An Evaluation of Description Logic for the Development of Product Models

Xenia Fiorentini, Sudarsan Rachuri, Hyowon Suh, Jaehyun Lee, Ram D Sriram

Manufacturing Systems Integration Division,
Design Process Group,
National Institute of Standards and Technology,
Gaithersburg, MD 20899, USA
{xenia.fiorentini, rachuri.sudarsan, hyowon.suh, lee.jaehyun, ram.sriram}@nist.gov

Abstract

The languages and logical formalisms developed by information scientists and logicians concentrate on the theory of languages and logical theorem proving. These languages, when used by domain experts to represent their domain of discourse, most often have issues related to the level of expressiveness and need specific extensions. In this paper we first analyze the requirements for the development of structured knowledge representation models for manufacturing products using ontologies. We then explore how these requirements can be satisfied through the levels of logical formalisms and expressivity of a structured knowledge representation model. We report our evaluation of Description Logic (DL) with respect to the requirements by giving an example of a product ontology developed with OWL (Ontology Web Language-Description Logic). In order to represent a product, we also need to combine both DL expressivity and domain-specific rules. Domain-specific rules are defined to add specific constraints in the knowledge base and we have used SWRL (Semantic Web Rule Language) for this purpose. We present a case study of an electro-mechanical product to validate the evaluation and further show how the OWL-DL reasoner together with the rule engine can enable reasoning of the product ontology. We finally discuss the open issues such as capabilities and limitations related to the usage of DL, OWL and SWRL for product modeling.

1 Introduction

In a typical industrial scenario a number of enterprises collaborate to accomplish various tasks by sharing resources, applications and services infrastructure throughout the product lifecycle. The elements that describe this scenario can primarily be grouped into i) entities, e.g., applications, persons and enterprises and ii) connections between these entities, e.g., data exchange and collaborations. In this network of entities, product models play a crucial role in achieving interoperability. To work together, the entities have to share a common model through which they can communicate. For example geometry representation models, such as STEP (Standard for the Exchange of Product model data) AP 203 [1], are used to exchange product geometry information between CAD systems, PDM (Product Data Management) schema [2] is used to share product data (identification, classification, structure and relationships, and properties of parts, work management data, etc.) between engineering information systems.

In this landscape of product models, semantic models can enable entities to achieve reliable and efficient collaboration. The tasks achievable with the semantic model include inferring new knowledge, querying, retrieving, and storing. To facilitate knowledge sharing, semantic models are used e.g., to eliminate ambiguities and to enable knowledge reuse by making domain assumptions explicit.

Developing product semantic models requires knowledge and information from different fields: domain knowledge, product modeling, programming, knowledge representation reasoning, etc. In this paper we focus on logic-based representation, which is currently in vogue for the semantic web applications.

Our approach is to match the requirements for product modeling to the expressivity provided by description logic. To meet the product modeling requirements we also need domain-specific rules. We provide some examples taken from a semantic model that we developed to test the expressivity of description logic (DL) for product modeling (Core Product Model and Open Assembly Model [3]). We hope to provide a critical and impartial assessment of the use of DL for product modeling.

This paper is structured as follows. In section 2 we review previous product ontologies and their representations, and emphasize the necessity of our evaluation. In section 3 we analyze the expressivity requirements for product model representation. In section 4 we describe how the expressivity of DL can support the expressivity requirements for the product model. OWL-DL 1.0 (Ontology Web Language) [4] and Protégé (tool) are currently very popular with ontology researchers. We discuss how these two meet DL requirements in section 5. In section 6 we introduce CPM/OAM as product model for evaluation. Section 7 presents how the DL expressivity is used in representing the product model. Section 8 provides some examples of reasoning mechanisms based on DL and domain specific rules. Section 9 describes the expressivity requirements that cannot be satisfied either with DL or with domain specific rules. The modeling language is included in our evaluation. We provide our conclusions in Section 10.

We use the following notations: classes are in Arial font (Artifact), the properties are in Arial font with leading character in lowercase (partOf), individuals are in Arial italics font with the leading character in lowercase (*myCar*).

2 Previous Research on Product Ontologies

Many of the product ontologies that are available in literature can be categorized into two groups according to their modeling scope. One is specific product ontologies, and another is generic product ontologies. Since the 1990's, many specific product ontologies have been proposed, and each of them was represented or implemented in different languages. So, specific product ontologies can be further classified according to their languages' expressivity. We classified them into three types: semi-structured representation, object-oriented representation, logic/rule representation.

STEP -the Standard for the Exchange of Product model data- is a comprehensive ISO standard (ISO-10303) that describes how to represent and exchange digital product information [5]. STEP has been very successful in specifying data models for domains such as solid modeling and finite-element geometry. STEP uses the EXPRESS language to represent information models. Product Data Markup Language (PDML) is a set of XML vocabularies for deploying product data on the internet [6]. PDML adopted STEP Integrated Resources for its information model, and it used XML instead of EXPRESS.

CONGEN [7], SHARED [8] and OAM product models [9] were specific product ontologies represented in an object-oriented representation. CONGEN (a framework for conceptual design) is the design application shell implemented as a part of the MIT DICE project. It provided a design knowledge representation scheme, and maintained design alternatives and context information. A product-process model, which included primitives such as context, specification, artifact, function, form, behavior, decision, goal and plan, formed the basis for the design knowledge representation. The DICE project also developed comprehensive engineering knowledge bases, called SHARED. Sudarsan et al. [9] proposed the Open Assembly Model (OAM) to provide a standard representation and exchange protocol for assembly and system-level tolerance information. OAM was extensible: it provided for tolerance representation and propagation, representation of kinematics, and engineering analysis at the system level. OAM was based on the Core Product Model (CPM), and it was represented in UML.

Generic product ontologies, such as the MOKA [10] product model and the NIST Core Product Model (CPM), were represented in object-oriented languages such as UML. The MOKA product model [11] supported five distinct views of a product: 1) structure, 2) function, 3) behavior, 4) technology, and 5) representation. In MOKA the product informal model was described using a class diagram in UML. Fenves et al. [12] proposed the CPM to provide a base-level product model that is open, nonproprietary, generic, extensible, independent of any one product development process and capable of capturing the full engineering context commonly shared in product development. Initially, they used UML to represent CPM, and mapped it to OWL-DL later on.

In the past decades, a number of researchers have developed product ontologies using logic-based formalisms. We discuss a few representative projects. There were also many specific product ontologies represented in logic or rule languages. Lin et al. [13] proposed a requirement ontology to manage customers’ requirements, product specifications and relations among them. They defined the requirement ontology in first order logic (FOL), and implemented it in Prolog in an object-oriented fashion. Borst and Akkermans [14] developed an engineering ontology based on the PHYSYS ontology. The PHYSYS ontology included three conceptual viewpoints on physical systems: 1) system layout, 2) physical processes underlying behavior, and 3) descriptive mathematical relations. They represented the ontology in KIF (Knowledge Interchange Format) [15]. Kitamura et al. [16] designed a functional concept ontology, which provided a rich vocabulary for functional representation. They applied the functional concept ontology to the automatic identification of functional structure of an existing product. They implemented their functional understanding system using Lisp. Kim et al. [17] defined an assembly ontology by enhancing the assembly relation mode using ontologies. Their ontology represents engineering, spatial, assembly, and joining relations in an assembly. They implemented their ontology using both OWL and SWRL.

Although the above product ontologies worked well for their specific purposes, a generic product ontology is still required to express product lifecycle knowledge uniformly and consistently. The concepts and relations involved in a generic product ontology have semantically different levels of abstraction. For example, a product's geometry or form are physical concepts, but the product's requirement, function or behavior are abstract concepts. A generic product ontology should include all of those concepts, and specify their semantics explicitly and logically.

Expressive logic and rule representation are required in order to specify semantics of all concepts and relations in a product lifecycle. However, no rigorous evaluation has been made on the applicability of logic for product representations. Nonetheless, many previous researches proposed their product ontologies using logic representations, and consequently no one knows whether their logic representations had enough expressiveness to represent their product ontologies or not. In the following sections we attempt to do such an evaluation.

3 Expressivity Requirements for Product Representation

In this paper we focus on product models that have to be shared along the whole product lifecycle. These models have to be generic enough to:

- Represent the core characteristics of the product.
- Be independent from specific product development processes.
- Be compatible with the detailed product representation that each application contains.

The development of such a high level interoperable model poses many challenges, such as encoding complex nature of interactions in product modeling and representing semantics. Hence, the information model for representing manufacturing product is inherently complex.

The following partial list sketches some of the issues with respect to each information element:

- *Function:* one aspect of what the artifact is supposed to do. The artifact satisfies the engineering requirements largely through its functions [3].
- *Behavior:* information supporting the simulation of the product under some given conditions. This simulation could be, for example, kinematics, dynamics and control systems.
- *Structure:* the individual parts that constitute the assembly, the hierarchy of the composition tree (parts-subassemblies-assembly) and the associated Bill of Materials (BOM).

- *Geometry and material:* a generic shape, chosen by the designer at early stages of the lifecycle and defined geometry and material captured in one or more CAD (Computer-Aided Design) systems.
- *Features:* a portion of the artifact's form that has some specific functions assigned to it. An artifact may have design features, analysis features, manufacturing features, etc., as determined by their respective functions [3].
- *Tolerances:* tolerance design is the process of deriving a description of geometric tolerance specifications for a product from a given set of desired properties of the product. Tolerancing includes both tolerance analysis and tolerance synthesis [3].

