximations for a Description Logic $DL - Lite_{\mathcal{F}}$.

In [FPA05] it was shown that OWL DL is non-AGM-compliant for the *contraction* and the *revision* operation. In order to to find a computationally tractable solution, only *belief base* revisions are considered in [HWKP06] instead of *belief set* revision. Because of the limited set of considered formulae, this approach is called *semi-revision* which is compliant to SHOIN (which corresponds to OWL DL).

The problem of updates to an OWL-DL knowledge base has been addressed in Swan with the introduction of the retract and assert operations (see Section 6.3). They provide an inference-sensitive update mechanism for an OWL knowledge base. Both operations are limited to updates on the ABox, which is comparable to the *semi-revision* approach.

This handling of updates belongs to the category of belief revisions although the discrimination between *updates* and *revisions* is a bit ambiguous. Whether a deletion is due to changes in the world or a revision of previous believes is not always clear. Nevertheless, the retract and assert operations offer a way for intensional updates. The completion of intensional updates is possible by the use of triggers which implement the intended behaviour.

Active Knowledge Bases

A number of systems exist that are specialised in the storage of RDF data. Amongst many others there can be named Jena [Jen], Sesame [Ses], Redland [Red], Brahms [Bra] or RDFDB [RDF]. All of these are capable to store, update, and query RDF data. RDF-Schema support is offered by Sesame and Brahms, of which the latter keeps all base facts and deductions in system memory in contrast to Sesame where both base facts and deductions are stored in a relational database. Jena comes with a built-in reasoning engine which can be configured to support a variety of schema languages like RDF-Schema, DAML+OIL or OWL. Moreover other DL reasoners (like Pellet) can be used by Jena.

With one exception, none of the above-mentioned systems allows for the specification of reactive behaviour. Only Jena has a very basic mechanism for the *detection* of change (event detection) which can be used to implement a trigger mechanism. The lack of reactive behaviour in nearly all of these RDF storage systems is surprising as RDF is expected to build the foundation of the Semantic Web where data sources and applications are highly distributed and subject to continuous change. Consequently, dynamic contents make it necessary that the changes become distributed, for example with the help of ECA rules.

Stemming from the area of active database research [Pat99] there have been numerous works in the recent years that incorporate the Event-Condition-Action paradigm in dynamic, highly-distributed data-driven application domains. In the context of this work, especially the question regarding the detection of *simple* events in knowledge bases is of interest. Although there is a large body of results about ECA rules from several research areas (e.g. Publish-Subscribe Systems, Sensor Networks or Event-Notification Systems) the majority of them are concerned with algorithmical aspects of *complex* event processing [Hin03, TSG$^+$06, CRW01] rather than how *simple* events are detected.

Apart from the many contributions by the (active) database community there is little research on active RDF knowledge bases. In [PPW04] the authors propose RDFTL as trigger definition language for RDF data which follows the ECA paradigm. RDFTL is designed for an *active* wrapper around a *passive* RDF repository (RDFSuite [ACK$^+$01]). The design of RDFTL follows both in syntax and rule execution semantics the standards of SQL3. Although RDFSuite also stores RDF-Schema metadata, the semantic problem of knowledge base updates remains unmentioned. The purpose of RDFTL is to enable reactive behaviour in a distributed environment in order to exchange information about the evolution of metadata. One drawback, however, is the *local* nature of the ECA rules as specified by RDFTL. For this reason it is impossible to detect distributed events, only local events can be detected. The notion of distributed execution of ECA rules is only true with regard to the action part, which can be executed at different peers.

In this respect the Swan architecture benefits greatly from the tight integration into the Mars framework: events can be detected either locally by means of knowledge base triggers (local ECA rules) or at arbitrary (also distributed) places by the use of higher level ECA rules, which are executed by the Mars ECA engine.

To the best of the authors knowledge no other implementation exists that is offering active rules with support for intentional updates in RDF knowledge bases. The work at hand offers a rule-based trigger mechanism with a syntax comparable to SQL triggers. With regard to the distinction between explicit and implicit updates also the trigger mechanism is twofold: one class of triggers reacts on explicit updates before reasoning on the data is performed. The other class of triggers reacts on changes to the model including the deductions.

The uniqueness of the trigger mechanism in SWAN makes it a novel contribution to knowledge base research in the Semantic Web.

Hybrid Reasoning

A short and comprehensive introduction to hybrid reasoning can be found in Section 3.7. The integration of different layers of inference has become a central issue for the architecture of the Semantic Web. This is reflected in the large number of investigations that deal with the integration of rules with ontologies. In the following some of those approaches in the field of knowledge bases and the Semantic Web are presented. A survey can also be found in [ADG$^+$05].

