This chapter introduces knowledge representation particularly from the viewpoint of configuration. Description Logics are described as a formal approach to represent conceptual knowledge and a configuration-specific terminology is created by defining modeling facilities together with a mapping to Description Logic elements that define their formal semantics. After this introduction, we describe how to build configuration models and how to reason about the represented knowledge.

3.1 Introduction

What is a knowledge representation? When knowledge about a domain is represented, this representation is a surrogate, a substitute for the things that exist in the domain of concern. There must be some form of correspondence specified between the surrogate and its referent in the world. This correspondence is the semantics for the representation [Davis et al., 1993].

What is a model? In general, a model is used to stand in place of some entity of the world. Therefore, a model is a representation, it imitates, resembles or stands for something else [Lee, 1999].

For being able to reason about the contents of a model it is necessary to define a collection of data structure types, a collection of operators or inferencing rules which can be applied to these data types, and a collection of general integrity rules, which implicitly or explicitly define consistency of the model [Codd, 1980]. The operators and integrity rules are essential for understanding how the data structure behaves.

Formalization is the process of identifying machine-processable aspects of information, including concepts with their attributes and values and relationships to other concepts [Shipman and McCall, 1994]. In a perfectly formalized representation there is no more hidden meaning or content to be uncovered, all relations between the elements of the representation are explicit and unambiguous [Heylighen, 1990].
There are three levels of formalization on the way from things that exist in the domain of interest to a representation of that domain:

1. The first level is **conceptualization**, which is an interpretation of the domain. Concepts are identified to represent actual instances by centralizing the common properties of these instances and grouping them according to the ontological structure of the concepts. A conceptualization describes an opinion about the domain which subjectively focusses on what is important about the domain and what is not. If the representation was as complex as the domain itself, it would become useless due to the fact that it would only duplicate the domain without any gain of speed or simplicity for reasoning about the domain.

2. The second level is **specification**, which is a precise definition of the concepts and relations between them. A specification contains attributes of the concepts and relations that exactly denote which concepts are related to one another. There may be more than one way to specify a conceptualization. A not very detailed specification could potentially refer to different conceptualizations. Such ambiguity is not admissible for the configuration task, however, because the configured concept instances are intended to be assembled in reality and therefore actual components must be identifiable based on the specification. Hence, a detailed and precise specification is a premise for building configuration models.

3. The third level is **representation**, which is the actual formalism in which the specification is expressed. Different representations can be different modeling languages, for example. A language is expressive enough to be suitable for modeling product knowledge if it provides an unambiguous and complete representation of the underlying specification. This means that although there may be multiple representations for one specification, the representation refers to exactly one specification and thus to the underlying conceptualization.

![Figure 3.1: Representation Layers.](image-url)
3.1 Of Ontologies, KB’s and Configuration Models

Figure 3.1 depicts how the three levels of formalization influence a representation of the underlying domain. There may be multiple conceptualizations for one domain as a conceptualization is a subjective interpretation of the domain. More than one specification can define the concepts and their attributes and relations to other concepts based on the same conceptualization. And there may be multiple ways to represent a specification. A representation may be textual or graphical and may be based on different languages. \( l \times m \times n \) potential representations of a single domain.

A knowledge representation formalism should be able to represent all relevant knowledge of a given application domain. A declarative semantics defines the meaning of the represented objects independently from programs that reason about this knowledge. Usually such a declarative semantics is given by a one-to-one correspondence between instances of concept definitions and corresponding real-world objects [Mylopoulos, 1990]. The structure of the model can be seen as a homomorphic map of the domain structure [Heylighen, 2001].

The model can be more or less adequate for a given domain, but never complete in the sense that it covers all potentially relevant criteria. “If we want to be able to represent anything, then we get further and further from the practicalities of frame organization, and deeper and deeper into the quagmire of logic and philosophy” [McDermott, 1993].