The second source of complexity is due to the abstraction principles needed to represent the information on products. The model may incorporate the following mechanisms [18]:

- *Generalization / specialization:* relationships built through intentional properties, e.g., "gear shifts are mechanical assemblies." This abstraction principle involves a hierarchical mechanism where concepts are categorized through the general knowledge of the problem.
- *Grouping / individualization:* relationships built through extensional properties, e.g., "manual gear shifts are gear shifts." In this case concepts are categorized through the specific knowledge of the represented domain and the group can contain heterogeneous things.
- *Classification / instantiation:* relationships between a real object (instance) and the concept it belongs to, e.g., "gear shift #1 is a gear shift." In modeling, particular attention needs to be paid to the establishment of the boundary between concepts and real objects.
- *Aggregation / decomposition:* part-of relationships between an element and its constituents, e.g., "gear shifts are part-of cars" and "gear shift #1 is part-of car #1."

Other types of part-of relationships may be required, for example, a given chemical compound "consists of" many other chemicals. For more details regarding general part-of formalisms, readers can refer to the General Extensional Mereology [19].

This paper outlines evaluation of the expressiveness of DL to capture both the information content and the abstraction principles discussed above, with the aim of developing a consistent formal model for manufacturing product assemblies. We use the terms expressiveness to mean both language expressiveness and processible expressiveness [20]. The language expressiveness is related to the language symbols, rules, conventions and vocabulary, while the processible expressiveness is related to the computability. For more detailed discussion on the issues related to the computational complexity, please refer to [21] [22].

4 DL expressivity in product modeling

Description Logic (DL) is a family of knowledge representation languages used to represent the knowledge of a domain in a structured fashion. The domain is modeled by means of concepts and roles, which denote, respectively, classes of objects and relationships between objects. The concepts and roles, together with knowledge specification mechanisms, form the knowledge base. Automatic reasoning procedures can be performed on the knowledge base.

DL is decidable, that is, there exists an automatic reasoning procedure such that, for every knowledge specification mechanism in the logic, the reasoning procedure is capable of deciding whether the mechanism is valid or not [23]. DL expressiveness enables explicit information representation and support inference mechanisms, i.e., mechanisms to find implicit consequences based on the explicit information.

4.1 DL for information representation

The DL formalism allows us not only to create concepts but also to create concept level hierarchies of the knowledge using is-a relationships (e.g., car is-a vehicle), to express complex roles (properties) between concepts (e.g., cars have exactly four wheels while bicycles have exactly two wheels) and to declare the membership of an individual in a concept (e.g., car #1 belongs to the concept of cars). Thus, the DL formalism helps us in satisfying some of the requirements inherent to the abstraction principles as well as information elements listed in Section 3:

- *Function, behavior, structure, geometry and material, features, tolerances* can be expressed in concepts and roles
- *Generalization / specialization* can be expressed as taxonomy structures in DL.
- *Grouping / individualization* where we can create in DL a complex concept that represents the grouping of concepts.
- *Classification / instantiation* where DL allows for the declaration of the membership of an individual in a concept.

4.2 DL for inference mechanisms

For inference mechanisms, consider the examples in Table 1. The first two questions are inherent to the generalization and grouping abstraction principles. The third and fourth questions address the classification abstraction principle, while the fifth question deals with the aggregation abstraction principle. In the second column we use the concepts of Vehicle, Car, Bicycle, Wheel and Engine to create our knowledge base and to query it. In the third column we present the DL mechanisms that allow for answers to those queries. In the fourth column we show answers to these queries.

Table 1: Examples of inference mechanisms

	Question	DL mechanisms	Answer
1	We subsume the concept of Car[1] in the concept of Bicycles[2]. Is it logically correct?	The “consistency checking” mechanism finds whether a concept admits at least one individual.	No, the model is inconsistent. There cannot be an individual that has four wheels and is a bicycle at the same time.

2	We introduce the concept of ElectricCar. What is its position in the hierarchy?	The "subsumption" mechanism finds implicit sub-concept relationships.	In the concept's hierarchy, the ElectricCar concept is a sub-concept of Car.
3	We declare *myCar* as an individual of the concept of Vehicle with four wheels. Is it a Car or a Bicycle?	The "realization reasoning" mechanism finds the most specific concept for each individual.	*myCar* has four wheels, so it is an individual of the concept of Car.
4	Which cars have the same kind of wheels?	The "retrieval" mechanism finds the individuals that are instances of a given concept or intersection of concepts.	The set of different instances of Car that have same kind of wheels.
5	We declare a wheel part-of a car but the engine powering that wheel is partOf another car. Is it logically correct?	DL can not help here since DL can not specify restrictions among properties' instances.	The individual of the Wheel concept is connected to the wrong individual of Car. The partOf property between the concepts of Wheel and Car is still correct.
[1] Car is a Vehicle with four wheels [2] Bicycle is a Vehicle with two wheels			

The DL formalism consists of four reasoning mechanisms [23]: consistency checking, subsumption, realization, and retrieval. Each of these provides the answer for one of the first four questions. The fifth question represents a different situation, it falls outside the scope of DL. To answer this question we need to represent appropriate role paths between individuals and not between classes. In other words it is the role path, going from the instance of Wheel to the instance of Car passing through the instance of Engine, which has to be constrained. To answer the fifth question, we have to introduce new elements in the representation: domain-specific rules.

4.3 Domain-specific rules

Domain-specific rules are defined to add specific constraints in a knowledge base. These rules are in the form of implications between an antecedent (body) and a consequent (head): whenever the conditions specified in the antecedent hold, then the conditions specified in the consequent must also hold. These rules not only allow the declaration of the membership of an individual to a concept, but also the declaration of properties with specific constraints between individuals. In the fifth example given in Table 1, a rule can state that if a wheel is powered by an engine and that engine is part-of a car, then the wheel has to be part-of the same car.

In order to represent knowledge in the assembly domain, i.e., to answer all five questions in Table 1, we need to combine both DL expressivity and domain-specific rules.

5 Languages and tools

Several modeling languages and tools can be considered to implement both the DL expressivity and the domain-specific rules.

5.1 Modeling Languages

We discuss the applicability of several commonly used modeling languages for DL expressivity. The candidate languages have been evaluated according to their capability of expressing DL axioms. It is outside the scope of this paper to provide an exhaustive and fair comparison of these modeling languages.

We have evaluated the following languages/frameworks:

- Unified Modeling Language (UML) [24]
- Entity-Relationship diagrams (ERD) [25]
- EXPRESS [26]
- Ontology Web Language (OWL-DL, version 1.0) [4].

In UML, the modeling elements are substantially aligned with the needs of object-oriented programming. UML, if evaluated according to its DL expressivity, lacks several features, for example, transitive and reflexive properties, which can be achieved only through Object Constraint Language- [OCL] constraints [27]. On the other hand, the expressivity of UML is enhanced by its meta-modeling architecture called Meta-Object Facility (MOF). This architecture is organized in four layers, from M3 to M0, where each layer provides precise constructs and rules for creating models in the successive layers [28]. The meta-modeling expressivity of UML is not currently considered in our evaluation and it will be carried out in the future.

ERD was developed for the organization of information within databases, therefore the correspondence with DL expressivity is even lower than for UML [25]. ERD, for example, does not provide any construct to express enumerated classes and negation.

In EXPRESS, the correspondence with DL is not high. EXPRESS misses some DL constructs, such as properties hierarchies and disjointness[1]. The expressive power of EXPRESS, on the other hand, is enhanced with algorithms not captured in the DL expressivity. These algorithms define the entities' behavior using functions, procedures, and rules.

OWL-DL was specifically created to express the DL constructs; therefore it is the most appropriate for our purpose. Each DL sublanguage is named with a combination of letters (acronyms), e.g., $\mathcal{ALC}$, $\mathcal{SHOIN}$, and $\mathcal{SHIQ}$,: each letter denotes its expressivity. OWL-DL 1.0 is classified as $\mathcal{SHOIN}^{(\mathcal{D})}$. Although decidable, OWL-DL could become intractable,

[1] The EXPRESS construct "ONE OF" only represents a local disjointness.

especially when dealing with large ontologies. The computational complexity of OWL-DL is outside the scope of this paper. The expressiveness of OWL is contained in its constructs: a model-theoretic semantic is specified for each construct. These language constructs have been designed with the aim of using DL for the semantic web to enable interoperability between systems through semantic data representation.

The use of the XML (Extensible Markup Language) syntax within OWL facilitates the exchange of models between agents, while the OWL features give the model the expressive power needed for ontological representation.

5.2 Modeling Tools

We evaluated the OWL expressiveness for product modeling by building a product ontology (see Section 6) and by performing inference mechanisms on it (see Section 7). We developed the ontology in OWL-DL version 1.0, using Protégé-OWL 3.3 [29] to edit it. We excluded OWL Lite and OWL Full from consideration because of their low formal expressivity and the hard computational problems, respectively. The Protégé ontology editor supports $\mathcal{SHOIN}^{(\mathcal{D})}$.