There have been several approaches for the integration of rule languages with conceptual languages, e.g. $\mathcal{AL}$-Log [DLNS91], and CARIN[LR96], where hybrid reasoning is realised by putting DL terms in logical rules. These systems have in common that the DL part of the knowledge base shares its individual constants with the Datalog programme (which adds the logic programming capabilities). But there is no hybridity in reasoning about the predicates (or other parts of the TBox). Consequently, there are strong limits to these reasoning systems, like in CARIN, where the occurence of DL terms in the rules is limited, (especially in the head of the rule), and therefore no new knowledge can be added to the DL part of the knowledge base.

Another hybrid reasoning system has been proposed with DLP [GHVD03]. One of its strong points is (compared to the aforementioned approaches) that it overcomes the separation of components and allows for bidirectional translation of premises and inferences. The DL part is limited to DHL (*Description Horn Logic*), an intersection of a decidable DL with Horn logic programmes. The combination of logic programmes and DHL works in both directions: rules can be layered on top of the knowledge base and have access to both individual constants and predicates or the other way round, the knowledge base is supplemented by access to the rules. The drawback is that DHL uses a severely restricted DL that has no cardinalities and no existential quantification. But it is still more expressive than RDFS, and it is tractable.

With regard to the ontology part of the hybrid rules systems, the extent of expressiveness that is supported is very different. Some of the systems support RDF and partly RDF-Schema. These are mostly approaches where the RDF data model and some of the RDF-Schema axioms are emulated by a rule engine. Examples are TRIPLE [TRI] where an XSB Prolog engine is used, or the Semantic Web Library of SWI-Prolog [SWI] which uses SWI-Prolog. The latter supports full RDF and RDF-Schema whereas TRIPLE has only limited support for RDF-Schema axioms. Both can hardly be compared to more sophisticated hybrid reasoning tools where two fully-fledged deductive systems are combined.

Jena [Jen] is a Semantic Web framework for managing ontologies which offers a highly modular access to its reasoning capabilities. Either external reasoning engines or one of several built-in rule engines can be chosen. While most of the built-in engines support a different range of axioms from RDF-Schema or OWL there is also a so-called generic rule reasoner, which can be used in a backward- or forward-chaining manner. The built-in engines for RDF-Schema and OWL make use of the same engine with a predefined set of rules. In case that the rule engine is used for both OWL and custom rules, the different engines become cascaded which means that only one of the engines is able to see the results of the other. The generic rules reasoner offers a number of built-in predicates. Object creation (BNodes) is only supported to a limited degree. Forward rules have no fixpoint check and hence sometimes do not terminate on data containing cycles. In backward rules there is no possibility for the removal of previously found derivations. The overall performance, especially in presence of cylic data and transitive rules is rather poor. For example, the calculation of composite railway connections as it was shown for F-Logic rules in Example 9.18 is not possible using the generic rules reasoner in Jena.

Another project aiming at the extension of OWL with rules is called the Semantic Web Rule Language (SWRL) [HPSB+04]. The central idea in SWRL is to overcome restrictions of OWL by adding rules as a new kind of axioms within OWL. The integration is hereby easy, syntactically, and also the semantics of SWRL are a straightforward extension to the semantics of OWL-DL. SWRL offers a rule mechanism that is comparable in expressiveness to basic Horn clauses. Rule head and body may consist of a conjunction of the atoms $C(x)$, $P(x, y)$, $sameAs(x, y)$ and $differentFrom(x, y)$. Hereby, for example, the uncle-relationship can be modelled. The simplicity of SWRL, which makes it so easy to be integrated into OWL, has also its downside. Many features leave to be desired, amongst them are disjunction, negation of atoms, non-monotonic features as negation as failure or defaults, and furthermore, there are no built-in predicates and therefore no arithmetics.

In HD-rules [DHM07] a XSD Prolog engine is coupled with a DL reasoner. The integration of the two deductive systems is realised by prolog-like rules that incorporate queries to the DL part by special predicates in the rule body. These rules are neither pure prolog rules nor OWL expressions. The hybrid rules are compiled into Prolog programmes and executed by a run-time system which interfaces the DL reasoner via the DIG interface. One of the advantages of this architecture is the exchangeability of reasoning engines: any Prolog engine with a standard interface to Java can be used, the same is true for any DIG-compliant DL reasoning engine. One major drawback, however, is that the expressiveness of an OWL ontology is severely limited by the DIG interface which supports, in its current specification, only a small subset of OWL DL.

The combination of rules and ontologies in SWAN is a typical *hybrid* solution in the sense of the definition that was given in Section 3.7. It is a combination of F-Logic and OWL DL, using FLORID for the rule part and PELLET for the ontology part. Since F-Logic is an extension of FOL, some subsets of DLs can be expressed in F-Logic. This aspect has already been studied in [Bal95] and [Bor96]. Hence, ontologies can be translated into F-Logic programmes and vice versa, hereby augmenting the DL part of the hybrid reasoning system with F-Logic features like default inheritance, basic arithmetics and object creation. F-Logic and OWL reasoning systems are kept separated in SWAN, each of them delivering further deductions in a process of an alternating fixpoint computation.

