UNIVERSITÀ DEGLI STUDI DI FIRENZE

FACOLTÀ DI INGEGNERIA

Tesi di laurea in Ingegneria Informatica

SVILUPPO DI UN'ONTOLOGIA PER SISTEMI DI AUTOMAZIONE RICONFIGURABILI IN DOMINI DI PRODUZIONE INDUSTRIALE

(ONTOLOGY DEVELOPMENT FOR RECONFIGURABLE AUTOMATION SYSTEMS IN MANUFACTURING DOMAINS)

Candidato
Leonardo Prosperi

Relatori
Prof. Paolo Frasconi

Prof. Giovanni Soda

Correlatore
Prof. Jacek Malec

ANNO ACCADEMICO 2005–2006

Contents

Introduction

In the last few years, manufacturing systems have become increasingly more complex. Little changes in production processes typically require an automatic full redesign of the system or an intervention of an expert operator, for the reconfiguration of the involved devices. The SIARAS project, which stands for Skill-based Inspection and Assembly for Reconfigurable Automation Systems, proposes a new type of approach for the reconfiguration of modern production lines.

Currently, the project is in its first stages of research at the Department of Computer Science, Faculty of Technology (LTH), in Lund, Sweden. Its main objective is the development of a reconfigurable automation system for variant-rich, low-volume manufacturing and highly automated production lines. To this end, it introduces the new concept of skill-based manufacturing, which requires that a system contains knowledge about its devices, and their skills and properties. The system, called the skill-server, will allow reconfiguration and reparameterization of the process, in order to have a production line which is both cost-efficient and flexible at the same time.

The object of study for this thesis is the design of a knowledge representation system, from which the skill-server would draw the necessary information.

In order to thoroughly represent connections among devices and skills, something as complex as an ontology is needed. An ontology provides formal relationships between the terms of a controlled vocabulary.

Among the many formal ontology languages, OWL offers good development support and guarantees. OWL, an acronym for Web Ontology Language, is a markup language of the RDF/XML category, easily parsed by computers. The OWL family provides three increasingly expressive sublanguages designed for use in different cases and necessities: OWL-Lite, OWL-DL, and OWL-Full. OWL-DL seems to be the best compromise, since it exceeds the expressive constraints of OWL-Lite, and possesses computational decidability which is unavailable with OWL-Full. OWL-DL is called so due to its correspondence with description logics. The structure of a description logic knowledge base, together with its associated reasoning services, are viewed as the core of contemporary knowledge representation systems.

Introduzione

Negli ultimi anni, i sistemi di produzione industriale sono diventati sempre più complessi. Piccoli cambiamenti nei processi produttivi richiedono tipicamente una completa riprogettazione del sistema in modo automatico o l'intervento di un operatore esperto, per la riconfigurazione dei dispositivi coinvolti. Il progetto SIARAS (Skill-based Inspection and Assembly for Reconfigurable Automation Systems) propone un nuovo tipo di approccio per la riconfigurazione delle moderne linee di produzione.

Attualmente, il progetto è nella prima fase di ricerca al Dipartimento di Computer Science, Faculty of Technology (LTH), a Lund, Svezia. Il suo principale obiettivo è lo sviluppo di un sistema di automazione riconfigurabile per linee di montaggio ad alta varianza, a basso volume di produzione e ad alta automazione. A questo proposito, introduce il nuovo concetto di produzione basata sulle abilità, il quale richiede che un sistema abbia conoscenza dei suoi dispositivi, le loro abilità e le loro proprietà. Il sistema, chiamato *skill-server*, permetterà la riconfigurazione e la riparametrizzazione del processo, al fine di avere una linea produttiva che sia economicamente efficiente e flessibile allo stesso tempo.

L'oggetto di studi per questa tesi è la progettazione di un sistema di rappresentazione della conoscenza, dal quale lo skill-server possa trarre le informazioni necessarie.

Al fine di rappresentare in dettaglio le connessioni tra dispositivi e abilità, qualcosa di complesso come un'ontologia è necessario. Un'ontologia fornisce le relazioni formali tra i termini di un vocabolario controllato.

Tra i molti linguaggi formali per rappresentare ontologia, OWL offre un buon supporto allo sviluppo e garanzie. OWL, un acronimo per Web Ontology Language, è un linguaggio di markup della categoria RDF/XML, facilmente interpretabile da parser. La famiglia di OWL fornisce tre sotto-linguaggi sempre più espressivi, progettati per usare in differenti casi e necessità: OWL-Lite, OWL-DL e OWL-Full. OWL-DL sembra essere il miglior compromesso, dato che supera le limitazioni espressive di OWL-Lite e possiede decidibilità computazionale, che non è disponibile con OWL-Full. OWL-DL è così chiamato per la sua corrispondenza con le logiche descrittive. La struttura di una base di conoscenza in logica descrittiva, insieme con i servizi di ragionamento associati, sono visti come la base dei moderni sistemi di rappresentazione della conoscenza.

Chapter 1

Knowledge representation

Knowledge representation is a branch of artificial intelligence that strives to represent a high-level description of the world, in such a way that this information is available and usable by reasoning applications, in order to be able to discover implicit consequences from an explicitly represented knowledge. This chapter will give a brief summary of the development of knowledge representation approaches, from semantic networks to ontologies.

1.1 History

The research in the field of knowledge representation has frequently been focusing on formalizing systems which are able to determine, starting from a knowledge codified in a hierarchical way, implicit inconspicuous consequences and to discover relations among the described entities. Several knowledge representation methods were already developed in the 1970s [1], and it is possible to identify two main categories of approaches:

- Declarative or propositional (logic-based), mainly established on predicate calculus and inference procedures.
- Procedural (non-logic-based), based on cognitive considerations derived from human execution of tasks.

The procedural method expresses the know-how, which is the knowledge of how to perform some task. This kind of approach is usually developed for specific duties, using *ad hoc* procedures and data structures, like *semantic networks* and *frames*. Although differences exist between them, both have a common basis in their features. In fact, they can both be considered network-based structures. The resulting model is mostly domain dependent and is inclined to be more specific than in propositional knowledge, although it could be used in different domains. There is an advantage of using procedural knowledge when it is not computationally feasible to try all logically possible ways of managing knowledge. Because the origins are human centered, the results can be more interesting from a practical viewpoint than the logical system. The semantic nets were introduced at the end of the 1960s, as a representational base for words of the English language, and the objective was to characterize structures of knowledge and reasoning in artificial systems. Similarly, frame-based systems were meant to obtain decisions from the network structure as a whole, not from the individual components.

Although these systems seem feasible, they were not satisfactory from a theoretical viewpoint for at least two reasons: the vagueness and the inconsistence of some constructions and the lack of a semantic level, independent from a particular implementation. A hierarchical structure of the network was introduced to improve representation and reasoning abilities, and the easiest way to implement such structure was to have a link that represented the *is-a* relation. As pointed out by Brachman [2], the *is-a* relationship made semantic networks an efficient storage scheme, since it defined a hierarchy over the concepts and allowed the inheritance of properties. When one concept is more specific than another, it inherits properties from that which is the most general. One important step in an accurate formalization of semantic networks was moving representative meaning from a semantic level to an epistemological level. Brachman succeeded to determine a set of primitives, independent from application or domain. These considerations lead Brachman and his colleagues to create the KL-ONE system [3]. This knowledge representation system did not only focus on description formation, but it introduced an assertion language, and predicate logic proved to be inadequate for this task. In fact, if predicate logic is used without some restrictions, then the information loses its structure and the expressive power is too high to allow efficient computational procedures. To face this problem, a new family of logics was introduced: description logics, which can be seen as sublanguages of first order predicate logic. These logics are used in knowledge representation systems to provide both a language for defining a knowledge base and inference rules to reason over it.

1.2 Knowledge base

The realization of knowledge-based systems requires a precise characterization of a knowledge base; this concerns characterizing the type of knowledge to be specified to the system. A knowledge base is a type of database for knowledge management. It is not a static collection of information, but a dynamic resource.

Inside a knowledge base it is possible to notice a clear distinction between general knowledge about the domain of interest (intensional knowledge), and specific knowledge for a particular problem (extensional knowledge). The need to distinguish general knowledge from specific knowledge causes for that description logic knowledge base to be divided into two types of components, *TBox* and *ABox*.

TBox The TBox (terminological box) contains intensional knowledge manifested in a taxonomy. TBox statements are made of declarations that describe attributes and properties of concepts. The natural structure associated with TBoxes is a semantic network, and that structure has nothing to do with any implementation. Intensional knowledge is often expected not to change, and therefore TBox statements are more static inside a knowledge base and are stored in a data model.

ABox The ABox (assertional box) contains extensional knowledge, that is specific to the individuals of the domain of discourse. Assertional knowledge is often thought to be dependent on one set of conditions, hence ABox statements are more dynamic and are usually stored as instance data.

The main reason for this division is that detachment can be useful when describing and formulating decision procedures for description logics. A reasoner may process the TBoxes and ABoxes separately, because some inference problems are connected to only one of them, independently of the other. In this way it is easier to reason only about a specific part of the knowledge base. A knowledge base structure can be specified through an *ontology*, which with the individuals of its classes establishes a knowledge base.

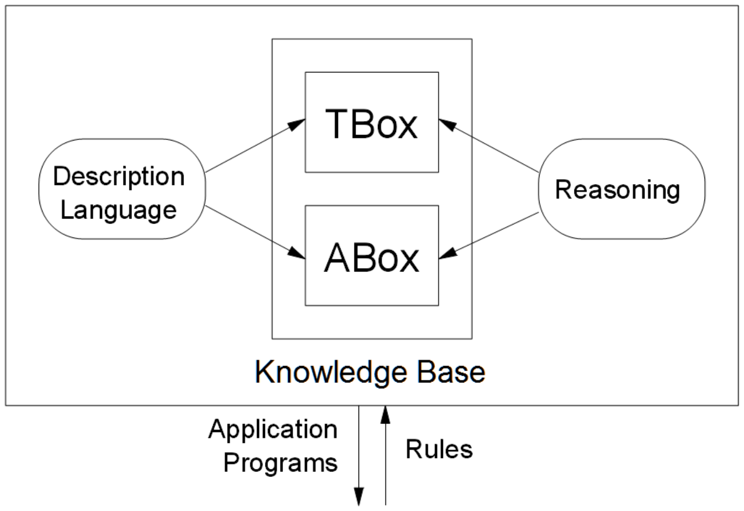


Figure 1.1. Architecture of a knowledge representation system based on description logics.

1.2.1 TBox

The intensional part of a knowledge base, structured upon description logics, is formed by a series of definitions of concepts beginning from descriptions. In turn, they are constructed from other concepts and constructors. Naturally there are some rules regarding the definitions. In particular:

- A concept is defined univocally once and only once.
- A concept cannot be defined by reference to itself.
- A concept cannot be defined through operation of a concept that it subsumes, because this still may have not been defined in the knowledge base.

The last two points can be reassumed imposing that, in general terms, that cyclic or self-referencing definitions are not allowed. Not only are these tasks vital in order to maintain the integrity of the semantics of the knowledge base, but they are also fundamental for the very existence of the procedures of inference and subsumption. Such procedures, in fact, are based on the expansion of the descriptions of the terms of the taxonomy in unique formulas, constituted by the union of several constructors. However, in a description there is a non-primitive concept, but its structured description, becomes in such a way replaced by its description, expanding the original description. In spite of this procedure, it can give rise to exponential descriptions [4], but there are no such practical descriptions to change the complexity of the reasoning. It is possible to manage the complexity of inference considering all expanded descriptions. Constructing the knowledge base in this way, the terminology comes to assume a hierarchical aspect, implicitly structured as a classification system. Every concept is placed exactly between the concepts that subsume it directly and those that it subsumes.

1.2.2 ABox

The ABox contains assertions that specify the individuals that are in the interest domain. The assertions can be of two fundamental types: conceptual assertions and assertions of role. Generally, the assertions in the knowledge base are specified in distinguished way, before the individuals of the domain are declared, and then all the respective relationships are indicated. The conceptual assertions define the belonging of an individual to one category (or concept) previously specified in a TBox. The typical procedures of inference of an ABox also involve the TBox by necessity. In fact, it is possible to see if an individual is an instance of a concept, and which concepts subsume it or which relations it has with other individuals.

1.3 Ontology

In philosophy, the word *ontology* means *study of being*, hence the study about the nature of existence. In artificial intelligence, the term has another meaning and a good definition is given by Tom Gruber in [5]. An ontology is *an explicit specification of a conceptualization.* A conceptualization is an elaboration of a concept that tries to give a simplified view of the world that has to be represented. Under this point of view, an ontology is a formal description of the objects and relationships among them, that belong to a specific domain.

The object-oriented nature of ontologies makes them limited in representing knowledge outside the domain for which they are conceived. For this reason they can also be called domain-specific ontologies. This is not a limitation, as long as the ontology is used for specific tasks. The research is trying to define high-level ontologies, that may describe general concepts defined independently from the domain of application and may be used in different application domains.

Ontology knowledge can be specified using four components: concepts, relationships, properties, and individuals.

- Concepts (class, object, category): an abstract set, or collection of objects. It can contain individuals or other classes.

- Attribute (property): asserts general facts about the elements of a class. Each attribute has at least a name and a value, and is used to store information that is specific to the object it is attached to.

- Relationship (role): an interaction between concepts or individuals. A relationship can be considered as an attribute whose value is another object in the ontology.

- Individuals (instances): are the members of the sets defined by concept; all these terms are used to represent elements in the domain. An ontology needs not include any individuals, but one of the general purposes of an ontology is to provide a means of classifying individuals, even if those individuals are not explicitly part of the ontology.

The issue of the definition of an *ontology language* arises considering ontology-related applications. An ontology language is a formal language used to encode the ontology. During last years of research, a lot of potential representation languages for ontology definition were defined, from natural language for highly informal ontologies to more formal languages for strict formal ontologies.

Not only do the representational constructs in a language have to be considered, but the reasoning that the language has to support, also has to be taken care of.

The research in description logics on a formal, logic-based semantic and an accurate analysis of the reasoning problems, makes this class of knowledge representation formalisms a starting point for defining ontology languages. The reasoning services required to support the building, integration, and development of high-quality ontologies are provided by description-logic systems.

Chapter 2

Description logics

Description logics are based on the concept language, a formal language that describes classes and relationships between elements of the classes. Knowledge bases are developed using description logics and are composed by formal expressions. These expressions can be divided into two separate categories: intensional (TBox) and extensional (ABox).

Both the TBox and ABox can be involved in an inferential process. Each type of query to the knowledge base activates an inferential process that produces answers deduced from the content of the knowledge base as logical consequences. For the characterization of a knowledge base, both the formal definition of the language and the definition of reasoning services are necessary.

2.1 Syntax and semantics

Description logic systems describe structured classes of objects through concept languages. The syntax and semantics, that are derived from such an approach, are defined in [1] and then retrieved by [6] and [7]. The base of concept languages are *concepts* and *roles*: a concept is a class of common elements, and a role is a binary relation between objects or attributes attached to objects.

Description logics are a family of different logics, distinguished by the set of constructors that they provide. Any type of language is named according to a convention introduced by Schmidt-Schauss and Smolka in [8]. Each constructor is associated with a different capital letter and the name of a language is composed by the prefix $\mathcal{AL}$, which is the acronym for *attributive language*. This is then followed by the letters corresponding to the constructors used in the language, for example: $\mathcal{ALC}$ or $\mathcal{ALCHOIF}$.

This type of name assignment makes a detailed classification of concept languages, due to the fact that the computational properties of the reasoning change enormously with the presence or absence of a particular constructor. Hence, it is necessary to accurately indicate the constructors which are used in a description logic.

The semantics of description logics are usually given in a Tarski-style form. Hence, the interpretations $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ are considered, where the non-empty set $\Delta^{\mathcal{I}}$ is any set of objects called *domain of the interpretation*, and $\cdot^{\mathcal{I}}$ is a function called *interpretation function*. This function is a mapping from non-logical symbols of description logics to elements and relations over $\Delta^{\mathcal{I}}$. It assigns a set $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ to every atomic concept A and a binary relation $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ to every role R. Therefore, given an interpretation $\mathcal{I}$ and its interpretation functions for atomic concepts, and roles, it is possible to find the interpretation of any concept [9]. The interpretation of a concept is also called its *extension*, and $\mathcal{I}$ is sometimes called an extension function. The only restriction on the interpretations is the *unique name assumption*, which imposes that different individual names have to be mapped into distinct elements of the domain.

A representation of description logic syntax and semantics is given in Tables 2.1 and 2.2.

Elementary descriptions are atomic concepts and atomic roles. Instead, complex descriptions are built inductively starting from these atomic concepts and roles, using concept constructors and role constructors. Concept constructors take atomic descriptions and transform them into more complex concept descriptions. Table 2.1 shows the syntax and semantics of common concept constructors.

The simplest language denoted by the prefix $\mathcal{AL}$ is close to the expressiveness of frame-based representation systems. It enables the specification of hierarchies of concepts through the conjunction of two concepts ($\sqcap$). Such a binary operator is the intersection of the two elements. The hierarchical structure comes from the fact that the conjunction $A \sqcap B$ is more specific than both A and B, because it denotes a smaller set of elements.

