Employing Ontologies for an Improved Development Process in Collaborative Engineering

vorgelegt von
Dipl.–Ing. Tania Tudorache
aus Sibiu

An der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
eingereicht zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
— Dr.–Ing. —

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender: Prof. Dr.-Ing. Uwe Nestmann, TU Berlin
Gutachter: Prof. Dr. Bernd Mahr, TU Berlin
Gutachter: Prof. Dr. Mark Musen, Stanford University

Tag der wissenschaftlichen Aussprache: 20. November 2006

Berlin 2006
D83

Kurzfassung

Ziel dieser Arbeit ist es, Produktentwicklungsprozesse im Maschinenbau zu verbessern. Gesamtziel solcher Entwicklungsprozesse ist die Erstellung eines Produkts in kürzester Zeit und mit minimalen Kosten welche alle Kundenanforderungen erfüllt. Um die häufig auftretende Komplexität solcher Prozesse zu bewältigen, wird typischerweise ein "divide-et-impera"-Ansatz verfolgt: Die Entwicklungsaufgabe wird in mehrere Bereiche unterteilt entsprechend den verschiedenen Aspekten eines Produkts. Zu vordefinierten Meilensteinen werden Entwurfsmodelle welche in den Bereichen entstehen miteinander synchronisiert, um die Konsistenz des Gesamtaufbaus sicherzustellen. Informationstechnologie (IT) spielt zur Bewältigung solcher Aufgaben eine zentrale Rolle.

Diese Arbeit adressiert zahlreiche Probleme, die sich in diesem Zusammenhang ergeben. Bei einer solchen Entwicklung kommen zahlreiche unterschiedliche IT Systeme zum Einsatz wie z.B. computergestützte Entwicklungswerkzeuge (CAD), um geometrische Eigenschaften zu modellieren oder Simulationswerkzeuge, um das Verhalten von Produkten darzustellen. Dabei hat jedes System typischerweise seine eigene Sicht auf das Produkt. Um die Konsistenz des Gesamtaufbaus sicherzustellen, muss eine Vielzahl von Sichtweisen in Einklang gebracht werden. Gemeinsames Arbeiten im Team erfordert eine regelmäßige Propagierung von Änderungen von einem System in andere Systeme. Derzeit erfolgt diese Propagierung weitestgehend händisch, was bei zunehmender Modell-Komplexität und –Größe zu hohen Fehlerraten führt. Erschwert wird die Propagierung durch den Einsatz unterschiedlichster Entwicklungsmethodiken für jedes System sowie durch den Einsatz unterschiedlichster Modellierungssprachen. Darüber hinaus bleiben viele Aspekte der Entwurfsmodelle implizit und sind nur den jeweiligen Entwicklern bekannt. Schließlich sind auch die Beziehungen zwischen verschiedenen Entwurfsmodellen implizit, was die Automatisierung der Änderungspropagierung deutlich erschwert.

Der konzeptionelle Beitrag dieser Arbeit besteht aus einem Rahmenwerk zur Entwicklung hochwertiger Entwurfsmodelle welches eine automatische Konsistenzprüfung von Entwurfsmodellen ermöglicht und eine automatische Propagierung von Änderungen zwischen Entwurfsmodellen unterstützt. Grundlage hierfür ist der Einsatz von Ontologien, um implizite Aspekte der Entwurfsmodelle explizit zu machen. Ontologien sind formale Beschreibungen eines Bezugsrahmens, welche relevante Gegenstände, die Eigenschaften der Gegenstände, die Beziehungen der Gegenstände untereinander sowie Grundannahmen des Bezugsrahmens in Form von Axiomen beschreiben. Ontologien unterstützen die Kommunikation zwischen Menschen, zwischen IT Systemen und zwischen Menschen und IT Systemen. Die Verwendung derselben Ontologie für verschiedene IT Systeme erlaubt beispielsweise eine Austauschbar-

keit der Entwurfsmodelle sowie eine Konsistenzprüfung des Gesamtsystems. Als Teil der Arbeit wurden mehrere Ontologien erstellt. Die „Engineering Ontology“ unterstützt die Anreicherung der Entwurfsmodelle durch weiteres Hintergrundwissen und hilft insbesondere beim Explizieren von Aspekten. Die „Requirements Ontology“ unterstützt beim Erfassen von Anforderungen an ein Produkt. Die Verwendung von Ontologien erlaubt das Anwenden von logischen Schlussfolgerungen, z.B. um die Erfüllung von Produktanforderungen automatisch zu überprüfen. Weiterhin können durch die Verwendung von Ontologien explizite Repräsentationen der Zuordnungen der Entwurfsmodelle untereinander erstellt werden, die das automatische Propagieren von Änderungen erlauben.

Die Konzepte wurden in zwei Softwareprototypen implementiert welche in drei unterschiedlichen Szenarien im Entwicklungsprozess eines Automobilherstellers angewendet wurden. In allen Szenarien konnte der vorteilhafte Einsatz von Ontologien zur Konsistenzprüfung und Änderungspropagierung gezeigt werden. Ferner konnte gezeigt werden, dass die Qualität und die Wiederverwendbarkeit der Entwurfsmodelle durch die Anreicherung mit Hintergrundwissen und Axiomen signifikant gesteigert wurde und die Austauschbarkeit zwischen den IT Systemen ermöglicht wurde.

Abstract

This work brings a contribution to the improvement of the product development process in the engineering domain. The overall goal of the design process is to realize in a short time and with minimal costs a high quality design solution that satisfies all customer requirements. In order to achieve this goal and to master the product complexity, a divide-and-conquer approach is applied: The design task is split in several development branches, each of them being concerned with only one aspect of the product. At different stages of the design process, the design models resulting from the different development branches are synchronized with each other in order to ensure the consistency of the overall design. However, checking the consistency of a design model and between different design models is a very challenging task. One of the reasons is that the engineering tools employed by the development branches operate on models that have different conceptualization of a product according to their own viewpoint. For instance, a computer-aided design (CAD) tool will model the geometrical characteristics of a product, while a simulation tool will model the behavior of the product. Although the two design models represent different viewpoints on the product, they must be consistent with each other, if a common implementation (i.e., the product) of the two design models should be realized.

Besides the consistency checking of design models, the propagation of design changes from one model to another is also important. These two tasks are very frequently done in the design process, and ideally they should be automated. However, there are several hindrances to an automation of these tasks. First, the engineering tools do not typically interoperate. Exchanging model information between engineering tools is often done manually, is error-prone and difficult, considering the size and complexity of the design models. Second, the design models are represented using different modeling languages with their own syntax and modeling methodology. Third, the design models operate with different conceptualization of the product according to their own view on the product. Very often many aspects of the design model remain implicit and are known only to the engineers who have actually built the model. Forth, there are no explicit or formal correspondences between the design models that could be used to asses the consistency between the models and to propagate the design changes.

The contribution that this work brings to the improvement of the design process is two fold. First, it provides a framework for building higher quality design models. Second, it provides a framework that supports the automation of the consistency checking between the design models and for change propagation. The two goals are achieved by using a formal approach to modeling engineering systems based on ontologies. Ontologies are formal descriptions of the objects, of their properties and relationships in a certain domain. Ontologies also con-

tain axioms that make domain assumption explicit both for humans and computers. Tools that commit to using the same ontology have a better chance to achieve interoperability and consistency between their models. As part of this work, an engineering ontology has been developed for representing engineering systems. The engineering ontology has been used to enrich the representation of the design models with domain knowledge. Another ontology has been developed for representing the requirements of a product in a formal way. Logical reasoning on the enriched design models has been used to check the fulfillment of the requirements. The second goal was achieved by defining in a formal way the consistency between different design models. This is supported by a mechanism for representing the correspondences (or mappings) between concepts in the design models. The mapping mechanism can be used to check the consistency of the design models or to propagate changes in one design model to another.

The concepts in this work have been implemented in two prototypes that have been used in three scenarios in the design process of the automotive domain. Employing ontologies to make domain assumptions explicit and for enriching design models with domain knowledge has proved to enhance the quality of the models, to enable the reuse of design models and to support the interoperation of the engineering tools.

Acknowledgments

Writing this dissertation has been a learning journey for me. I am very grateful to all the people who have lightened my way and who helped me not lose the focus on the important things in life. Besides learning many interesting things about my research field, this experience also taught me lessons that will always enrich my life and which will hopefully help me become a better person.

I would have never accomplished this work without the guidance and support of Professor Bernd Mahr, who has accepted to advise my dissertation. In our numerous and very exciting discussions I could better understand different aspects of my work and to see connections to other domains, which were not obvious to me. Professor Mahr always supported me and gave me hope in the harder times. I am also very grateful to Professor Mark Musen for reviewing my dissertation and for putting pressure on me to finish writing it in time. Special thanks to Professor Uwe Nestmann for accepting to preside the PhD Committee.

I want to thank all my former colleagues from DaimlerChrysler, who have supported me and who gave me insight in the practical aspects of my research field. Thank you very much to my former team managers, Frank Müller and Michael Heinrich, who encouraged me even when I wanted to give up. Dr. Helen Leemhuis and Shahram Hami-Nobari were always very supportive to me. I have also learned a lot from the lively and very interesting discussions with Dr. Rüdiger Klein. It was a great pleasure for me to work with Andi Diaconu-Muresan and Adrian Mocan, who did parts of the applications' implementation that I have used in my dissertation. I am also grateful to Andi for the very nice and long bicycle tours around Berlin, in which I have found out that having a compass and a map is not a guarantee for finding the right way. During all these years, I have learned a lot from Ulrike Gramann about German language and culture; she was always a great support and an inspiration for me. Thank you very much to Felicia and Flavius Copaciu, Tudor Groza, and Jennifer Vendetti for proof-reading my dissertation. Luna Alani was always very kind and helpful to me. I am also grateful to all my colleagues from Stanford Medical Informatics, who have encouraged me in the last phases of writing up my dissertation.

I want to thank my family and friends for their enormous support throughout these years. Without them I would have never succeeded this learning journey. My mother gave me an example of perseverance in life and taught me to believe in my own abilities. My father helped me discover the computer-scientist in me. Lorena is a great friend, who helped me stay in touch with reality when I was too overwhelmed by the task that I have engaged in.