Compared to the aforementioned hybrid reasoning systems, the SWAN architecture allows for a more flexible usage of the rule component. As it was already described in Section 12.2, the rule part becomes activated by user-defined triggers instead of a continuous application of hybrid reasoning. As long as no critical updates (with regard to *static facts*) have been detected there is no need for the additional F-Logic reasoning capabilities, consequently leading to a faster execution of knowledge base updates. If, however, the discrimination between *static facts* and *dynamic facts* is not possible in an ontology, it is still possible to *continuously* apply hybrid reasoning upon every update (or not at all).

The discussion about the Swan architecture is now concluded with an outlook to further work.

12.6 Further Work

Detection Of State Change

Trigger activation depends on the detection of state changes in the knowledge base. The trigger mechanism of the Swan architecture (see Section 7) is based on the calculation of the difference of two specialised theories (called an *Ont-Model* in Jena). On every update a copy of the specialised theory is created before the update. After the execution the difference between those theories is calculated. The set of differences reflects the changes caused by the update. Again, triggers may react upon these changes which might cause further updates. Each update makes it necessary to repeat the calculation of the difference of two specialised theories.

The copying of a specialised theory is merely a copy of the ABox and TBox contents. Derivations that have been made for the original specialised theory are not copied. It is the computation of the difference of two specialised theories which is an expensive operation. Each specialised theory has to be grounded (see Section 7.3), that is all possible conclusions from the theory have to be drawn resulting in a simple graph structure consisting of positive ground atomic facts. This grounding operation can be, depending on the size of the knowledge base and the complexity of the ontology, quite expensive.

Although usually only a small fragment of the knowledge base is changed the grounding of the whole theory is necessary in each of the copies.

Incremental reasoning is a feature for reasoning engines that tries to re-use previous calculations in later steps of the reasoning process. Considering that in DL an addition can never cause the retraction of some previous entailment, many of the previous calculations can be used again. In the same way an axiom that becomes deleted cannot cause a subsumption that did not hold before to become true.

It is easy to see how incremental reasoning could contribute to a faster update mechanism in Swan. Most of the computational time during the update is consumed by the reasoning engine. The preservation of previous deductions in the copy of a theory would have a considerable effect on the performance of the knowledge base updates.

There is ongoing work on incremental reasoning for the Pellet reasoning engine [PHWS06]. As this is only an internal optimisation in Pellet it is not clear whether and how this implementation could be used for the aforementionend purposes in Swan. Future investigations will have to show how this can be used to improve the performance of updates in Swan.

Translations Between Open and Closed Worlds

In Section 9 a description is given for those features in DLs that cannot be expressed in F-Logic. One important characteristic of some DLs in this respect is existential quantification which cannot be expressed in F-Logic. Therefore, rule reasoning cannot be used to draw conclusions about knowledge that is known to exist but not yet available. This is a collison of *open* and *closed world assumptions* which make it impossible to translate this aspect between the different formalisms.

How hybrid reasoning is affected by the differences between the reasoning formalisms is depicted in Section 9.4.5. Moreover, a short description of the temporal expansion of existentially quantified expressions in SWAN is given there. This expansion creates temporary objects for all relationships that are asserted by the OWL ontology but that are not known yet. These temporary objects are only given to FLORID but not to the OWL knowledge base.

Although it is hereby possible for F-Logic programmes to draw conclusions about those objects that are asserted by the ontology but do not exist as statements, this solution introduces new problems in connection with infinite structures. Infinite structures occur when the TBox contains cyclic concept definitions. Consider the following concept definition:

$$\mathsf{Child} \equiv \exists \mathsf{hasParent}.\mathsf{Child}$$

The completion procedure needs to be blocked because the cyclic definition of the concept Child would not allow the procedure to terminate. In the hybrid reasoning engine of SWAN the completion procedure becomes blocked in case that an recursion of object completion is detected. This is, however, only a partial detection as only direct recursions are identified (like the one that is caused by the definition of Child). Compare this definition to the following example:

$$A \equiv \exists a.C$$
$$C \equiv \exists c.A$$

Here, A and C are also cyclic concept definitions, but an instance of A will be completed by a relationship a to a temporary object being an instance of C. In turn, instances of C have to be completed by the relationship c to temporary objects being instances of A (and so on). In order to detect the recursive pattern in the resulting completion tree it is necessary to apply more sophisticated blocking algorithms. The present solution in SWAN allows only for the detection of very simple patterns where the concept on the left-hand side of a concept definition is the same (and only) concept as on the right hand side. Although all cyclic definitions in the TBox are best avoided a more mature blocking procedure is feasible. Related work on blocking mechanisms can be found in various publications, e.g. about conjunctive queries for DL [HST00, HT00, OCE06], epistemic queries for DL [CLLR06] or hybrid reasoning with rules and DL [MSS05].

ACA Meta-Service

In Mars, the handling of abstract actions is left to the application domain nodes. On the rule level, only abstract action definitions are used. On the one hand, this ensures the modularity of the architecture, and service components can be integrated easily. On the other hand, domain services have to provide an infrastructure for the mapping of the abstract actions to knowledge base updates. While the Swan architecture offers an ACA rule mapping (see Chapter 8) this will not be the case for ordinary Web Services.