According to [Wartofsky, 1979], there are no intrinsic relational properties that mark out particular entities as being suitable for representation. Anything can be used to represent anything else. The efficiency of a model depends on its mathematical form, which must be sufficiently detailed to embody all information needed for an unambiguous solution to the problem, but not so complex as to make the search for this solution needlessly difficult. A basic idea from the field of Artificial Intelligence (AI) is that mental activities can be reduced to some form of information processing and that this process is independent of the detailed physical structure of the environment [Heylighen, 1990]. For the configuration task the most important aspect is that components of the domain can be identified based on selected objects from the representation.

3.2 Of Ontologies, Knowledge Bases and Configuration Models

This section discusses the terms ontology, knowledge base and configuration model. These terms are sometimes used interchangeably in literature, but the original intentions are different. To prevent from irritations, we briefly describe the three terms are discuss their use within this thesis.

Ontology

The word ontology has gained popularity within the knowledge engineering community since research in the area of the Semantic Web gains interest.
The term “ontology” stems from philosophy, notably from early Greece. Plato and Aristotle contend that all nouns refer to entities. Ontology studies being or existence and their basic categories and relationships, to determine what entities and what types of entities exist.\footnote{http://en.wikipedia.org/wiki/Ontology} Ontology thus has a strong implication on conceptualizations of reality. Following the Aristotelian paradigm, every conceptualization has to start from entities that are actually present. The Platonic paradigm, on the other hand, assumes the primacy of abstract ideas. The latter is called \textit{modus operandi} and is typical for knowledge representation in Description Logics (see Section 3.4).

In computer science, an ontology is a data model that formally defines a common set of terms that are used to describe and represent a domain. Ontologies are intended to be used by automated tools to improve services like search or knowledge management and to draw inferences, i.e. getting new information from old (reasoning and inferencing is detailed in Subsection 3.7). \textit{Ontology engineering} describes building a common vocabulary, providing backbone information, answering competence questions (i.e. inferencing), standardization of terminology or meanings of concepts, transformation of data bases considering the differences of conceptual schemata, and reusing knowledge [Mizoguchi and Ikeda, 1996].

[Guarino and Giaretta, 1995] identifies two roles of an ontology: \textit{conceptualization} to denote a semantic structure which reflects a particular conceptual system, and \textit{ontology theory} to denote a logical theory intended to express ontology knowledge. The underlying intuition is that ontological theories resemble knowledge bases. Conceptualizations are the semantic counterpart: the same ontological theory may commit to different conceptualizations, and vice versa.

A conceptualization can be seen analogous to a database schema. But interestingly, in the database community the distinction between the specification of the conceptual vocabulary (the database schema) and content (the database itself) is apparently clearer than for knowledge bases [Barley et al., 1997].

Ontologies provide a "shared and common" understanding of a domain and facilitate "knowledge sharing and reuse" [Fensel, 2001]. An ontology is an explicit specification of a \textit{conceptualization}: the objects and other entities that are assumed to exist in some area of interest and the relationships that hold among them [Genesereth and Nilsson, 1987, Gruber, 1992].

Standardization of knowledge formats facilitates model-based problem solving. And after all, one of the basic mechanisms of human knowledge representation and processing is the division of the world into classes or concepts which usually are given with a hierarchical structure [Baader et al., 1991]. Individuals of the world are represented as instances of these concepts.

**Knowledge Base**

A knowledge base represents a particular domain or field of knowledge. It provides the means for computerized collection, organization, and retrieval of
knowledge. Simple knowledge bases are commonly used to capture explicit knowledge while more advanced knowledge bases are also capable of having automated deductive reasoning applied to them for deriving new knowledge from the explicitly stored knowledge.

A knowledge base may use an ontology to specify its structure (entity types and relationships) and its classification scheme [Gruber, 1991]. An ontology, together with a set of instances of its classes constitutes a knowledge base. Therefore, a knowledge base consists of two components: the ontology containing a domain-independent vocabulary, and the "core" containing knowledge about the domain. An ontology need only describe a vocabulary for talking about a domain, whereas a knowledge base may include the knowledge needed to solve a problem or answer arbitrary queries about a domain [McCarthy and Hayes, 1969].