Anca and Gabi showed me that dreams can be lived and inspired me for the future. One of my closest partners in this journey was my friend, Felicia, to whom I am very grateful for her help. I wouldn't have written this if it wasn't for the support of my wonderful husband, Csongor, who showed me understanding and who was beside me through good and bad times all these years.

Thank you everybody!

Contents

1. Introduction and Background

1.1. Background

The demanding market situation in the very innovative and customer-oriented automotive industry, imposes new challenges on the product development process. In order to be successful in this market, three challenging goals have to be achieved at the same time: a shorter time to market, improved product quality and low product costs. Several factors and trends make the accomplishment of these goals even more difficult, including the high complexity of products, the variant explosion due to excessive mass customization, and a high innovation rate due to customer expectancy for new functionalities. These functionalities are realized by means of mechatronics – a word made up by mechanics and electronics – in which the close integration of mechanical, electrical engineering, and information technology makes possible new fundamental solutions with an improved cost/benefit ratio and opens possibilities for new, as yet unknown products [VDI 2206, 2004].

In response to the new trends and their inherent challenges, the product development process adapted by evolving to a digital (computer-based) and integrated process [Krause, 1999; Ehrlenspiel, 2003]. The information technology thereby plays a key role. According to Karcher et al. [2001] only companies that own information systems in the product development process, that are flexible, and are adaptable to the changing conditions, will be able to survive in hard market competition.

The IT processes and systems have to fulfill new requirements because of the interdisciplinary and hence complex nature of new mechatronic products. Compared to the traditional purely mechanical systems, the increased complexity of the new products results from the greater number of components that need to be coupled together and from the heterogeneous nature of mechatronics, where components from different disciplines need to work together in order to fulfill the functionality of the product. This challenge can be mastered only by using a model-based approach in the development process [Struss and Price, 2004]. The promised goal is a better way to manage the complexity of products and a reduction in the design iterations and their inherent costs. For this purpose, a new development methodology for mechatronic systems has been developed [VDI 2206, 2004], which is based on methods from systems engineering, software development, and quality assurance. The methodology stresses some important desiderata for an efficient development process, such as an integrated approach for

product development supported by model-based techniques, early functional validation, and quality assurance in all development phases.

One of the methods to speed up and improve the design process is to reuse predefined components out of a model library. In many cases, the architecture of the system to be designed is roughly known at the beginning of the process. Engineers work with functional solution templates [Motta, 1998] to guide the design process, which realizes great reductions in the complexity of design. This approach is supported by the parametric design task, which has been formally defined in [Wielinga et al., 1995] and applied in different frameworks [Motta et al., 1999]. The goal of the parametric design task is to find a structure composed of parameterized building blocks that satisfies all the requirements and does not violate any constraints.

1.2. The Challenge

In each development phase, the engineers employ specific methods and tools that support them in fulfilling their engineering tasks. The development methodologies for the different disciplines have been established and used for a long time. Until recently, there was no need for integrating them. However, with the emergence of mechatronic products, engineers were faced with a new challenge: They had to deal with a multitude of heterogenous models originating from different engineering tools and to make sure that in the end all models offer a coherent view of the product. There are no mature processes and tools that enable the exchange of models between the different parallel developments.

Some of the tools employed in the development of a complex engineering product are: computer aided design tools (CAD) – used for geometrical design, computer aided engineering tools (CAE) – involved in the analysis of the different properties of a product, different simulation tools, functional modeling tools, requirement engineering tools and so on [Ehrlenspiel, 2003]. Each of these tools provides a custom-tailored solution for a particular task in a particular stage of development, which creates a complex network of partial solutions. All of these systems islands are usually proprietary and do not interoperate. The data exchange between pairs of engineering tools is based on hard-coded translation rules, is error-prone and difficult.

In the development of mechatronic systems, the main challenge is the insurance and testing of the overall product functionality, which requires the integration of inter-disciplinary components into a mechatronic system. This requires in turn the cross-domain communication and cooperation between the different disciplines and the interoperability of their associated IT systems. At present, there are no environments for the integrated development of mechatronic systems. The KOMFORCE reference model, proposed in Gausemeier et al. defines four integration levels for the IT systems needed to achieve an integrated development platform: the method level, process level, model level, and system level.

The engineering tools involved in the development process operate with different models of a

product. Although the models are developed by separate teams in a concurrent fashion, they represent the same product under different viewpoints. In such a scenario, the consistency of the models becomes a crucial issue. The risk of developing inconsistent models is very high and may produce huge costs if inconsistencies or incompatibilities between models are discovered too late in the development process.

The current model-based design faces several impediments: tools store large quantities of design data in proprietary representations, the handling of different models and views on the product is hard to manage, the risk of redundancy and inconsistencies is sizeable, which also prevents an efficient development process.

The design models are developed by different teams for different purposes and to solve different tasks. Therefore, each design model only describes the aspects of the product that are relevant to their specific task. Figure 1.1 shows the challenges that must be tackled in order to enable an efficient development process. First, there are different modeling languages in which the design models are specified with their own syntax and modeling methodology. Second, the models have different conceptualizations of the product according to their own view of the world. Very often many aspects of the model remain implicit and are known only to the engineers who have built the model. It is hard to integrate different models at the conceptual level because the semantics of the models remains implicit and hidden in domain assumptions. Third, there are no explicit or formal correspondences between the model elements to enable the automation of the tasks which involve several design models, such as consistency checking, change propagation or model generation.

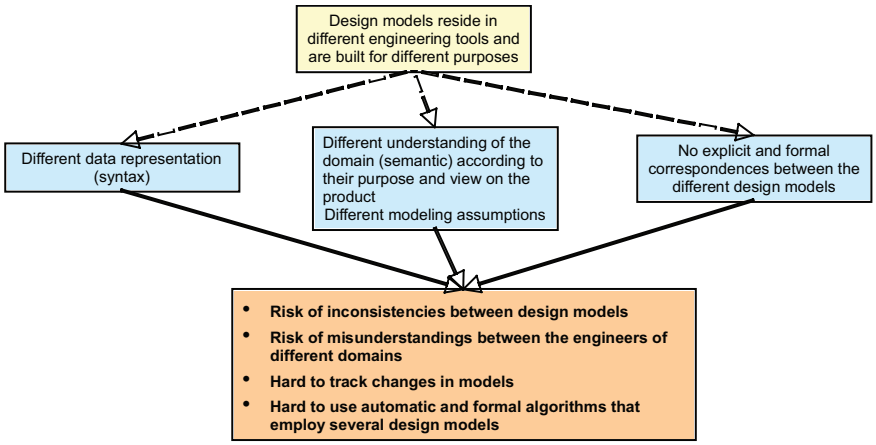

Figure 1.1.: Challenges in concurrent development process.

1.3. Research Question

The central question investigated by this thesis is:

> *How can an improved concurrent model-based design process be supported, in which design models describing different views of the same product are built in parallel development branches, and the consistency of models needs to be maintained throughout several design iterations?*

This research raises three concrete questions that will be investigated in this work:

Q1. How is the consistency of a design model and the consistency between models in different viewpoints defined?
In order to support an efficient design process, the consistency of design models needs to be maintained in different design iterations. Another type of consistency is the one between different design models, which is crucial in a collaborative development process. A formal definition of the consistency is needed in order to support the automation of the consistency checking process.

Q2. How can the representation of the current design models be improved, so that it will support a more efficient construction of consistent design models?
The current representation of design models relies on many implicit domain assumptions. Checking their consistency is often done in a manual way, which involves a great effort and is also very error-prone. This approach is not feasible for a highly dynamic development process, in which the consistency between design models is checked at every design iteration. A formal representation of the design models that also takes into consideration the domain assumptions is needed in order to support the consistency checking for a design model and between several design models.

Q3. How can the consistency between different design models be checked in an automated way?
There are several obstacles in checking the consistency of models in an automated way, ranging from purely syntactic issues to conceptual differences in the design models. A solution needs to be found that tackles the differences in the design models and supports automatic consistency checking of the models.

1.4. Contributions

This thesis contributes to improving the model-based design process in the engineering domain in two ways:

1. By enabling the construction of higher quality design models
2. By improving the processes that operate with the models.

The first goal is achieved by using a more precise and formal description of engineering models using ontologies. An ontology is a formal representation of an abstract, simplified view of a domain that describes the objects, concepts and relationships between them that hold in that domain [Gruber, 1993]. Ontologies support knowledge sharing and reuse and provide a guarantee of consistency between models that commit to the same ontology. The second goal is contributed by an approach to maintaining the consistency between design models that describe the same product from different viewpoints.

The contributions are fleshed out in the following bullet points.

- **A formal definition of the consistency between design models**
  Ensuring the consistency between different design models is crucial in a collaborative development environment. The later the incompatibilities or inconsistencies between models are discovered, the higher the costs for correcting them. Automated methods for checking the consistency between the design models are needed in order to enable an efficient development process. A contribution of this work is the investigation of the consistency between engineering models from a formal perspective. I have extended the definitions from the Reference Model for Open Distributed Processing (RM-ODP) [RM-ODP/1, 1998], which already provides the main concepts involved in a distributed development environment, such as specifications, realizations and viewpoints. I have applied the consistency definition from RM-ODP to the engineering development process and put forth first order logic definitions. I have also identified two roles that the design model plays in the development process: as a realization and as a specification. I have also discussed the challenges encountered when dealing with multiple viewpoints of the same product. Taking into consideration the different conceptualizations of the viewpoints, I gave a formal definition of consistency between design models using concepts from distributed first order logic.
- **The Engineering Ontologies**
  I have used ontologies to enrich the representation of the design models with domain knowledge and to support the sharing of knowledge between different models. I have identified that all design models in engineering are using similar representation patterns, such as part-whole relationships, connection relationships and constraints. In order to improve the knowledge sharing between the models, I have developed three ontologies, Components, Connections and Constraints that may be used as upper ontologies for the representation of the design models from different viewpoints. I have also described a part-whole modeling pattern that is suitable for representing the system decomposition in a design model. The Requirements and Constraints ontologies may be used in modeling the requirements of a design model and to check whether the model is consistent.