Another category of constructors in $\mathcal{AL}$ enables the specification of attributes with the unqualified existential ($\exists R.\top$) and qualified universal ($\forall R.C$) quantifiers. These allow the creation of expressions to build new concepts in terms of the roles. In particular, expression $\exists R.\top$ denotes the set of elements related to some other element by the role R, while $\forall R.C$ denotes the set of elements which are related by the role R exclusively to elements of the concept C.

Note that, in $\mathcal{AL}$, negation can only be applied to atomic concepts, and only the top concept is allowed in the scope of an existential quantification over a role. In addition, the concept language $\mathcal{AL}$ provides the symbols $\top$ for the full domain and $\bot$ the empty-set, respectively.

More expressive languages are obtained by adding further constructors to $\mathcal{AL}$.

The union of concepts, indicated by the letter $\mathcal{U}$, is written as $C \sqcup D$, and

Table 2.1. Syntax and semantics of concept expression constructors.

Description	Syntax	Semantics
concept name	A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
top	$\top$	$\Delta^{\mathcal{I}}$
bottom	$\bot$	$\emptyset$
intersection	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
union ($\mathcal{U}$)	$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
universal quantification	$\forall R.C$	$\{x \mid \forall y (x,y) \in R^{\mathcal{I}} \Rightarrow y \in C^{\mathcal{I}}\}$
existential quantification ($\mathcal{E}$)	$\exists R.C$	$\{x \mid \exists y (x,y) \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$
general negation ($\mathcal{C}$)	$\neg C$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
(unqualified) number restriction ($\mathcal{N}$)	$\leq nR$	$\{x \mid \sharp\{y \mid (x,y) \in R^{\mathcal{I}}\} \leq n\}$
	$\geq nR$	$\{x \mid \sharp\{y \mid (x,y) \in R^{\mathcal{I}}\} \geq n\}$
functionality ($\mathcal{F}$)	$\leq 1R$	$\{x \mid \sharp\{y \mid (x,y) \in R^{\mathcal{I}}\} \leq 1\}$
qualified number restriction ($\mathcal{Q}$)	$\leq nR.C$	$\{x \mid \sharp\{y \mid (x,y) \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\} \leq n\}$
	$\geq nR.C$	$\{x \mid \sharp\{y \mid (x,y) \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\} \geq n\}$
one-of ($\mathcal{O}$)	I	$I^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ with $\sharp I^{\mathcal{I}} = 1$

interpreted as:

$$(C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$$

The negation of arbitrary concepts is indicated by the letter $\mathcal{C}$ (for complement) and it is written as $\neg C$, and interpreted as:

$$(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$$

Full existential quantification is indicated by the letter $\mathcal{E}$ and it is written as $\exists R.C$, and interpreted as:

$$(\exists R.C)^{\mathcal{I}} = \{x | \exists y (x, y) \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$$

Note that $\exists R.C$ differs from $\exists R.\top$. In the latter case, arbitrary concepts can happen in the range of the existential quantifier, while the first case identifies a class of all elements that have at least a relation R with an element of the class C.

Number restrictions can vary with respect to the type of restrictions. A normal number restriction is indicated by the letter $\mathcal{N}$ and it can be written as $\geq nR$ (at-least restriction) and $\leq nR$ (at-most restriction), where n is a non-negative integer. They are interpreted, respectively, as:

$$(\geq nR)^{\mathcal{I}} = \{x | \sharp\{y | (x, y) \in R^{\mathcal{I}}\} \geq n\}$$
$$(\leq nR)^{\mathcal{I}} = \{x | \sharp\{y | (x, y) \in R^{\mathcal{I}}\} \leq n\}$$

where $\sharp$ denotes the cardinality of a set.

A special case is, if $n = 1$, the number restriction is called *functionality* and it is indicated by the letter $\mathcal{F}$.

If the number restriction is *qualified* for a concept C, the restriction is indicated by the letter $\mathcal{Q}$ and and it can be written as $\geq nR.C$ and $\leq nR.C$. They are interpreted respectively as:

$$(\geq nR.C)^{\mathcal{I}} = \{x | \sharp\{y | (x,y) \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\} \geq n\}$$
$$(\leq nR.C)^{\mathcal{I}} = \{x | \sharp\{y | (x,y) \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\} \leq n\}$$

Through the concept constructor *one-of*, denoted by $\mathcal{O}$, it is possible to enumerate the individuals belonging to the concept. Sometimes, it is convenient to allow *individual names* or *nominals* not only in the ABox, but also in the description language. The *one-of* constructor can be written $\{a_1, ..., a_n\}$, where $a_1, ..., a_n$ are individual names. Such a set concept is interpreted as:

$$\{a_1, ..., a_n\}^{\mathcal{I}} = \{{a_1}^{\mathcal{I}}, ..., {a_n}^{\mathcal{I}}\}$$

Role constructors take atomic concept descriptions and transform them into more complex role descriptions. Table 2.2 shows the syntax and semantics of common role constructors.

Table 2.2. Syntax and semantics of role expression constructors.

Description	Syntax	Semantics
role name	P	$P^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
role intersection	$R \sqcap Q$	$R^{\mathcal{I}} \cap Q^{\mathcal{I}}$
role union	$R \sqcup Q$	$R^{\mathcal{I}} \cup Q^{\mathcal{I}}$
role composition	$R \circ Q$	$\{(a,c) \| \exists b.(a,b) \in R^{\mathcal{I}} \wedge (b,c) \in Q^{\mathcal{I}}$
inverse role ($\mathcal{I}$)	R^{-1}	$\{(y,x) \| (x,y) \in R^{\mathcal{I}}\ \}$

Some constructs can be expressed in first-order predicate logic, such as composition, union, and intersection.

Contrarily, other role constructors require a more detailed description. If a role can satisfy the transitive property, in this case, $\mathcal{AL}$ extended with transitive roles is denoted by $\mathcal{AL}_{\mathcal{R}^+}$.

In description logic languages, it is possible to set constraints using a bidirectional relation using inverse roles, denoted by $\mathcal{I}$. Without the inverse operator for roles, binary relations can only be used asymmetrically. Hence, the inverse operator is important because it overtakes that asymmetry.

A set of axioms of the form $R \subseteq S$ where both R and S are atomic is called *role hierarchy*. Such a hierarchy obviously imposes restrictions on the interpretation of roles. The fact that the knowledge base can contain a role hierarchy is indicated by appending $\mathcal{H}$ to the name of the description logic.

2.2 Description logic languages

From the semantic point of view, not all possible languages are distinct. The semantics enforces the equivalences $C \sqcup D = \neg(\neg C \sqcap \neg D)$ and $\exists R.C = \neg\forall R.\neg C$. Hence, union $\mathcal{U}$ and full existential $\mathcal{E}$ quantification can be expressed using negation $\mathcal{C}$. Contrarily, the use of union and full existential quantification makes it possible to express negation of concepts. Therefore, union and full existential quantification are present in each language that contains negation. It follows, that all $\mathcal{AL}$-languages can be written using the letter $\mathcal{C}$ instead of the letters $\mathcal{UE}$ in language names. For instance, it is possible to write $\mathcal{ALC}$ instead of $\mathcal{ALUE}$.

The smallest propositionally closed description logic is $\mathcal{ALC}$ (Attributive Language with Complements):

$$\mathcal{ALC} ::= \bot | A | \neg C | C \vee D | C \wedge D | \exists R.C | \forall R.C$$

The addition of a new constructor to the description logic language entails the addition of a letter to the description logic name, making it possible to create a lot of combinations. Several additions are independent of one another, such as $\mathcal{O}$, $\mathcal{I}$, and $\mathcal{H}$. These can be added to any type of description logic, but their simultaneous use increases the computational complexity, in case of satisfiability problems on the knowledge base.

The number restrictions of concept constructors, which are $\mathcal{F}$, $\mathcal{N}$, and $\mathcal{Q}$, are more expressive. Only one of them can be used to characterize the description logic. The letter $\mathcal{F}$ means functionality, but it is sometimes employed to express the presence of feature (dis)agreement constructor, rather than functionality. The letter $\mathcal{N}$ generalizes $\mathcal{F}$-supporting logics with number restrictions, and $\mathcal{Q}$ qualifies the number restrictions with the concepts.

Some short references have been introduced to extend the $\mathcal{AL}$ family language, in order to avoid very long names for expressive description logics.

The description logic $\mathcal{ALC}$ extended with transitive role $\mathcal{ALC}_{\mathcal{R}^+}$ is abbreviated with $\mathcal{S}$.

In turn, $\mathcal{SH}$ extended with complex role inclusions, expressed in the form of $R \circ S \subseteq R$ or $R \circ S \subseteq S$, is abbreviated with $\mathcal{R}$.

The use of role constructors increases the variety of description logics even further. This does not directly affect the name of the logic, but sometimes it is possible that the role constructors are indicated after the name, e.g. $\mathcal{ALC}(\neg, \cap, \cup)$. Also, in this case the increase of language expressiveness with new role constructors increases the computational complexity. In some cases, allowing more complex roles inside number restrictions can easily cause undecidability.

2.3 Reasoning with description logics

As formerly stated, a description logic knowledge base is a pair $KB =< TBox, ABox >$. Once the knowledge base KB is defined, TBox and ABox are also defined. Inside the knowledge base, it is possible to define the concept of logical implication, hence, a statement α is logically implied (or entailed) by the knowledge base KB if and only if it is true in every model of KB. This is written as $KB \models \alpha$.

Starting from simple logic implications, it is possible to reason with description logic knowledge bases. The reasoning is the essential process that allows description logic systems to extract implicit data from the knowledge base.

The reasoning services can be different from each other, depending on the purpose of the description logic system and their implemented reasoning techniques. Reasoning services can be classified as simple services, which concerns whether or not a statement has a truth value, or more complex services.

Reasoning services range from checking concept satisfiability or verifying the subsumption between two concepts, to more complex services such as searching all the individuals that are in a given class, even if those individuals are not explicitly in that class.

Some reasoning tasks that only consider a TBox are:

- **Knowledge base satisfiability**: a knowledge base KB is satisfiable ($KB \models \top \not\equiv \bot$) if there is at least one non-empty model for KB.

- **Concept satisfiability**: a class $C \not\equiv \bot$ is satisfiable with respect to KB ($KB \models C$), if a model of KB exists, and where the interpretation of the concept C is not empty ($C^{\mathcal{I}} \neq \emptyset$).

- **Subsumption**: a class D subsumes a class C with respect to KB ($KB \models C \sqsubseteq D$), if each interpretation of C is in that of D ($C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$) for all models of KB.

- **Equivalence**: two classes C and D are equivalent with respect to KB , if $C^{\mathcal{I}} = D^{\mathcal{I}}$ for all models of KB. It is possible to write $KB \models C \equiv D$.

- **Disjointness**: two classes C and D are disjoint with respect to KB, if $C^{\mathcal{I}} \cap D^{\mathcal{I}} = \emptyset$ for all models of KB.

Description logics can offer both intersection and full complement, since all inferences have reference to satisfiability. Therefore, algorithms for checking satisfiability are sufficient to obtain decision procedures for any of the five discussed inferences.

The formerly mentioned inferences are usually used during the design phase of the TBox in order to determine if all classes are satisfiable and that expected subsumptions are verified. After the design of the TBox, all classes are generally satisfiable.

The ABox represents a particular state of a domain, introducing individual names and their properties. ABoxes are composed of statements of the form: $C(a)$ or $R(a, b)$. The class assertions $C(a)$ means that the interpretation of individual a is in the interpretation of the concept C ($a^{\mathcal{I}} \in C^{\mathcal{I}}$). The property assertions (roles) $R(a, b)$ means that the pair composed by the interpretations of individuals a and b is in the interpretation of role R ($a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$).

As formerly stated for the TBox, an interpretation $\mathcal{I}$ is a model for an ABox if it satisfies each element of the ABox.

Reasoning tasks usually considered for ABoxes are the following:

- **Consistency**: an ABox A is consistent with respect to a TBox T, if an interpretation exists that is a model of both A and T. A is consistent, if it is consistent with respect to the empty TBox.

- **Instance checking**: an assertion $C(a)$ is entailed by KB ($KB \models C(a)$) if in every model of KB the interpretation that satisfies KB also satisfies $C(a)$.

- **Retrieval problem**: given KB and a concept C, this task finds all individuals a such that $KB \models C(a)$. The dual problem is to find all named classes C for an individual a for which $KB \models C(a)$.

- **Role fillers**: given a role R and an individual a in KB, role fillers retrieve all individuals x which are related with a by R ($\{x|KB \models R(a,x)\}$). Similarly, it is possible to retrieve the set of all named roles R between two individuals a and b, asking if the pair (a,b) is a filler of R.

ABox consistency can be reduced to concept satisfiability, if a language has the $\mathcal{O}$ constructor. Otherwise, checking the instance is usually harder than concept satisfiability.

The retrieval problem gathers all individuals in the knowledge base that are instances of a given concept in every model of the knowledge base, which can be viewed as a query answering service.

One of the most common tasks is the classification. It consists of representing the class taxonomy inferred by the knowledge base. The taxonomy is a graph, whose nodes are the class names, and whose edges are the subsumption relations between the classes.

2.4 Complexity of reasoning with description logics

In description logics, the complexity of reasoning refers to the computational complexity of the knowledge base satisfiability problem, but it can be also considered for consistency problems.

The complexity depends mostly on the type of constructors provided by the description logic language. Hence, the complexity is directly connected to the expressiveness of a language.

The modularity of the description logic language facilitates the addition of new constructors. This makes it possible to use specific language constructors for different applications, depending on the required level of expressiveness.

Nowadays, the main purpose of the research on description logics is that of extending the expressiveness of description logic languages, while maintaining the decidability and soundness of the reasoning services. The undecidability limits and the complexity of the reasoning are well-known, but there is a lot work to do on the definition of practical algorithms, and on the analysis of the interaction between the different constructors.

According to the computational complexity theory, a set of problems of related complexity is described by a *complexity class*. Typically, a complexity class is the set of the computational problems which require the certain amount of resources to be solved. More formally, it is defined as the set of problems which require an $O(f(n))$ amount of resource R to be solved by an abstract machine M, where n is the size of the input.

It is possible to divide the description logic problems into three complexity classes, whose definitions are based on [10]:

- PSpace-complete: PSpace is the complexity class whose problems which require a polynomial amount of memory and unlimited time to be solved by a deterministic Turing machine.

  A decision problem is PSpace-complete if it is PSpace, and each problem PSpace is reducible to it in polynomial time. These problems are suspected to be outside of P and NP, but that is not known. PSpace can be correlated with P and NP through the following relationship:

$$P \subseteq NP \subseteq \text{PSpace}$$

- ExpTime-complete: the complexity class ExpTime is the set of all decision problems which require a $O(2^{p(n)})$ time to be solved by a deterministic Turing machine, where $p(n)$ is a polynomial function of n.

  A decision problem is ExpTime-complete if it is ExpTime, and each problem in ExpTime is reducible to it in polynomial time. The following relationships are known:

$$P \subseteq NP \subseteq \textsc{PSpace} \subseteq \textsc{ExpTime}$$
$$P \subsetneq \textsc{ExpTime}$$

It is possible to express ExpTime as the space class APSpace, this class of problems requires a polynomial space to be solved by an alternating Turing machine. Due to alternating Turing machine characteristics, it is possible to see that PSpace $\subseteq$ ExpTime.

- NExpTime-complete: the complexity class NExpTime is the class of decision problems which require a $O(2^{p(n)})$ time and unlimited space to be solved by a non-deterministic Turing machine, where $p(n)$ is a polynomial function of n. The following relationship is valid:

$$\textsc{ExpTime} \subseteq \textsc{NExpTime}$$

It is also known that if $P = NP$, then ExpTime = NExpTime.

As previously stated, these decision problems can be solved by some algorithms, therefore they are also decidable problems.

The computational complexity depends both on the type of problem and on the type of description logic language [11]. Figure 2.1 [11] shows how the complexity of a concept satisfiability problem with general TBox changes when languages change. A *general TBox* is a finite set of concept inclusions, $C \subseteq D$ for arbitrary concepts C and D. The base language used in Figure 2.1 is $\mathcal{ALC}$, but it is still the same domain division for more complex languages, such as $\mathcal{ALCH}$, $\mathcal{S}$, or $\mathcal{SH}$.