The representation effort is reused and pays off when instantiated for new applications. But the benefit does not come from reusing entire knowledge bases. Knowledge-based systems will always require application-specific extensions [Chaudhri et al., 1998]. General objects and relationships underlying application-specific facts can be identified to organize knowledge and enable inheritance from these constructs. Different names exist for this extraction of general, domain-independent knowledge, for example upper ontology [Russell and Norvig, 2003] or top-level ontology [Guarino, 1998].

Configuration Model

What is a configuration model? The purpose of a configuration model is not to represent the whole world, but rather a part of it – the so-called domain. A domain is a specific area of interest, that part of the universe about which the system is reasoning. This means that the rest of the universe can be ignored and need not be represented. It has no direct implications on the system [Heylighen, 1990]. For product configuration the application domain is the area in which products are configured using the configuration model. A configuration model does not allow different interpretations of the product domain. This would result in ambiguity for deciding which components to assemble for the desired product.

Definition. A configuration model is a representation of the product domain, its components and properties, and the structure according to which the components can be assembled to form a product.

Discussion

There are similarities between configuration model, ontology and knowledge base; but each of them has its own purpose and properties:

---

2 [Stojanovic, 2004] gave this definition of a domain being a specific area of interest for ontologies.
• A configuration model does not contain both concepts and instances, like ontologies and knowledge bases do. [Noy and Klein, 2004] argue that in ontologies schema and data are often intermingled in such a way that it is hard to distinguish where the ontology ends and where the instances begin. For the purpose of configuration, concept instances are intended to be instantiated dynamically during the configuration process.

• Knowledge bases are systems that maintain knowledge about the domain of interest. Typically, they represent beliefs about the domain, in propositional logic or other formalizations. These beliefs can both support and decline facts, i.e. contain facts and negation of facts, respectively. A configuration model represents components from which products can be assembled in a closed world and does not describe information about potential components that are absent in the domain of concern. This means that a configuration model specifies supporting facts only, no declining facts.

• Configuration models comprise conceptual representations of components and constraints that describe restrictions on ways in which components can be combined. This means that reasoning about configuration models is somewhat more complex.

The following sections discuss modeling facilities developed for model-based reasoning and structure-based configuration in particular.

### 3.3 Foundation for Configuration

A lot of effort has been put into formalizing configuration models over the last decades. Different ideas for modeling knowledge about a domain of concern have come up. But most formalisms have a common basis. A configuration model (sometimes called *product model* or *system model*) uniquely identifies the components of a domain, their properties and structure.

Conceptualizing a domain, specifying the conceptualization, representing the specification in a configuration model, and configuring products involve four levels of reasoning [Floch and Gulla, 1996, Männistö and Sulonen, 1999]:

• Basic facilities are *objects, characteristics, inheritance*, etc.

• Modeling facilities that use these basic facilities are *concepts, attributes, specialization relations, composition relations* and *constraints*.

• A *configuration model* formalizes knowledge about a product domain by implicitly representing all potentially configurable products.

• *Product individuals* are assembled from *instances* of concepts that are instantiated according to the concept definitions and the given requirements.
The modeling facilities used for representing product domains in structure-based configuration have emerged from early work on frames, semantic networks and concept languages.

*Frames* have been introduced by [Minsky, 1974] as data structures that represent prototypical situations and objects. A frame collects all necessary information for treating a situation or object in one place. The notion of *concepts* for denoting objects in the domain of concern is derived from frames.

*Semantic networks* [Quillian, 1968] have been developed for representing the semantics of natural language. But they allow to represent concepts of any kind in a graph structure where nodes describe concepts and edges assign properties to concepts (property edges) and introduce hierarchical relationships among concepts (is-a edges). Properties are inherited along is-a edges. The formalism of semantic networks has greatly influenced knowledge representation, most notably Description Logics.

The knowledge representation system KL-ONE was the first to use a *concept language* [Brachman and Schmolze, 1985]. Subsequently many systems based on the idea of KL-ONE have been built [Baader and Hollunder, 1991], most notably the family of Description Logics. Central formalism of Description Logics is term *subsumption*: a relation between two terms denoting a generalization / specialization in the sense that a more general concept subsumes a more specific concept. The formal model-theoretic semantics of concept languages provide means for investigating soundness and completeness of inference algorithms [Brachman and Levesque, 1984].