- **The mapping framework**
  I have developed a framework for mapping between different ontologies, which describe design models in different viewpoints. The mapping framework allows the definition of correspondences (a.k.a., mappings) between general model components, which may be instantiated and applied to interrelate concrete design models. The centerpiece of the framework is a *Mapping* ontology, which stores the mappings between the ontologies in a declarative way. It provides support for mapping on paths in the ontologies, which was necessary in order to map between complex system structures. The mapping framework may be used for performing consistency checking, change propagation between the models, or model skeleton generation. It also supports the composition of mappings which proved to be an important feature for improving the mappings modularity and reusability.

1.5. Organization of the Thesis

The thesis is organized in chapters as follows:

Chapter 2 describes the context of research. It provides an overview of the mechatronic development process and identifies the major challenges in the process.

Chapter 3 defines formally the consistency between engineering models by extending concepts from RM-ODP. It also shows the different roles a design model plays in the development process. It also gives a definition of consistency at conceptual level between different design models using concepts from distributed first order logic [Ghidini and Serafini, 1998].

Chapter 4 describes the engineering ontologies: the *Components* ontology, used for modeling part-whole relationships; the *Connections* ontology, used to model connections between components of a system; the *Requirements* ontology, used to capture requirements on the design model and to relate them to the model; and the *Constraints* ontology, used to represent constraints between properties of the design models.

Chapter 5 describes the mapping framework, which is used to interrelate different ontologies of design models. It describes the Mapping ontology and shows how the mappings can be composed to map complex systems.

Chapter 6 shows two uses cases in which the developed concepts have been implemented and validated. The first use case is from the requirements engineering domain. The second use case shows how the mapping between design models can be used to support consistency checking.

Finally, in Chapter 7 I discuss conlusions and point out future work.

2. Context of Research

2.1. A Motivating Scenario

In the following I will show a motivating scenario taken from the development of an automatic transmission gearbox of a car. The automatic transmission is a very complex mechatronic component that possesses all characteristics of a typical mechatronic product: It is a very complex system because it is built out of a large number of components of different types (mechanical, hydraulic, electrical and software) that all need to interoperate together in order to achieve the overall functionality of the product.

During the development of a mechatronic product, different design models are realized by domain experts that model a certain aspect of the product, like for example, geometrical models, functional models, multi-body models, hydraulic models, software models, etc. They are used both in synthesis and analysis steps involved in the development process of mechatronic products as described in Section 2.2.1.

The models are built in a concurrent fashion by teams of engineers with different expertise and interests in domain-specific tools. Even if at the first glance the models seem to be independent of each other, they represent the same product from different perspectives. The same characteristic of a product, such as for example, the inertia of a component, might be described differently in different modeling tools: It may have different names, data representations, modeling assumptions, or it may be only implicit in one tool and can be inferred by other characteristics of the model, while in the second tool it may be represented explicitly.

The differences between the models enumerated above are critical, if the models did not have to interact with each other and exchange their data. I will give in the following an example of such a situation.

In the development of the automatic transmission, two models of the systems are realized in a parallel fashion: a geometrical model designed in a CAD tool, also called Digital Mock-up and a functional model designed in a simulation tool, also called Functional Mock-up, shown in Figure 2.1.

The geometric model is built using feature technology and describes the geometry of the automatic transmission. It is used to test whether the product can be physically produced. It contains information related to features, for example, a drill hole, the positioning of geometric elements relative one to anther, information about how they are connected physically, material

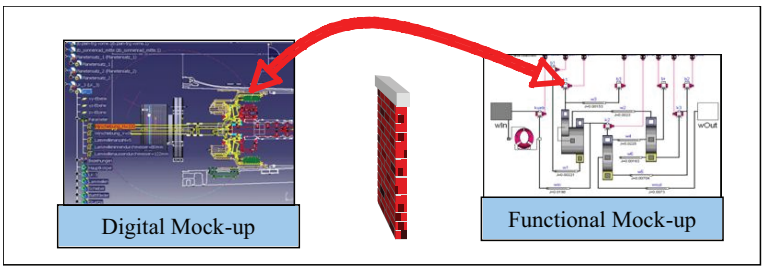

Figure 2.1.: The digital and functional model of a gearbox are developed by different teams in a parallel fashion.

information, etc. An excerpt of the geometrical model for a planetary set is shown in the right hand side of Figure 2.2.

The functional model is a mathematical model that describes the behavior of the automatic transmission and is used for the functional validation of the design using simulation. It describes the function of the system, which is built out of functional components. The functional components have ports through which they communicate with other components and their behavior is described by mathematical equations. An example of a functional component of the automatic transmission gear box is the *IdealPlanetary* with no inertia. It is defined by its components (sun, carrier, ring), its properties (the transmission ratio) and its behavior described by mathematical equations. The definition of the *IdealPlanetary* in Modelica[1], an object-oriented language for describing physical systems, is shown in the left hand side of Figure 2.2.

The models are used to test different types of requirements that the products must satisfy. For example, the geometrical models are used to test if the product can be realized from the technological point of view, or if the total weight of the product will conform to the imposed restrictions, while the functional model is used to test if the functional requirements can be satisfied, how the system will behave in different situations, how the system reacts to different faults, etc.

The two models describe the same automatic transmission gear box, but from different perspectives. There are elements in the two models that describe the same physical component, which is not obvious by only comparing the names of elements used in the two models. For example, the *sun* component of the *IdealPlanetary* in the functional model describes the same physical component as the *Sonnenrad* from the geometrical model. The models do not share all their elements and data. Each of them contains elements and data that are only relevant for their particular domain and are not present in the other model. For example, the functional model contains the mathematical description of the behavior of the component, which is not present in the geometrical model, since it would be irrelevant for the domain and for the goal

[1] http://www.modelica.org

```
model IdealPlanetary "Ideal planetary gear box"
  parameter Real ratio=100/50
    "number of ring_teeth/sun_teeth (e.g. ratio=100/50)";

  // kinematic relationship
  Interfaces.Flange_a sun "sun flange (flange axis directed INTO cut plane)" ;
  Interfaces.Flange_a carrier
    "carrier flange (flange axis directed INTO cut plane)" ;
  Interfaces.Flange_b ring
    "ring flange (flange axis directed OUT OF cut plane)" ;
  ;
equation
  (1 + ratio)*carrier.phi = sun.phi + ratio*ring.phi;

  // torque balance (no inertias)
  ring.tau = ratio*sun.tau;
  carrier.tau = -(1 + ratio)*sun.tau;
end IdealPlanetary;
```

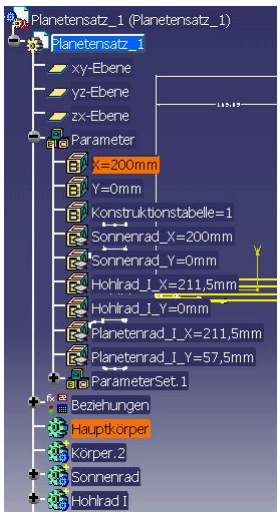

Figure 2.2.: The left half of the figure shows an excerpt from the gearbox functional model in Modelica. The right half of the figure shows a screenshot from a CAD tool of a geometrical model for the same gearbox.

of the geometrical model.

Another example is that the geometrical model contains in the description of the multiple-disk clutch used by the automatic transmission the number of clutch disks, since this is an important geometric characteristic of the clutch. The functional model does not contain this information and describes the multiple-disk clutch using a parameter, *pressure2torque* that depends on the number of clutch disks from the geometrical model. But this relationship is not explicitly stated in any of the models. This knowledge actually describes how the two models relate to each other and it is necessary in order to test, whether the two models are consistent with each other.

The risk of parallel development of the design models is that possible incompatibilities between the models are discovered only very late in the process. For this reason, at certain milestones, the design models must be synchronized with each other. Model synchronization is necessary when a change occurs in one model, for example if a component has been replaced with another one, then the other model needs to be updated to reflect this change. One change in a model may trigger other changes, if the requirements of the models are not satisfied.

An example of a change scenario is described briefly in the following. A new sport variant of a car should be constructed that uses the same automatic transmission gear box and a more powerful engine. In the first step, the engineers will update the functional model of the car, which contains the automatic transmission model as a component, with a car engine with an increased power, meaning that the torque generated by the engine will also be bigger. In the

second step, the functional simulation of the model will indicate that for this case, the clutch will slip after changing gears because it is not able to transfer the bigger torque. This means that the functional requirements are not satisfied by the model. In order to solve this problem, the engineers must find a way to transfer the bigger torque without the clutch slipping. One solution is to use a "stronger" clutch, which can be physically realized by adding a new disk in the clutch. Since this is a constructive change, it must be done in the geometrical model. In order to test again the behavior of the model, the changes in the geometrical model have to be reflected back in the functional model. This can be realized by a model synchronization procedure that is aware of the dependencies between elements in the two models. For this particular example, the *pressure2torque* parameter used in the functional model of the automatic transmission must be computed out of several parameters in the geometrical model. This is a highly iterative process and goes on until the model has reached the desired level of maturity and the domain requirements are satisfied.

An important feature of the design process is that the models are built out of model fragments defined in domain specific model libraries. The model libraries contribute to a more rapid development and to an increased model quality by reusing existing components that have already been validated in other designs.

There are different impediments in the current parallel development process. One of them, which has been illustrated in this section, is model synchronization. In many cases, model synchronization is done manually by exchanging Excel sheets or text files. It remains the tedious task of the domain engineer to understand and interpret the changes done in the other model and to integrate them manually in his own model. This process has to be done at each synchronization cycle. It is not automated because the dependencies between the models are not stated explicitly anywhere. Usually domain engineers do not have an understanding of the other domain models and it is hard for them to track what effects their changes might have on other models, especially if the models have a great complexity.

2.2. Overview of the Product Development Process

2.2.1. The Development Methodology for Mechatronics

Mechatronics is an engineering science, in which the functionality of a technical system is realized by a tight interoperation of mechanical, electronic, and data-processing components.