The division of complexity space in ExpTime and NExpTime is present also in other cases. The check of ABox consistency with general TBox has the same complexity. Hence, for languages with operators $\mathcal{O}$ and $\mathcal{I}$, and a number restriction ($\mathcal{F}$, $\mathcal{N}$, or $\mathcal{Q}$) and for a general TBox, the best known

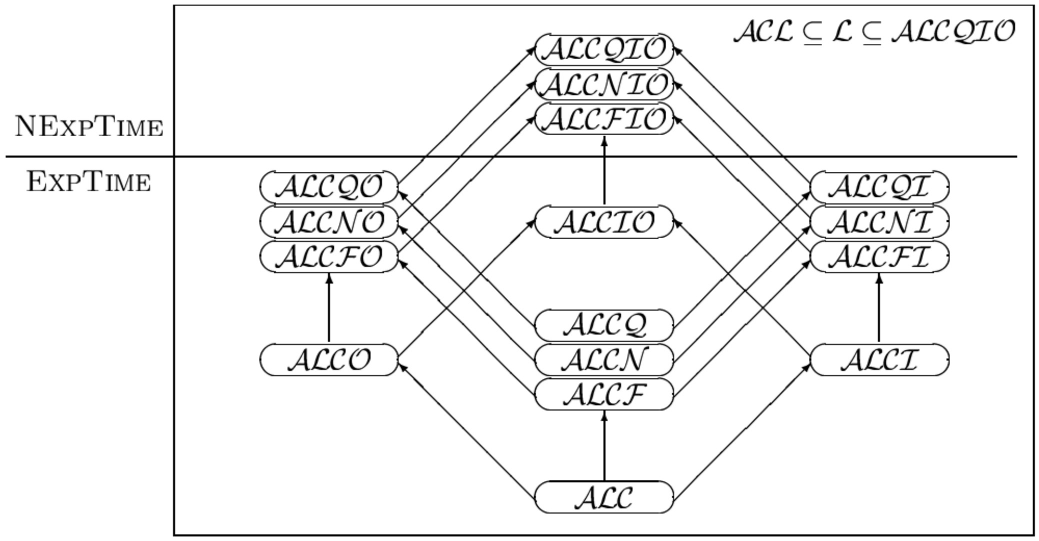


Figure 2.1. Complexity of concept satisfiability problem with general TBox.

algorithms require exponential time and also exponential space, unless the equality PSpace = ExpTime is true.

The assumption that the TBox is empty further simplifies the development of reasoning procedures. In case the TBox is empty and using $\mathcal{SH}$ as base language, the previous complexity schema is still applicable, both for ABox consistency and concept satisfiability problems. While in case of empty TBox and more simple base languages, such as $\mathcal{ALC}$ or $\mathcal{S}$, the complexity division is shown in Figure 2.2 [11].

In this case, the complexity that before was ExpTime becomes PSpace, except if $\mathcal{O}$ and $\mathcal{I}$ constructors are both present.

For several description logics, the definition of their computational complexity is still an open problem. For some other languages, such as $\mathcal{R}$ or $\mathcal{RIQ}$, satisfiability and consistency problems are established only as decidable. The complexity of reasoning problems can be further increased by the presence of a cyclic TBox. The operator μ, introduced in order to represent cyclic TBox, is the *least fixed-point* operator. If it is applied to decidable

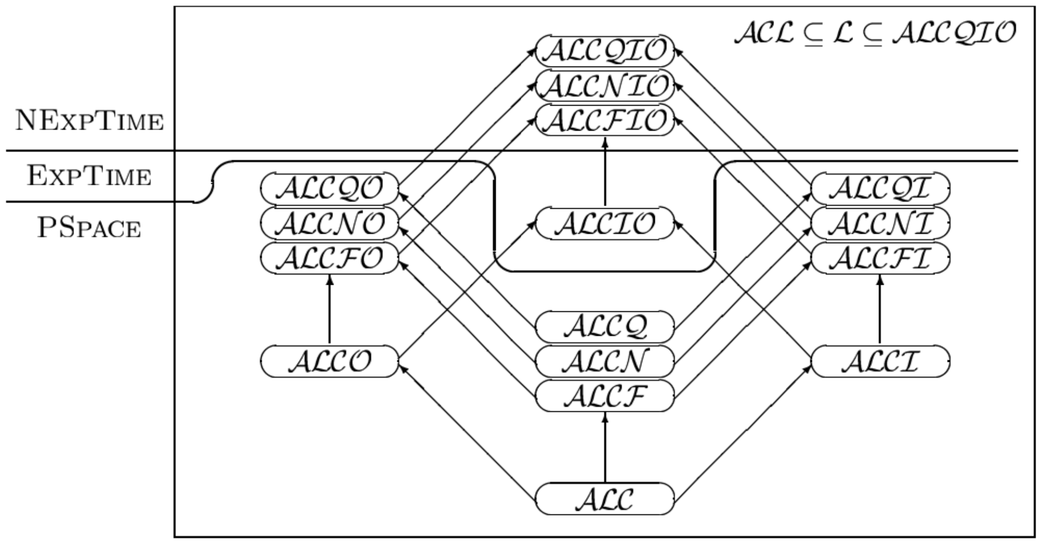


Figure 2.2. Complexity of concept satisfiability problem with empty TBox.

languages, it can turn them into undecidability of the description logic, this is the case of the $\mu\mathcal{ALCOIF}$ language.

Because description logics are a knowledge representation formalisms, and since a knowledge representation system usually has to always answer the queries of a user in reasonable time, the reasoning procedures should always terminate, both for positive and for negative answers. Since the guarantee of an answer in finite time needs not imply that the answer is given in reasonable time, investigating the computational complexity of a given description logic is an important issue. On one side, expressive description logics have often inference problems of high complexity, or in the worst cases they can be undecidable. On the other side, less expressive description logics, that have efficient reasoning procedures, cannot be expressive enough to represent the important concepts of a given application [6]. This trade-off between the expressiveness of description logics and the complexity of their reasoning problems is one important issues for description logics research.

Chapter 3

OWL Web Ontology Language

OWL (acronym for Web Ontology Language) is a markup language used to represent a knowledge domain through ontologies. OWL is derived conceptually from the DAML+OIL Web Ontology Language, and it is a vocabulary extension of Resource Description Framework (RDF). OWL specification is supported by the World Wide Web Consortium (W3C), and the explanation of OWL is based on their specifications [12]. OWL has three increasingly expressive sublanguages designed for use in specific implementations, they are: OWL-Lite, OWL-DL, and OWL-Full.

3.1 OWL syntax

OWL is an ontology language built upon Resource Description Framework (RDF), which is a specification for a metadata model, typically implemented as an application of XML.

The RDF metadata model represents resources in the form of triples, where each triple contains, conventionally written in the following order: subject, predicate, object. The predicate is also known as the property of the triple. These triples represent a labeled, directed pseudo-graph, therefore an OWL ontology is an RDF graph.

Accordingly to any RDF graph, an OWL ontology graph can be written in several different syntactic forms, as reported in [13]. However, the meaning of an OWL ontology is exclusively determined by the RDF graph. Therefore, it is possible to use other syntactic RDF/XML forms, as long as these result in the same main set of RDF triples. Such other syntactic forms would then provide exactly the same meaning as the syntactic form used in building the ontology.

As OWL is a vocabulary extension of RDF Semantics, the meaning given to an RDF graph by OWL includes the meaning given to the graph by RDF. These types of ontologies are called OWL-Full, and they can thus include arbitrary RDF content, which is treated in a manner consistent with that of RDF. The OWL language also provides two less expressive sublanguages: OWL-DL and OWL-Lite. They extend the RDF vocabulary, but also impose restrictions on the use of this vocabulary. Hence, RDF documents are OWL-Full, unless they are specifically developed to be OWL-DL or OWL-Lite.

A standard initial component of an ontology includes a set of XML namespace declarations enclosed in an opening `rdf:RDF` tag. These provide a means to unambiguously interpret identifiers and make the rest of the ontology presentation much more readable.

Among namespace declarations, the following line is present:

```
xmlns:owl ="http://www.w3.org/2002/07/owl#"
```

This is a conventional OWL declaration, used to introduce the OWL vocabulary. Elements prefixed with `owl:` should be understood to be referring to that namespace.

3.1.1 Classes

Classes provide an abstract set for grouping elements with similar characteristics. Each OWL class is associated with a set of individuals, called *instances*. In OWL-Lite and OWL-DL an individual cannot be a class at the same time, because classes and individuals form disjoint domains. An OWL class is syntactically represented as a named instance of `owl:Class`, a subclass of `rdfs:Class`.

```
<owl:Class rdf:ID="MainSensorSkill"/>
```

The only exceptions are two predefined identifiers: the classes `owl:Thing` and `owl:Nothing`. The latter is the empty set, while the former is the set of all individuals. Consequently, every OWL class is a subclass of `owl:Thing` and `owl:Nothing` is a subclass of every class.

Classes are described through a set of constructors (see Table 3.1).

The first three constructors in the table can be viewed as representing the AND, OR and NOT operators for classes. The three operators get the standard set-operator names: intersection (`intersectionOf`), union (`unionOf`), and complement (`complementOf`). `owl:unionOf` and `owl:complementOf` are not part of OWL-Lite, while `intersectionOf` is limited in its use.

A class described with `owl:oneOf` is a list of individuals that are the instances of the class, because it contains exactly the enumerated individuals. The list of individuals can be represented with the RDF construct

Table 3.1. Class constructors.

OWL syntax	DL syntax	FOL syntax
`intersectionOf`	$C_1 \sqcap \ldots \sqcap C_n$	$C_1(x) \wedge \ldots \wedge C_n(x)$
`unionOf`	$C_1 \sqcup \ldots \sqcup C_n$	$C_1(x) \vee \ldots \vee C_n(x)$
`complementOf`	$\neg C$	$\neg C(x)$
`oneOf`	$\{x_1\} \sqcup \ldots \sqcup \{x_n\}$	$x = x_1 \vee \ldots \vee x = x_n$
`allValuesFrom`	$\forall P.C$	$\forall y.P(x,y) \rightarrow C(y)$
`someValuesFrom`	$\exists P.C$	$\exists y.P(x,y) \wedge C(y)$
`hasValues`	$\exists P.\{y_i\}$	$P(x,y_i) \wedge C(y_i)$
`maxCardinality`	$\leq nP$	$\exists^{\leq n} y.P(x,y)$
`minCardinality`	$\geq nP$	$\exists^{\geq n} y.P(x,y)$
`cardinality`	$= nP$	$\exists^{=n} y.P(x,y)$

`rdf:parseType=Collection`, which provides a convenient syntax for writing down a set of list elements. `owl:oneOf` constructor is not part of OWL-Lite.

A class defined by an `owl:allValuesFrom` constraint is a class of all individuals for which all values of the property under consideration are data values within the specified data range. The `owl:someValuesFrom` constraint entails that for each instance of the class that is being defined, there exists at least one value of the property that fulfills the constraint. In OWL-Lite the only type of class description allowed as object of `owl:allValuesFrom` and `owl:someValuesFrom` is a class name.

With a `owl:hasValue` constraint, a class can be composed by individuals for which the property concerned has at least one value semantically equal to a predefined value. Also `owl:hasValue` is not included in OWL-Lite.

Any instance of a class may have an arbitrary number of values for a particular property. To make a property mandatory, to allow only a specific number of values for that property, or to impose that a property must not occur, cardinality constraints can be used.

The cardinality constraints link a restriction class to a data value belonging to a non-negative value. A restriction containing an `owl:maxCardinality` constraint describes a class of all individuals that have at most N semantically distinct values for the property concerned, where N is the value of the cardinality constraint. Similarly, a restriction containing an `owl:minCardinality` constraint describes a class of all individuals that have at least N semantically distinct values for the property concerned. And a restriction containing an `owl:cardinality` constraint describes a class of all individuals that have exactly N semantically distinct values. OWL-Lite allows the use of all three types of cardinality constraints, but only when used with the values 0 or 1.

OWL classes can also be described through *class axioms*, as they contain additional components that state necessary and/or sufficient characteristics of a class. OWL contains three constructs for combining class descriptions into class axioms:

- `rdfs:subClassOf` allows one to say that a class is a subset of another class.

- `owl:equivalentClass` allows one to say that a class has exactly the same members as another class.

- `owl:disjointWith` allows one to say that a class has no members in common with another class.

3.1.2 Properties

OWL distinguishes between two main categories of properties:

- *Object properties* link individuals to individuals.

- *Datatype properties* link individuals to data values.

An object property is made using predefined class `owl:ObjectProperty`, while a datatype property uses class `owl:DatatypeProperty`.

Both `owl:ObjectProperty` and `owl:DatatypeProperty` are subclasses of the RDF class `rdf:Property`. Henceforth, object properties are indicated as relationships, in order not to overload the term *property*. OWL supports several constructs for property:

- RDF Schema constructs: `rdfs:subPropertyOf`, `rdfs:domain` and `rdfs:range`;
- relations to other properties: `owl:equivalentProperty` and `owl:inverseOf`;
- global cardinality constraints: `owl:FunctionalProperty` and `owl:InverseFunctionalProperty`;
- logical property characteristics: `owl:SymmetricProperty` and `owl:TransitiveProperty`.

`rdfs:subPropertyOf` defines that a property is a sub-property of some other property. This states that all instances (pairs) contained in a sub-property are also members of the property, from which the sub-property is derived. Sub-property can be applied to both datatype properties and object properties. In OWL-DL, the subject and object of a sub-property statement must be both the same kind of property.

`rdfs:domain` asserts that the subjects of such property statements must belong to the indicated class.

`rdfs:range` asserts that the values of the property must belong to the indicated class description or to data values in the specified data range.

Multiple domain and range restriction are allowed and they are interpreted as a conjunction of all domains or intersection of all ranges. Multiple alternative ranges or domains can be specified by using the `owl:unionOf` constructor among classes.

The `owl:equivalentProperty` construct can be used to state that two properties have the same property extension. Equivalent properties have the same values, but they can have different intensional meaning. As this requires that properties are treated as individuals, that are only allowed in OWL-Full.

Relationships have a direction, from domain to range. Therefore, it useful to define relations in both directions. The `owl:inverseOf` construct can be used to define such an inverse relation. Syntactically, `owl:inverseOf` is a built-in OWL property with `owl:ObjectProperty` as its domain and range.

A functional property is a property that can have only one value for each instance. Both relationships and datatype properties can be declared as functional. Hence, OWL defines the built-in class `owl:FunctionalProperty` as a special subclass of the RDF class `rdf:Property`.

Whereas a property can be declared to be inverse-functional, then the object of a property statement uniquely determines the subject. Formally, an inverse-functional property is specified by declaring the property to be an instance of the built-in OWL class `owl:InverseFunctionalProperty`, which is a subclass of the OWL class `owl:ObjectProperty`. In OWL-DL object properties and datatype properties are disjoint, so an inverse-functional property cannot be defined for datatype properties.

Both `owl:FunctionalProperty` and `owl:InverseFunctionalProperty` specify global cardinality constraints. This is different from the cardinality constraints contained in property restrictions. The latter are class descriptions and they are only implemented on the property when applied to that class.

When one defines a property as a transitive property, this means that if a property has two pairs of instances: (x,y) and (y,z), then it is possible to infer that the pair (x,z) is also an instance of that property.

Syntactically, a property is defined as being transitive by making it an instance of the built-in OWL class `owl:TransitiveProperty`, which is defined as a subclass of `owl:ObjectProperty`.

A symmetric property, instead, is a property that has as instances both the pair (x,y) and the pair (y,x). Syntactically, a property is defined as symmetric by making it an instance of the built-in OWL class `owl:SymmetricProperty`, a subclass of `owl:ObjectProperty`.

Obviusly, the domain and range of a symmetric property are the same.

3.1.3 Individuals

Individuals are defined with individual axioms, also called facts. These typically are statements indicating class membership of individuals and property values of individuals.

Several languages have a unique names assumption, so different names refer to different things. In many applications, such an assumption is not possible. For example, the same device could be referred to in many different ways. For this reason OWL does not make this assumption.

OWL provides three constructs for stating facts about the identity of individuals:

- `owl:sameAs` is used to state that two identifiers refer to the same individual.
- `owl:differentFrom` is used to state that two identifiers refer to different individuals.
- `owl:AllDifferent` provides an idiom for stating that a list of individuals are all different.

The OWL `owl:sameAs` property links an individual to another individual. Such an `owl:sameAs` statement indicates that two identifiers actually refer to the same thing, the individuals have the same identity.

The OWL `owl:differentFrom` property links an individual to another individual. An `owl:differentFrom` statement indicates that two identifier references refer to two different individuals.

Because `owl:differentFrom` creates a high number of statements, OWL also provides the special constructor `owl:AllDifferent`. `owl:AllDifferent` is a special built-in OWL class, for which the property `owl:distinctMembers` is defined, which links an instance of `owl:AllDifferent` to a list of individuals. The intended meaning of such a statement is that individuals in the list are all different from each other.

3.2 OWL expressiveness

OWL has several characteristics in common with description logics, but also it has some differences. The first difference between description logics and OWL is due to the syntax of OWL. OWL information is saved in RDF/XML documents and parsed into RDF graphs composed of triples. Because RDF graphs express a poor syntax, some description logic constructs in OWL can be saved into many triples. However, because of the way RDF graphs are structured, it is possible to create cyclic syntactic structures in OWL. This is an advantage because these structures are not allowed in description logics. The second difference between OWL and description logics is that OWL contains characteristics that do not correspond to the description logic framework. For example, OWL classes are objects in the knowledge domain and can be instances of other concepts, including themselves. These two differences make a semantic analysis of OWL different from the semantic analysis of description logics.

For this reason, there are defined subsets of OWL that are much more related to description logics. The larger of these subsets, called OWL-DL, restricts OWL in two ways. First, classes, properties, and individuals have to be separated in the semantics for OWL-DL. Second, unusual syntactic constructs, such as descriptions with syntactic cycles inside, are not allowed in OWL-DL. These two restrictions make OWL-DL much closer to a description logic.

OWL-DL can be defined through two forms of semantic specification: a direct model-theoretic semantics, and an RDF-compatible model-theoretic semantics. The two have a strong connection, but the specification clearly states that the direct model-theoretic semantics has the preference. The semantics for OWL-DL is quite standard by description logic standards. The OWL semantic domain is a set whose elements can be disjointly divided into abstract objects and datatype values. Datatypes in OWL are derived from a subset of the built-in XML Schema datatypes.