We used the reasoning engine RACERPro [30] for doing inference in DL. We chose RACERPro because it includes subsumption and classification algorithms which are typical DL reasoning algorithms. Further, it is easily accessible through the OWL Plug-in in Protégé. For domain-specific rules we used the Semantic Web Rule Language (SWRL) [31]. Since the general combination SWRL and OWL-DL is undecidable, we selected only the DL-safe portion of SWRL. The rules are edited directly in Protégé-OWL through the SWRLTab - an extension to the editor - and then executed by Jess [32], a rule engine for the Java platform that supports reasoning on declarative rules. We used the Jess Bridge for the following:

- merge SWRL rules and relevant OWL data
- input them to the Jess engine, and
- return the new inferred information to the OWL ontology.

Figure 1 depicts how the OWL data and the SWRL rules are connected. The bold arrows indicate the flow of information (initial data, rules and final data) while the dashed arrows indicate how the Jess Bridge enables that flow.

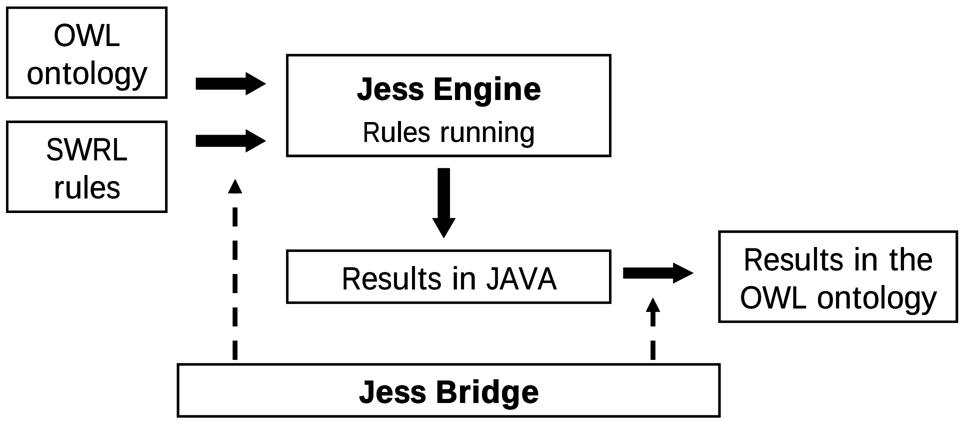

Figure 1: Connection between the OWL ontology and SWRL rules

In the next sections we will describe, with an example, how we used the selected languages and tools to develop a product ontology. The concepts and their relationships in CPM/OAM described in Section 6 will be the source of the examples used in Section 7 and 8.

6 Description of the information model

The Core Product Model (CPM) [3] was intended to form a base for representing a product model that could respond to the demands of the next generation CAD systems, besides providing improved interoperability among future software. The Open Assembly Model (OAM) is the CPM extension for assembly and tolerances representation. Based on CPM and OAM [3], [33] presents two ontological models. These two models were developed at the National Institute of Standards and Technology (NIST) as part of the ongoing work related to product representation for lifecycle management [3]. A brief description of these two models is given below.

6.1 Core Product Model

The concepts (classes in OWL) in CPM are grouped into four categories (see Figure 2):

- Classes that provide supporting information for the objects (abstract classes, i.e. classes that do no have direct instances): CoreProductModel, CommonCoreObject, CommonCoreRelationship, CoreProperty and CoreEntity.
- Physical or conceptual objects classes: Artifact, Feature, Port, Specification, Requirement, Function, Flow, Behavior, Form, Geometry and Material.
- Classes that describe associations (relationships) among the objects: Constraint, EntityAssociation, Usage and Trace.
- Classes that are commonly used by other classes (utility classes): Information, ProcessInformation and Rationale.

The hierarchy of classes begins from CommonCoreEntity. This class represents real objects and relationships or associations between them. The two subclasses of

CommonCoreEntity are CommonCoreObject and CommonCoreRelationship. CommonCoreObject is the parent class for all the object classes. CommonCoreRelationship and its specializations, the EntityAssociation, Constraint, Usage and Trace relationships, can be applied to individuals of classes derived from this class. CommonCoreRelationship is the base class from which all association classes are specialized. It also serves as an association to the CommonCoreObject class. CoreEntity is an abstract class from which the classes Artifact and Feature are specialized. EntityAssociation relationships may be applied to entities in this class. CoreProperty is an abstract class from which the classes Function, Flow, Form, and Material are specialized. Constraint relationships may be applied to individuals of this class.

The information elements listed in Section 3 that are included in CPM are Function, Behavior, Geometry , Material and Features. The structure of Assembly is partially defined in the CPM: the relationship between an Assembly and its components is defined in the relationship partOf but the connections between the components are outside the scope of CPM. For further details, please, refer to [3].

6.2 Open Assembly Model

OAM incorporates information about assembly relationships and component composition (we used the term "structure" in Section 3 to identify both); the representation of the latter is by the class ArtifactAssociation, which represents the assembly relationship that generally involves two or more Artifacts. ArtifactAssociation is specialized into the following classes: PositionOrientation, Relative-Motion and Connection. ArtifactAssociation is directly connected to Assembly to allow the possibility to check the assembly relationship involved in the Assembly through the property ArtifactAssociation2Assembly (see Figure 3).

An assembly is a composition of its subassemblies and parts. The Assembly and Part classes are subclasses of the CPM Artifact class. A Part is the lowest level component. Each assembly component (whether a subassembly or part) is made up of one or more features, represented in the model by OAMFeature, a subclass of the CPM Feature class. OAMFeature has tolerance information, represented by the class Tolerance. Tolerance is one of the information elements we identify among the requirements in Section 3.

The class AssemblyFeatureAssociation (AFA) represents the association between mating assembly features through which relevant Artifacts are associated. The class ArtifactAssociation is the aggregation of AssemblyFeatureAssociation. The class AssemblyFeatureAssociationRepresentation (AFAR) represents the assembly relationship between two or more assembly features. This class is an aggregation of ParametricAssemblyConstraints, KinematicPair, and/or KinematicPath between assembly features. KinematicPair defines the kinematic constraints between two adjacent Artifacts (links) at a joint. KinematicPath provides the description of the kinematic motion. For further details, please, refer to [3].

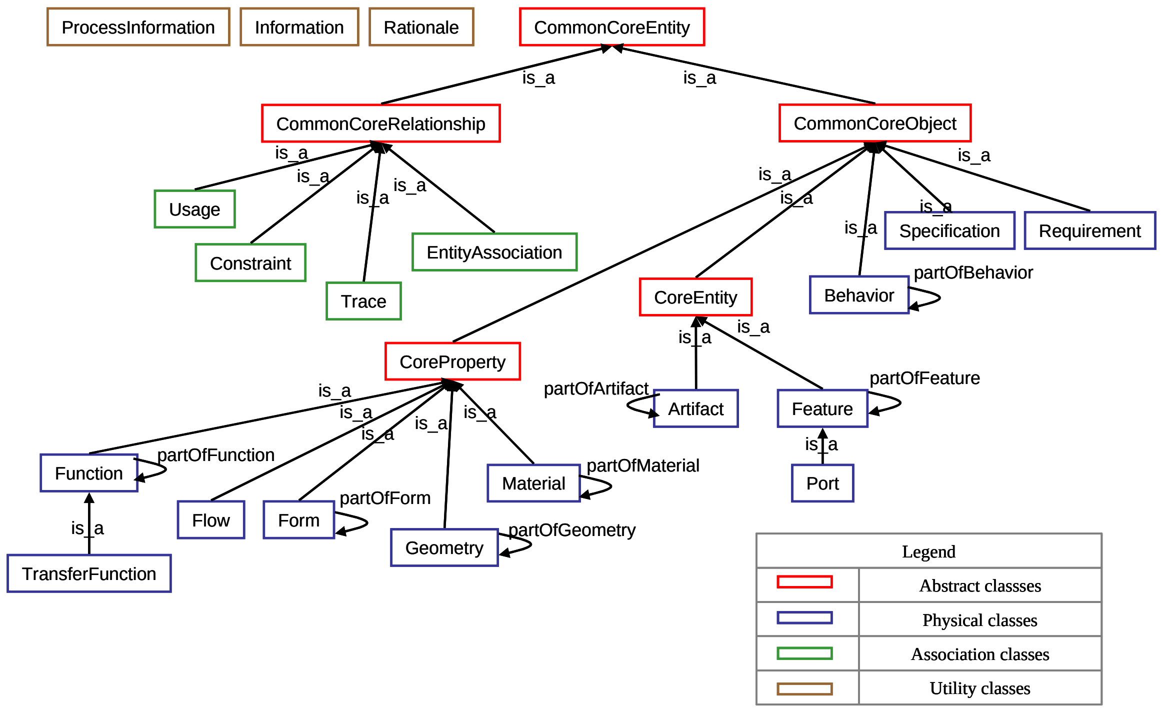

Figure 2: Core Product Model, OWL version (highlighting is-a and part-of relationships)