There are several definitions of mechatronics, but a well accepted one is given by [Harashima et al., 1996] that will also be used throughout this work:

Definition:
"*[Mechatronics is]... the synergetic integration of mechanical engineering with electronics*

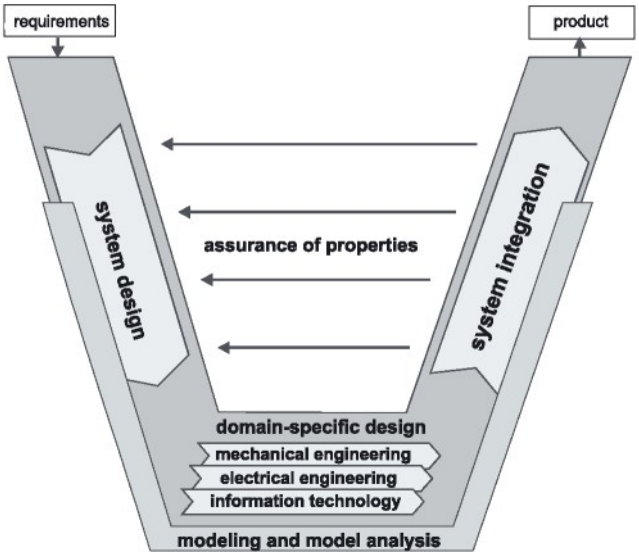

Figure 2.3.: The V-Model of the mechatronic product development process.

and intelligent computer control in the design and manufacturing of industrial products and processes"

Although many mature and well-known design methodologies for mechanical engineering have been established (an overview and comparison can be found in [Pahl and Beitz, 1996; Roth, 1982]), they could not be applied to the mechatronics system due to its complex and interdisciplinary nature. This problem was solved in 2004, when VDI[2] developed and published a new methodology for the design of mechatronics systems, known as VDI 2206 [VDI 2206, 2004].

The methodology relies on a flexible procedural model that is supported by three elements described briefly in the following paragraphs:

- The V development model as a macro-cycle
- General problem solving cycle on the micro-level
- Predefined process modules for handling recurrent working steps

The **V model** (Figure 2.3) as a development macro cycle has been adopted from the software development process and describes the general procedure for developing mechatronic systems [VDI 2206, 2004; Gausemeier, 2005].

[2] Verein Deutscher Ingenieure

The starting point is the definition of **product requirements** which will be used to measure and asses the designed product later. According to Stevens et al. [1998], requirements can be classified into user and system requirements.

User requirements are the first step towards the definition of a system. User requirements are usually short, non-technical and in a form that users can easily understand and correct. They contain aspects such as capacity, comfort, safety, etc.

System requirements evolve from user requirements defining what the system must do to meet them. System requirements describe what a system will do, not how it will be done. The systems requirements form a model of the system, acting as an intermediate step between the user requirements and the design, and are often expressed using functional terms ("communicate", "transport", "supply power", etc).
One important aspect is the traceability of the user requirements to users and the traceability of system requirements to user requirements.

System requirements contain both formal and descriptive information, which serve different purposes [Stevens et al., 1998]:

- They give an abstract view of the system.
- They allow trade-offs, exploration, and optimization before committing to a design.
- They demonstrate to users that their needs are reflected in the development.
- They provide a solid foundation for design.

In the **system design** phase, a cross-domain solution concept is established that describes the main physical and logical operation characteristics of the product. According to VDI 2206 [2004], identifying system requirements is also a part of the system design phase. The overall product function is decomposed in subfunctions, such as "*driving*", "*transmission*", etc.

The subfunctions are linked to each other by flows of material, energy and information [Pahl and Beitz, 1996] to form together the functional structure that describes the behavior and which can also be used to detect inconsistencies between the functions. The aim is to associate to each subfunction operating principles and solution elements that can perform the function. This is done in an alternation of analysis and synthesis steps (See below the development micro-cycle). An example of a subfunction is "*transfer torque*" which could be realized by the physical effect "*friction*" against a cylindrical working surface. In accordance with Coulomb's Law ($F_F = \mu \cdot F_N$), depending on the way in which the normal force is applied, the function will lead to the selection of a shrink fit or a clamp connection as the working principle. This example is take from [Pahl and Beitz, 1996], where also rules on how to asses and select the best solution are given.

In the **domain-specific design** phase, the developed principle solution is further concretized in domain-specific solutions (such as, mechanical, electrical and information domains). For each domain there are well-established development methodologies that are employed.

The **system integration** phase aggregates the results from the different domains to form an overall system. This is a very critical phase because the interactions between the partial solutions must be achieved. System integration is understood as a means to bring together parts (functions, components, subsystems) to form a superordinate whole. Other activities that are carried out in this phase are identifying incompatibilities of the part solutions, eliminating theses incompatibilities, and finding an optimum overall solution.

The **assurance of properties** phase ensures that the actual system properties coincide with the desired system properties. This verification is done regularly during the development process based on predefined requirements and solution principles. Modeling and model analysis are activities that are part of several phases in the design process, which operate on computer models of the product. The design of mechatronics system without the assistance of computer-based models and computer aided tools is not possible due to their complex and inter-disciplinary nature.

In the system design, domain specific design and system integration phases of the mechatronic development process, it is essential to test if the developed system conforms to the requirements. This step is crucial, because incompatibilities and conflicts in design have to be determined early in the process, so that costly iterations in the later development phases are avoided.

According to the mechatronics development methodology, the assurance of properties process encompasses two activities: verification and validation, which describe different stages of ensuring the required system properties.

Verification answers the question "*Are we building the product right?*" Verification is the process of determining that a model implementation accurately represents the developer's conceptual description and specifications that the model was designed to [DoD, 2001]. Verification is done generally using formal algorithms.

Validation answers the question "*Are we building the right product?*" Validation is the process of determining the manner and degree to which a model is an accurate representation of the real world from the perspective of the intended uses of the model, and of establishing the level of confidence that should be placed on this assessment [DoD, 2001]. Validation is usually not done in a formal manner.

The end result of the V macro-cycle is a complete design specification. The V macro-cycle may also be applied for different maturity levels of the product, such as the laboratory specimen, the functional specimen, the pilot-run specimen, etc.

The **micro-cycle** used in each phase of the V macro-cycle is problem-solving, which is borrowed from systems engineering. Its phases are shown in Figure 2.4. For a detailed overview,

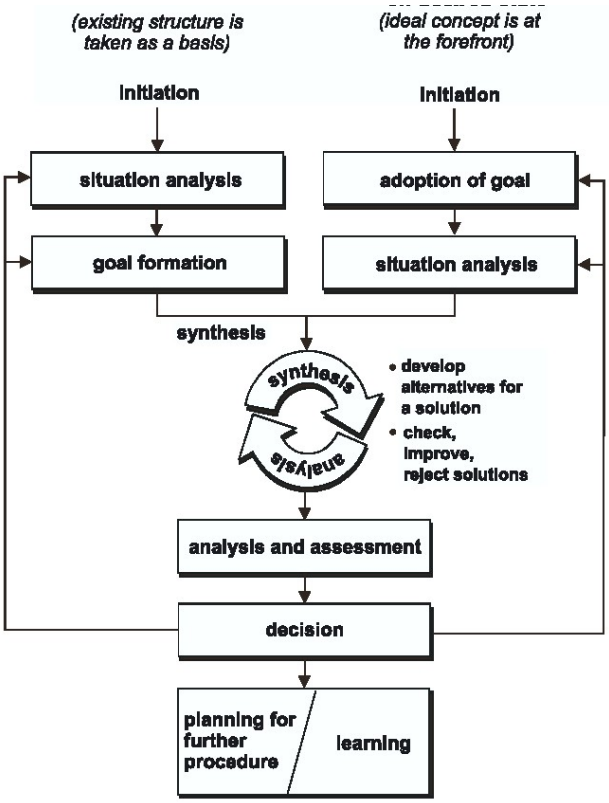

Figure 2.4.: The problem solving cycle applied in different phases of the product development.

please consult [Daenzer and Huber, 2002; Hubka, 1984; Patzak, 1982]. A relevant aspect is that the process involves permanent iterations between analysis and synthesis steps with the final goal to obtain an optimized design.

2.2.2. Challenges and Trends in the Current Product Development Process

The demanding market situation imposes new challenges on the product development process, especially in the very innovative and customer-oriented automotive industry. Shorter time to market, a high complexity of products, variant explosion due to mass customization, new functionalities realized through electronic control units and software require new strategies and tools to solve new challenges such as: front-loaded development, assembly oriented design, concurrent engineering, parametric and knowledge-based design.

Three overall goals are the root of challenges in product development: a shorter time-to-

market, an increased product quality and reduced production costs. While trying to achieve these goals, the manufacturers have to tackle conflicting subgoals. Products become more complex by the introduction of electronic and software components that fulfill many critical functions. Meanwhile, the processes and IT systems have to be adapted, so that they accommodate the development methodologies of the newly introduced disciplines (such as electronics and software). Products have to be brought to market in a much shorter time, which also means that the product development process is much shortened. Collaborative and simultaneous engineering are used to achieve this goal, but they come with other challenges related to the synchronization and integration of partial design models from different development branches.

In the following, I will give a brief description of the current situation and trends in the product development process for the automotive industry, together with the key challenges that motivate the need for employing new technologies with the goal of achieving a more effective product development. Needs and requirements in the current development process are also identified.

Innovations through a high number of mechatronic products One trend that could be observed in the last years is the increase of the number of mechatronic systems used in the automotive products, breaking with traditional design, in which all parts used to be purely mechanical. Mechatronics is an engineering science, in which the functionality of a technical system is realized by a tight interoperation of mechanical, electronic and data processing components. The importance of mechatronics comes from the fact that about 90% of innovations in the automotive domain originate from the field of electronics, out of which 80% is actually implemented in software [IAM, 2002]. After microprocessors were introduced for engine control, they were also used in chassis systems, e.g. ABS[3] and ESP[4], then in the body, e.g. for the air conditioning or vehicle access system. Other mechatronic products are the electrically actuated ones (X-by-wire), such as “shift-by-wire” or “brake-by-wire”.

The mechatronic break system (“brake-by-wire”) is an example of the tight coupling between different types of components, which is used to achieve the braking function of a car. The breaking system in a passenger car is built out of four mechatronic wheel brake modules [Roos and Wikander, 2003]. The information flow between the break pedal is electronic and the power is transferred to the brake modules through the vehicle’s electrical system. The whole break functionality, for example, ABS and traction control system can be implemented in a flexible way through control algorithms implemented in the software.