Although it is possible to integrate arbitrary Web Services by the use of opaque rule component definitions (see Section 4.3.2) this is not a satisfactory solution. Opaque components limit the generality of ECA rules, therefore they are best avoided if possible.

In [JH04] a meta-service for event notification is proposed. Although event-notification is different from action forwarding there are similarities. The problem in both cases is how to integrate heterogeneous services. A meta-service as proposed is expected to accomplish mappings from abstract specifications to application-specific formalisms. This would be an additional service component in Mars. Any service could be integrated given that an appropriate mapping rule is registered at the ACA meta-service.

The advantage of this approach is that the meta-service provides for a uniform mapping infrastructure (which otherwise would have to be implemented at every service) reducing the task of service integration to the development of rule mappings. There would be no need for implementing a mapping infrastructure for every new service. Also, in the majority of cases it is not possible to modify an external service. Here, the integration would greatly benefit from a meta-service infrastructure.

Chapter 13

Conclusions

In this work, the architecture of an application node for the Semantic Web was presentend. The architecture realises an active OWL knowledge base, which exhibits a number of distinct features novel to this kind of knowledge management applications.

The knowledge base uses RDF as a data model and OWL DL for the description of the domain ontology. The concepts and relationships that are used in the domain are defined in this ontology. It was shown how reasoning allows to derive new information from given facts and also, what limitations have to be kept in mind with respect to OWL reasoning. These considerations led to the conclusion that a supplementary inference mechanism is desirable. For this purpose, the SWAN architecture was extended to a hybrid reasoning engine. In addition to the OWL reasoner, which is integrated into the knowledge base, an F-Logic reasoner is utilised for supplementary deductions. The problems and benefits of this approach were analysed in detail. It was demonstrated that this implementation of hybrid reasoning is able to overcome some of the restrictions that are intrinsic to OWL.

Furthermore, the knowledge base offers sophisticated update operations. The presence of intensional knowledge requires the distinction between implicit and explicit updates. The differences were described and a formal characterisation of knowledge base updates was given. In short, the update operations insert, delete, and modify can be used for the specification of explicit updates, whereas assert and retract can be used for the specification of intensional updates.

It was shown how the trigger mechanism in SWAN complements the feature of intensional updates. The specification of triggers realises the idea of an active OWL knowledge base which is a novel contribution to knowledge management in the Semantic Web. Knowledge base triggers can be used not only for the completion of intensional updates, but also for maintaining the integrity of the knowledge base.

Another feature of the architecure of this domain node is the ease of integration into the event-driven environment of MARS. The SWAN architecture enables the execution of abstract actions that are given in terms of the domain ontology instead of explicit update commands to the knowledge base. The execution of abstract actions relies on the definition of translation rules, called ACA rules. These rules, together with the knowledge base triggers, define the behaviour of the domain node. There is a logical characterisation of this behaviour, which allows for reasoning about the behaviour of the knowledge base.

A prototype of this architecture was implemented. In this work it was demonstrated by an example scenario how the concepts of SWAN and MARS can be used for the modelling of an application domain.

The SWAN architecture integrates all these features: Knowledge base triggers for an OWL knowledge base, hybrid reasoning combining F-Logic and OWL, and the execution of ACA rules for the translation of abstract actions. This is a unique combination of components which makes SWAN a valuable contribution to the Semantic Web.

List of Figures

Bibliography

[ACK$^+$01] So Alexaki, Vassilis Christophides, Greg Karvounarakis, Dimitris Plexousakis, and Karsten Tolle. The ICS-FORTH RDFSuite: Managing Voluminous RDF Description Bases. pages 1–13, 2001.

[ADG$^+$05] Grigoris Antoniou, Carlos V. Damásio, Benjamin Grosof, Ian Horrocks, Michael Kifer, Jan Maluszynski, and Peter F. Patel-Schneider. Combining Rules and Ontologies. A survey., 2005.

[AF94] J.F. Allen and G. Ferguson. Actions and Events in Interval Temporal Logic. Technical Report 521, University of Rochester, 1994.

[AG08] Renzo Angles and Claudio Gutiérrez. Survey of Graph Database Models. *ACM Comput. Surv.*, 40(1), 2008.

[AGM85] Carlos E. Alchourrón, Peter Gärdenfors, and David Makinson. On the Logic of Theory Change: Partial Meet Contraction and Revision Functions. *J. Symb. Log.*, 50(2):510–530, 1985.

[Bal95] Mira Balaban. The F-Logic Approach for Description Languages. *Annals of Mathematics and Artificial Intelligence*, 15:15–19, 1995.

[BCM$^+$03] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider, editors. *The Description Logic Handbook*. Cambridge University Press, 2003.

[BFK$^+$07] Erik Behrends, Oliver Fritzen, Tobias Knabke, Wolfgang May, and Franz Schenk. Rule-Based Active Domain Brokering for the Semantic Web. Number 4524, pages 250–268. LNCS, 2007.