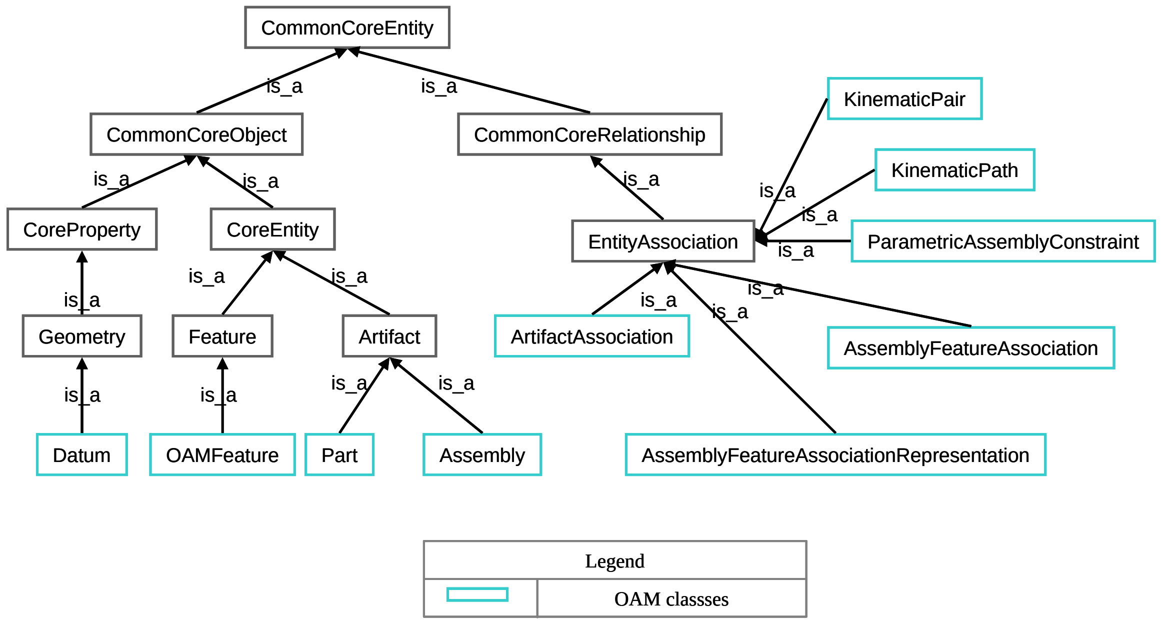

Figure 3: Open Assembly Model, OWL version (highlighting is-a relationship)

7 Expressivity capabilities

In this section we will discuss the detail evaluation of OWL-DL1.0 with respect to CPM/OAM with the expressivity of DL and capability of supplementary rules (domain-specific rules).

7.1 Expressivity of DL

In Table 2 we give relationship between the DL expressivity included in OWL and the expressions (classes) defined in CPM/OAM model. The first column of the table shows the notation for expressivity of OWL-DL, i.e., $\mathcal{SHOIN}^{(\mathcal{D})}$ [23]. The third column in Table 2 indicates the expressivity associated with each letter of the DL notation. The fourth and the fifth column provide respectively the description and the related expressions of CPM/OAM.

Table 2: Examples of DL expressivity in product modeling

Notation	No	Expressivity	Description	DL-related expressions of CPM/OAM
$\mathcal{AL}$	1	Universal concept	The concept that contains all the individuals.	The concept of Thing: included in every OWL ontology.
	2	Bottom concept	The concept without any individual.	The concept of Nothing: included in every OWL ontology.
	3	Atomic concept	A concept name.	The concept of Assembly.
	4	Atomic negation	The negation of an atomic concept.	The concept of Part consists of those individuals that are not Assemblies.
	5	Value restriction	All the individuals that are in the relationship with the described concept belong to a specified concept.	A Feature can be connected through the property feature2ParametricAssemblyConstraint only to the concept ParametricAssemblyConstraint.
	6	Intersection of concepts	The set of individuals belonging to both the concepts.	An OAMFeature is the intersection between the concept of Feature and the concept of the individuals connected at least with one AFA through the property feature2AFA. An OAMFeature can be automatically recognized by giving the definition of that concept.

Notation	No	Expressivity	Description	DL-related expressions of CPM/OAM
$\mathcal{S}$	7	Transitive properties	For all individuals a, b, and c, if a is related to b and b is related to c, then a is related to c.	We introduce two properties: artifactHasPart (transitive) and artifactHasPart_direct (not transitive) to address the issue of undecideability. The property artifactHasPart is used to connect an Assembly with all its components. If you specify cardinality constraints for transitive properties than it becomes undecideable. For this reason, we also define the property artifactHasPart_direct which is not transitive. This allows to specify cardinality constraints on this property. artifactHasPart_direct is used to connect the Assembly to its direct subassemblies or Parts.
$\mathcal{H}$	8	Role hierarchy	If P1 is a subproperty of P2, then the property extension of P1 (a set of pairs) should be a subset of the property extension of P2 (also a set of pairs).	The property artifactHasPart is a subproperty of the generic property hasPart.
$\mathcal{O}$	9	Enumerated classes	The concept is made of exactly the enumerated individuals.	Two CommonCoreObject can be linked through the CommonCoreRelationships "*alternativeOf*", "*isSameAs*", "*versionOf*", "*isBasedOn*", "*derivedFrom*": these are the enumerated individuals of the range class of the link.
$\mathcal{I}$	10	Inverse properties	For all individuals a and b, iff a is related to b, then b is related to a through the inverse property.	The property partOf is the inverse of artifactHasPart. When an Assembly is connected to its component through artifactHasPart, the component will be connected to the Assembly through partOf.
$\mathcal{N}$	11	Cardinality restrictions	The concept is constrained to have a number of values of a particular property.	An ArtifactAssociation has to link at least 2 Artifacts. An inconsistency will be identified if not.
$\mathcal{F}$	12	Functional properties	The individuals of certain concepts have unique property fillers for a given property.	A KinematicPair can be referred only to one AFAR.
$\mathcal{E}$	13	Full existential quantification	The set of all individuals in the domain which has at least one specified R-successor.	A DatumFeature has to have some connections with AssemblyFeatureAssociation. If it doesn't the reasoner will recognize an inconsistency.
$\mathcal{U}$	14	Concept union (disjunction)	The set of the individuals belonging at least to one of the disjointed concepts.	The property that connects ArtifactAssociation to the assembled components has as range the concept union of Assembly and Part.

Notation	No	Expressivity	Description	DL-related expressions of CPM/OAM
$(\mathcal{D})$	15	Datatype properties	Property for which the value is a data literal, such as a string or a number.	CommonCoreEntities have names: the property links CommonCoreEntity to a string.

All OWL DL 1.0 constructs ($\mathcal{SHOIN}^{(\mathcal{D})}$) are required for CPM/OAM but only the inference mechanisms of $\mathcal{SHIN}^{(\mathcal{D})}$ are performed since the reasoning capability of RacerPro in the presence of enumerated classes ($\mathcal{O}$) is incomplete [34]. In our ongoing research we are evaluating the Pellet reasoner ($\mathcal{SHOIQ}^{(\mathcal{D})}$) because it is more compatible with OWL 1.1 ($\mathcal{SROIQ}^{(\mathcal{D})}$), the newest version of OWL [35].

With the set of axioms listed in Table 2, the reasoner is able to:

- Query and search the model.
- Check its consistency.
- Perform inference on the classes' hierarchy.
- Perform inference on the membership of the individuals to the classes.

In OWL-DL 1.0 a property is declared in terms of its domain, range and characteristics such as transitivity or reflexivity. In cases where it is required to impose specific conditions or restrictions we need to specify some rules (see question 5 in Table 1). For example, consider Figure 4. Class Car and class Person are connected through the property hasOwner, class Person and class Garage are connected through the property isRenter and class Car and class Garage are connected through the property isParked. To infer that a particular person's car is parked in the garage the person rents we need a rule to specify this explicitly.

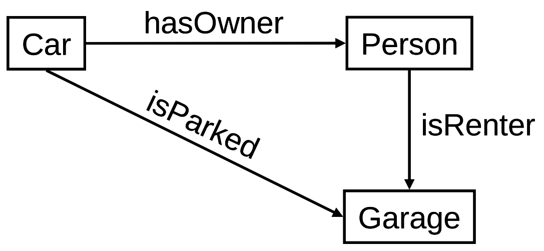

Figure 4: Example of a case where rules are needed

In OWL 1.1 the above rule can also be achieved through property chains, e.g., the property isParked is the combination of the properties hasOwner and isRenter, but in our opinion not in all cases the rules can be replaced by property chains.

7.2 Expressivity of Rules

We use SWRL [31] rules in order to:

- create properties between individuals,
- associate individuals to classes: we use this capability to associate an individual to a class for detecting inconsistencies in the ontology.

We classify these rules into four groups:

- property rules
- association rules
- partOf rules, and
- acyclic rules.

In the following diagrams, we use rectangles to identify classes and ovals to identify individuals. All the diagrams representing the rules (Figures from 6 to 9) are related to the following portion in Figure 5 of the CPM/OAM models and they all share the same legend.

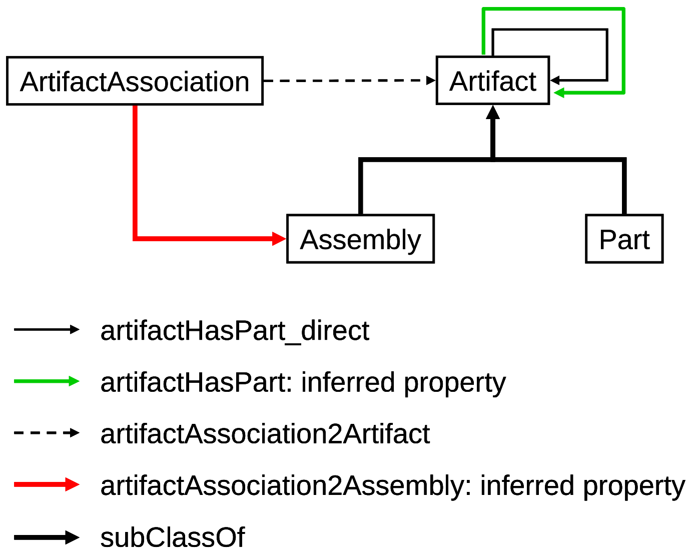
ArtifactAssociation
Artifact
Assembly
Part
artifactHasPart_direct
artifactHasPart: inferred property
artifactAssociation2Artifact
artifactAssociation2Assembly: inferred property
subClassOf

Figure 5: CPM and OAM for representing assembly structure

The class Artifact is specialized into Assembly and Part. The relationship part-of between two Artifacts is expressed through the properties artifactHasPart_direct and artifactHasPart. ArtifactAssociation, i.e., the class that represents the assembly relationships, is connected through the property artifactAssociation2Artifact to two or more Artifacts. As the inferred relationship from this, the class ArtifactAssociation is also connected through the property artifactAssociation2Assembly to the Assembly that contains the assembly relationships.