Despite that, Ian Horrock has shown in [14] how to translate OWL-DL entailment into $\mathcal{SHOIN}(\mathcal{D})$ unsatisfiability. The first step of his proof is to translate an entailment between OWL-DL ontologies into an entailment between knowledge bases in $\mathcal{SHOIN}^+(\mathcal{D})$. Then $\mathcal{SHOIN}^+(\mathcal{D})$ entailment is transformed into unsatisfiability of $\mathcal{SHOIN}(\mathcal{D})$ knowledge bases. It should be noted that concept existence axioms are eliminated in this last step, leaving a $\mathcal{SHOIN}(\mathcal{D})$ knowledge base.

The whole translation from OWL-DL entailment to $\mathcal{SHOIN}(\mathcal{D})$ can be performed in polynomial time and it results in a polynomial number of knowledge base satisfiability problems, each of them is polynomial in the size of the initial OWL-DL entailment. Thus OWL-DL entailment is in the same complexity class as knowledge base satisfiability in $\mathcal{SHOIN}(\mathcal{D})$. Unfortunately, $\mathcal{SHOIN}(\mathcal{D})$ is a difficult description logic. Most problems in $\mathcal{SHOIN}(\mathcal{D})$, including knowledge base satisfiability, are in NExpTime. Sometimes it is also possible to make an inexact translation from $\mathcal{SHOIN}(\mathcal{D})$ to $\mathcal{SHIN}(\mathcal{D})$ that turns nominals into atomic concept names.

OWL-Lite is a subset of OWL-DL, which allows an increased simplicity of implementation. It is obtained by removing several constructors from OWL-DL, and by limiting the use of some of the remaining constructors. This also makes parsing and other syntactic manipulations easier. Due to the fact that OWL-Lite does not have the *one-of* construct, then inference is easier in OWL-Lite than in OWL-DL, making easier to design practical algorithms.

Similarly as happened with OWL-DL and $\mathcal{SHOIN}(\mathcal{D})$, OWL-Lite entailment can be transformed into knowledge base unsatisfiability in $\mathcal{SHIF}(\mathcal{D})$. The transformation can be computed in polynomial time and it results in only linear size increase [14]. While the knowledge base satisfiability in $\mathcal{SHIF}(\mathcal{D})$ is in ExpTime, this means that entailment in OWL-Lite can be computed in exponential time.

Table 3.2. OWL expressiveness summary.

OWL language	DL expressiveness
OWL-DL	$\mathcal{SHOIN}(\mathcal{D})$
OWL-Lite	$\mathcal{SHIF}(\mathcal{D})$

Ontology entailment in the OWL-DL and OWL-Lite ontology languages can be reduced to knowledge base satisfiability in the $\mathcal{SHOIN}(\mathcal{D})$ and $\mathcal{SHIF}(\mathcal{D})$ description logics, respectively, as it is summarized in Table 3.2. Though some constructs in these languages go further than the standard description logic constructs, this is still true [14]. These mappings show that the complexity of ontology entailment in OWL-DL and OWL-Lite is in NExpTime and ExpTime, respectively (the same as for knowledge base satisfiability in $\mathcal{SHOIN}(\mathcal{D})$ and $\mathcal{SHIF}(\mathcal{D})$, respectively). The mapping of OWL-Lite to $\mathcal{SHIF}(\mathcal{D})$ also means that already-known reasoning algorithms for $\mathcal{SHIF}(\mathcal{D})$ can be used to determine ontology entailment in OWL-Lite. The mapping from OWL-DL to $\mathcal{SHOIN}(\mathcal{D})$ can also be used to provide complete reasoning services for OWL-DL.

3.3 Reasoning in an OWL ontology

Given an ontology, it is essential to provide and receive tools, services, and information. Through reasoning it is possible to answer queries to ontology classes and instances, so as to find more general/specific classes or retrieve individuals/tuples matching a given query.

The worst case complexity initially leads to the conjecture that expressive description logics might be of limited practical applicability. However, although the theoretical complexity results are discouraging, empirical analysis of real use cases have shown that the types of construct which lead to worst case intractability rarely occur in practice [1]. Using actual reasoning systems, it has been demonstrated [1] that, even with very expressive logics, highly optimized implementations can provide acceptable performance in realistic applications.

A description logic reasoner offers several inferencing services, such as computing the inferred superclasses of a class, determining whether or not a class is consistent (a class is inconsistent if it cannot have any instances), deciding whether or not one class is subsumed by another and so on. Several description logic reasoners are available, some of the popular ones are: Pellet, FaCT++ and RacerPro. The nature of the project requires to use open source software, therefore only Pellet and FaCT++ will be used.

3.3.1 Protégé

Protégé [15] is a development platform with a set of tools to construct domain models and knowledge-based applications with ontologies. The central part of Protégé utilizes a set of knowledge-modeling structures and functions that support the creation, visualization, and manipulation of ontologies in several representation formats. Protégé can be modified to provide domain-friendly support for inserting data and building knowledge models.

Protégé can be expanded through a plug-in architecture and a Java-based Application Programming Interface (API) for assembling knowledge-based applications and tools, all these functions compensate for the missing of an integrated development environment. However, the Protégé platform supports two main ways of modeling ontologies: Protégé-Frames editor and Protégé-OWL editor. The latter is an extension of Protégé that supports the Web Ontology Language. The Protégé-OWL editor allows the user to:

- Load and save OWL and RDF ontologies.
- Edit and visualize classes, properties and rules.
- Define logical class characteristics as OWL expressions.
- Execute reasoners such as description logic classifiers.
- Edit OWL individuals for Semantic Web markup.

The Protégé-OWL plugin does not make any distinction between editing OWL-Lite and OWL-DL ontologies. However, it offers the choice to constrain the ontology being edited to OWL-DL, or allow the expressiveness of OWL-Full. In addiction to providing an API to manage and modify OWL ontologies, Protégé-OWL uses also a reasoning API to access an external DL Implementation Group (DIG) compliant reasoner. This enables inferences to be made about classes and individuals in an ontology.

OWL-DL has its basis in description logics, which are decidable subsets of first order logic. For a particular task, a logic is decidable if it is possible to design an algorithm that will terminate in a finite number of steps. In a description logic it is possible to define an algorithm that calculates whether or not one concept is a subclass of another concept, which is guaranteed to terminate after a finite number of steps. It is possible to perform automated reasoning over the ontology using a description logic reasoner, because an OWL-DL ontology can be translated into a description logic representation.

3.3.2 DIG interface

The DIG Interface [16] is a standardized XML interface to description logics systems developed by the DL Implementation Group (DIG). The interface defines a simple protocol along with an XML Schema that describes a concept language and accompanying operations.

The interface is not intended as a difficult specification of a reasoning service. Rather, it provides a minimal set of operations (e.g. satisfiability and subsumption checking and classification reasoning) that have been shown to be useful in applications.

Protégé-OWL provides an API that can be used to interact with an external DIG reasoner. Fortunately, the Protégé-OWL reasoning API abstracts away from the DIG language/reasoner, meaning that it is not really necessary to know the fine grained details of DIG, or that the reasoning services are provided by an external DIG reasoner. A number of reasoners including FaCT++, RACER, and Pellet provide support for DIG, through a http connection. The advantage of DIG is that applications can communicate with any DIG compliant reasoner, without needing to know specific reasoner details or reasoner interaction protocols.

However, the current DIG specification has some problems. The expressiveness offered by DIG 1.1 is not sufficient to capture general OWL-DL ontologies, in particular datatype support is lacking in DIG 1.1 and there is a lack of complete correspondence between DIG's concept of relations and OWL properties. The problem in the DIG 1.1 version will be eventually solved in the upcoming DIG 2.0, but meanwhile, the current DIG interface can still be used, but when it is clear where the deficiency is.

3.3.3 Reasoner

In a general architecture for problem solving, a reasoner is a program that provides a reasoning algorithm. It can handle some form of logical inference on a knowledge base, and as a result it gives information about consistency of ontology and classification of taxonomy, unsatisfiable concepts, or it simply answers queries.

Reasoners are used mainly to infer information that is not explicitly contained within the ontology. There are also several standard reasoner services: consistency, subsumption, equivalence, and instantiation checks.

A reasoner can be referred to as a classifier, in order to provide a automatized classification of ontology elements.

3.3.3.1 Pellet

Pellet [17] is a Java based OWL-DL reasoner based on the tableaux algorithms developed for expressive description logics. It supports the full expressiveness of OWL-DL including reasoning about nominals (enumerated classes). Therefore, OWL constructs `owl:oneOf` and `owl:hasValue` can be used freely. Currently, Pellet is the first and complete description logic reasoner that can handle those expressiveness. Pellet ensures soundness and completeness by incorporating the recently developed decision procedure for $\mathcal{SHOIQ}$, that has expressiveness of OWL-DL plus qualified cardinality restrictions in description logic terminology.

Pellet has a number of features specific for OWL requirements:

- Ontology analysis and repair;

- Conjunctive ABox query;

- Datatype Reasoning;
- Ontology Debugging .

Pellet has an interface to support OWL-API applications. There is also a DIGServer, so that Pellet can be used with DIG applications, as an external reasoner for Protégé.

3.3.3.2 FaCT++

FaCT++ [18] is the new generation of the FaCT (Fast Classification of Terminologies) OWL-DL reasoner. FaCT++ has an optimized tableaux algorithms implementation, which has now become the standard for description logic systems. Unlike FaCT, that uses Common Lisp, FaCT++ is implemented using C++ in order to create a more efficient software tool, and to maximize portability. FaCT++ is a description logic classifier that can also be used for modal logic satisfiability testing.

Its expressive logic is $\mathcal{SHOIQ}$, so it is sufficiently expressive to be used as a reasoner for the description logic. FaCT++ also supports reasoning with arbitrary knowledge bases (i.e., those containing general concept inclusion axioms). Also FaCT++ works through a DIG interface.

The FaCT system has been used to analyze the practicability of using the algorithm for subsumption reasoning, and results show that in spite of the logic intractability in the worst case, the algorithm can provide acceptable performance in realistic applications.

3.3.3.3 Algernon

Algernon [19] is an inference engine that supports both forward and backward chaining rules. It automatically activates forward chaining rules when new data is stored in the knowledge base. It automatically activates backward chaining rules when the knowledge base is queried. If a forward chaining rule contains a step that queries the knowledge base, Algernon will automatically switch to backward chaining mode in order to satisfy the query using any applicable backward chaining rules.

Algernon guarantees that rules will not be activated more than once for each identical assertion or query. The first time a query is made, backward chaining rules will be used to infer the answer. If later an identical query is processed, the result will be looked up in the knowledge base rather than wasting time re-executing rules to infer knowledge that is already in the knowledge base. Algernon correctly handles special cases such as new rules being added between queries, and some situations where frames are deleted after a query is made. However, it does not have full support for non-monotonic reasoning.

For efficiency, rules are assigned to classes or relations. They will only execute when an instance of the corresponding class or relation is modified in the knowledge base. A rule can contain commands that perform knowledge base retrievals, assertions, class, instance and slot creation commands, and other Algernon operators that print output, retrieve the current date, call external Java or LISP [20] routines.

Rules can be stored in external files or in a knowledge base.

The main features of Algernon are:

- Supports interleaved forward and backward chaining.
- Direct manipulation of Protégé data. Requires no mapping to an external fact base.
- Contains operators that create and delete classes, instances and slots.
- Base language corresponds directly to knowledge base traversal.

Algernon is useful in any system that needs to process information stored in a frame-based knowledge base. Algernon is implemented in Java and has a full API that allows access to its functionality. It also has a Protégé tab plugin that allows all operations to be performed from within the Protégé GUI. An Algernon server can also be activated to perform knowledge base operations under the direction of a remote client.

Chapter
4

Ontology design

This chapter tries to explain how the ontology for the SIARAS project was developed, by providing evidence and explaining how the choices were made. There is more than one possible ontology to represent a specific knowledge domain. Ontology structure is defined by semantic information that has to be shared and by the type of reasoning that has to be used with this information. The steps of development are described in the next paragraphs, beginning from the analysis of the SIARAS requests, continuing into the construction of a taxonomy, and ending with definition of attributes and instances.

4.1 Focus on SIARAS requirements

Nowadays, according to SIARAS research plan [21], modern large-scale manufacturing facilities are characterized by a high degree of automation, and accomplished by extensive engineering but limited by cost and complexity. That is, the actual production is executed by machines whereas the planning and the set-up of the production line is accomplished by humans. With increasing product variants and shorter product cycles, the frequency of necessary reconfigurations of a manufacturing system will further increase and the dynamic reconfiguration of these complex production processes will become a key technology.

This is the reason why the main objective of the SIARAS consortium is research and development of new technologies and algorithms for the fast and efficient setup of a manufacturing process with the restriction to reuse existing components of a production line. The selected approach is a skill-based representation of manufacturing processes, where production units have embedded knowledge about their skills and properties, in order to solve a given manufacturing task or part of it.

Main concepts represented within the domain are:

Device : is a concrete tool intended to be used in a production line, such as a sensor, a actuator or a robotic arm.

Skill : describes the functional ability of a device. It is an action/function that a device can do to perform a simple operation. For example, a gripper can *grasp* an object, thus a *grasp*-skill can perform a *grasp*-operation within a production line, while it cannot perform complex tasks made up of more operations.

Property : is an attribute that helps to characterize skills and devices. Device properties can be *cost* or *weight*, while skill properties can be *response* *time* or *accuracy*. Some properties can be distinctive features of a type of device/skill, such as *number of fingers* for a *finger gripper*.

Workpiece : is the precessed item, a physical description of an object.

Task : is the description of a production line or a single process, intended to provide a product or a service. A simple task or *operation* can be performed by a single skill, therefore a device performs an operation through one of its skills. Decomposition of a task leads to a set of operations. Each operation is performed by one skill, which in turn belongs to a device.

Tasks and workpieces would not be represented in detail inside the ontology, because they do not belong to the knowledge domain. They are specific for every production line. Workpieces require a dedicated description to highlight geometrical details, by using CAD or X3D formats. Tasks can be represented as graph charts, individualizing requirements and time constraints. Through the ontology it is possible, however, to represent the structure of devices involved in production, and the structure of their skills.

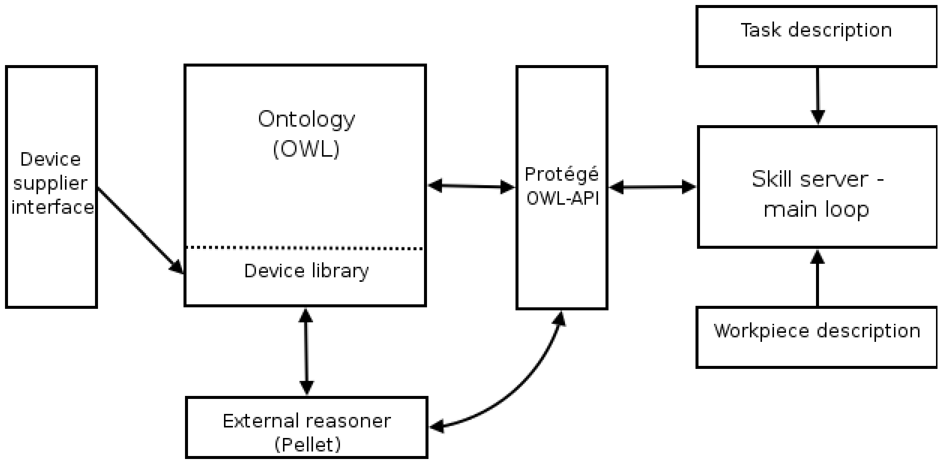


Figure 4.1. Skill server architecture.

In SIARAS project, the central part of a skill-based automation systems is the *skill server*. In Figure 4.1 the building blocks of the current skill server approximation, focused on the ontology side, are shown. The main loop is the core of the skill server, and it is the part that actually performs (or at least initiates) the reasoning.

It is presumed that the skill server has the complete information about the current process implementation. In particular, the skill server has the description of the available devices (with skills and properties) and the description of processes (with tasks and conditions). It also has the link from the tasks to the skills and the temporal order in which the skills of the devices are called. When the product, and consequently the model of the workpiece are modified, an engineer modifies the process description accordingly. When the process description is changed, almost always the process implementation has to be changed, too. The skill server assists and supports the user in this situation, and the user has to confirm each choice that the skill server suggests.

The goal of the skill server is to analyze and manage all the information about the product line in order to provide a correct reconfiguration. Reconfiguration of the production process always requires some modifications to it and its implementation. Some modifications need to be done manually, while others, as parameter optimizations, are better done automatically. All these activities lead to requests for the skill server. A prerequisite for a successful implementation is the collaboration of manufacturers, tool suppliers, and application developers, so all of them must have a clear idea about knowledge domain.

Moreover, due to its practical use and requirements from industrial partners, the ontology has to be articulated with a description logic language. In particular, OWL-DL is used because the reasoning algorithms have to be decidable (all computations will finish in finite time).

4.2 Use case: windshield fitting

Several companies which are part of SIARAS consortium presented some use cases [22] to show which kind of problems have to be solved. In particular, one of the use cases that they provided shows a simple task of windshield assembly into a car frame. The windshield assembly represents a simple sub-task of a main task, that could be a car assembly.