[BFMS06] Erik Behrends, Oliver Fritzen, Wolfgang May, and Franz Schenk. Combining ECA Rules with Process Algebras for the Semantic Web. In *Rule Markup Languages (RuleML)*, pages 29–38. IEEE, 2006.

[BFMS08] Erik Behrends, Oliver Fritzen, Wolfgang May, and Franz Schenk. Embedding Event Algebras and Process for ECA Rules for the Semantic Web. *Fundamenta Informaticae*, (82):237–263, 2008.

[BH95] Franz Baader and Bernhard Hollunder. Embedding defaults into terminological knowledge representation formalisms. *Journal of Automated Reasoning*, 14:149–180, 1995.

[BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American Magazine*, 2001.

[Bor96] Alex Borgida. On the Relative Expressiveness of Description Logics and Predicate Logics. *Artificial Intelligence*, 82:353–367, 1996.

[Bra] Brahms: Main-memory storage for RDF/S. http://lsdis.cs.uga.edu/projects/semdis/brahms/.

[BS85] Ronald J. Brachman and James G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171–216, 1985.

[CKW93] Weidong Chen, Michael Kifer, and David S. Warren. HiLog: A Foundation for Higher-Order Logic Programming. *Journal of Logic Programming*, 15:187–230, 1993.

[CLLR06] Diego Calvanese, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Epistemic First-Order Queries over Description Logic Knowledge Bases. In *Description Logics*, 2006.

[CM94] Sharma Chakravarthy and D. Mishra. Snoop: An Expressive Event Specification Language for Active Databases. *Data Knowledge Engineering*, 14(1):1–26, 1994.

[Cod70] E.F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970.

[CRW01] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19:332–383, 2001.

[CS98] Jan Chomicki and Gunter Saake, editors. *Logics for Databases and Information Systems*. Kluwer, 1998.

[DFF$^+$99] Alin Deutsch, Mary F. Fernández, Daniela Florescu, Alon Y. Levy, and Dan Suciu. A Query Language for XML. *Computer Networks*, 31(11-16):1155–1169, 1999.

[DHM07] Wlodzimierz Drabent, Jakob Henriksson, and Jan Maluszynski. HD-rules: a hybrid system interfacing Prolog with DL-reasoners. In *Proceedings of 2nd International Workshop on Applications of Logic Programming to the Web, Semantic Web and Semantic Web Services, Porto, Portugal (13th September 2007)*, volume 287, pages 76–90, 2007.

[DIG] Description Logic Implementation Group (DIG). http://dl.kr.org/dig/.

[DLNS91] Francesco M. Donini, Maurizio Lenzerini, Daniele Nardi, and Andrea Schaerf. A hybrid system with datalog and concept languages. In *Trends in Artificial Intelligence; AI*IA'91*, number 549 in LNCS, pages 88–97. Springer, 1991.

[DLNS98] Francesco M. Donini, Maurizio Lenzerini, Daniele Nardi, and Andrea Schaerf. AL-log: Integrating Datalog and Description Logics. *Journal of Intelligent Information Systems*, 10(3):227–252, 1998.

[DOS03] M.C. Daconta, L.J. Obrst, and K.T. Smith. *The Semantic Web: A Guide to the Future of XML, Web Services, and Knowledge Management.* John Wiley and Sons., 2003.

[EG92] Thomas Eiter and Georg Gottlob. On the Complexity of Propositional Knowledge Base Revision, Updates, and Counterfactuals. *Artificial Intelligence*, 57:227–270, 1992.

[EIST06] Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. Effective Integration of Declarative Rules with External Evaluations for Semantic Web Reasoning. In *Proceedings of 3rd European Semantic Web Conference, Budva, Montenegro (11th–14th June 2006)*, volume 4011 of *LNCS*, pages 273–287, 2006.

[FaC] FaCT++: OWL DL Reasoner. http://owl.man.ac.uk/factplusplus/.

[FAD$^+$99] Dieter Fensel, Jürgen Angele, Stefan Decker, Michael Erdmann, Hans-Peter Schnurr, Steffen Staab, Rudi Studer, and Andreas Witt. On2broker: Semantic-Based Access to Information Sources at the WWW. In *Proceedings of the World Conference on the WWW and Internet (WebNet 99)*, 1999.

[FLB$^+$06] Tim Furche, Benedikt Linse, François Bry, Dimitris Plexousakis, and Georg Gottlob. RDF Querying: Language Constructs and Evaluation Methods Compared. In *Proceedings of Summer School Reasoning Web 2006, Lisbon, Portugal (4th–8th September 2006)*, volume 4126 of *LNCS*, pages 1–52. REWERSE, 2006.

[FLO98] Florid homepage.
http://www.informatik.uni-freiburg.de/~dbis/florid/,
1998.

[FPA05] Giorgos Flouris, Dimitris Plexousakis, and Grigoris Antoniou. On Applying the AGM Theory to DLs and OWL. In *In Proc. of Int. Semantic Web Conf*, pages 216–231, 2005.