[Calvanese et al., 1999] show that the core of most conceptual modeling formalisms is very similar and can be transformed into one another. The Description Logic $\text{ALUNI}$ acts as basic formalism and the authors define mappings that map elements from *frame-based systems* [Fikes and Kehler, 1985], semantic data models like the *Entity-Relationship model* (*ER* model) [Chen, 1976], and *object-oriented* (*OO*) data models [Kim, 1990] to the corresponding elements of $\text{ALUNI}$.

Knowledge-based systems that rely on Description Logics are especially appropriate for applications that have an evolving schema or that involve incrementally evolving descriptions allowing the user to maintain a partial, incomplete view of the domain of discourse in which information is incrementally acquired. Advanced reasoning mechanisms like classification and consistency checking facilitate dealing with both evolving schemata and incrementally evolving descriptions, which are the normal state of affairs in design and configuration specification efforts [Brachman et al., 1990]. [Borgida, 1995] indicates how one can achieve enhanced access to knowledge by using Description Logics for schema design and integration, queries, answers, updates, rules, and constraints. [Schröder et al., 1996] use a Description-Logics based approach for a theoretical analysis of the configuration tools PLAKON [Cunis et al., 1989] and its successor KONWERK [Günter and Hotz, 1999].
3.4 Description Logics

*Description Logics (DLs)* are a family of knowledge representation languages that have been studied extensively in Artificial Intelligence over the last two decades. A Description Logic is an object-oriented knowledge representation formalism for modeling a domain in terms of *concepts, roles* (properties of concepts) and *individuals* (instances of concepts). Concepts are specified by necessary and sufficient conditions satisfied by individuals in the set. These conditions are specified in *concept expressions* that are built from a set of *constructors* [Baader et al., 2003].

The choice of concept constructors is tailored to the expressive purpose at hand, tempered by the desired computational properties of the reasoner, especially its decidability [Nebel and Smolka, 1991]. Choosing a subset of concept constructors leads to Description Logics of more restricted expressiveness, but at the same time more efficient reasoning [Brachman and Levesque, 1984].

3.4.1 The Knowledge Representation Language

**SWRL-ALCQI+(D)**

It is understood that much of the product configuration process involves recognition, formulation and satisfaction of constraints [Lin and Chen, 2002]. In this section we present the hybrid knowledge representation language SWRL-\(\text{ALCQI}+(D)\) that consists of the Description Logic \(\text{ALCQI}+(D)\) for representing conceptual knowledge, and a SWRL component for expressing quantified constraints.

The basic idea underlying the hybrid formalism of integrating rules and Description Logics (often called \(r\)-hybrid systems [Rosati, 2005]) is to deal with a common representation of the domain constituted by a structural component (the \(\text{ALCQI}+(D)\) component) and a rule component (the SWRL component). The interaction between the two subsystems is obtained by allowing variables in SWRL rules to range over the set of instances of a specified concept of the corresponding Description Logic.

3.4.2 The \(\text{ALCQI}+(D)\) Component

The Description Logic \(\text{ALCQI}+\) is the standard attributive language \(\text{AL}\) plus support for reasoning with complements (\(C\)), quantified number restrictions (\(Q\)) and inverse roles (\(I\)). The + stands for transitive roles that are required for reasoning about whole subtrees of concepts.

Fillers of *functional roles* are expressed by using *concrete domains* (\(D\)) that represent values of number or string domains. Functional roles describe properties that inherently belong to an individual and are not defined by related individuals; henceforth functional roles are called *attributes*.3
3.4. Description Logics

Considered to be already sufficiently structured by the predicates \(<, \leq, =, \neq, >, \geq\), etc. Therefore, it is not appropriate to form new classes of concrete objects (values) using the concept language [Baader and Hanschke, 1991]. In addition, a direct representation makes it possible to use existing reasoners of the concrete domain. Knowledge representation in CLASSIC [Brachman et al., 1990], for example, allows to refer to so-called host values, i.e. concrete individuals that are interpreted by the underlying programming language.