As preconditions, the skill server has a model of the current process, which contains data on the windshield, the car body, and the working parameters of the devices. The assembly consists of a robotic arm with a gripper as its terminal point, that has to grasp a windshield and then move it until it fits inside the car frame. The final movement should be aided by distance sensors to perform measurements until the accuracy of the position error is below a certain threshold. After the assembly, the robotic arm does a blind movement to the base position, ready to start the task again.

The use case supposes that the user changes the process to fit a different kind of window to the car body, such as changing the material property of the window, making it distinctly heavier.

After the change, the skill server consults the device repository, determining whether or not the current gripper is still able to handle the new window. If the current gripper and robotic arm are still capable of managing the new windshield, the skill server provides a re-parametrization of devices. If the robotic arm, to which the gripper is attached, is not sufficient to lift and move the window, this should be adjusted. Using data from the current gripper and the new weight requirements, the skill server selects a new gripper from the device repository. Criteria such as robustness and accuracy used in the selection of the original gripper is applied to select the new one.

The next step of skill server is to insert the new gripper into the process and propagate the changes. Assuming the gripper has a different size, a new trajectory is needed for it to trace in order to avoid colliding with the car frame. Once the new trajectory is calculated, it is possible that the quality measurement sensors need re-calibration.

Since there could be some data missing, a full verification/validation has not been possible, and the new process is shown to the user in order to fill in the missing values and make any additional changes. After this is done, the process is verified and the skill server has adapted the process to take the new window into account.

Analyzing the requests of the task, a schematic formulation of assembly is done using a sequential function chart, as shown in Figure 4.2, taken from [23].

JGrafChart is the software currently used to construct the task descriptions as sequential function charts. The software saves the chart description in a straightforward XML-format which can be loaded into the skill server through a module written for this specific purpose.

In order to preserve the formalism, the chart is connected to the ontology through the optional naming of the steps in JGrafChart. Temporarily, this choice is based on the assumption that step names do not have semantics attached to them. The connection between the ontology and a sequential function chart is created by giving each step in the chart a name corresponding to a skill in the ontology. The chart also contains further information about the skills and the performing devices. Optionally, a suffix consisting of a hash (#) followed by a number is used to indicate which device is performing the skill in the current scenario. Every step with the same suffix is assumed to be performed by the same device.

As in Figure 4.2, the square blocks are skills needed to perform the task. These skills are simple operations performed by devices, *DetectObject*, *MoveFromTo*, *CheckPosition*, *Grasp*, and *Release*. The two *MoveFromTo* skills are to be associated with the same device (presumably a robotic arm) and, similarly, the *Grasp* and *Release* skills should be connected to the same device.

At the moment, the conditions of the transitions do not have any special

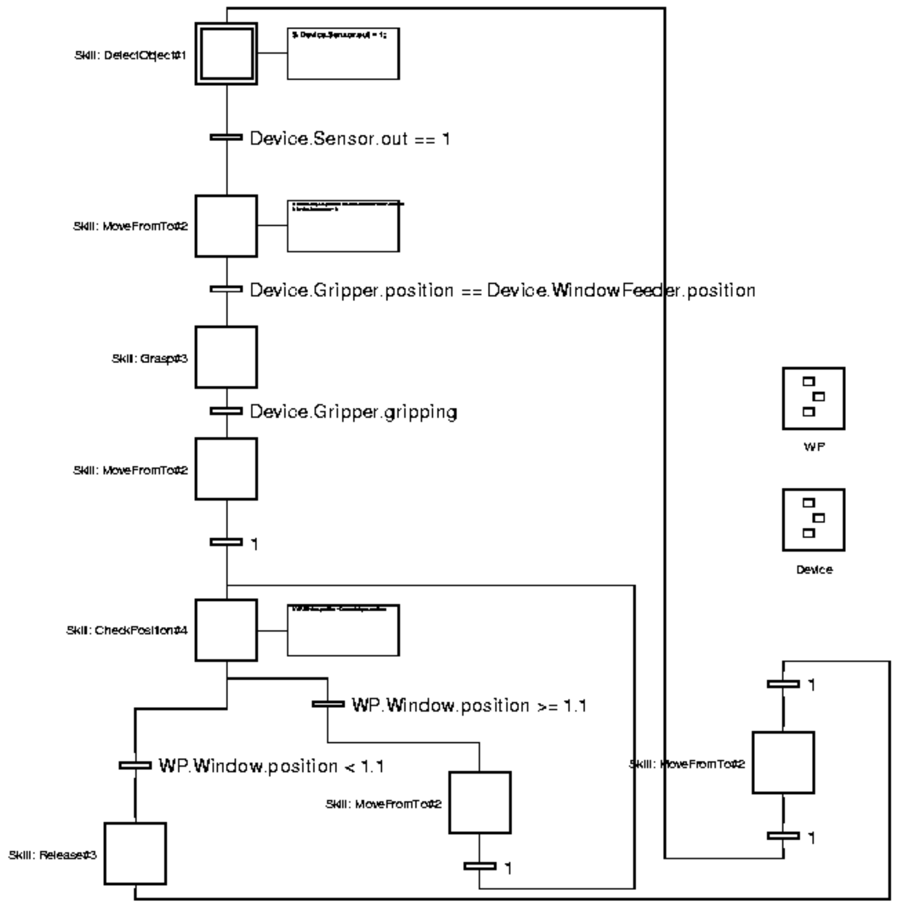


Figure 4.2. Sequential function chart of a windshield assembly.

meaning; they are simply expressions which, if they evaluate to be nonzero, allow the simulation to pass in the next step. However, the skill server will make more use of them by, for example, including a state description and/or pre- and post-conditions of the operations in the guard. Conditions can also be related to skill and device properties.

4.3 Taxonomy

The need to represent devices and skills results in the definition of a controlled vocabulary of terms in a hierarchical structure. This can be done using a taxonomy, generally defined as the science of classifying things. It is possible to view a taxonomy as an ontology without attributes or properties, so it is the start of building an ontology. One of the easiest ways to represent information is in a hierarchical structure, however, a taxonomy may also refer to other structures than hierarchies, such as network structures.

A hierarchical taxonomy is a tree structure, and at the top of this structure is a single classification, the root node, that applies to all objects. Nodes below this root are more specific classifications that apply to subsets of the total set of classified objects.

In the upper level of the ontology there are the four main concepts (or classes): *Device*, *Skill*, *Task* and *Workpiece*. The figure 4.3 shows the four classes, where the *owl:Thing* class is the superclass of every OWL class. Links in the figure mean membership and in Protégé this relationship is called *is-a*.

Starting from this, *Device* and *Skill* classes can be developed as a tree-like structures that clearly depicts how objects are related to each other. In such a structure, each object is the *child* of a *parent* class. The used terminology is provided by industrial partners, in regards to sensors [24][25], grippers [26] and robots [27].

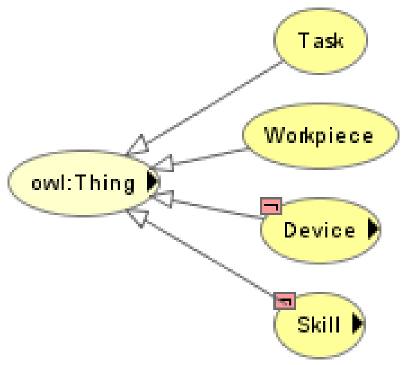


Figure 4.3. Upper level of the ontology.

Figure 4.4 represents a partial expansion of the *Device* class. A more detailed characterization of each device is shown as the device becomes closer to the leaves of the tree.

Similarly, the *Skill* class can also be split into sub-classes and following indications from SIARAS documents [24], the skills of devices can be partitioned into three categories as shown in Figure 4.5.

The main skills are typically used during normal operation of the device and they enable the device to perform certain tasks.

The additional skills are applied before normal operation starts and they can change the settings of the device and the properties of its skills. Additional skills are mainly used for setup, teaching, and training of a device, in order to provide information about installation and parametrization of a new device in a manufacturing line.

The diagnostic skills serve to supervise the operations and the physical properties of the device and also its environment. The diagnostic skills can be used during setup, concurrently with normal operation, or when an error has occurred.

Only main skills describe abilities of a device to carry out a task while, contrarily, the additional and diagnostic skills tend only to characterize devices from the supplier's point of view, as they are not influential for the

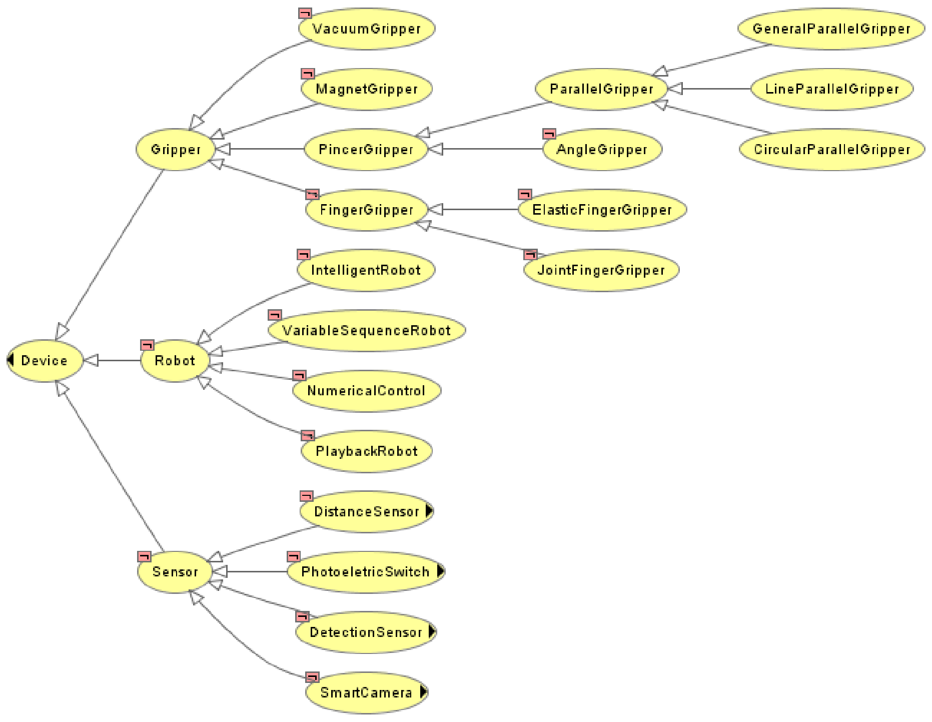


Figure 4.4. Partial expansion of the Device class tree.

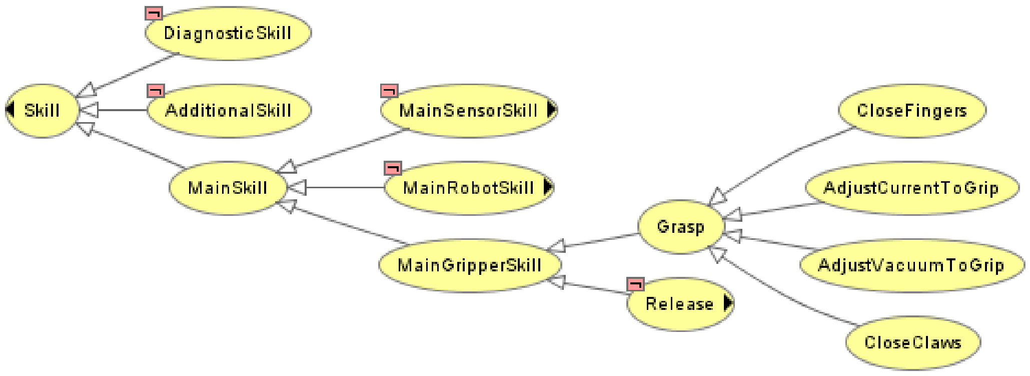


Figure 4.5. Partial expansion of the Skill class tree.

skill server regarding task composition. Main skills are classified in relation to device structure, using a hierarchical tree-like structure similar to that of devices, so that sensor skills are separated from those of the gripper and robot.

Task and *Workpiece* are simple classes in the ontology because they have an external description. Their presence is needed to represent basic ideas of the concepts.

By default, OWL classes are assumed to overlap. It is not possible to assume that an element is not a member of a particular class simply because it has not been asserted to be a member of that class. In order to separate a group of classes it is needed to make them disjoint from one another. This ensures that an individual, which has been asserted to be a member of one of the classes in the group, cannot be a member of any other classes in that group. Each device is disjoint from one another, and also from skills. That is done defining the `owl:disjointWith` statement among each class and its siblings. This means that it is not possible for an individual to be a member of a combination of these classes, because it would not make sense for an individual to be a gripper and a sensor.

The ontology schema might resemble at a representational level a database schema, and instances might be interpreted as database tuples. However, the fundamental difference is that the ontology is supposed to capture some aspects of *real-world* or domain semantics [28], and also represent ontological commitment forming the basis of semantic normalization.

4.4 Relationships

Relationships, or *Object Properties* as they are indicated in OWL, link individuals to individuals, they tend to link classes among themselves in order to make it easier to create concept connections. Skill server has to be able to answer several questions, such as:

- Which skills can perform this task?
- Which skills does this device have?
- Which devices can provide this skill?
- Which devices are needed to perform this task?

As shown in Figure 4.6, the center of reasoning is between *Device* class and *Skill* class. In fact, a device is characterized by its skills. *hasSkill* and *isSkillOf* relations join the two classes together through a bidirectional link. They are inverse of one another, so they have inverted domain and range. The `owl:inverseOf` construction can be used to define such an inverse connection between relations (`owl:ObjectProperty`):

```
<owl:ObjectProperty rdf:about="#hasSkill">
   <rdfs:range rdf:resource="#Skill"/>
   <rdfs:domain rdf:resource="#Device"/>
   <owl:inverseOf>
      <owl:ObjectProperty rdf:about="#isSkillOf"/>
   </owl:inverseOf>
</owl:ObjectProperty>
```

while the `rdfs:range` and `rdfs:domain` constructions, based on RDF syntax, indicate where the two relations operate.

The *canBePerformedBy* link connects *Task* to *Skill* indicating which skills can perform a specific task. Using both *canBePerformedBy* and *isSkillOf* relation, it is therefore possible to go back from a task to all skills and devices needed to perform it.

The relations: *hasMainSkill*, *hasAdditionalSkill* and *hasDiagnosticSkill*, are sub-relations of *hasSkill*, and have a different range, a subset of *hasSkill* range. The three relations have their own inverse relations, that derive from *isSkillOf* relation. Albeit *hasAdditionalSkill* and *hasDiagnosticSkill* participate only marginally in the fulfillment of the task, they provide new details about choice of the devices.

Looking at the code below, the structure of *hasMainSkill* relation appears plain. Others sub-relations have similar code. Unlike *hasSkill*, it has a smaller range and it is defined as a sub-relations of *hasSkill*, using `rdgs:subPropertyOf` constructor.

```
<owl:ObjectProperty rdf:about="#hasMainSkill">
   <rdfs:range rdf:resource="#MainSkill"/>
   <rdfs:domain rdf:resource="#Device"/>
   <rdfs:subPropertyOf>
      <owl:ObjectProperty rdf:about="#hasSkill"/>
   </rdfs:subPropertyOf>
   <owl:inverseOf>
      <owl:ObjectProperty rdf:about="#isMainSkillOf"/>
   </owl:inverseOf>
</owl:ObjectProperty>
```

In particular, the *hasAdditionalSkill* relation notifies the skill server how to configure the device, and then the skill server can reparametrize the device or instruct an operator to do so. Meanwhile, through *hasDiagnosticSkill* relation, the skill server obtains information about the correct functioning of the device in the product line, regarding faults or failure detection.

All these properties are one-point to multi-point links, because a task can require more than one skill to be performed. Some devices can have more

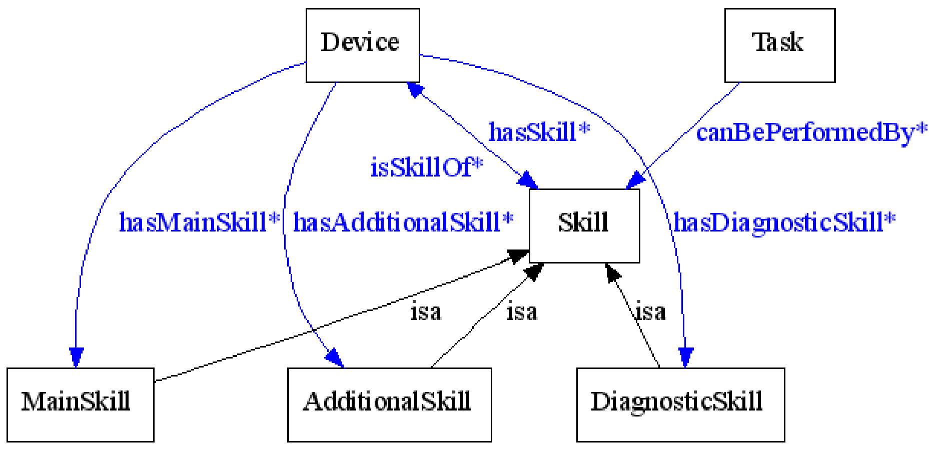


Figure 4.6. Relationships among concepts at upper level.

than one skill, and the same skill can be owned by more than one device. As an example, two movement sensors can be produced by different manufacturers. In this case two devices have different physical characteristics, but they have same skill of determining movement. Similarly, a device can have more than one skill, such as a smart camera having as many skills as have been programmed, or a gripper that is natively definite as a device that can grasp and release.