Property rules enable inferring new properties between individuals. The property rules add additional semantics in the ontology. In the following example, artifactAssociation2Assembly property (inferred) associates the ArtifactAssociation directly to the Assembly. Figure 6 shows how this rule connects the master *assembly_1* to the ArtifactAssociations existing between its subcomponents *assembly_2*, *part_1*, and *assembly_3*. In this way, we can explicitly make it available how many *artifactAssociation* are involved in *assembly_1*.

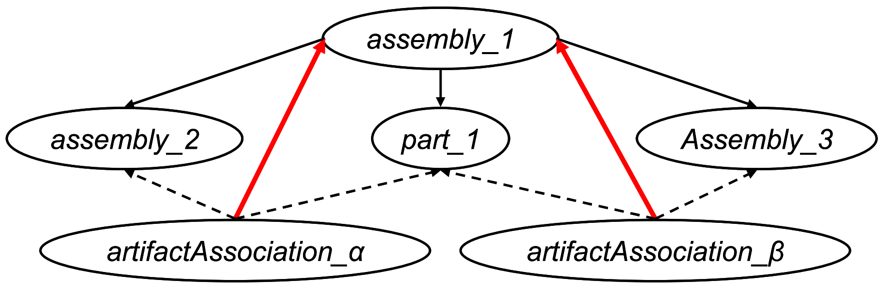

Figure 6: Example of a property rule

Association rules represent the binary relationships between association classes and object classes. In Figure 7 a minimum cardinality 2 is applied in the OWL model, and then a SWRL rule specifies that if two different individuals of the association class are connected to the same individuals of the object class then these two association individuals are the same (sameAs). In this way a unique ArtifactAssociation can be connected to the same individuals of Artifact.

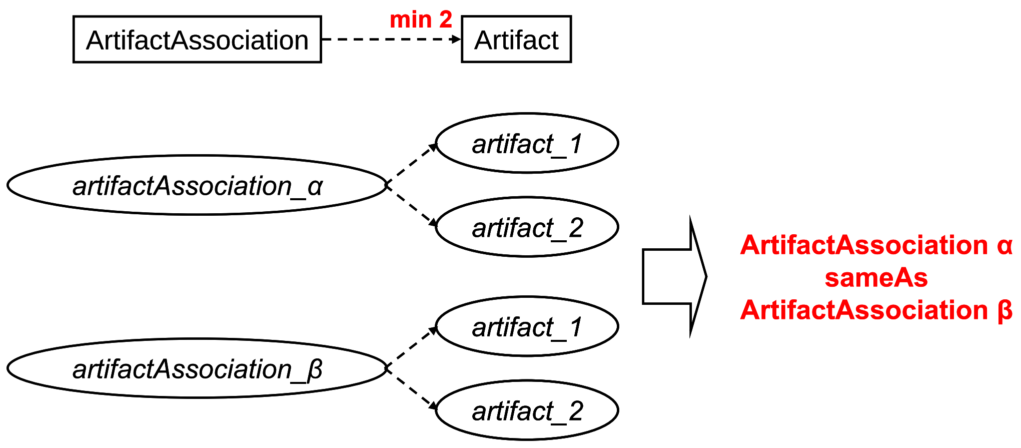

Figure 7: Example of an association rule

PartOf rules create the right assembly structure, i.e., enable the assemblies to distinguish between the direct and the indirect partOf properties. After executing the partOf rules, the indirect property links an assembly with all its parts (example in Figure 8).

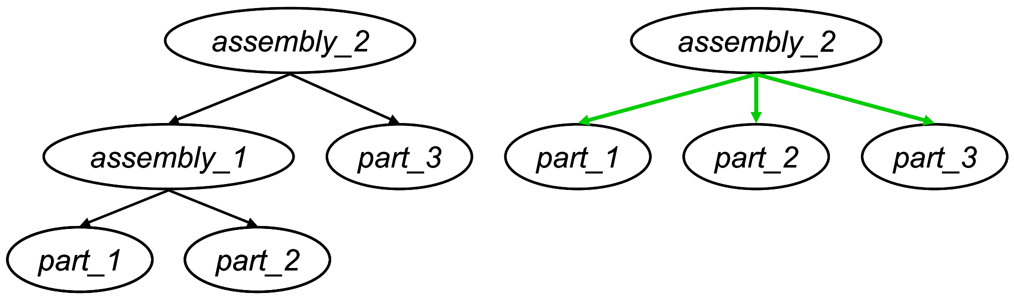

Figure 8: Example of a partOf rule

Acyclic rules instantiate classes of the kind *not-allowed*, to identify the individuals that, although declared, are included in a part-of cycle. Since no inference mechanism can delete wrong information from the ontology, we insert the wrong information in the not-allowed classes through the acyclic rules. Since the not-allowed classes are declared disjoint from the original ones, the reasoner will detect an inconsistency.

Consider the example in Figure 9: the *assembly_2* is composed by itself (*assembly_2* is composed by *assembly_1* that is in turn composed by *assembly_2*).

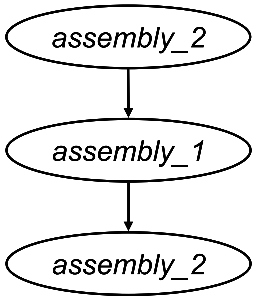

Figure 9: Example of an acyclic rule

In this example, both *assembly_2* and *assembly_1* are individuals of the same class. Since the DL axioms can only apply to classes, no axiom can constrain the relationship between instances of the same class. In this case we can insert any axiom to declare that the part-of relationship is acyclic. For this reason, we create the NotAllowedAssembly class and instantiate it through the acyclic rules. Since the classes NotAllowedAssembly and Assembly are disjoint, the reasoner detects an inconsistency.

We give practical examples of the reasoning mechanisms employed in the next section, where a case study is presented for the exploration of the potentialities of the ontology assembly representation.

8 Reasonings Examples

We tested the reasoning capabilities of DL for product modelling by instantiating the OAM model with a planetary gear system. Figure 10 and the summary presented below are taken from [12] [3].

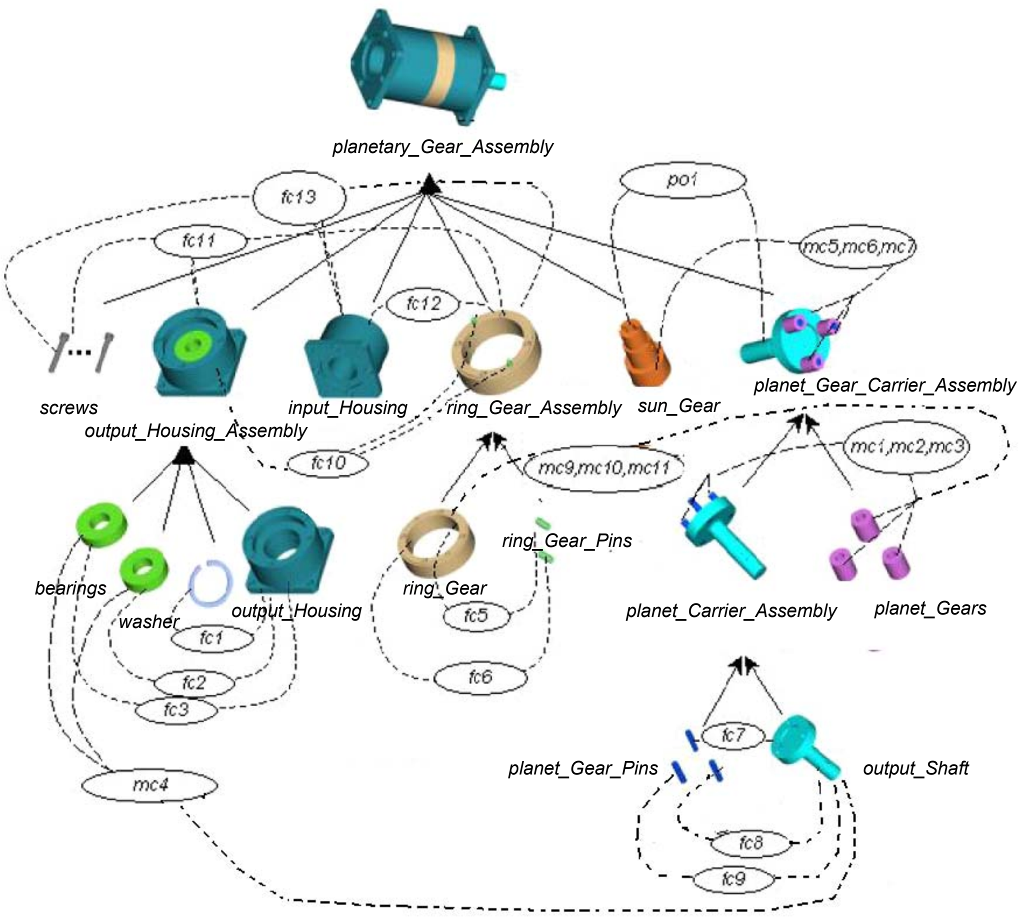

Figure 10: Case study: planetary gear system

The planetary gear system is composed of two parts and three subassemblies. The parts include the input-housing and the sungear. The three subassemblies include: (1) the output end assembly comprising two bearings, a washer, and the output housing; (2) the ring gear assembly comprising a ring gear and two ring-gear pins; and (3) the planet gear holder assembly comprising three planet gears and a planet carrier assembly, which further decomposes into the output shaft and three planet-gear pins. In total there are 30 different parts. The connections and pairs between different Artifacts are of different types: fixed connection (fc), movable connection (mc) or position orientation (po).