Semantics

The semantics of the Description Logic \(\mathcal{ALCQI}+(\mathcal{D})\) is given in terms of a Tarski style model theoretic semantics using an interpretation. Concepts are interpreted as subsets of a domain and roles as binary relations over that domain. Intuitively, \(\neg C\) represents negation is interpreted as set complement with respect to the domain of interpretation. \(C \cap D\) represents the conjunction of two concepts and is interpreted as set intersection, while \(C \cup D\) represents the disjunction of two concepts and is interpreted as set union. \(\exists r \cdot C\) is called existential quantification and \(\forall r \cdot C\) is called universal quantification over roles and denote those objects of the interpretation domain that are connected through role \(r\) only to instances of the concept \(C\). \((\geq n \ r)\) and \((\leq n \ r)\) are called number restrictions and impose restrictions on the minimum and maximum number of objects to which their instances are connected through role \(r\). When the minimum and maximum number are the same, \((= n \ r)\) denotes an exact number. \(r^{-}\) is the inverse role and represents the inverse of the binary relation denoted by \(r\).

Table 3.1 shows the concrete syntax and semantics of our conceptual language. More formally, an interpretation \(\mathcal{I} = (\Delta^I, \Delta^D, \cdot^I)\) consists of an abstract domain \(\Delta^I\), a concrete domain \(\Delta^D\) and an interpretation function \(\cdot^I\). The abstract and concrete domains must be disjoint, i.e. \(\Delta^I \cap \Delta^D = \emptyset\). The interpretation function maps every concept \(C\) to a subset \(C^\mathcal{I}\) of \(\Delta^I\) and every role \(r\) to a subset \(r^\mathcal{I}\) of \(\Delta^I \times (\Delta^I \cup \Delta^D)\) according to the semantic rules specified in Table 3.1. \#M denotes the cardinality of \(M\). The ordering on strings is the standard lexicographic one.

Table 3.2 shows terminological and assertional axioms. Concept inclusion is denoted by \(C \sqsubseteq D\) (concept \(C\) is a subset of concept \(D\)) and concept equivalence is denoted by \(C \equiv D\). An element \(a\) of concept \(C\) is asserted by expressing \(C(a)\) and two elements \(a, b\) that are related via role \(r\) are asserted by expressing \(r(a, b)\) (element \(b\) is the filler of role \(r\) defined for element \(a\)).

We assume a set of concept names \(N_C\), a set of property names \(N_P\) and a set of constraint names \(N_T\). The set of property names consists of two disjoint subsets \(N_A\) and \(N_R\) denoting attribute names and composition relation names, respectively \((N_P = N_A \cup N_R)\). There is exactly one root concept \(\top\), which is the topmost concept. The root concept is the only concept definition without a parent.
3. Knowledge Representation

Constructor | Syntax | Semantics |
---|---|---|
**top** | $\top$ | $\Delta^I$ |
**bottom** | $\bot$ | $\emptyset$ |
**concept** | $C$ | $C^I \subseteq \Delta^I$ |
**negation** | $\neg C$ | $\Delta^I \setminus C^I$ |
**conjunction** | $C \cap D$ | $C^I \cap D^I$ |
**disjunction** | $C \cup D$ | $C^I \cup D^I$ |
**role** | $r$ | $r^I \subseteq \Delta^I \times (\Delta^I \cup \Delta^D)$ |
**inverse role** | $r^-$ | $\{(i, i') \mid (i', i) \in r^I\}$ |
**transitive closure** | $r^+$ | $\bigcup_{n \geq 1} (r^I)^n$ |
**existential quantification** | $\exists r.C$ | $\{i \mid \exists i'.(i, i') \in r^I \land i' \in C^I\}$ |
**universal quantification** | $\forall r.C$ | $\{i \mid \forall i'.(i, i') \in r^I \Rightarrow i' \in C^I\}$ |
**qualified number restriction** | $(\geq n \, r.C)$ | $\{i \mid \#{i'.(i, i')} \in r^I \land i' \in C^I\} \geq n$ |
**concrete value** | $\exists r.d$ | $\{i \mid \exists i'.(i, i') \in r^I \land i' \in \Delta^D \land d(i')\}$ |
**concrete value-type** | $\forall r.d$ | $\{i \mid \exists i'.(i, i') \in r^I \Rightarrow i' \in \Delta^D \land d(i')\}$ |

Table 3.1: Concrete syntax and semantics of constructors from the Description Logic $\mathcal{ALCQI}+(D)$.