[GHV06] Claudio Gutierrez, Carlos Hurtado, and Alejandro Vaisman. The Meaning of Erasing in RDF under the Katsuno-Mendelzon Approach. In *Workshop on the Web and Databases*, 2006.

[GHVD03] Benjamin Grosof, Ian Horrocks, Raphael Volz, and Stefan Decker. Description Logic Programs: Combining Logic Programs with Description Logic. In *12th. WWW Conference*, pages 48–57, 2003.

[GJS92] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Composite Event Specification in Active Databases: Model & Implementation. In *Proceedings of the 18th International Conference on Very Large Databases*, 1992.

[GL93] M. Gelfond and V. Lifschitz. Representing Action and Change by Logic Programs. *Journal of Logic Programming*, 1993.

[GR89] Adele Goldberg and David Robson, editors. *Smalltalk: The Language.* Addison-Wesley, 1989.

[Gru93] T.R. Gruber. A Translation Approach to Portable Ontologies. *Knowledge Acquisition*, 5(2):199–220, 1993.

[Hay79] P.J. Hayes. The Naive Physics Manifesto. 1979.

[HFB$^+$00] I. Horrocks, D. Fensel, J. Broekstra, S. Decker, M. Erdmann, C. Goble, F. van Harmelen, M. Klein, S. Staab, R. Studer, and E.Motta. The ontology inference layer OIL. Technical report, On-To-Knowledge, 2000. Available at http://www.ontoknowledge.org/oil.

[Hin03] Annika Hinze. Efficient Filtering of Composite Events. In *Proceedings of the 20th British National Database Conference*, 2003.

[HLTB04] I. Horrocks, L. Li, D. Turi, and S. Bechhofer. The Instance Store: Description Logic Reasoning with Large Numbers of Individuals, 2004.

[HM01] Volker Haarslev and Ralf Möller. Racer system description. In *Int. Joint Conf. on Automated Reasoning (IJCAR)*, number 2083 in LNAI, pages 701–705. Springer, 2001. http://www.racer-systems.com.

[HO01] Volker Haarslev and Ralf M Oller. High Performance Reasoning With Very Large Knowledge Bases: A Practical Case Study. pages 161–168, 2001.

[Hor] John F. Horty. Some Direct Theories of Nonmonotonic Inheritance. In *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 3.

[Hor98] Ian Horrocks. Using an expressive description logic: Fact or fiction? In *Principles of Knowledge Representation and Reasoning (KR)*, pages 636–647, 1998. http://www.ca.man.ac.uk/~horrocks/FaCT.

[HPS04] Ian Horrocks and Peter F. Patel-Schneider. Reducing OWL entailment to description logic satisfiability. *Journal of Web Semantics*, 1(4):345–357, 2004.

[HPSB$^+$04] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. SWRL: Semantic Web Rules Language Combining OWL and RuleML. http://www.w3.org/Submission/SWRL/, 2004.

[HPSvH04] Ian Horrocks, Peter F. Patel-Schneider, and Frank van Harmelen. From SHIQ and RDF to OWL: The making of a web ontology language. *Journal of Web Semantics*, 1(1):7–26, 2004.

[HS07] Ian Horrocks and Ulrike Sattler. A tableau decision procedure for SHOIQ(D). In *J. of Automated Reasoning*, 2007.

[HST00] Ian Horrocks, Ulrike Sattler, and Stephan Tobies. Reasoning with Individuals for the Description Logic SHIQ. In *Intl. Conf. on Automated Deduction (CADE)*, number 1831 in LNCS, pages 482–496, 2000.

[HT00] Ian Horrocks and Sergio Tessaris. A Conjunctive Query Language for Description Logic Aboxes. In *Nat. Conf. on Artificial Intelligence (AAAI)*, pages 399–404, 2000.

[HV02] Annika Hinze and Agnès Voisard. A Parameterized Algebra for Event Notification Services. In *TIME '02: Proceedings of the Ninth International Symposium on Temporal Representation and Reasoning (TIME'02)*, page 61, Washington, DC, USA, 2002. IEEE Computer Society.

[HWKP06] Christian Halaschek-Wiener, Yarden Katz, and Bijan Parsia. Belief Base Revision For Expressive Description Logics. In *In Proc. of Workshop on OWL Experiences and Directions*, 2006.

[Jen] Jena: A Java Framework for Semantic Web Applications. http://jena.sourceforge.net.

[JH04] Doris Jung and Annika Hinze. A Meta-service for Event Notification. In *Proceedings of Summer School Reasoning Web 2006, Lisbon, Portugal (4th–8th September 2006)*, volume 3290 of *LNCS*, pages 283–300, 2004.

[Kat07] Heiko Kattenstroth. Knowledge Management with OWL and F-Logic: A Combination of Description Logic Reasoning with F-Logic Rules. Diploma Thesis, Univ. Göttingen, 2007.