So, the ontology has enough information to know which devices can carry out a task, but not how to use devices in the current scenario, as this is an assignment of the skill server. A task is performed by an ordered n-tuple of skills. This information is provided to the skill server using an external SFC file. However, *canBePerformedBy* relation cannot identify the right sequence or the skills of an tuple, it can only indicate a set of skills.

All subclasses that belong to *Device* and *MainSkill* inherit the same type of relationships. This could create some ambiguous links, so it could happen that a gripper could have a color detection skill. To avoid that, restrictions are imposed on skills and devices.

4.5 Restrictions

A restriction is a special kind of class description. It describes a class, meaning that all individuals of that class have to satisfy the restriction. OWL distinguishes two kinds of restrictions on relations between classes: value constraints and cardinality constraints. The latter introduces a further level of complexity in the logic. For this reason, they have not been used in the ontology.

A restriction can be used to specify which relations are mandatory for a set or subset of classes. In particular, for devices it is required that they have at least one main skill and at least one property.

Device: hasMainSkill ***some*** *MainSkill*

Device: hasProperty ***some*** *Property*

A value constraint puts restrictions on the range or domain of the relation when applied to a particular class description. This allows devices to be defined with more precision, that is, a color sensor can only detect color, so it has only that skill, and all this can be translated in logic:

ColorSensor: hasMainSkill ***only*** *DetectColor*

In OWL, the `owl:allValuesFrom` construct appears regarding *ColorSensor*:

```
<owl:Restriction>
  <owl:onProperty>
    <owl:ObjectProperty rdf:about="#hasMainSkill"/>
  </owl:onProperty>
  <owl:allValuesFrom rdf:resource="#DetectColor"/>
</owl:Restriction>
```

If a device has more than one skill, it is sufficient to use **or** operator among skills to specify which ones a device has. The same type of restrictions cannot be applied to the skills, because the ontology can be enhanced inserting new devices that do not exist yet. These can have more skills that already belong to previous devices. Each skill has its own *isMainSkillOf* relation, which is constrained by `owl:someValuesFrom` restriction. The information, which comes from this restriction, suggests that there is at least one type of device with that skill.

The effects of restrictions in the device hierarchy allow the thorough specification of the skills for each device, relative to its proximity to the bottom of the hierarchy. In Figure 4.7, it is possible to see how the skills become more detailed in relation to the devices.

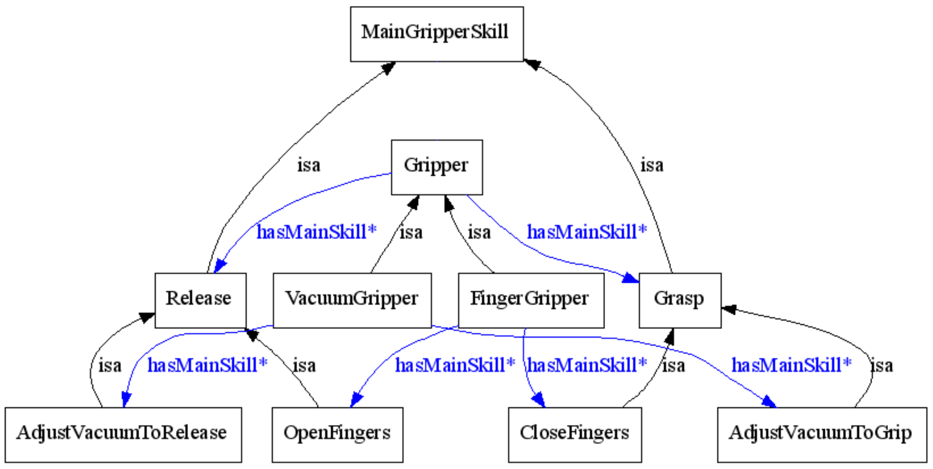


Figure 4.7. Effect of restrictions on grippers.

All of the classes in the ontology only have necessary conditions to describe them. Necessary conditions can be interpreted as: if something is a member of this class then it is necessary to fulfill these conditions. A class that only has necessary conditions is known as a *primitive class*, while a class that has has at least one set of necessary and sufficient conditions is known as a *defined class*.

4.6 Properties

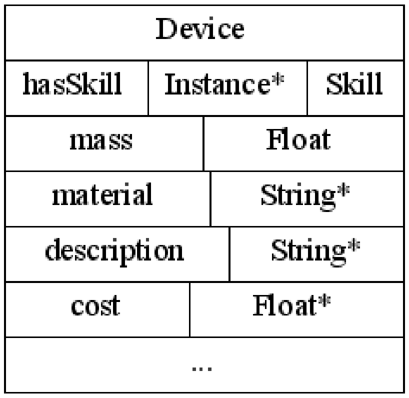
Both devices and skills are characterized by attributes and values. OWL allows the definition of *Datatype properties* for each class, and this *Datatype properties* can set attributes for a class by selecting admissible data-types available in OWL (integer, float, string, etc.).

All properties, if not specifically indicated are inheritable. *Device* properties are inherited by all devices that belong to the *Device* class. Each group of devices used in manufacturing has some particular properties, and so for each branch, new attributes are added to improve the overview of devices. In this way, a property like *typeOfFinger* is assigned only to *FingerGripper* class, and not to other device classes. If *FingerGripper* has a further classification, all its subclasses inherit its properties.

As with devices, skills also have their own properties. Sometimes it is hard to define which properties are specific for devices and which are for skills. Part of the assessment of these properties includes checking device datasheets, however, it is also possible to attribute a property both to devices and skills. Inheritance can also be utilized to distribute properties for *Skill* classes.

If a property can only have one value for each instance, such a property can be declared as functional. For this purpose, OWL defines the built-in class `owl:FunctionalProperty`. This construct increases the complexity of reasoning problems, because it limits the cardinality of a property to be at most one. With the introduction of functional properties, each description logic that originates from $\mathcal{ALC}$ becomes an issue of $\mathcal{ALCF}$ logic, whose complexity of reasoning is at least PSPACE-complete [29]. Several physical properties, such as *mass* or *maxAmbientTemperature*, are unique for each device, but their uniqueness gives a lot of information at the cost of algorithmic complexity.

In figure 4.8, it is shown a partial representation of datatype properties for a *Device*, where properties marked by star ($*$) are not functional.



Device		
hasSkill	Instance*	Skill
mass	Float	
material	String*	
description	String*	
cost	Float*	
...		

Figure 4.8. Relationships among property, skill and device classes.

In addition to the RDF datatypes, OWL-DL provides one additional construct for defining a range of data values. In fact `owl:oneOf` allows specification of a class via a direct enumeration of its members, which is useful to limit values that a property can have. For instance, *numberOfFingers* property can be an integer, with only values: 3,4,5. As described by Tobies in [30], the presence of nominals together with inverse and functional properties makes the logic become $\mathcal{ALCOIF}$, of which complexity is NExpTime-complete. Currently, the presence of nominals is only concerned with values that are not directly involved in the main reasoning procedure.

4.7 Individuals

Individuals represent objects in the domain of interest, where they fill classes with real data. Each individual has its own values on *datatype properties* and it can have relations with other individuals through object properties. In this application, individuals that belong to *Device* hierarchy, are devices used in manufacturing. For example, a new gripper "modelABB" can be inserted into gripper class and all its property values are indicated. After that, its specific skills (like *Grasp* and *Release*) are defined in *Skill* hierarchy. Also for skills, it is possible to specify their values and connect them together through *hasSkill* and *isSkillOf*. In the case that a new model of vacuum gripper will be added, new skills can be created more specifically than for a generic gripper (*AdjustVacuumToGrip* and *AdjustVacuumToRelease*). Similarly, the ontology will not allow inappropriate skills to associate, such as *CloseClaws* or *OpenClaws*.

Among the main functionalities offered by Protégé, there is the control of consistency of the classes of the ontology. A class is said to be consistent if individuals belonging to it can exist. If an ontology has inconsistent classes, then some errors have been made in the design phase. In order to execute the control of consistency of the ontology from Protégé, it is necessary to use an external reasoner, possibly through DIG interface, that analyzes the ontology and evidences possible inconsistent classes.

An important difference between OWL and Protégé is that OWL does not use the *unique name assumption*. This means that different names could actually refer to the same individual, just because two names are different does not mean they refer to different individuals. In OWL, it must be explicitly stated that individuals are the same as each other. It is possible to define the same device with several names and use the construct `owl:sameAs`, to specify that all instances concern the same device. That has sometimes a negative effect, because some external reasoners and DIG interface support unique name assumption.

The OWL file allows the insertion of individuals together with the ontology, inside XML structure. Another possible solution is to create a separate file only with individuals, as this is the presupposition to create a database or a device library.

4.8 Properties as classes

During the development of the skill server prototype, the necessity of reasoning about properties emerged. The ontology has to provide answers to another sequence of questions:

- Which properties does this device have?
- Which properties does this skill have?
- What is such property related to?

As states in [31], it is necessary to identify how reasoning can be performed. Three different levels of reasoning were detected:

- Reasoning without domain knowledge is the lowest level. Reasoning just consists of comparing numerical or text values of properties with the conditions required in the task. For example: Is the sensor response time smaller than 100ms? On this level, a specific domain knowledge is not required. The skill server does not take in regard the whole task, it just evaluates a condition between tasks.
- Reasoning with general domain knowledge. On an intermediate level, reasoning manages with general concepts, not specific to a domain, such as timing or geometric arrangements. Unlike in low level reasoning, the reasoner needs general domain knowledge as rules, which dictate how to order action in time or how to avoid collisions in space.
- Reasoning with specific domain knowledge. This high level reasoning deals with domain-specific knowledge. For example: the reasoner knows about the physical dependencies between the properties such as aperture, exposure time and working distance of a camera.

Evidently, the lowest level of reasoning without a domain knowledge is the easiest one. The selection of skills/devices which fulfill a given set of numerical conditions is performed by the latter. The use of skill properties establishes the connection between skills and conditions formulated in the task representation made using JGrafChart. There is no need for a deeper understanding of the skill properties, for example the resolution of a camera is just a value and it is not related to a physical model of image capturing. This lowest level of reasoning, without domain knowledge, could be used also to chose parameters of skills/devices.

The necessity to reason about general domain knowledge requires a deeper analysis of properties. Mechanisms are needed to register the skill properties used by these reasoners in a structured way. Mainly, it is also necessary to allow properties to be instantiated independently of skills and devices.

On the highest level of reasoning, physical models are needed to evaluate the performance of skills/devices. In these cases, specific reasoning tools are needed, and they can be plugged into the skill server as modules, called *utility functions*.

Utility functions define conditions and effects of every operation in a particular situation, performing reasoning that is domain or device specific. They also need an explicit interface specifying applicability criteria, required data, and obtainable results. Each partner in SIARAS consortium and device manufacturer has to provide their product-dependent utility functions, in order to increase domain-specific reasoning.

The ontology, shown up to now, considers properties only as attributes of devices and skills. Properties are created when a new class is created, while values of properties are declared when a new individual is instanced. The values of properties represent a parametric description of an instanced device or skill.

Representing properties also allows classes to register properties independently of skills and devices. Therefore another class, called *Property*, is added to the four main classes that are already existing. Inside this class there is a list of subclasses that represent the vocabulary of attributes. A datatype property *value* is assigned to every property class, and the interpretation of *value* is due to the skill server.

Because the properties can be associated with both skills and devices, there is a bi-directional relation between properties and devices, and between properties and skills, as shown in Figure 4.9.

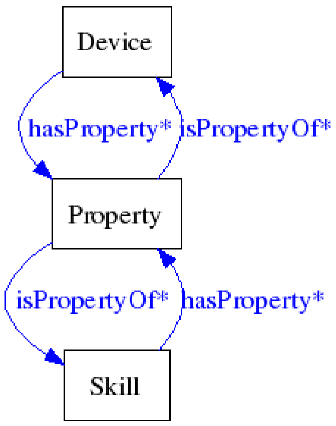


Figure 4.9. Relationships among property, skill and device classes.

hasProperty relation has a domain composed by the union of *Device* and *Skill* classes, as shown in the code below, where there is also defined the inverse relation *isPropertyOf*. That is made necessary by the nature of properties where, in fact, they can be assigned indifferently to both devices and skills together as well.

```
<owl:ObjectProperty rdf:about="#hasProperty">
  <rdfs:range rdf:resource="#Property"/>
  <owl:inverseOf>
    <owl:ObjectProperty rdf:ID="isPropertyOf"/>
  </owl:inverseOf>
  <rdfs:domain>
    <owl:Class>
```

```
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Skill"/>
        <owl:Class rdf:about="#Device"/>
      </owl:unionOf>
   </owl:Class>
  </rdfs:domain>
</owl:ObjectProperty>
```

Through restrictions it is possible to specify which properties are mandatory for some devices or skills. Unlike datatype properties, it is not possible to represent some concepts, such as functionality and enumeration. It is not possible to define that a property has only one value, because assigning functionality or setting the cardinality to *hasProperty* relation, only limits the number of properties, but without specifying which properties have to have only one instance. Skill server should directly manage the cardinality and the values of the properties.

The names of property classes have the first capital letter, so the skill server can distinguish them from datatype properties, and consequently it can reason about both of them. Hence, the skill server can also use the datatype property cardinality, to manage the property classes.

With this implementation, when a new device and its skills are instanced, it is necessary to define parameters inside datatype values and also it is necessary to instance properties as individuals of property class.

Chapter 5

Ontology usage

The developed ontology represents the knowledge base to which device manufacturers will introduce their device descriptions and characteristics, that can be received and processed by the skill server. Engineers continuously develop new devices, hence the ontology has to guarantee expandability. It has to be possible to add new definitions to the ontology and more knowledge to its definition, without altering the set of well-defined properties that are already guaranteed. When this information is inside the ontology, it needs to be retrieved. That can be done directly with skill server or using any tools able to query the ontology.

5.1 Support for introducing new elements

Device manufacturers have two different ways to insert new elements into the ontology. They can directly insert new devices into the current ontology structure, or they can extend the structure in order to create a more appropriate position to place new devices.

5.1.1 New devices

The assignment of introducing new elements is charged to the device producers. They have to create instances for their devices and skills.

Actually, Protégé has a tab for inserting individuals. Using this tab, the producer has to select the appropriate class to locate the device. Obviously, the deeper the class is placed in the hierarchical tree, the more accurate the classification. At this point the interface shows all properties available for that type of device, giving a support for inserting values in accordance with datatypes and enumerations, and limiting the freedom of the user to choose values.

Then, besides the datatype properties, the user has to indicate the relations among the classes. Some relations are mandatory, such as *hasMainSkill* and *hasProperty*, as shown in section 4.5. It is possible to indicate further skills and properties, but their choice is limited automatically by restrictions made on the range.

Using the Protégé individuals tab to insert a new device, the ontology maintains its consistency. However, with current reasoners the classification is not complete for individuals. To use individuals in class descriptions can sometimes cause unexpected results by the reasoner. Unfortunately, at this moment the user does not have any type of support to fill up fields in the form, and there is no warning if the user leaves them empty. An interface is needed that interacts with the user or an automatic text analyzer which reads information from product data-sheets, in order to create new instances.

5.1.2 New classes of devices

It is possible that device manufacturers consider that current structure is not representative for their new product. If this is the case, then the structure can be extended, but a central coordination is needed in order to avoid the creation of inconsistent concepts or duplicates.

Each addition has to be done using a revision control system. Through this system it is possible to manage multiple additions and revisions of the information. Revision control systems are often used in engineering and software development to manage in-progress development of digital data-like application source code and other information that is worked on by a team of people.

Each partner has to be in agreement with the changes, so that a member can gain knowledge of new characteristics of the ontology, and update the vocabulary of terms.

The skill server keeps track of all the updates made to the ontology. But since the utility functions for new devices are not updated, the skill server can only provide a low level reasoning on such devices, comparing numerical values of properties with the conditions required in the tasks.

The simplest addition that can be done is a new device/skill property. Before creating a new property, it does not have to be synonymous of another one or represent the same meaning. Moreover, a property has to also be consistent with the class, to which it is applied, and its subclasses. When a new property is created, it becomes available for all pre-existing instances, and producers can set missing values of such property.

The creation of a completely new type of devices, or the introduction of a type of device not previously considered, is more problematic than inserting a new property. An instance is not sufficient to represent the features that a new type of device can introduce. It is necessary to create a new class inside the ontology.

In fact, the current ontology proposes only three main categories of devices: sensors, grippers, and robot arms. It can be necessary to insert a new family of products, such as actuators.

The addition of a widespread category of devices can be done by merging another ontology to the current one. The merged ontology however, has to have the ternary structure: device-skill-property. Anyhow, if skills and properties are already present into the ontology, the merged ontology can be composed only by device classes, which have to be associated to their skills and properties through existing relationships.

The merging of two related ontologies is obtained by taking the union of the terms and the axioms defining them, and using XML namespaces to avoid name collisions. Then it is needed to add linking axioms that relate the terms in one ontology to the terms in the other through the terms in the merge.