<table>
<thead>
<tr>
<th>Axiom</th>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>concept inclusion</td>
<td>$C \sqsubseteq D$</td>
<td>$C^I \subseteq D^I$</td>
</tr>
<tr>
<td>concept equality</td>
<td>$C \equiv D$</td>
<td>$C^I = D^I$</td>
</tr>
<tr>
<td>concept assertion</td>
<td>$C(a)$</td>
<td>$a^I \in C^I$</td>
</tr>
<tr>
<td>role assertion</td>
<td>$r(a, b)$</td>
<td>$(a^I, b^I) \in r^I$</td>
</tr>
</tbody>
</table>

Table 3.2: Concrete syntax and semantics of terminological and assertional axioms from the Description Logic $\mathcal{ALCQI}+(D)$.

3.4.3 The SWRL Component

*Semantic Web Rule Language (SWRL)* [Horrocks and Patel-Schneider, 2004] is the proposed standard for specifying rules in future releases of the *Web Ontology Language (OWL)*. A SWRL rule has the basic form of an implication, i.e. “if $A$ then $C$” (or using the concrete syntax: “$A \Rightarrow C$”), where $A$ is called antecedent and $C$ is called consequent. The antecedent defines a set of variables in conjunctive normal form, where some variables may be universally or existentially quantified. The consequent restricts possible values for some of the variables.

Decidability and complexity of r-hybrid systems are a crucial issue. In fact, the interaction does not preserve decidability: starting from a decidable DL and a decidable rule component, reasoning in the system obtained by combining the two components may not be a decidable problem. However, we can trade in a little expressivity for decidability: the combination of any Description Logic
with DL-safe rules is proven to be decidable [Rosati, 2005]. DL-safe rules allow classes and properties from the Description Logic component to appear freely in the antecedent or consequent of the rule, with the only restriction being that they can be applied only to explicitly named instances.

### Semantics

The model-theoretic semantics for SWRL is a straightforward extension of the semantics for the Description Logic given in the previous subsection. The basic idea is that we define an extension of the interpretation that also maps variables to elements of the domain (see Table 3.3).

<table>
<thead>
<tr>
<th>Atom</th>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>existential quantification</td>
<td>$\exists x$</td>
<td>${ ?x^I \mid # { ?x^I \in (\Delta^I \cup \Delta^D) } \geq 1 }$</td>
</tr>
<tr>
<td>universal quantification</td>
<td>$\forall x$</td>
<td>${ ?x^I \mid ?x^I \in (\Delta^I \cup \Delta^D) }$</td>
</tr>
<tr>
<td>concept variable</td>
<td>$C(?x)$</td>
<td>$C \in N_C \land ?x^I \in C^I$</td>
</tr>
<tr>
<td>role variable</td>
<td>$r(?x, ?y)$</td>
<td>$r \in N_P \land (\forall x^I, ?y^I) \in r^I$</td>
</tr>
</tbody>
</table>

Table 3.3: Concrete syntax and semantics of atom satisfaction for SWRL constraints.

More formally, an interpretation $I$ satisfies an antecedent $A$ if, and only if, $A$ is empty or $I$ satisfies every atom in $A$. An interpretation $I$ satisfies a consequent $C$ if, and only if, $C$ is not empty and $I$ satisfies every atom in $C$. A constraint is satisfied by an interpretation $I$ if, and only if, for every satisfied antecedent, the consequent is also satisfied.

The semantic conditions that define an interpretation are unchanged: an interpretation satisfies the terminological and assertional knowledge if, and only if, it satisfies every axiom (including SWRL rules).