The tight interoperation between cross-domain components (mechanical, electrical and software) to achieve new functionalities of a product requires many changes in the current IT

[3] ABS = Anti-lock Braking System
[4] Electronic Stability Program

systems. The software development process has to be integrated into an existing purely mechanical process and system world. New tools and processes have to be defined that can handle the complexity of new products.

New data representations have to be conceived to support the tasks in the mechatronic development process, such as checking the consistency between different models of a product. This task is essential, especially in the integration phase, in which different models have to be aggregated to form a design specification that fulfills all its requirements.

Heterogeneous IT Systems Another aspect of the new development process is the overwhelming increase in the IT systems that support the product design in the automotive industry. Each of these systems provides a custom-tailored solution for a particular task in a particular stage of the product life cycle, which creates a complex network of partial solutions. All these system islands are usually proprietary and do not support a uniform data model. The data exchange between pairs of engineering tools is based on hard-coded translation rules, is error-prone, and is difficult. The change propagation between the design models is not supported in the current IT systems environment.

Virtual Prototyping Using test models and prototypes is an important aspect of the development process that allows product trials to be performed before mass production. Building and testing physical prototypes is very time and cost intensive. Consequently, there are many efforts to minimize the number of physical prototypes to be constructed. This is realized by employing virtual prototypes, which are computer models used in different computer-aided tools and which can support the virtual testing process. Virtual prototyping has many advantages in the development of mechatronic products, in which the interactions and increasing complexity introduced by coupling of previously independent subsystems from different disciplines, require model validation in early design phases. This enables the investigation of different design configurations with low time and cost investments.

Top-down design methodology: the assembly-oriented design Another trend in the current development process in the automotive industry is that of assembly-oriented design. The traditional design process, in the times of drawing-board design, was a top-down process, meaning that first an initial design at product level was created and then it was further refined into the design of atomic parts. The introduction of 3D CAD (Computer Aided Design) tools changed the development process from a top-down to a bottom-up approach. In this case, the atomic part level of a product is first being designed, and then the parts are aggregated to form a complex product.

In the current development process, the traditional top-down approach has come back in force, because the importance of a conceptual and systematic design has been recognized. New

design methodologies, such as [VDI 2206, 2004], have been developed and are already supported in CAD tools. However, the engineering data management (EDM) tools remain mostly bottom-up and part-oriented. The complex relationships between the individual parts of an assembly are often implicit and hard to manage [Burr et al., 2003].

Concurrent and Simultaneous engineering A natural consequence of the reduced development times is the concurrent and simultaneous engineering that involves the parallelization of development activities. Simultaneous engineering is based on a simultaneous and parallel proceeding in all phases of the product development. A development phase is started before the previous one has ended. Concurrent engineering means that several engineers are working in parallel to solve the same task. The task is decomposed in subtasks that are solved by different teams. This is realized by task decomposition and allocation, and at the end of the process, the integration of the partial solutions in an overall solution. This raises several challenges. One of them, which will be addressed in this work, is the synchronization of different engineering models that implies a consistent and automated knowledge transfer between the applications. Since the different engineering applications are focused on solving specific partial problems, the question arises on how to insure a true interoperability between these applications under consideration of their specific point of view on the domain and on their own interpretation of the domain. The semantics given to the models in the different engineering tools is often hidden and not transparent to a machine for automated processing.

Front-loaded development It is well known that the greatest proportion of costs is determined by decisions made in early phases of product development. Wrong decisions made in early phases, are propagated to all the following design steps. The late discovery of these failures results in high costs. Many design activities that have been previously done in the later phases, for example verification and validation, or planning activities, are now moved, or "loaded" to the early phases of development. This implies also a greater responsibility that is transported in the early design phases, because each design decision has to consider the consequences and effects on the later phases of development. The product design has to be validated very early in the process, which may require several iterations between design and analysis tools, until a design solution is obtained that satisfies the requirements. A transparent and consistent data exchange between the engineering tools is needed to support the iterative design process.

Function-oriented design Function-oriented design has emerged from the need of a more efficient development process that supports modularization and reuse at the conceptual level. In function-oriented design, the function of a product plays a central role [Leemhuis, 2004; Krause et al., 2003]. Well defined relations between the product functions and requirements

and between the product function and the part structure are valuable for a systematic documentation of the product [Krause et al., 2003]. An efficient requirements engineering process is needed to support an improved development process.

In mechatronic products, functionality of a technical system is realized through a tight coupling between mechanical, electronic and information processing components. From beginning of the system design, an integrated functional view of the system has to be achieved. The driving force in this process is represented by the functional requirements. Starting from the product specification, the functional requirements describe the product characteristics and parameters; they are then further used in the design layout phase as specification parameters, and later they are involved in checking whether the design conforms to the specified requirements.

Good data models and representation capabilities are needed in order to support the knowledge transfer throughout the development process. Requirements refinement and traceability, as well as the representation of the allocation of requirements to systems are essential in order to support all the phases of the development process.

2.3. Models

2.3.1. Definition of Model

The word "**model**" comes from Middle French word "*modelle*". This stems from an Old Italian word "*modello*", which is assumed to come from the Vulgar Latin "*modellus*". This emerged from Latin "*modulus*" (meaning "measure", "quantity", "scale"), which is a diminutive of "*modus*".

A model is an abstract, simplified representation of an original that can be seen along two dimensions: The model is a "model of something" and meanwhile it could be a "model for something".

According to the general model theory of Stachowiak, a model has the following three properties [Stachowiak, 1973]:

- *Mapping feature*: A model is associated, or mapped to an original. They are representations or reproductions of the originals that may be in their turn also models. The originals can be artificial or natural.
- *Reduction feature*: A model does not contain all attributes of the original, but only a selection of the original's properties that are relevant to the model builder or to the model user.
- *Pragmatic feature*: A model should be able to replace the original in the context of the purpose for which it has been created.

From the first two features, models are abstractions and simplifications of the reality that show only aspects of it. The mapping feature implies that a model is a "projection" of the reality, in which certain aspects are lost during the process of abstraction. What is retained depends on the purpose of the model.

Models are always built for a purpose. They are not only used to determine certain properties of an artifact, but are also a means for controlling, communication and instruction.
According to the theory of engineering systems [Hubka, 1984], models can be classified with respect to their means of representation in:

- *Iconic models* are a 2D or 3D representation of the original that may also be scaled, such as, drawings, spatial models of machines, photos. There is a considerable similarity between the original and the model.
- *Analogical models* have only certain properties similar to the original. With the help of these models, the static and dynamical properties of the original can be emulated. Examples of these types of models are graphs and diagrams.
- *Symbolic models* represent the static and dynamical properties with the help of words or mathematical symbols.

Another classification that is of interest for engineering domain is in physical and mathematical models. Physical models are real, physical objects that mimic some properties of the real system that are built to answer questions about the system. It is common that physical models of systems are built, sometimes called also Physical Mock-Up (PMU), that have the same shape or appearance as the real objects to be studied, for example with respect to aerodynamics or esthetics.

Mathematical models are often employed in engineering for systems analysis. They are descriptions of a system, where the relationships between the variables are in form of mathematical expressions. Variables can represent measurable quantities, like size, length, temperature, etc. The behavior of the systems is usually described by mathematical equations.

Mathematical models can be further classified according to different axes [Tiller, 2001]:

- *Linear vs. nonlinear models*: A model is linear if the equations or constraints in the model are all linear. If one or more equations or constraints are non-linear, then the model is known as nonlinear.
- *Dynamic vs. static models*: A static model does not take account of the time component. Static models are used to describe steady-state or equilibrium situations, meaning that the outputs do not change, if the input changes. Dynamic models include time in the models and are typically represented using difference equations or differential equations.

- *Deterministic vs. probabilistic models*: Deterministic models perform in the same way for a given set of initial conditions, which does not hold in a probabilistic model because randomness is also present.

- *Quantitative vs. qualitative models*: Quantitative models have their variables represented numerically according to a given measurable scale. Qualitative models do not have the same precision and use classification of variable values into a finite set.

2.3.2. Characteristics of Good Engineering Models

[Neelamkavil, 1987] gave an illustrative definition of a model for the engineering domain: "*A model is a simplified representation of a system intended to enhance our ability to understand, predict and possibly control the behavior of a system*".

A good model is one that helps us achieve the purpose for which it was designed. A good model for engineering design has several characteristics [Stevens et al., 1998], which are briefly described below.

For modeling and simulation purpose, models provide an abstracted view of the world that contains only elements relevant for the analysis. Models must be *predictive and explanatory* – meaning that they should show only the essence of what is analyzed and should reveal new information to its user.

Models are used to take design decisions. That is why it is important that one can trust a model that it represents the reality correctly. A model should be *faithful*. Even if it is not complete, it should state the elements which are not modeled and the assumptions that are made when building the model. The model should be *verifiable* against reality. One method for model verification is to test the model against information, which was not used during the modeling process.

A model should be as *simple* as possible, meaning that only relevant information that helps to achieve the purpose for which the model was built, should be kept in the model.

Clarity is also an important characteristic of good models. The model should show unequivocally what will happen in the real world.

Models should be built in such a way that they are *neutral* to any solutions. This means that the modeler should not build the model so that it will reflect his own preference. Any viable solution should emerge from the model.

Models should also be *tractable*, which means that they can provide results with a reasonable effort or in reasonable time. A model that needs an unrealistic amount of resources or too much time to get to a result is not useful.

Other requirements for a good model in engineering domain can be derived from the development methodology for mechatronic systems (Section 2.2.1) or from the challenges and trends

in the development of engineering systems (Section 2.2.2). They are briefly outlined in the following paragraphs.

Models should be *maintainable*. The increased product complexity, their hybrid nature and the fact that models evolve and change very often during the design process requires adequate methods for managing their life cycle. Another requirement on the models is that they should be *modular* and *compositional*. This may be realized by defining clear interfaces to the environment, through which models may communicate with other models. The model of a complex system may be built up from models of smaller interconnected systems, which also support the reusability of models.

2.3.3. Models of Systems in Engineering

According to the ISO Standard [ISO/IEC 15288, 2002], a **system** is a combination of interacting elements organized to achieve one or more goals. Nonetheless, in engineering science, a system is also an object or collection of objects whose properties we want to study [Fritzson, 2004]. The central element in the latter definition is the fact that we want to study the system. The reasons for "studying" a system in engineering are usually to better understand it, to investigate its properties, or in order to build it.