To represent the planetary gear system we declare in total 187 individuals and 277 properties between the individuals. These individuals comprise not only the Artifacts but also their Features, their Geometries, their Tolerances and their connections through the association classes. Out of the 187 individuals, 70 are declared to belong to the class Thing, parent of all the classes in the ontology. The reasoner, using the classes and properties axioms, classifies these 70 individuals into their proper classes.

The inference mechanisms deal with not only the individuals in the ontology but also the properties between the individuals. The editor Protégé-OWL automatically defines the inverse and the parent properties. Since all the properties in the model have their inverse, Protégé automatically defines 277 inverse properties, one for each declared direct property. Moreover, Protégé defines all the properties parents of the asserted properties. After performing the DL reasoner, the rule-based inference found additional 170 properties that are added to the ontology.

In the following sections, we provide three examples of the inference mechanisms we used: the first is based on description logic, the second is based on domain-specific rules while the third combines both DL and rules.

8.1 Example of DL reasoning

Table 3 presents an example of description logic reasoning from the case study. The class Artifact and its subclasses are the main focus. In OAM we describe the class Part with a necessary and sufficient condition: Parts are Artifacts without subassemblies. In other words, in DL expressivity, the concept of Part is the intersection between the concept of Artifact and the concept of the Thing having cardinality 0 on the property artifactHasPart_direct ($\mathcal{AL}$ expressivity, number 6 in Table 2). This property has as domain (the class owning the property) and as range (the class of the values of the property) the class Artifact.

Moreover, we describe the class Assembly with a necessary condition: Assemblies must have at least two Artifacts connected through the inherited property artifactHasPart_direct. In other words, in DL expressivity, we apply a cardinality restriction ($\mathcal{N}$ expressivity, number 11 in Table 2) to the concept of Assembly.

We then define Assembly and Part as partitions of the class Artifact, i.e., the concept of Artifact is made by the union ($\mathcal{U}$ expressivity, number 14 in Table 2) of the disjoint concepts Assembly and Part. As a result, an instance of Artifact (*planet_Carrier_Assembly* in this example) composed by other Artifacts (*output_Shaft*, *planet_Gear_Pin_1*, *planet_Gear_Pin_2*, *planet_Gear_Pin_3*) is inferred to be an individual of Assembly.

Table 3: Example of DL reasoning

AIM	Infer that an Artifact composed by other Artifacts is an Assembly
CLASSES	Artifact, Assembly
PROPERTIES	artifactHasPart_direct (Range: Artifact , Domain: Artifact)
RESTRICTION	On Assembly: artifactHasPart_direct min 2 (an Artifact is an Assembly only if it is related with at least 2 other Artifacts)
INPUT	An individual of Artifact (*planet_Carrier_Assembly*) is composed through artifactHasPart_direct by 4 individuals of Part (*output_Shaft, planet_Gear_Pin_1, planet_Gear_Pin_2, planet_Gear_Pin_3*)
OUTPUT	*planet_Carrier_Assembly* is reclassified as an individual of the class Assembly rather then general Artifact.

8.2 Example of rule-based reasoning

Table 4 presents an example of rule-based reasoning from the case study: the structure of an Assembly is described by its parts/subassemblies and by the relationship between its components.

Table 4: Example of rule-based reasoning

AIM	Infer the relation between Assembly and ArtifactAssociation
CLASSES	Assembly, Part, ArtifactAssociation
PROPERTIES	artifactAssociation2Assembly (Range: ArtifactAssociation, Domain: Assembly)
RULES	If the components of an Assembly are linked through an ArtifactAssociation, then relate that ArtifactAssociation to the Assembly (see Table 5)
INPUT	An individual of Assembly (*output_Housing_Assembly*) is composed of *bearing_1*, *bearing_2*, *output_Housing* and *washer* through artifactHasPart_direct. These individuals are connected with individuals of the class ArtifactAssociation
OUTPUT	*output_Housing_Assembly* is linked with the corresponding individuals of ArtifactAssociation (*fc_1, fc_2, fc_3*) through the artifactAssociation2Assembly property.

In this example, *output_Housing_Assembly* is composed of *bearing_1*, *bearing_2*, *output_Housing* and *washer*. The ArtifactAssociations connect *washer* with *output_Housing* (*fc_1*), *bearing_1* with *output_Housing* (*fc_2*) and *bearing_2* with *output_Housing* (*fc_3*).

The aim of the reasoning is to correctly relate the *output_Housing_Assembly* to the ArtifactAssociations involved in the Assembly. In this case we can not use OWL declarations since the condition for creating the new relation is dependent on the specific properties each individual possesses. For this reason we have to resort to SWRL rules.

In this example we need four different property rules (see Table 5). Each of them takes into account a different scenario:

- *Rule 1* is applied when the description of the Assembly is detailed (the AssemblyAssociation connects two or more Parts) and the Assembly has at least one subassembly that is a Part. The antecedent of the rule indicates that one Part is directly part-of the Assembly while the other Part is indirectly connected to the Assembly.
- *Rule 2* is applied when the description is detailed but the ArtifactAssociation exists between Parts that are not directly subassemblies of the Assembly. This means that the Assembly is composed of other subassemblies and each subassembly has a Part involved in the Assembly. In the antecedent of Rule 2 we explore the indirect property to search these Parts in the subassemblies.

- *Rule 3* is applied when the description is not detailed so that the Assembly is composed of two or more subassemblies connected together.
- *Rule 4* is similar to the third but is useful when the Assembly is made of a Part and a subassembly.

Table 5: Rules needed to connect Assembly with its ArtifactAssociations

Rule 1	Rule 2	Rule 3	Rule 4

artifactHasPart_direct(?x, ?y)
Part(?y)
artifactHasPart(?x, ?z)
Part(?z)
differentFrom(?y, ?z)
artifact2AA(?y, ?a)
artifact2AA(?z, ?a)
	artifactHasPart_direct(?x, ?y)
	Assembly(?y)
	artifactHasPart_direct(?x, ?z)
	Assembly(?z)
	differentFrom(?y, ?z)
	artifactHasPart(?y, ?q)
	Part(?q)
	artifactHasPart(?z, ?r)
	Part(?r)
	differentFrom(?q, ?r)
	artifact2AA(?q, ?a)
	artifact2AA(?r, ?a)
		artifactHasPart_direct(?x, ?y)
		Assembly(?y)
		artifactHasPart_direct(?x, ?z)
		Assembly(?z)
		artifact2AA(?y, ?a)
		artifact2AA(?z, ?a)
			artifactHasPart_direct(?x, ?y)
			Assembly(?y)
			artifactHasPart_direct(?x, ?z)
			Part(?z)
			artifact2AA(?y, ?a)
			artifact2AA(?z, ?a)

→ Assembly2ArtifactAssociation(?x, ?a)			

8.3 Example of combining DL and rule-based reasonings

Table 6 presents an example of combining both DL and rule-based reasoning. The focus is the composition hierarchy of an Assembly. The goal in this example is to avoid cyclic composition hierarchies, i.e., hierarchies in which an assembly contains itself. Since composition hierarchies are constituted by individuals of the same class Assembly, we can not use any DL axiom to impose the acyclicity constraint on the hierarchy. The use of domain-specific rules is the only solution (see Section 4).

We create in the product ontology the class NotAllowedAssembly, disjoint from the class Assembly. NotAllowedAssembly will contain all the individuals of the Assembly involved in a cyclic hierarchy composition (*planetary_Gear_System_Assembly* in the case of Table 6). We create a SWRL rule to automatically instantiate this class. After executing the rule, the individuals involved in the cyclic hierarchy will belong to both the classes Assembly and NotAllowedAssembly. Since these two classes are declared disjoint, the DL reasoner will detect an inconsistency.

Table 6: Example of combining DL and rule-based reasonings

AIM	Infer an inconsistency in case of a cyclic composition of an Assembly
CLASSES	Assembly, NotAllowedAssembly
PROPERTIES	artifactHasPart (Range: Artifact , Domain: Artifact)
RULES	If an Assembly is composed of itself, then the Assembly will belong to the class NotAllowedAssembly
RESTRICTION	Assembly and NotAllowedAssembly are disjoint classes
INPUT	An individual of Assembly (*planetary_Gear_System_Assembly*) contains the subassembly *planet_Gear_Holder*, that in turn contains the *planetary_Gear_System_Assembly*
OUTPUT	*planetary_Gear_System_Assembly* belongs to both the classes Assembly and NotAllowedAssembly: an inconsistency is detected

9 Expressivity limitations

The process for selecting the appropriate logical formalism for product ontology modeling is a question of trade-off between expressivity, decidability and computational complexity. To help the reader in this process, we provide in this paper an evaluation of description logic sublanguages and domain-specific rules. In this section we complete our evaluation by highlighting the limits of their expressivity capabilities. In other words, we show some of the expressivity requirements that can not be satisfied with OWL and SWRL, both widely used in the world wide web.