[Kem01] C. Kemke. About the Ontology of Actions. Technical Report MCCS-01-328, Computing Research Laboratory, New Mexico State University, 2001.

[KL89] Michael Kifer and Georg Lausen. F-Logic: A higher-order language for reasoning about objects, inheritance and scheme. In *SIGMOD*, pages 134–146, 1989.

[KLW95] Michael Kifer, Georg Lausen, and James Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42(4):741–843, 1995.

[KM91] Hirofumi Katsuno and Alberto O. Mendelzon. On the Difference Between Updating a Knowledge Base and Revising it. pages 387–394. Morgan Kaufmann, 1991.

[LLMW06] Hongkai Liu, Carsten Lutz, Maja Milicic, and Frank Wolter. Updating Description Logic ABoxes. In *KR*, pages 46–56, 2006.

[LPR07] Maurizio Lenzerini, Antonella Poggi, and Riccardo Rosati. On the Approximation of Instance Level Update and Erasure in Description Logics. In *In Proc. of AAAI 2007*, 2007.

[LR96] Alon Y. Levy and Marie-Christine Rousset. Carin: A representation language combining horn rules and description logics. In *Europ. Conf. on Artificial Intelligence (ECAI)*, pages 328–334, 1996.

[MA92] Z. Manna and A.Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer Verlag, 1992.

[MAB04] Wolfgang May, José Júlio Alferes, and François Bry. Towards Generic Query, Update, and Event Languages for the Semantic Web, 2004.

[Mak94] David Makinson. General Patterns in Nonmonotonic Reasoning. pages 35–110, 1994.

[MB02] Steven J. Mellor and Marc J. Balcer. *Executable UML: Model-Driven Architecture*. Addison-Wesley, 2002.

[MB08] Georgios Meditskos and Nick Bassiliades. Combining a DL Reasoner and a Rule Engine for Improving Entailment-based OWL Reasoning. In *ISWC*. ACM, to be published 2008.

[McC80] John McCarthy. Circumscription—A Form of Non-Monotonic Reasoning. *Artificial Intelligence*, 13:27–39, 1980.

[MEMS] Manikya Madhu, Babu Eadara, Adam Malinowski, and Junichi Suzuki. Matilda: A Distributed UML Virtual Machine for Model-Driven Software Development.

[MF02] Deborah L. McGuiness and Richard Fikes. DAML+OIL: An Ontology Language for the Semantic Web. *IEEE Intelligent Systems*, 17(5):72–80, 2002.

[Mil83] R. Milner. Calculi for Synchrony and Asynchrony. *Theoretical Computer Science*, pages 267–310, 1983.

[MK01] Wolfgang May and Paul-Th. Kandzia. Nonmonotonic Inheritance in Object-Oriented Deductive Database Languages. *Journal of Logic and Computation*, 11(4), July 2001.

[MS06] Boris Motik and Ulrike Sattler. A Comparison of Reasoning Techniques for Querying Large Description Logic ABoxes. In Miki Hermann and Andrei Voronkov, editors, *Proc. of the 13th Int. Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR 2006)*, volume 4246 of *LNCS*, pages 227–241, Phnom Penh, Cambodia, November 13–17 2006. Springer.

[MSCK05] M. Magiridou, S. Sahtouris, V. Christophides, and M. Koubarakis. RUL: A Declarative Update Language for RDF. In *In Procs. 4th Intern. Conf. on the Semantic Web (ISWC-2005*, pages 506–521, 2005.

[MSK07] Wolfgang May, Franz Schenk, and Heiko Kattenstroth. Combining OWL with F-Logic Rules and Defaults. In *ALPSWS*, number 287, pages 60–75, 2007.

[MSS05] Boris Motik, Ulrike Sattler, and Rudi Studer. Query Answering for OWL-DL with Rules. *J. Web Sem.*, 3(1):41–60, 2005.

[MST93] V. Wiktor Marek, Grigori F. Schwarz, and Miroslaw Truszczynski. Modal Nonmonotonic Logics: Ranges, Characterization, Computation. *J. ACM*, 40(4):961–988, 1993.

[MSvL06] Wolfgang May, Franz Schenk, and Elke von Lienen. Extending an OWL Web Node with Reactive Behavior. In *PPSWR*, pages 134–148, 2006.

[OCE06] Magdalena Ortiz, Diego Calvanese, and Thomas Eiter. Characterizing Data Complexity for Conjunctive Query Answering in Expressive Description Logics. In *Nat. Conf. on Artificial Intelligence (AAAI)*, 2006.

[OMG] OMG Unified Modeling Language Specification. `www.omg.org`.

[OWL] OWL reference: Datatypes. `http://www.w3.org/TR/owl-ref/#rdf-datatype`.

[OWL04] OWL Web Ontology Language. `http://www.w3.org/TR/owl-features/`, 2004.

[Pat99] Norman W. Paton, editor. *Active Rules in Database Systems*. Springer, New York, 1999.

[Pel] Pellet: An OWL DL reasoner. Maryland Information and Network Dynamics Lab, `http://www.mindswap.org/2003/pellet`.