A merged ontology is not only a link between two related ontologies but also a new ontology for further merging with other ontologies in the ontology community. Ontology merging often requires ontology experts' intervention and maintenance, although automated reasoning by an inference engine can be conducted in the merged ontology.

When making a new class of devices, two eventualities are possible. If the new class is disjoint from the others, it can be placed in the most appropriate branch of the tree structure. The class inherits all the datatype properties and relations from its parent class. Moreover, new properties or relations can be attributed to such a class, in order to increase devices description.

Another eventuality is that the new device is not disjoint from the others. Hence, it can fit into two or more classes, that are already defined in the ontology as disjoint.

For example, it is possible to suppose that a new type of gripper is designed. A gripper that has a double gripping capability, it is able to grasp using both magnetic force and fingers. This device can be defined as a gripper, but it is not possible to define it neither as a pure finger gripper or a pure magnet gripper.

The new class of devices can be collocated into the ontology in three different ways:

1. The devices are instanced in an upper level class.
2. A new class is defined and the devices are instanced in that.
3. The devices are instances as the union of two classes, using multiple inheritance.

In the first case, the magnetic finger gripper is instanced as a generic gripper. This does not require the addition of a class into the current ontology, but it tries to fit elements in the current structure as instances without modifying the structure. Specific skills, such as *AdjustCurrentToGrip*, *AdjustCurrentToRelease*, *OpenFingers*, and *CloseFingers*, can be assigned to the gripper as shown in Figure 5.1. But other gripper skills are not prevented, such as *CloseClaws*, or others. The category of magnetic finger gripper is not defined through its skills, but only through the instanced skills. The new device does not have reference to neither to *MagnetGripper* nor to *FingerGripper*.

This type of approach does not allow for the assignment of physical datatype properties, specific for a type of grippers, such as *numberOfFingers* or *typeOfMagnet*. Only physical properties for generic grippers can be assigned to the device. However, it is possible to specify a link to class properties *NumberOfFingers* or *TypeOfMagnet*, because there are not impediments on the choice of such properties. Contrarily, datatype properties of specific skills can still be specified.

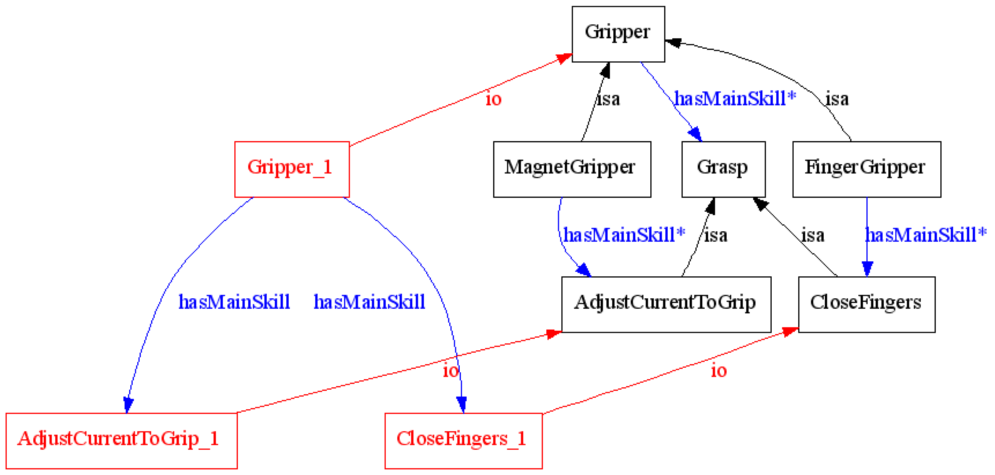


Figure 5.1. First case: instance in an upper class.

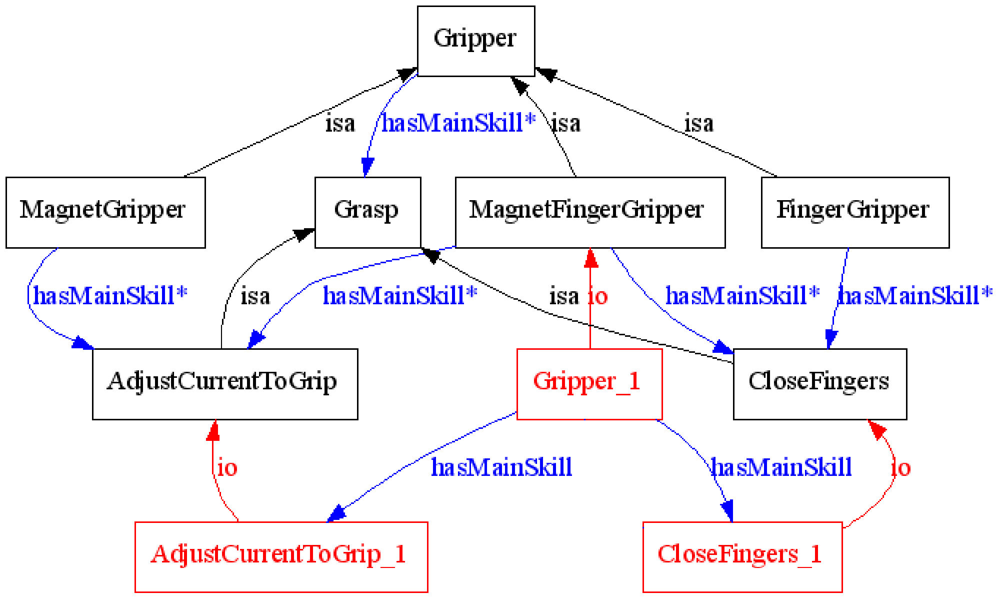


Figure 5.2. Second case: instance in a new class.

The second case plans to create a new class, which is defined as a new type of gripper *MagnetFingerGripper*, as shown in Figure 5.2.

This class has to be disjoint from its sibling classes, in particular: *MagnetGripper* and *FingerGripper*. A device instanced in such class is neither a finger gripper nor a magnetic gripper, it is another category of device.

Through restrictions, it is possible to specify which skills the device can have, and in this case it has *AdjustCurrentToGrip*, *AdjustCurrentToRelease*, *OpenFingers* , and *CloseFingers*.

Other properties, already available for other devices, can also be assigned to the new class, or completely new properties can be created. Unlike the first case, the new devices have a better description of their characteristics and a well defined range for their skills.

Once the new class is created, all producers can create instances in such class or move instances from upper classes if they have used the first approach. Moving instances from an upper class to a lower class is often not applicable, because the lower classes have more restrictions on the range of relations. In this case the lower class is purposely created to receive instances from the upper class, which has sets of instances with the same characteristics. Hence, these instances can be moved without creating inconsistences.

From the first two cases, obviously the second one appears preferable, because it can provide a more detailed description of devices, whereas, in the first case the device is instanced in a generic category, it is without a suitable representation of its characteristics and properties.

On the other hand, the creation of too many classes might render the ontology too widespread to correctly classify all devices. Since a class represents a category of devices, the worst case is to have only one device in each class. Before creating a new class of devices there is an opportunity to define it if the new device is already categorized with some more skills or it is the first specimen of a category of products. Only in this second case, it is necessary to create a new class.

In the third approach, the ontology can have classes that have many superclasses (multiple inheritance). It is nearly always a good idea to construct the class hierarchy as a simple tree. Often a manually constructed hierarchy has no more than one superclass, and this is also the case of the proposed ontology. Computing and maintaining multiple inheritance can be a job for the reasoner. This technique helps to keep the ontology in a maintainable and modular state. This not only allows the reuse of the ontology by other ontologies and applications, it also minimizes human errors that are inherent in maintaining a multiple inheritance hierarchy.

Multiple inheritance in OWL is the same as the feature of object-oriented programming languages in which a class can inherit behaviors and features from more than one superclass. This contrasts with single inheritance, where a class inherits from only one superclass.

Multiple inheritance can cause some problematic situations, so there are both benefits and risks. If, on one side, it presents a better representation, then on the other it increases complexity and ambiguity in some situations.

Before applying multiple inheritance, it is necessary to make the involved classes disjoint. In order to make two classes disjoint, the `owl:disjointFrom` constructors need to be removed from the two class descriptions. Sometimes two classes are also disjoint from each other, because the restrictions on the relations make them logically disjoint. This is the case of *MagnetGripper* and *FingerGripper* classes.

Therefore, multiple inheritance is not always applicable, because it is necessary that two or more classes are logically not disjoint.

5.2 Use of the ontology as a device library

The OWL file allows the insertion of individuals together along with the classes definition, inside XML structure. Another possible solution is to create a separate file with only individuals. This is the presupposition to create a database or a device library.

The library has a dynamic role, because the product suppliers have to update it with new device specifics. The addition of new individuals is bound by the structure and the restrictions of the ontology.

The use of the device library makes it possible for the skill server to read and use the database to search for devices and skills. Each company can create its own database, in which the skill server will do local searches. If a device (or a skill) is not able to satisfy requests, it can search in a distributed system of device libraries, provided by several companies.

5.2.1 Queries in device library

The representation of knowledge only, is not enough. User agents have to have an access to the resources inside the ontology. This need to be able to query the ontology has given rise to several tools, specializing in querying (meta)data based on web standards, such as RDF. Most of the query languages are based on the triple data model. They represent statements of the form of triples, such as <subject, predicate, object>. Others can have different encoding, such as a tree or graph data model, which makes them able to perform more complex queries over the resource description graph.

Five classes of queries have been identified by [28], they are classified by the provided functionality:

- Selection and extraction query: this type of query can recover information represented in the data, where the choice can be based on the content, structure or position within the whole set of data.

- Reduction query: this type is the complementary to the previous type, because it lets specify what has not to be returned as result, instead of specifying what to return.
- Restructuring query: this query is able to change both the value of the data and the structure.
- Aggregation query: aggregation of data is a simple form of generation of new knowledge.
- Combination and inference query: using these queries it is possible combine existing information that is not explicitly connected.

Among alternatives for the reasoner to be used in querying the ontology, Algernon is one of possible choices to be considered. The Algernon tab allows execution of queries and assertions within the Protégé, working in the same memory space. It accepts file-based command line interrogations as well.

Algernon query syntax is based on a triple data model, as *(predicate subject object)*, where the question mark (?) introduces a variable and the colon (:) introduces a command or a keyword. A triple is called clause, and a path is a sequence of clauses. Algernon allows the user to retrieve or confirm information in the system, store and delete information, and have control of inference.

The following queries show how information can be gleaned from the ontology, comparing numerical or text values of datatype properties. The language used for tests is LISP [20]. The versatility of this language permits an easy access to the resources, but the level of reasoning that can be performed by skill server with this query is low. The skill server can search for devices or skills, comparing numerical or text values of properties with the parameters required in the task description. A high level of reasoning is possible with external modules, called *utility functions*, that contain device descriptions and algorithms specific for their skill.

Some examples of questions which can be asked of the skill server regarding devices, skills and their properties, and their translation in the Algernon syntax are as follows:

- Which sensors have a main skill with response time of less than 100ms?

```
((:instance Sensor ?y)(hasMainSkill ?y ?mf)
(responseTime ?mf ?rtime)(:test (:lisp (< ?rtime 100))))
```

Or equivalent:

- Which sensor skills have a response time of less than 100ms?

```
((:instance MainSensorSkill ?y)(isSkillOf ?y ?d)
(responseTime ?y ?rtime)(:test (:lisp (< ?rtime 100))))
```

In output there are devices too, because there is *isSkillOf* relation, that makes devices visible.

It is possible to make the search domain more specific:

- Which distance sensors have a main function with response time of less than 100ms?

```
((:instance DistanceSensor ?y)(hasMainSkill ?y ?mf)
(responseTime ?mf ?rtime)(:test (:lisp (< ?rtime 100))))
```

- Which detecting color skills have a response time of less than 100ms?

```
((:instance DetectColor ?y)(responseTime ?y ?rtime)
(:test (:lisp (< ?rtime 100))))
```

It is possible to also compare a string, not only a float:

- Which distance sensors have a diagnostic skill that has as its output “malfunctioning”?

```
((:instance DistanceSensor ?y)
(hasDiagnosticSkill ?y ?df)(out ?df ?output)
(:test (:lisp (string= ?output "malfunctioning"))))
```

It is possible to check device and skill properties together, making a cross check:

- Which sensor skills have a response time of less than 100ms and belong to a device with a mass of less than 100g?

```
((:instance MainSensorSkill ?y)(isSkillOf ?y ?d)
(responseTime ?y ?rtime)(mass ?d ?m)
(:test (:lisp (and (< ?rtime 100)(< ?m 100)))))
```

Through the Algernon syntax, it is also possible to create longer paths of clauses in order to research elements inside the ontology.

5.2.2 Skill server interaction

The actual version of skill server is in the form of a prototype, where its *main loop* is limited in its functionality. By now, there are no possibilities to connect utility functions in order to provide a high level reasoning about devices.

In the main loop, four main steps are performed:

- The ontology contained in the OWL file is loaded into the memory, through the Protégé-OWL libraries.
- Through the DIG interface, the skill server establishes a connection with an external reasoner, which computes relationships between skills and devices.
- The sequential function chart is loaded into the memory, and parsed to produce a graph of the model of the production line.
- In the final step, the skill server determines which devices can perform the skills present in the task.

Actually, the skill server handles only the classes of devices and skills. There is no access yet to the instanced devices in the ontology nor to the real devices used in the production line.

Therefore as output, the skill server provides a list of devices that can be used to perform the given task. The list currently displays for each operation the involved device class and all its subclasses, as shown in Table 5.1.

The skill server can collect, in groups, all the operations that are performed by the same device, albeit the device uses different skills. In the table, operations *S3* and *S7* are carried out by two skills *Grasp* and *Release*. These two skills belong to the *Gripper* and all its subclasses.

When the utility functions will be available, the list of devices will be composed by instanced devices that are able to perform the given operation.

Table 5.1. Detail of the skill server output.

```
...
Steps: u'S3', u'S7' w/ Skills: Grasp, Release
  Devices:
    VacuumGripper
    JointFingerGripper
    MagnetGripper
    GeneralParallelGripper
    LineParallelGripper
    ParallelGripper
    ElasticFingerGripper
    PincerGripper
    CircularParallelGripper
    AngleGripper
    Gripper
    FingerGripper
...
```

Hence, the skill server might compare the characteristics of the device and its skills with the parameters that are present in the task description.

For example, in case that a gripper has to grasp a workpiece, it is necessary to verify that the workpiece is *grippable* by the gripper. In the windshield fitting case, the skill server has to use the gripper utility functions to determine whether the gripper can handle the glass surface and lift up the workpiece. In the positive case the skill server will search the best parametrization for the gripper. Once a gripper is defined as a possible choice, the skill server will verify whether all the other involved devices are compatible with the new device. It will check if the robot arm is mechanically and electronically compatible with the gripper, and it the sensors can detect gripper movements with a given accuracy.

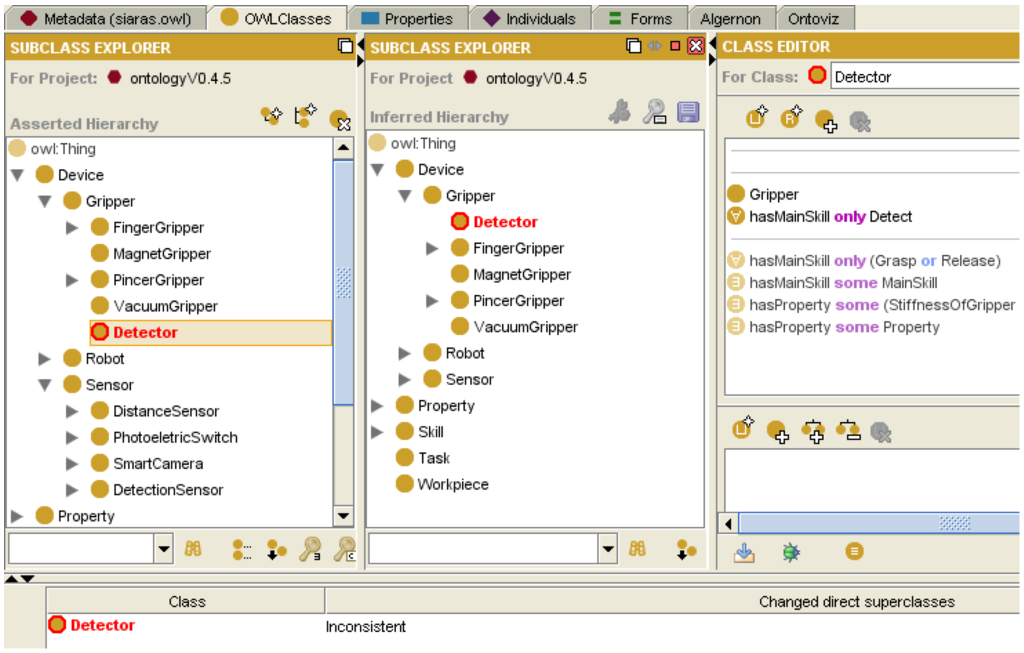
5.3 Categorization of devices using inference

Each type of addition to the ontology has to follow a principle of categorization. It is necessary that manual or automatic addition be as accurate as possible. However, the reasoners can help by giving feedback to the user in case of an error.