OWL and SWRL are continuously evolving to meet the needs of application-specific communities (e.g., the bio-ontologies community). OWL 1.1, an extension of OWL-DL 1.0, is the typical example of this evolution. This extension enhances the expressivity capability of OWL 1.0 [http://www.webont.org/owl/1.1/overview.html] by including new DL constructs such as:

- qualified cardinality restrictions
- reflexive, irreflexive, symmetric, and asymmetric properties, local reflexivity restrictions
- disjoint properties
- property chain inclusion axioms
- user-defined datatypes

Table 7 shows some examples on how the expressivity of these new constructs can be used for product modeling.

Table 7: Example of OWL 1.1 expressivity in product modeling

Expressivity	Examples of Axioms using the Expressivity
Qualified cardinality restrictions	a DieselCar has exactly 1 DieselEngine as engine.
Irreflexive properties	a Car can not be a part of itself
Disjoint properties	the properties partOf and artifactHasPart cannot hold between the same instances of Artifact. If *artifact_1* is a partOf artifact_*2*, then it is impossible that *artifact_1* artifactHasPart *artifact_2*.
Property chain	if a Wheel is powered by an Engine that is a partOf a Car, then the Wheel is partOf of the Car
User-defined datatypes	the price of an EconomicCar is represented by an integer less than 15000.

OWL 1.1. is supported by Protégé 4 but a SWRL tab for this new version of Protégé has not yet been developed. Hence, we decided to develop our product ontology in OWL 1.0.

9.1 DL issues

Our major concern for DL is regarding the Open World Assumption (OWA) which is the assumption that the knowledge we represent in an ontology is considered to be incomplete by default, i.e., whatever we do not declare in the ontology is unknown and therefore not possible to draw any conclusion. Now, although the OWA allows for easy reusability and extensibility of the ontology, it is sometimes useful in the product domain to close that open world [36].

For example, consider the declaration for the class Car with a minimum cardinality constraint on the number of Seats it has to contain: a Car must have at least two Seats. In this case we can still create an instance of Car (*car_1*) that doesn't have any Seat. *car_1*, till proven otherwise, will have at least two seats (recorded in the memory of the inference engine) but these seats are yet to be specified in the ontology because our knowledge on these seats is incomplete. This reflects exactly our knowledge at the early design stage of that Car, when the seats haven't been designed yet. However, if we represent the same knowledge of that Car for developing a catalogue, an inconsistency should be detected. Therefore we would need a new reasoner that can apply the Close World Assumption (CWA) conveniently. Depending on the context, we could then apply either the OWA or the CWA.

Another issue related to the OWA appears in the OAM ontology presented in Section 6, where the concept of Part is defined as the intersection between the concept of Artifact and the concept of Thing having cardinality 0 on the property artifactHasPart_direct. Even with this definition, an instance of Artifact without subassemblies (i.e., not holding the property artifactHasPart_direct) is not inferred to be a Part.

To reason on the concept of Part we would need to define it with a Negation as Failure (NaF): a Part is an Artifact that does **not** hold the property artifactHasPart_direct. NaF would allow us to infer that an instance belongs to the class Part, until we do not find proof that a particular instance holds the property artifactHasPart_direct. Unfortunately, NaF, or non-monotonic negation, as a consequence of the OWA, is not provided in DL. Therefore we would need the capability for expressing NaF and also the reasoners' ability to handle NaF (as already provided by most of the first order logic reasoners).

9.2 SWRL issues

The issue we described in the previous example could be overcome by declaring a domain-specific rule that if an instance of Artifact does not hold the property artifactHasPart_direct, then that instance belongs to the class Part. The rule would look like:

Artifact (?x) and *not* artifactHasPart_direct (?x, ?y) → Part (?x)

Unfortunately, SWRL does not provide a syntax to express negation, so it is impossible to express this rule. Introducing negation as failure for domain-specific rules is a requirement for the representation of product models.

Moreover SWRL supports only procedural rules, i.e., rules of the form of an implication between an antecedent and a consequent [31]. These rules are called procedural because they are on the form *if-then*, where the negation of the consequent does not imply the negation of the antecedent. As an example, consider the possibility of defining an EconomyCar with SWRL:

Car (?x) and hasPrice (?x, ?y) and swrlb:lessThan(?y, 15000) → EconomyCar (?x)

If we then introduce an instance of ExpensiveCar (the negation of the EconomyCar concept), nothing is inferred about its price. We would need a rule engine to be able to logically interpret the rule and therefore able to infer that an ExpensiveCar has a price higher than $15000. This kind of rule engine would allow us to reason that if condition A implies condition B, then the negation of the condition B implies the negation of the condition A:

A → B,

notB → not A.

Now, consider again the rule that asserts that a Car with a price less than $15000 is an EconomyCar. In an ontology we can define the concept of EconomyCar as a Car with a price less than $12000 using an appropriate axiom. If we create an instance of Car with a price equal to $13000, first the reasoner will assert the consistency (validated through the axiom) of the ontology and then the Jess engine will infer that that instance is an EconomyCar using the appropriate SWRL rule. To detect the inconsistency between what is asserted with the ontology declarations and the rules, we have to run the reasoner again. This example shows that there is a strong need for a concurrent reasoner-engine which can check the consistency of the rules and the ontology together.

9.3 OWL issues

The other major problem in using OWL for product modeling is dealing with datatypes and datatype properties (**($\mathcal{D}$)** notation in Table 2). At the beginning of this section we already discussed the utility of user-defined datatypes, a feature already implemented in OWL 1.1. We still cannot define classes in terms of restrictions on: 1) more than one datatype property at a time, and 2) one property in multiple ways. N-ary predicates, the term used to identify these restrictions, would allow us to express, for example, 1) the retail price of a Car is always greater than the manufacturing cost of that Car, and 2) the weight of an Assembly is equal to the sum of the weights of its subassemblies/parts (mathematically defining the summation). Mathematical functions are also required while computing the Behavior of the Artifact.

Currently, datatype properties can only be declared to be functional ($\mathcal{F}$ notation in Table 2), whereas we could specify more characteristics, namely functional, inverse functional, symmetric and transitive for object properties (properties holding between individuals). But we need to declare a datatype property as inverse functional to avoid many to one relationships. For example, if we could declare the id property of the Artifacts (unique identifier) as both functional and inverse functional, the effect of that will be that each Artifact will have a unique id and each id will refer to a unique Artifact.

The major obstacle we found in dealing with object properties is their hierarchy (built through the subproperty mechanism). For example, consider the case in which we want to declare that: Car hasEngine exactly 1 Engine ($\mathcal{N}$ notation in Table 2). In our ontology we could assert that: 1) the property hasPart has as domain the class Car and as range the class ElectroMechanicalComponents, and 2) the property hasEngine, which is a subproperty of hasPart, has as domain the class Car and as range the class Engine, subclass of ElectroMechanicalComponents (see Figure 11).

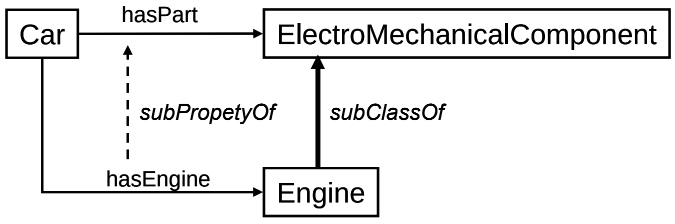

Figure 11: Example of a property hierarchy system

Now, although we assert these restrictions, we can still connect a Car to multiple Engines through the property hasPart, therefore avoiding using the property hasEngine for which the cardinality was specified. In this way an instance of Car containing more than one Engine will not create any inconsistency in the model. To identify the inconsistency, we would require a mechanism to “define” properties themselves. With this mechanism we could declare, in the previous example, that the property hasEngine is equivalent to the property hasPart connecting Car to Engine. A reasoner could then infer the most specific property that connects two instances.

One major concern of product modelers is the scalability of OWL ontologies. Ontology scalability is defined in terms of three aspects of the ontology [37; 38]: 1) ontology size, i.e., the number of classes, instances and properties contained in the ontology, 2) ontology complication, i.e., complexity of the logical relationship contained in the ontology, and 3) ontology modularity, i.e., knowledge reusability. Choosing an ontology size is the task of product modelers: they have to find the right compromise between comprehensive domain coverage and a suitable number of classes, instances and properties. In terms of ontology complication and modularity, we would need two different kinds of mechanisms: 1) mechanisms for partial expressivity usage, and 2) mechanisms for partial import.

To manage ontology complication, a mechanism for partial expressivity usage would allow us to specify all the definitions and restrictions in the product ontology and then select, depending on the context of usage, only the part of the expressivity we are interested in. If, for example, a set of axioms considerably increases the reasoning time, we would like to exclude that set of axioms from the ontology when that set is not relevant in our context of usage.