[PHWS06] Bijan Parsia, Christian Halaschek-Wiener, and Evren Sirin. Towards Incremental Reasoning Through Updates in OWL DL. In *Reasoning on the Web, RoW 2006*, 2006.

[Poo94] David Poole. Default Logic. In Dov Gabbay, Christopher J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming, Volume 3: Nonmonotonic Reasoning and Uncertain Reasoning*, pages 189–215. Oxford University Press, Oxford, 1994.

[PPW04] George Papamarkos, Ra Poulovassilis, and Peter T. Wood. RDFTL: An Event-Condition-Action Language for RDF. In *In Proc. 3rd Int. Workshop on Web Dynamics (in conjunction with WWW2004*, 2004.

[RDF] RDFDB: An RDF Database. `http://www.guha.com/rdfdb/`.

[RDF00a] Resource Description Framework (RDF). `http://www.w3.org/RDF`, 2000.

[RDF00b] Resource Description Framework (RDF) Schema specification. http://www.w3.org/TR/rdf-schema/, 2000.

[Red] Redland RDF Libraries. http://librdf.org/.

[Rei80] Raymond Reiter. A Logic for Default Reasoning. *Arificial Intelligence*, 12:81–123, 1980.

[RFBLO01] Dirk Riehle, Steven Fraleigh, Dirk Bucka-Lasen, and Nosa Omorogbe. The Architecture of a UML Virtual Machine. In *Conference on Object-Oriented Programming Systems, Languages and Applications*. ACM Press, 2001.

[Ros06] Riccardo Rosati. DL+log: Tight Integration of Description Logics and Disjunctive Datalog. In *KR*, pages 68–78, 2006.

[RSS02] Mathieu Roger, Ana Simonet, and Michel Simonet. Toward Updates in Description Logics. In *Description Logics*, 2002.

[Ses] Sesame: A Framework for storing, querying and reasoning with RDF and RDF Schema. http://www.openrdf.org/doc/sesame/users/index.html.

[Smi] Barry Smith. Ontology and Information Systems. http://ontology.buffalo.edu/ontology(PIC).pdf.

[Sow] John Sowa. Agents. http://www.jfsowa.com/ontology/agents.htm.

[Sow00] John Sowa. *Knowledge Representation: Logical, Philosophical, and Computational Foundations*. Brooks/Cole Publishing Co., Pacific Grove, CA., 2000.

[SPQ06] SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/, 2006.

[SWI] SWI-Prolog Semantic Web Library. http://www.swi-prolog.org/packages/semweb.html.

[TRI] TRIPLE. A RDF Query, Inference, and Transformation Language for the Semantic Web. http://triple.semanticweb.org/.

[TSG$^+$06] Goce Trajcevski, Peter Scheuermann, Oliviu Ghica, Annika Hinze, and Agnes Voisard. Evolving Triggers for Dynamic Environments. In *In Proc. Int. Conf. on Extending Database Technology (EDBT*, pages 1039–1048, 2006.

[Wel03] Christopher Welty. Ontology Research. *AI Magazine*, 24(3):11–12, 2003.

[ZU99] D. Zimmer and R. Unland. On the Semantics of Complex Events in Active Database Management Systems. In *Proceedings of the 15th International Conference on Data Engineering*, pages 392–399. IEEE Computer Society Press, 1999.

Acknowledgements

The writing of this thesis was not always an easy task. But there are people who made this journey easier with words of encouragement and more intellectually satisfying by offering different points of view.

In this respect, I would like to express my gratitude to Professor Wolfgang May, who supervised this thesis. I am deeply indebted to you for your constant help, for the constructive criticism, and for all the stimulating discussions which helped me to weed out the weak points in my argumentation.

Also I want to acknowledge the support that I received from partner, Anke. You were a permanent source of motivation and gave me great support. Moreover, I dare not imagine what this work would look like without your invaluable advice on questions of style and language.

Although not being mentioned, all other helpful hands and minds have not been forgotten. I am deeply grateful to everyone who helped in the completion of this work.

Curriculum Vitae

Franz Schenk
born 03.06.1971 in Tegernsee

12/2003 -	Research Assistant, Universität Göttingen
07/2002 - 11/2003	Research Assistant, Universität Kassel
11/2001 - 02/2002	Scientific Programming
	NovelScience, Göttingen
04/1997 - 10/2001	Georg-August-Universität Göttingen
	Degree in Biology (Dipl.Biol.)
04/1995 - 04/1997	FU Berlin: Biology
10/1993 - 12/1994	Civil Service
09/1991 - 07/1993	Gymnasium Tegernsee (Abitur)
10/1990 - 03/1991	FH Weihenstephan: Biotechnology
10/1989 - 09/1990	FH München: Technische Physik
1987-1989	Fachoberschule Bad Tölz (Fachabitur)
1983-1987	Realschule Miesbach (Mittlere Reife)
1977-1983	Elementary School, Holzkirchen