Systems are built hierarchically: they consist of interacting elements that can be in turn other systems, too. Therefore, element and system are relative terms.
The **elements** of a system are interconnected and interact with each other in order to fulfill the overall goal. In order to completely define a system from the perspective of systems theory [Hubka, 1984], some more concepts will be introduced:
purpose, behavior, structure, environment, input, output, properties and state. A model of a system is shown in Figure 2.5.

The **purpose** or **intention** of a system is described as a set of interrelated goals. Some of the goals may be in a hierarchical relation (goal and sub-goals).

The **behavior** of a system is defined as the set of temporally successive states of the system. The behavior describes how a system responds to changes in its environment and/or in its own state. In engineering domain, to achieve a certain behavior of a system is a top-level goal. The **function** of a system is related to the way in which the behavior of the object is used to accomplish its intended use. The function of a system is seen as a stable property, a desired operation mode or effect, that is however not always achieved because the system may not behave as it should. The term function is always put in relation to the wanted effect.

The term **structure** refers to the internal organization, the ordering or the construction of a system. It is often also seen as a network of elements. The structure is formed by the set of elements of a system and the set of relations between these elements. This can be formalized as follows:

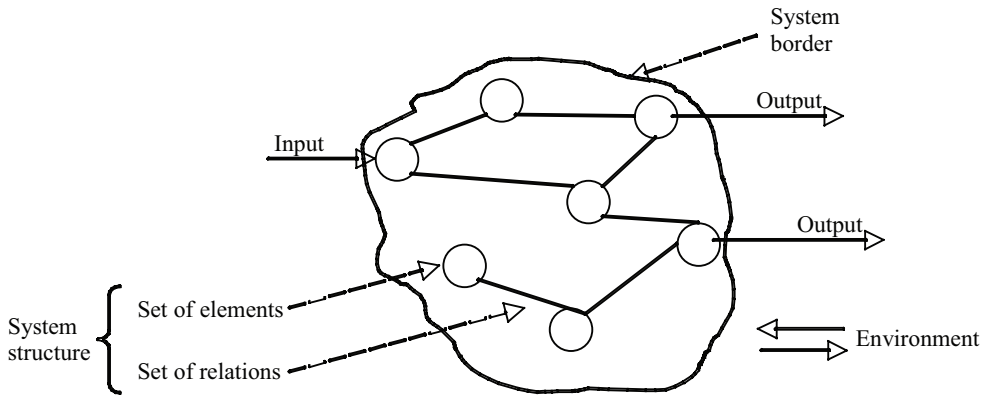

Figure 2.5.: A model of a system.

If,

$E = \{e_1, e_2 \dots e_n\}$; E is the set of elements; and
$R = \{r_1, r_2 \dots r_m\}$; R is the set of relations between the elements;

Then, the structure of the system is: $S = E \cup R$.

Structure and behavior are the most important properties of a system. The structure and function of a system are not independent of each other: The behavior of the system is determined by its structure. The opposite is not true, because a certain behavior can be achieved by different structures.

The **environment** of a system is theoretically everything that is not considered to be part of the system. But practically, the environment is the set of the systems that interact directly with elements of the system. For example, an output of an element of the system is input for an external system.

Through **inputs** and **outputs** the system interacts with the environment. There are many definitions of inputs and outputs from the point of view of different disciplines. Input and outputs describe different types of interrelations, both wanted and unwanted (disturbances) or other types of interrelations, like matter, energy or information flows.

In many systems, it is hard to make a difference between inputs and outputs, because the same variable acts as input and output. This is the case of acausal behavior, in which the relationship between variables does not have a causal direction, such as the case of relationships described by equations. What is input and what is output in a system is a choice of the observer who is guided by his interest of study.

In engineering domain, the inputs of a system are variables of the environment that influence

the behavior of the system [Fritzson, 2004]. These inputs may or may not be controllable by us. The outputs of the system are variables that are determined by the system and may influence the environment.

The system, its elements and relations have **properties** that define the system more precisely, such as, size, mass, speed, form, etc. A property is defined as a feature that is owned by an object and that characterizes that object.

The totality of all properties values of a system at a defined time moment is called the **system state**. The state can be imagined as a vector that contains the properties as its elements. By the definition of the system state only the relevant properties of a system are considered.

An important aspect of systems is that they should be **observable** [Fritzson, 2004]. Some systems are also **controllable**, meaning that we can influence their behavior by varying their inputs.

Hubka [1984] identified three typical tasks in the system design process:

System synthesis: If the behavior of the system and other requirements are known, find a system structure that behaves as stated and fulfills the requirements.

System analysis: If the structure of a system is known, find its behavior.

Black-Box Problem: If a system is given as input, for which the structure is not known, or just partially know, find the behavior and possibly the structure of the system.

2.4. Model-based Analysis and Design in Engineering Domain

2.4.1. Model-Based Design

The design task is defined by [Daenzer and Huber, 2002] as "the conceiving of a whole, a solution concept, the identifying or finding the solution elements required for this and the intellectual, model-based joining together and connecting of these elements to form a whole". Designing is according to this definition an activity that starting from requirements leads to a concretization of a technical system.

Model-based design uses models as a central component. In the course of the development process, several types of models are built that describe a specific aspect of the system and that are used in solving a specific task. Examples of model types are requirement models for describing the system requirements, CAD models that are used for describing the form of the system, or behavioral models that depict the behavior and function of the system.

For mechatronic systems, functional and behavioral models play a special role, because they are used for the integration of the models from different disciplines.

Like all phases of the mechatronic development process (see Section 2.2.1), the model-based design is also done according to the problem solving micro-cycle and includes iterations between the model synthesis and model analysis phases.

Many definitions have been given for the terms synthesis and analysis [Chakrabarti, 2002]. Even if it is argued whether analysis should follow synthesis or vice versa, there is agreement that both activities are needed in the design process. In the following, the definitions used in systems engineering will be used.

Synthesis comes from Greek and means originally combining, assembling, mixing, compounding of anything into a new whole. Pahl and Beitz [1996] define synthesis as "*putting together parts or elements to produce new effects and to demonstrate that these effects create an overall order*". An important aspect of the synthesis activity is that individual solutions or findings are put together in order to build an overall whole. In systems engineering, synthesis involves finding a structure of a system that fulfills the given requirements and that has the desired behavior. Synthesis is usually seen as the opposite of analysis.

Analysis means in the general sense the examination of something in detail in order to understand it better or to win knowledge from it. In systems engineering, analysis refers to the analysis of systems behavior when the system structure is already known (See Section 2.3.3).

Synthesis refers in model-based design to the activity of modeling, or building a model for a system and of its components. One concern is the quality of the models that are being built. Only a model that describes a system realistically can provide in the model-analysis phase results that can be transferred to the real system, or reliable knowledge about the system.

In the methodology for developing mechatronic systems [VDI 2206, 2004], several abstraction levels are used in the modeling of the behavior of a system (Figure 2.6). For example, topological models are used to describe the topology of the system, which means the arrangement and the inter-linking of the function-performing elements. The function structure can be used as input for defining a topological model. The function structure describes the solution-neutral decomposition of the overall function to elementary functions. There are different methodologies for determining the topology of a system out of the function structure, such as the one described in [Pahl and Beitz, 1996; Roth, 1982].

The topological model is used to create a physical and mathematical model. The physical and mathematical models are the basis for the behavioral description of the system. The numerical model is a preparation of the mathematical model so that it can be algorithmically handled by a simulation. The numerical model is provided with concrete numerical values.

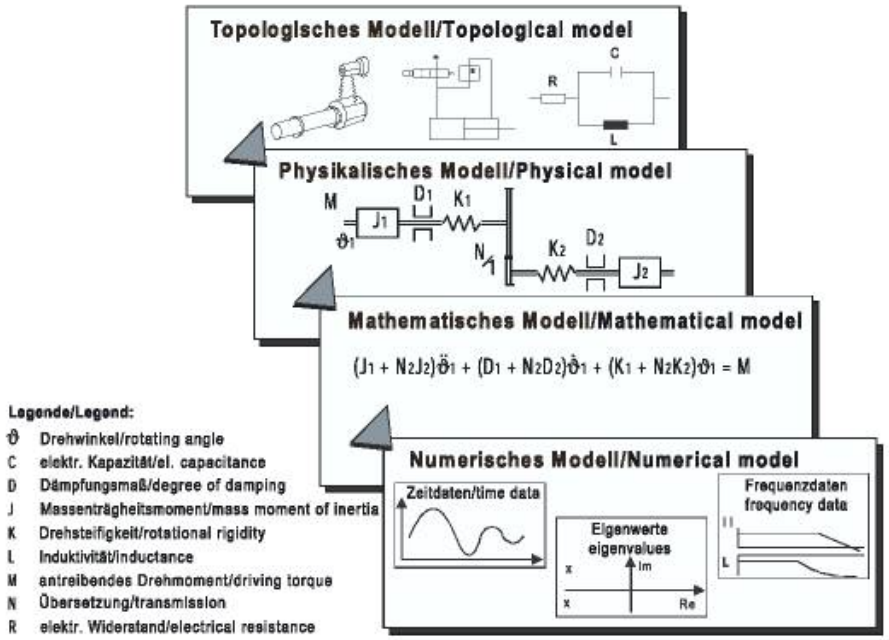

Figure 2.6.: Models are built out of other models. The image is adapted from [VDI 2206, 2004].

2.4.2. Model-Based Analysis

In the traditional development methodologies, the functionality of a car was realized only by mechanical components. This has changed in the past years, when more and more functionality has moved from the mechanical realization to one based on electrical systems, electronics (EE) and software. The consequences are a much greater product complexity due to the interdisciplinary development that could only be managed by a model-based development process, which made early integration and validation of design possible.

Model-based analysis is concerned with the investigation of the systems properties, like for example its behavior. Using a model, it is possible to analyze system states in which the real system cannot or must not be brought [VDI 2206, 2004]. Models can refer here to computer-based models or physical models. Some analysis cannot be made using computer-aided tools due to the high complexity of the model or of the environment.