### 3.5 Modeling Facilities

This section presents a more product-modeling centered terminology and maps modeling facilities for specifying a product domain to the corresponding elements of the knowledge representation language SWRL-$\text{ALCQI}^+(D)$. This mapping provides formal grounds for the knowledge representation of product knowledge.

The language presented in the following subsections will be used throughout the remainder of this thesis for representing and reasoning about configurable products. In some places the semantics on which we rely slightly deviates from the typical semantics known from Description Logics. These places are indicated and the deviation is described accordingly.
3.5.1 Concepts

A concept is a description which gathers common features of a set of objects in the domain. Concepts are interpreted as sets, which means that concept conjunction can be interpreted as set intersection, concept disjunction as set union and negation of concepts as set complement [Baader and Hollunder, 1991].

Concepts are intersubjectively realizable (and in that sense objective) cognitive structures. They are not objects (and therefore not individuals) but rather patterns that are instantiated (as individuals) in a subjective process [Cocchiarella, 1995]. Often, the terms object, class, frame, or simply item or component are used to describe the same or a similar meaning. But throughout this work we stick to the term concept because it makes a clear distinction between the conceptualization (the abstract meaning) and the instantiation (actual instances) of objects.

Concepts are modeled containing two different hierarchical relationships: is-a and has-parts. The taxonomic is-a relation and the partonomic has-parts relation are processed differently. The former addresses commonalities and differences of the concept definitions while the latter involves spatio-temporal and functional correlation. For a better understanding we sketch the basic notions of the two relations now. A detailed definition follows in Subsections 3.5.3 and 3.5.5, respectively.

- **Is-a** is used to model a generalization / specialization hierarchy, also called taxonomy. Every object has exactly one ancestor and can have an arbitrary number of descendants. Hence, the taxonomy is a tree structure.

- **Has-parts** forms compositions into a composition hierarchy, also called partonomy. The inverse relation of a has-parts relation is called part-of relation. Objects are either primitive (sometimes called atomic) or composite. This means that they reside at the leaves of the composition hierarchy or are the root of a subgraph, respectively.

Every concept carries a unique name which identifies it within the domain, specifies exactly one parent concept of which it is a specialization, if not the root concept, and an arbitrary number of attributes and composition relations, collectively called properties. Concept names are denoted with upper case strings, e.g. C, D or Car.

**Note.** Description Logics allow to specify multiple parents for a single concept definition. We restrict the expressivity of the taxonomy to tree structures in order to reduce computational complexity. The possibilities to represent a product domain are only slightly restricted. In most cases components can be grouped according to their functionality and modeled in tree structures without loss of expressivity.
3.5.2 Instances

Concepts describe a set of objects to which multiple concept instances may belong. Instances are instance of concepts (e.g. $i \in C$) and inherit all properties from the concept definitions. The important property of instances is that they have an identity, which allows them to be distinguished from one another and to be counted. This means that instances with different names, even with the same definition, will be different instances. Instance names are denoted with lower case strings, e.g. $C(i)$ or Car(myCar).

Concepts are static descriptions that are not altered during the configuration process while instances are created dynamically within the process and configured until their properties are fully specified according to the configuration problem. Property values may be partly specified and are only allowed to specify subsets of the original values defined in the concepts of which the instance is an instance.

The instance-of relation is a binary relation that maps instances to concepts [Chaudhri et al., 1998]. A concept instance is direct instance of at least one concept, whereas it is indirect instance also of all superconcepts of these concepts.

3.5.3 Specialization Relations

Concept inclusion (or hyponymy) describes that a concept is a special type of another concept, for example “cars are types of vehicles”. The more special type of concept is subsumed by the more general type of concept. With specialization relations a taxonomy of concepts can be created helping to avoid the specification of redundant information because all subconcepts of a concept inherit all properties defined for this concept. Inheritance is a way to define new concepts using concepts that have already been defined. The new concept, also known as derived concept, takes over (i.e. inherits) properties of the pre-existing concept, which is referred to as its parent. Inheritance is typically accomplished either by refining one or more properties exposed by the parent, or by adding new properties to those exposed by an parent.\(^4\)

**Note.** The notion of property inheritance along the specialization hierarchy facilitates top-down configuration approaches due to the fact that restrictions that apply to instances of a concept must also apply to instances of all specializations of that concept.