To enable ontology modularity, the only OWL mechanism available is the “import” mechanism. This mechanism allows us to reuse the concepts defined in one particular ontology (the imported ontology) in other ontology (the importing ontology).

Unfortunately this mechanism will import all the concepts from the imported ontology to the importing one. To make an ontology more scalable, we need a mechanism to be able to import only the concepts (partial import) we are interested in.

10 Summary and Conclusions

To ensure interoperability of different systems sharing product information across its different stages of lifecycle, a multiple view of the product model is required. Such a model development is influenced by three factors: logical formalisms, computer interpretable languages and product information.

Usually logicians focus on logical theorem proving, computer scientists focus on theory of languages, while product engineers focus on product information representation. The aim of this paper is to evaluate DL for product modeling taking into account the interrelations between logical formalisms, computer interpretable languages and product information.

First, we identified the expressivity requirements for representing products. We grouped these requirements into information elements and abstraction principles. Second, we discussed the expressivity of description logic with respect to the requirements for representing products. DL provides mechanisms for both explicit knowledge specification and implicit knowledge inference. These mechanisms, in the case of DL, are decidable, so that a user can generally define product model with DL. Since some expressivity requirements can not be satisfied through DL, we chose domain-specific rules to increase the expressivity in the product model. Description logic and domain-specific rules are combined together to accommodate the level of expressivity required in product modeling. For the expression of DL, we chose OWL-DL and for domain-specific rules, we chose SWRL, because it is compatible with the OWL editor Protégé.

Third, we used OWL and SWRL to build product ontologies: ontological version of Core Product Model (CPM) representing a generic product and Open Assembly Model (OAM), extension of the CPM for representing mechanical assemblies. We use these ontologies to show how the expressivity of DL with domain-specific rules satisfies the requirements of the product modeling. We described an instantiated ontology with a planetary gear system use case to show how the level of expressivity in the model allows explicit knowledge specification and implicit knowledge inference.

Finally, we concluded our evaluation by listing the expressivity limitations of DL, OWL and SWRL. The issues presented in our evaluation provide future research directions for developing high expressive product models. Developing such a model requires close collaboration between product engineers, computer scientists and logicians. The community that is using DL and its implementation languages can guide the development of new expressivity features. These new expressivity features can then be finally tested by that community.

This paper outlined how DL with rules satisfies the expressivity requirements for developing a consistent formal model for mechanical products. We believe this study can be extended to understand how to choose appropriate logical frameworks (OWL DL to OWL Full) for developing product ontologies. Moreover, since OWL, the reasoners and the available tools are evolving, we are constantly evaluating our approach.

Disclaimer

11 References

1. ISO 10303-203: 1994, Industrial automation systems and integration -- Product data representation and exchange -- Part 203: Application Protocol: Configuration controlled 3D design of mechanical parts and assemblies.

2. The STEP PDM Usage Guide. http://www.wikistep.org/index.php/PDM_Usage_Guide, last visited 2008.

3. Sudarsan, R., Baysal, M. M., Roy, U., Foufou, S., Bock, C., Fenves, S. J., Subrahmanian, E., Lyons K.W, and Sriram, R. D., "Information models for product representation: core and assembly models," *International Journal of Product Development*, Vol. 2, No. 3, 2005, pp. 207-235.

4. Web Ontology Language (OWL), 2005. http://www.w3.org/2004/OWL/ .

5. ISO 10303-1: 1994, Industrial automation systems and integration -- Product data representation and exchange -- Part 1: Overview and fundamental principles.

6. Burkett, W. C., "Product data markup language: a new paradigm for product data exchange and integration," *Computer-Aided Design*, No. 33, 2001, pp. 489-500.

7. Gorti, R. and Sriram, R. D., "From symbol to form: a framework for conceptual design," *Computer-Aided Design*, Vol. 28, No. 11, 1996, pp. 853-870.

8. Gorti, S. R., Gupta, A., Kim, G. J., Sriram, R. D., and Wong, A., "An object-oriented representation for product and design processes," *Computer-Aided Design*, Vol. 30, No. 7, 1998, pp. 489-501.

9. Sudarsan, R., Fenves, S. J., Sriram, R. D., and Wang, F., "A product information modeling framework for product lifecycle management," *Computer-Aided Design*, Vol. 37, No. 13, 2005, pp. 1399-1411.

10. MOKA, *Managing Engineering Knowledge: MOKA Methodology for Knowledge Based Engineering Applications*, Wiley,2001.

11. Brimble, R., and Sellini, F., "The MOKA modelling language," Vol. Knowledge Engineering and Knowledge Management, Proceedings,2000, pp. 49-56.

12. Fenves, S., Foufou, S., Bock, C., Bouillon, N., and Sriram, R. D., "CPM2: A Revised Core Product Model for Representing Design Information ," National Institute of Standards and Technology, NISTIR 7185, Gaithersburg, MD 20899, USA, 2004.

13. Lin, J. X., Fox, M. S., and Bilgic, T., "A requirement ontology for engineering design," *Concurrent Engineering Research and Applications*, Vol. 4, No. 3, 1996, pp. 279-291.

14. Borst, P., Akkermans, H., and Top, J., "Engineering ontologies," *Internation Journal of Human-Computer Studies*, Vol. 46, No. 2-3, 1997, pp. 365-406.

15. KIF-Knowledge Interchange Format, 2006. http://logic.stanford.edu/kif/kif.html .

16. Kitamura, Y., "A functional concept ontology and its application to automatic identification of functional structures," *Advanced Engineering Informatics*, Vol. 16, No. 2, 2002, pp. 145-163.

17. Kim, K. Y., Manley, D. G., and Yang, H., "Ontology-based assembly design and information sharing for collaborative product development," *Computer-Aided Design*, Vol. 38, No. 12, 2006, pp. 1233-1250.

18. Taivalsaari, A., "On the notion of inheritance," *ACM Computing Surveys*, Vol. 28, No. 3, 1996, pp. 438-479.

19. Artale, A., Franconi, E., Guarino, N., and Pazzi, L., "Part-whole relations in object-centered systems: an overview," *Data & Knowledge Engineering*, Vol. 20, No. 3, 1996, pp. 347-383.

20. Sudarsan, R., Subrahmanian, E., Bouras, A., Fenves, S., Foufou, S., and Sriram, R. D., "Information sharing and exchange in the context of product lifecycle management: Role of standards," *Computer-Aided Design*, Vol. to appear, 2008.

21. Ma, L., Mei, J., Pan, Y., Kulkarni, K., Fokoue, A., and Ranganathan, A.. Semantic Web Technologies and Data Management, 2007. http://www.w3.org/2007/03/RdfRDB/papers/ma.pdf .

22. Dolby, J., Fokoue, A., Kalyanpur, A., Kershenbaum, A., Schonberg, E., Srinivas, K., and Ma, L.. Scalable Semantic Retrieval Through Summarization and Refinement, 2007. http://domino.research.ibm.com/comm/research_projects.nsf/pages/iaa.index.html/$FILE/techReport2007.pdf .

23. Baader, F., Calavanese, D., McGuinnes, D., Nardi, D., and Patel-Schneider, *The description logic handbook*, Cambridge University Press2003.

24. OMG. UML 2.0 Superstructure Specification, 2003. http://www.omg.org/cgi-bin/doc?ptc/03-08-02 .

25. Chen, P. P., "The Entity-Relationship Model: Toward a Unified View of Data," *ACM Transactions on Database Systems*, Vol. 1, No. 1, 1976, pp. 9-36.

26. Schenck, D., and Wilson, P. R., *Information modeling: the EXPRESS way*, Oxford University Press, New York,1994.

27. Berardi, D., Cal, A., Calvanese, D., and De Giacomo, G.. Reasoning on UML Class Diagrams. Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza," 2003. http://www.dis.uniroma1.it/~degiacom/didattica/esslli03/

28. Meta Object Facility (MOF) Specification, 2002. http://www.omg.org/docs/formal/02-04-03.pdf .

29. Protégé. http://protege.stanford.edu/, last visited 2008.

30. RacerPro. http://www.racer-systems.com/index.phtml, last visited 2008.

31. SWRL, W3C Member Submission 2004. http://www.w3.org/Submission/SWRL/ .

32. Sandia National Laboratories. Jess Engine. http://herzberg.ca.sandia.gov/, last visited 2008.

33. Fiorentini, X., Gambino, I., Liang, V., Foufou, S., Rachuri, S., Bock, C., and Mahesh, M., "Towards an ontology for open assembly model," International Conference on Product Lifecycle Management,2007, pp. 445-456.

34. W3C. Representing Specified Values in OWL: "value partitions" and "value sets", 2005. http://www.w3.org/TR/swbp-specified-values/ .

35. Liebig, T., "Reasoning with OWL: System Support and Insights," Computer Science Faculty, Ulm University, Technical report 2006-04, Sept. 2006.

36. Sirin, E., Smith, M., and Wallace, E.. Opening, Closing Worlds - On Integrity Constraints. http://clarkparsia.com/weblog/2008/08/21/owl-integrity-constraints-survey/, last visited 2008.

37. Jarrar, M., and Meersman, R., "Scalability and knowledge reusability in ontology modeling," International conference on Infrastructure for e-Business, e-Education, e-Science, and e-Medicine, Rome,2002.

38. Wache, H., Serafini, L., and Gracia-Castro, R., "Survey of Scalability Techniques for Reasoning with Ontologies, 2004. " http://www.sti-innsbruck.at/results/browse/deliverables/details/?uid=245 .