Among the main functions offered by a reasoner, there is the control of consistency of the classes of the ontology. If an ontology has inconsistent classes, that some error has been made in the the design phase, and that it is an opportunity to correct it before distributing or using the ontology in an operative skill server. In order to execute the control of consistency of a ontology from Protégé, it is necessary to press the button *Check consistency* on the Protégé toolbar. Having started a reasoner, the ontology can be sent to the reasoner to automatically compute the classification hierarchy, and also to check the logical consistency of the ontology. Protégé passes the ontology to the reasoner that analyzes it and evidences eventual inconsistent classes.

In Protégé the manually constructed class hierarchy is called the asserted hierarchy. The class hierarchy that is automatically computed by the reasoner is called the inferred hierarchy.

For one example, a class *Detector* has been created in *Gripper* class, therefore the restriction *Detector: hasMainSkill* ***only*** *Detect* has been assigned. Because grippers can only have *Grasp* and *Release* skills, and *Detect* skill is disjoint from them, the *Detector* class is an empty set. Hence, Protégé signals that the *Detector* class is inconsistent, marking it with a red circle in the inferred hierarchy.

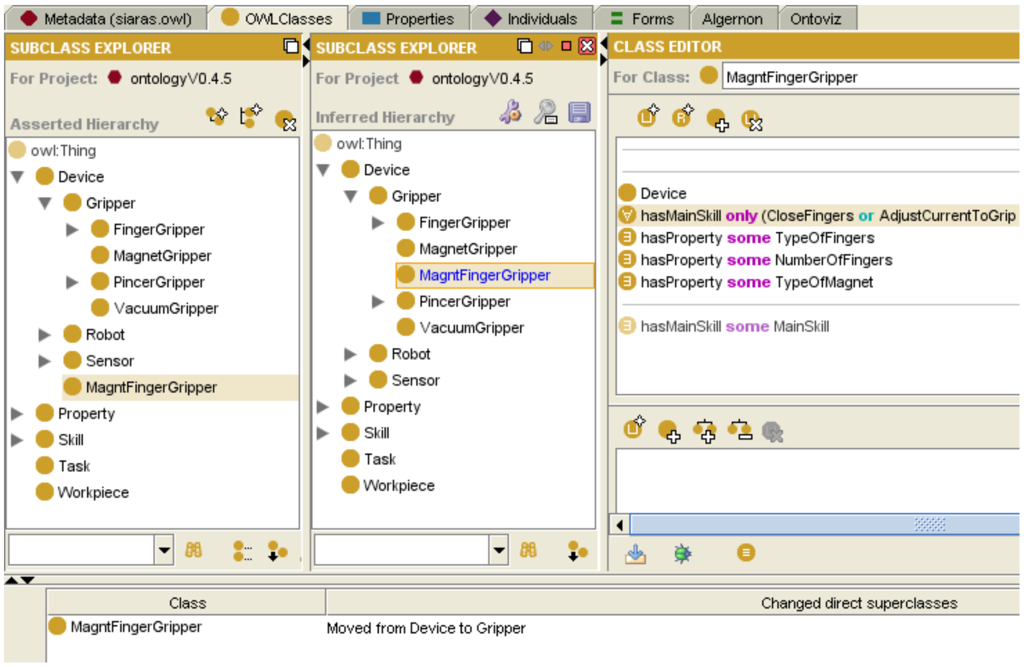


Gripper
hasMainSkill **only** Detect

hasMainSkill **only** (Grasp **or** Release)
hasMainSkill **some** MainSkill
hasProperty **some** (StiffnessOfGripper
hasProperty **some** Property

Class	Changed direct superclasses
Detector	Inconsistent

Figure 5.3. Protégé interface: *Check consistency* results.



Device
hasMainSkill **only** (CloseFingers **or** AdjustCurrentToGrip
hasProperty **some** TypeOfFingers
hasProperty **some** NumberOfFingers
hasProperty **some** TypeOfMagnet

hasMainSkill **some** MainSkill

Class	Changed direct superclasses
MagntFingerGripper	Moved from Device to Gripper

Figure 5.4. Protégé interface: *Classify taxonomy* results.

As a second example, another function that can be useful in order to understand what a reasoner can do to help the categorization is the classification of the ontology. Therefore, like the consistency control, the classification is one function offered by the reasoner. It allows the user to obtain the inferred hierarchy of the classes of the ontology which can vary from the one declared by its creator. In order to execute the classification from Protégé, it is necessary to press the button *Classify taxonomy* on the toolbar. Protégé passes the ontology to the reasoner which analyzes it and generates the inferred hierarchy that becomes visualized in addition to the one declared by the creator.

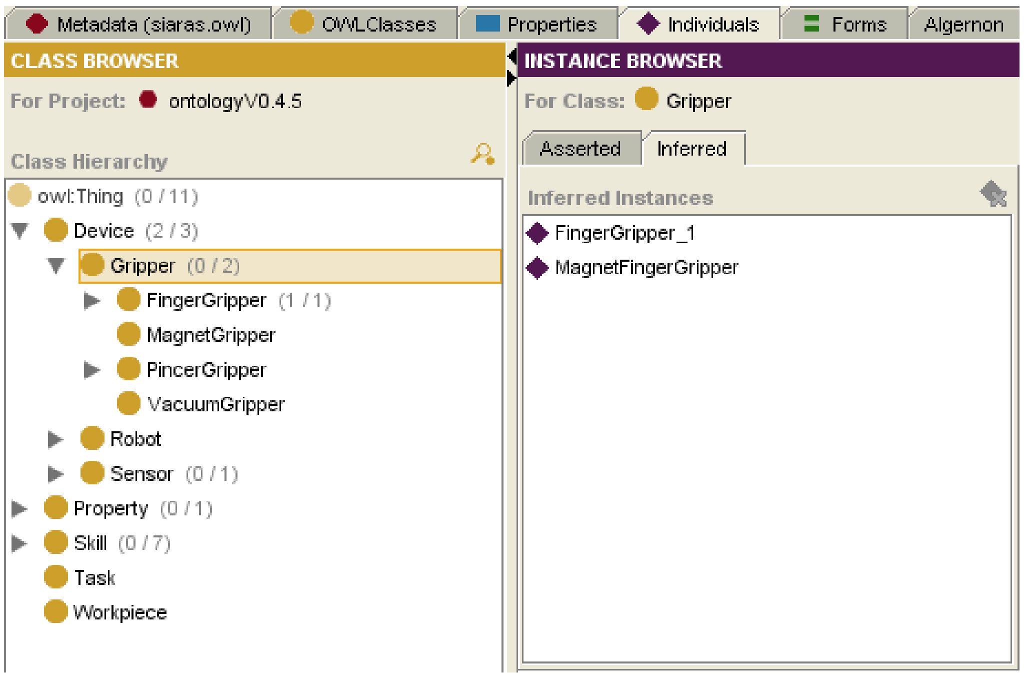
Using the first example, *MagnetFingerGripper* class is located as the direct subclass of *Device* class. When the inferred hierarchy has been computed, an inferred hierarchy window will open next to the existing asserted hierarchy window as shown in Figure 5.4.

Classifying the ontology after having inserted the defined class *MagnetFingerGripper*, the obtained result in the inferred hierarchy is that the *MagnetFingerGripper* class is moved from *Device* to *Gripper* class. If a class has been reclassified then the class name will appear in a blue color in the inferred hierarchy. The user can update the structure of the ontology with the inferred hierarchy, so as to preserve a more logically correct structure.

The last built-in function is reachable by pressing the button *Compute inferred types* on the Protégé toolbar. This computation is only available when individuals exist in the ontology. While the taxonomy classification finds a better collocation for the class, the computation of inferred types re-collocates the instances in new classes.

As a third example, a new device is instanced in the *Device* class. Such device, *MagnetFingerGripper*, is denoted as having only gripper skills.

Making a computation of inferred types, the obtained result is visible in the Figure 5.5. In the asserted hierarchy, each class is labeled with the values in the form (n, m), where n is the number of asserted instances belonging to



For Project: ontologyV0.4.5

Class Hierarchy

owl:Thing (0 / 11)
Device (2 / 3)
Gripper (0 / 2)
FingerGripper (1 / 1)
MagnetGripper
PincerGripper
VacuumGripper
Robot
Sensor (0 / 1)
Property (0 / 1)
Skill (0 / 7)
Task
Workpiece

For Class: Gripper

Inferred Instances

FingerGripper_1
MagnetFingerGripper

Figure 5.5. Protégé interface: *Compute inferred types* results.

such class, and m is the number of inferred instances.

In the list of inferred instances of *Gripper* class there are two elements, while the class has no asserted instances.

The first element is the instance that is in the subclass *FingerGripper*, through the subsumption it is an inferred instance of all its super-classes.

The second element is *MagnetFingerGripper*, which has been re-collocated by the reasoner in *Gripper* class.

Chapter 6

Conclusion and future directions

6.1 Work evaluation

The developed ontology follows the SIARAS Consortium requirements. In fact, it represents the manufacturing vocabulary from a *skill point of view*, where the three elements: skills, devices, and properties, are separated. Having connected the devices with their skills and properties, it is easy to identify the tasks that can be achieved with available devices.

The ontology also provides a high level representation of properties. Properties have a double representation, where the first one is defined as a set of instances of the built-in OWL class `owl:DatatypeProperty`. This renders the ontology able to manage a high range of values regarding device and skill characteristics. Contrarily, a double representation increases the complexity of reasoning algorithms.

Pellet developers offer an on-line tool, that provides various reasoning services for OWL documents, such as OWL species, consistency checking, satisfiability, and entailment tests.

Applying the test to the current ontology, the result is:

OWL Species: DL
DL Expressivity: ALCHOIF(D)
Consistent: Yes

Beginning from the base description logic $\mathcal{ALC}$, the logic also has other constructors: $\mathcal{H}$ (role hierarchy), $\mathcal{O}$ (nominals), $\mathcal{I}$ (inverse role), and $\mathcal{F}$ (functionality).

The representation of properties as classes allows the skill server to have a vocabulary of properties. However, it lacks high expressiveness in value and range representation. Following SIARAS requirements, the actual skill server version only supports this type of representation.

Performing the same test on the ontology without datatypes properties produces the following results:

OWL Species: DL
DL Expressivity: ALCHI(D)
Consistent: Yes

As expected, the new expressiveness is $\mathcal{ALCHI}$, because by removing datatype properties, functionality and nominals have also been removed.

Device manufacturers have the assignment to populate the ontology with instances and new classes, but it becomes clear that the ontology can never be complete.

However, the ternary-tree structure easily allows expansion, but each change can make the ontology inconsistent. In order to prevent such a problem, it is important to know the computational complexity to verify the ontology.

An inconsistent ontology is also quite easy for a reasoner to discover, if it can process the whole ontology. In fact, in tableau reasoners, as Pellet and FaCT++, unsatisfiability testing is reduced to a consistency test by assuming that there is a member of the class to be tested, and doing a consistency check on the resultant knowledge base.

However, unlike with simple unsatisfiable classes, it is very difficult for a reasoner to do further work with an inconsistent ontology. Since any result follows a contradiction, no other results from the reasoner are useful.

6.2 Future directions

The work done in this thesis could be extended with the categorization of new devices, such as actuators, conveyor belts, and all devices that are in a production line. These additions allow a wider representation of the manufacturing domain. Theoretically, the range is infinite, but limitations can be imposed upon device and skill representation.

Instances are needed to create a database of devices. The instances can be added to the ontology manually but, since the addition of new instances is quite trivial, it can be assigned to an external tool. Here, the tool is able to extract information from data-sheets and documents and insert new terms in the ontology. The reasoners will have the assignment of verifying the ontology consistency and, in case of negative feedback, the user intervention is needed.

Now, the instances are memorized in the OWL file using XML references.

The next step could be to move instances into a relational database management system, in order for them to be more reachable by several skill servers, and with the possibility of also creating a distributed database system.

The availability of properties as classes allows further development of relationships among properties. In the future, it might be possible to represent dependencies among the properties. For example, physical properties such as aperture, exposure time and working distance of a camera sensor could be represented through a relationship. Regardless, continued research of the skill server development along with the SIARAS ontology is needed in order to use the newly introduced functionalities.

Acknowledgments

I would like to begin saying that the time I have spent in Sweden was one of the most beautiful and interesting periods of my life. So I thank Sweden and its people for having received me with their kindness in such a marvelous place.

I would like to thank, first, Professor Jacek Malec for his kind welcome at LTH and for his supervision of my work. I would also like to thank all the people at Computer Science Department of LTH for their courtesy and support: Ola Angelsmark, Slawomir Nowaczyk, Pierre Nugues, and Klas Nilsson. After that, I would like to thank all the people involved in SIARAS project.

Special thanks go to my Italian supervisors Professor Paolo Frasconi and Professor Giovanni Soda which gave me the opportunity to complete my work when I came back to Italy.

My appreciation also goes then, to my old friends and the new friends I have met in Sweden with which I have enjoyed my time there.

Finally, I would like to show my gratitude to my family. I have been able to enjoy this very interesting and beautiful experience only thanks to their continuous support and efforts. A big thanks to my girlfriend Jill for her help.

Bibliography

[1] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications.* Cambridge University Press, 2003.

[2] R.J. Brachman. What ISA is and isn't: An analysis of taxonomic links in semantic networks. *IEEE Computer*, vol. 16:30–36, Oct 1983.

[3] R.J. Brachman and J. Schmolze. An Overview of the KL-ONE Knowledge Representation System. *Cognitive Science*, vol. 9:171–216, Apr 1985.

[4] Bernhard Nebel. Terminological reasoning is inherently intractable. *Artificial Intelligence*, 43:235–249, 1990.

[5] Thomas Gruber. Toward Principles for the Design of Ontologies Used for Knowledge Sharing. *Internationa Journal of Human-Computer Studies*, vol. 43(5-6):907–928, 1995.

[6] Sergio Tessaris. *Questions and answers: reasoning and querying in Description Logic.* PhD thesis, University of Manchester, 2001.

[7] Raphael Volz. *Web Ontology Reasoning with Logic Databases*. PhD thesis, Universität Karlsruhe (TH), 2004.

[8] Manfred Schmidt-Schaub and Gert Smolka. Attributive concept descriptions with complements. *Artif. Intell.*, 48(1):1–26, 1991.

[9] Ronald Brachman and Hector Levesque. *Knowledge Representation and Reasoning*. Morgan Kaufmann, 2004.

[10] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing Company, 1994.

[11] Evgeny Zolin. Complexity of reasoning in Description Logics `http://www.cs.man.ac.uk/~ezolin/logic/complexity.html`, 2004.

[12] Mike Dean, Guus Schreiber, Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. OWL Web Ontology Language Reference. W3C Recommendation. `http://www.w3.org/tr/owl-ref/`, 2004.

[13] Dave Beckett. RDF/XML Syntax Specification (Revised). W3C Recommendation. `http://www.w3.org/tr/2004/rec-rdf-syntax-grammar-20040210/`, 2004.

[14] I. Horrocks and P. Patel-Schneider. Reducing OWL entailment to description logic satisfiability. In *Proc. of the 2nd International Semantic Web Conference (ISWC)*, 2003.

[15] Stanford Medical Informatics. The Protégé; Ontology Editor and Knowledge Acquisition System
`http://protege.stanford.edu/`, 2006.

[16] Daniele Turi. DIG Interface
`http://dig.sourceforge.net/`, 2006.

[17] Mindswap. Pellet OWL Reasoner
`http://www.mindswap.org/2003/pellet/`, 2006.

[18] Sean Bechhofer. OWL: FaCT++
`http://owl.man.ac.uk/factplusplus/`, 2006.

[19] Micheal Hewett. Algernon - Rule-Based Programming
`http://algernon-j.sourceforge.net/`, 2006.

[20] Paul Graham. *ANSI Common LISP.* Prentice Hall, 1995.

[21] Kai Modrich et al. Sixth framework programme priority, Annex I - Description of Work. SIARAS, 11 May 2005.

[22] Ola Angelsmark. Use cases for the skill server, version 0.6. SIARAS - Lund University, 10 January 2006.

[23] Ola Angelsmark. A first approximation of the skill server, version 0.1.6. SIARAS - Lund University, 12 June 2006.

[24] Albrecht Ströle. Structured Description of Skills and Properties, Version 0.2. SIARAS - Sick AG, 23 November 2005.

[25] Albrecht Ströle. Tasks and Skills of Optical Sensors, Version 1.1. SIARAS - Sick AG, 10 October 2005.

[26] Matthias Bengel. Tasks and skills of grippers, version 0.1. SIARAS - Fraunhofer IPA, 6 October 2005.

[27] James A. Rehg. *Introduction to Robotics in CIM Systems (5th Edition).* Prentice Hall, 2002.

[28] George G. Savii. Evaluation of methods and software tools for skill representation, version 1.2. SIARAS - Universitatea Politehnica din Timisoara, 8 April 2006.

[29] C. Lutz. PSpace reasoning with the description logic $\mathcal{ALCF}(\mathcal{D})$. *Logic Journal of the IGPL*, 10(5):535–568, 2002.

[30] Stephan Tobies. The complexity of reasoning with cardinality restrictions and nominals in expressive description logics. *Journal of Artificial Intelligence Research*, 12:199–217, 2000.

[31] Jens Pannekamp. Levels of reasoning and location of domain knowledge, version 0.2. SIARAS - Fraunhofer IPA, 13 June 2006.