The taxonomic hierarchy is defined by $D \sqsubseteq C$. Concept $C$ is called parent and concept $D$ is called child. The specialization relation is transitive: concept $D$ is a direct subconcept of concept $C$ and an indirect subconcept of the parent of $C$, and so on. The transitive closure over the taxonomy can be computed with: $D \sqsubseteq C, E \sqsubseteq D \rightarrow E \sqsubseteq C$.

A concept $D$ is a subconcept of concept $C$ if, and only if, every potential instance of $D$ is also an instance of $C$ ($i \in D \rightarrow i \in C$). Forming the logical

\(^4\) [http://en.wikipedia.org/wiki/Inheritance_(computer_science)]
conjunction of two concepts is enough to decide over subsumption: \( C \) subsumes \( D \) if, and only if, \( C \sqcap D \equiv D \). A superconcept relation (also called is-a relation or generalization) between concepts is defined as the inverse of the subconcept relation (also called specialization).

The taxonomy is well-formed if, and only if, it does not contain cycles. A concept has exactly one parent (if not the root concept), and can have an arbitrary number of children. Multiple inheritance is explicitly ruled out with the definition of a tree structure.

**Note.** Description Logics allow multiple inheritance by specifying more than one concept as parents of a concept definition. By restricting the taxonomy to be a tree structure (see the definition of concepts) we rule out multiple inheritance.

### 3.5.4 Attributes

Attributes define characteristics of concepts and are denoted as roles with lower case names and a concrete domain as filler, e.g. \( a: \text{Integer}, b: \text{Real} \) or \( \text{color}: \text{String} \). The name is uniquely identifiable within the taxonomy.

Unary predicates of a concrete domain are used to restrict the value of an attribute (for example \( \text{size} \geq 17 \) states that the value of the size attribute must be at least 17, which may be a valid restriction for a Tire) and binary predicates of a concrete domain are used to compare the values of two attributes (we will see examples for binary predicates of a concrete domain later, in Subsection 3.5.6).

An attribute value is restricted to a set of pre-defined value domains. Three value domains are pre-defined for specifying attribute values: integer numbers, real numbers and strings. In implementations, however, the potential values of concrete domains are dictated by the programming language.

**Integers** The domain of integer numbers \( \mathbb{Z} \) can be interpreted by the binary predicates \( <, \leq, =, \neq, >, \geq \) and the unary predicates \( <, \leq, =, \neq, >, \geq \) for \( z \in \mathbb{Z} \). Integer numbers \( z \in \mathbb{Z} \), integer intervals \( [z_1; z_2] \) with \( z_1, z_2 \in \mathbb{Z} \land z_1 \leq z_2 \), and integer sets \( \{z_1, \ldots, z_n\} \) with \( z_1, \ldots, z_n \in \mathbb{Z} \land z_i \neq z_j \forall 1 \leq i, j \leq n \) can be specified.

**Reals** The domain of real numbers \( \mathbb{R} \) can be interpreted by the binary predicates \( <, \leq, =, \neq, >, \geq \) and the unary predicates \( <, \leq, =, \neq, >, \geq \) for \( r \in \mathbb{R} \). Real numbers \( r \in \mathbb{R} \), real intervals \( [r_1; r_2] \) with \( r_1, r_2 \in \mathbb{R} \land r_1 \leq r_2 \), and real sets \( \{r_1, \ldots, r_n\} \) with \( r_1, \ldots, r_n \in \mathbb{R} \land r_i \neq r_j \forall 1 \leq i, j \leq n \) can be specified.

**Strings** The domain of strings \( S \) includes sequences of symbols a-z, A-Z, 0-9, -, _, whitespace and further special characters like braces. It can be interpreted by the binary predicates \( =, \neq \) and the unary predicates \( =_s, \neq_s \) for \( s \in S \). Strings \( s \in S \) and string sets \( \{s_1, \ldots, s_n\} \) with \( s_1, \ldots, s_n \in S \) can be specified.