In an ideal case, model-based analysis should accompany the whole development process. In the design phase, it is a good method for evaluating and for validating different design variants. Meanwhile, it plays the central role in the integration phase, when different subsystems must be brought together to form a working system. With the aid of model-based analysis, the

validation of the integrated system can be made. Simulation is a method used in model-based analysis for quality ensurance and in the property assurance phase.

The functional validation of a mechatronic product cannot be done without the use of model-based analysis and mathematical models due to the high interdisciplinary character of mechatronics. The mathematical models of mechanics, electric/electronics and software can be easily integrated and simulated together. Their integration is tested in a virtual experiment.

In contrast to real experiments, in which physical realizations of the system under investigation are present and the physical loads are replicated (for example, in mechanics, movements, forces and moments; in electronics, resistive, inductive and capacitive loads are replicated), in virtual experiments, the system is represented by mathematical models and its behavior is simulated. It is very common that a combination of virtual and real parts of a system is used in a hardware-in-the-loop (HiL) testing.

In the initial phases of the development, only models of the system and its components exist. In the next steps, the models become more concrete, and after they have been validated, they serve as a specification for the design of the real hardware.

One of the methods of improving the quality and reducing the costs of the product development process is to perform the functional validation of the systems in the early phases of the development (front-loading), in which the iteration cycles are much shorter and the costs are much smaller than in later phases.

Model-based analysis contributes to this goal by enabling an integrated functional validation of models from different disciplines. The steps of model-based analysis are shown in Figure 2.7. During the analysis process, parts of the system model are replaced iteratively with real prototypes or production components. At the end of this cycle, all individual components and subsystems have been tested and validated in an overall system.

2.4.3. Uses and Benefits of Model-Based Analysis and Design

The development of complex mechatronic products is not possible without the support of modeling and simulation techniques. Model-based design offers important advantages in terms of time and costs by employing modeling and computer-aided analysis. Building models takes up more time and costs initially, but the advantages that are brought to the entire development process are significant.

If traditionally in the development of a mechatronic product, the design of the control unit was done in the early design phases, the test and implementation of the embedded control system had to wait until late in the process due to the unavailability of production prototypes needed for testing the behavior of the control software. The integration of hardware and software took place very late in the development cycle. This led to a late discovery of errors

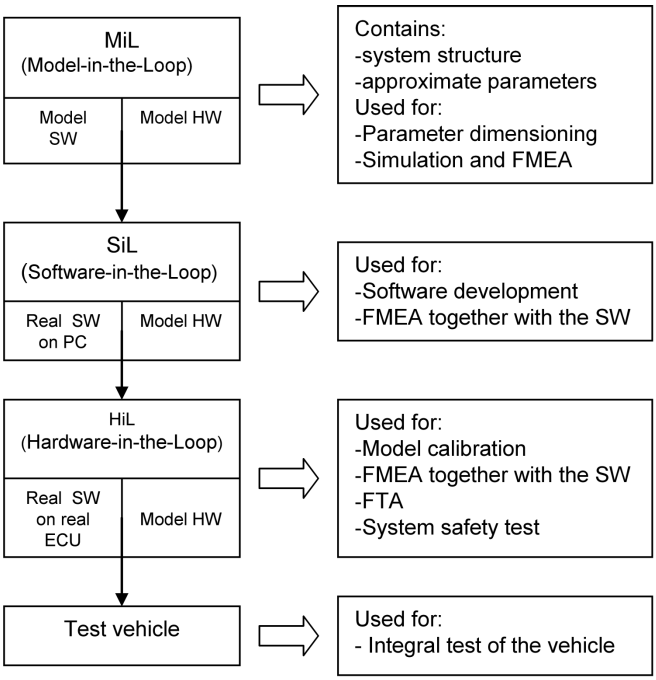

Figure 2.7.: The sequence of operations used in model-based analysis.

that caused production delays due to new iterations in the development process, which in turn implied additional expenses in code updates and verification tests.

With the aid of model-based design and the new development methodologies, engineers are able to overcome these drawbacks. Computer models and new modeling and simulation tools enable the checking and analysis of the behavior of a system long before the completion of the first prototype. Iterations take place in the early stages of the development to assure the product properties, to identify problems early and to reduce risk. Different testing methods have been developed that allow the testing of real and virtual components in a common environment, like software-in-the-loop (SiL) or hardware-in-the-loop (HiL).

Building prototypes of products is an important step in the development process. Prototypes are needed for systems or components in order to reduce risk in design. The end product of prototyping is knowledge about the product and design in different forms. For example, a result of a prototyping might be that a specific material is not suited for the construction of a component. However, building physical prototypes is a very time-consuming and cost-intensive process. For this reason, there are many efforts to reduce the number of physical prototypes. Virtual prototyping – that is, the analysis of computer models of objects that are in development [VDI 2206, 2004] – can support this effectively.

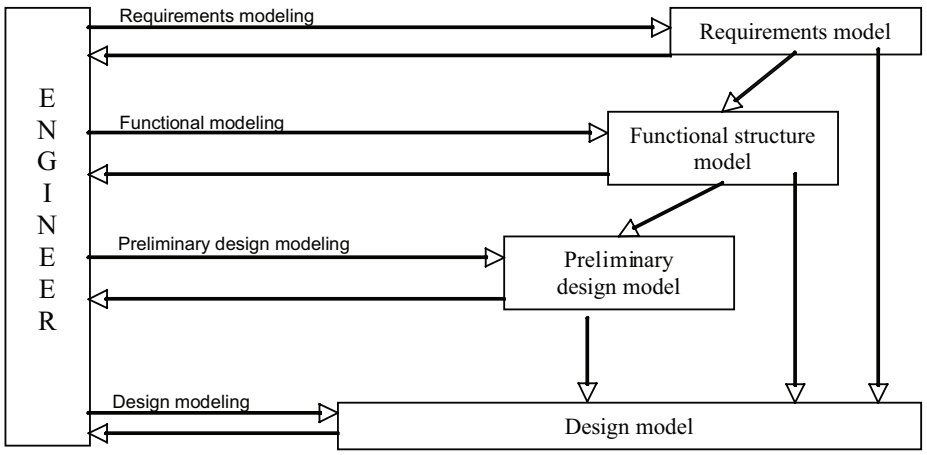

Figure 2.8.: Evolution of models from requirements to design.

The virtual prototyping technique uses models as a central element, on which different analysis techniques are then applied. For example, complex 3D geometrical models, also called digital mock-ups, may be used to test aspects such as kinematics, dynamics, and strength of a system. Other types of models are functional models, which are used to integrate and validate the functionality of the overall system, as well as in safety and reliability analysis. Ideally, models of earlier development phases are used to build up models of later phases. An example of how models are developed and of how they built-up one on another is given in Figure 2.8, based on the approach of Rude [1998].

There are numerous benefits of using models and a system-oriented development methodology. Benefits from the development process are propagated further to supplier and customer advantages.

Some of the benefits are enumerated below:

- *Improved expressiveness and rigor* – By using standardized modeling languages and techniques, product aspects can be represented using appropriated models, such as, 3D models, functional models, behavioral models, etc.

- *Improved understanding of complex products* – Models may represent different levels of abstraction of a complex product. By means of refinement, models can be further concretized. The system-oriented development methodology also contributes to a better understanding of the models by providing an overall view on the system. A system is decomposed in several smaller subsystem, which are more manageable.

- *Improved traceability* – Traceability is very important feature in the development process. The history of design changes can be traced from version to version. If a change has to be made in a functional parameter of a component, this leads to a functional requirement that is affected by this change that can be further traced back to a user requirement. This means a more effective translation and transparency of the user needs into the systems requirements and further to the design.

- *Improved quality* – By the early validation of the design using different methods, a better quality of the design models may be achieved. Model-based analysis ensures a more robust design, in which all aspects of quality (safety, performance, reliability, durability, conformance to requirements) are systematically tested.

- *Shorter development time and smaller costs* – Unnecessary design iterations in late phases are avoided, due to the front-loading of the development process. Costs are also reduced by a systematic approach to verification and validation of models.

- *Enhanced communication with the suppliers* – Due to the rigor and unambiguous nature of models, the exchange of information, with models as specifications of components, is much improved in the communication with the suppliers.

- *Reuse of models* – Libraries of validated models of components enhance the quality of models and reduce the time needed to build a new model. Building a new design becomes a matter of combining and configuring predefined components from a model library.

2.5. A Model of the Design Task

This section presents a model of the design task and is organized as follows: Section 2.5.1 presents the activities in the design task an their interrelationship. Section 2.5.2 describes the parametric design task. Section 2.5.3 illustrates the collaborative design task that is the context, in which this thesis aims to bring a contribution.

2.5.1. The Design Task

Design is an activity that produces a design solution to solve certain tasks or fulfill certain goals. In this sense, design is a goal-oriented activity [Motta, 1998]. After the requirements clarification phase, design engineers have the task of finding a design solution that satisfies the goals under consideration of different types of constraints coming from different material, technological, economic, legal, etc. considerations [Pahl and Beitz, 1996]. After a design solution has been found, a design specification is produced. The physical realization of the artifact is the responsibility of the manufacturing engineer.

A simplified model of the design process can be seen in Figure 2.9. Design is seen as a problem solving task as in other approaches [Pahl and Beitz, 1996; Stevens et al., 1998; VDI 2206, 2004]. The design task should not be understood as a sequential process, but as a rather iterative one, in which analysis and synthesis steps are performed several times. The requirements and constraints are not a constant set defined at the beginning of the process, but one that is also evolving and changing during the design process.

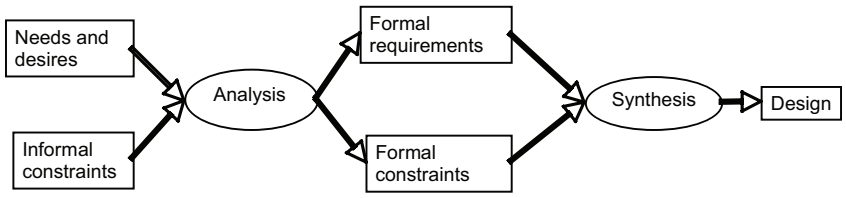

Figure 2.9.: A simplified model of the design process. Adapted from [Breuker and Van de Velde, 1994].

The result of the design process is a structure that fulfills the stated goal. The general design process starts with the definition of user needs and desires, also called **user requirements** [Stevens et al., 1998] that are specified in the non-technical vocabulary of the users. They describe what the system should do and not how the solution should be implemented.

For example, a user requirement for building an passenger car might be that "*The average fuel consumption in city traffic should be less than 6 liters of fuel in 100 km.*"

Besides the user requirements that usually cover operational and functional aspects, other requirements are derived from different sources, such as, constructions norms, legislations, technological aspects, marketing, service and maintenance, etc.

Constraints on the design may also be informally specified. The difference between requirements and constraints is that requirements are usually stated as "wanted" properties that a design should have in order to be an acceptable solution [Wielinga et al., 1995], such as "*The car should have an automatic transmission*", while constraints are "negative" properties that add no extra capability to the solution, but indicate limitations on a possible design solutions. Another distinction is that very often user requirements are related to the desired functionality of the device, while constraints are usually non-functional.

The result of the analysis step of user requirements is a formal representation, also known as **system requirements**. The formal representation is used in the synthesis process that has as a result a structure, which must fulfill the initial goal of design.

2.5.2. Parametric Design Task in Engineering Domain

A type of design that is especially of interest for the engineering domain is the **parametric design task**, which is formally defined by [Wielinga et al., 1995] and applied in different frameworks [Motta et al., 1999; Motta, 1998]. The parametric design task has as a goal finding of a structure composed of parameterized building blocks that are constructible and that satisfy all the requirements. For this purpose, the realization space ("Variantenraum") and the satisfaction space ("Erfüllungsraum") have been defined. A design solution must be found in the intersection of the realization space and the satisfaction space.

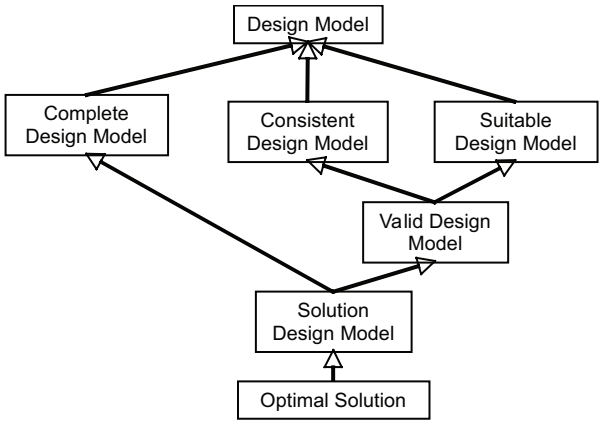

Figure 2.10.: The taxonomy of the design models according to [Motta, 1998].

Very often, the architecture of the system to be designed is roughly known at the beginning of the process and the designers work with functional solution templates [Motta, 1998] to guide the design process, which brings a great reduction of the complexity of design. In the parametric design task, the solution space is further reduced, and parameterized solution templates are used. The design process implies assigning values to the design parameters such that no constraints are violated and all requirements are fulfilled.

A model for the parametric design task and a terminology for it has been proposed by [Motta, 1998; Wielinga et al., 1995; Motta et al., 1999]. The taxonomy of design models types defined by [Motta, 1998] is shown in Figure 2.10.

A design model is **consistent** if none of its constraints is violated. A design model is **suitable** if it fulfills all requirements. A design model is **valid** if it is both consistent and suitable. A design is **complete** if each design parameter has assigned a value in the allowed value domain set. A solution to a design model must be valid and complete. Also optimality can be defined if a cost and preference function over the solution space is defined.

In modern engineering process, reusing previous designs and experience has become an essential way of improving the design process. Besides its constructive nature, design has become also a configuration task. Instead of "reinventing the wheel", engineers have defined general functional solutions that can be used to fulfill different functions. These general pieces of functional solutions, model fragments, or building blocks are assembled together to form a system that must fulfill a complex function.

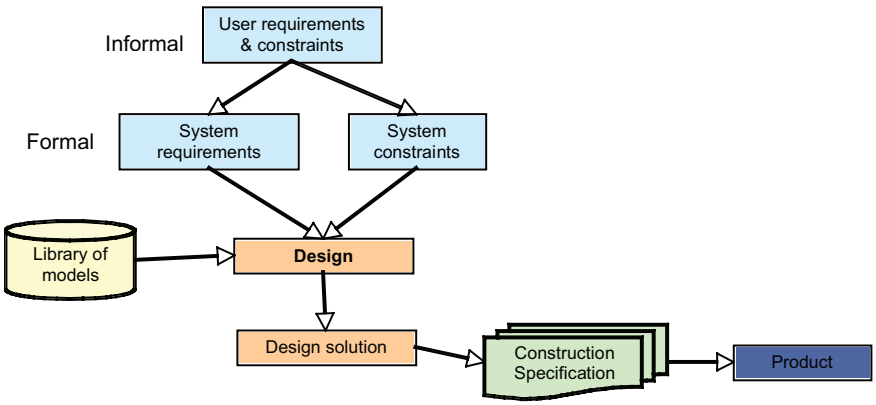

Figure 2.11.: Reusing template components out of a model library in the design process.

The model fragments are stored in a library of models (See Figure 2.11). Because the size of the model library tends to grow rapidly in time, good knowledge and model management is needed for effective use of the libraries. Engineers should be able to retrieve the suitable model fragments based on different criteria. Libraries of models are a means for reducing the complexity of the design process. They also contribute to the increase of model quality, because the model fragments in the library are already validated and "approved" components.

Finding a design solution implies assembling different model fragments in order to achieve a design that fulfills all requirements and that does not violate any constraint. Usually several design solutions variants are found during the design process. There are different methods [Pahl and Beitz, 1996] that can be used to find the optimal solution.

After choosing the optimal solution, a construction specification is generated, which is used to build a prototype that can be tested against the user requirements. If the prototype fails, new design iterations are necessary in order to correct the failure. Such design iterations are extremely expensive, especially if failures are identified only late in the development cycle. For this reason, having good quality and consistent models is crucial for an improved and cost-efficient design process.

2.5.3. The Collaborative Design Process

The design process presented in the previous section illustrates the steps performed in only one development branch. The modern development process is highly collaborative and concurrent in nature due to the market requirement to reduce total development time.

Different types of workflows are used to support a shorter development time, such as simultaneous and concurrent engineering. Simultaneous engineering implies all phases of product development proceeding in parallel. A development phase is started before the previous one has ended. Another type of workflow is that of concurrent engineering, in which different teams are working to solve the same task together in a parallel fashion. The task is decomposed into subtasks, and each team works on solving one of the subtasks. In the end, they need to integrate the partial solutions resulting from the different development branches. They need to make sure that the partial solutions are consistent with each other and that they can form one consistent design specification.

The concurrent design cycle is shown in Figure 2.12. The first step is similar to the one in the simple design cycle, acquiring user requirements and constraints. There are different criteria that guide how development branches will be decomposed. One branching possibility is in functional and geometrical design; another one is based on the structure of the product. In case of the automotive industry, an example of structural branching is the design of the motor, of the transmission, of the chassis, etc. It is also common that a combination of the branching criteria is used (such as geometrical design of transmission and functional design of transmission).

After completing the requirements decomposition, the design process is started in a parallel fashion in different development branches. Each of them employs their own methods, languages, tools and model libraries to obtain a design solution that fulfils the branch requirements.

The next step is the most challenging and critical one: Integrating the different design solution to achieve one consistent design that fulfils all requirements and that can be used as a specification to build the product. There are several challenges in synchronizing or integrating the models built in the different development branches: Models are built in different modeling environments using specific languages; Engineers understand only their own models and do not know how changes in their models may affect models in other development branches; Many implicit modeling assumptions are made; There are no explicit correspondences between the models that would support an automated model synchronization.

The remainder of the thesis will address two of the most important challenges identified in the design process: **Consistency of design models** and **consistency among design models** from different development branches.

Consistency of a design model can be checked by having a formal representation of the design model, a domain theory, and a formal representation of constraints. The formal representation

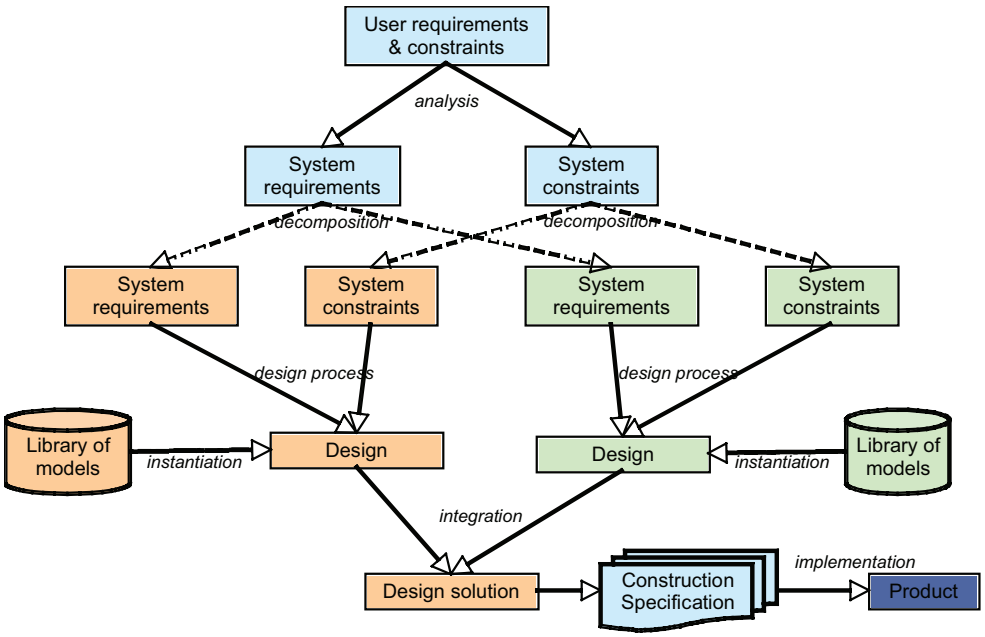

Figure 2.12.: The concurrent design process.

of design models for the engineering domain is discussed in Chapter 4.

Consistency among design models may be achieved by representing explicitly the correspondences between the models that can be checked by formal methods. This is supported by the mapping framework, which is presented in Chapter 5. Two applications that have been used to validate the concepts in this thesis are presented in Chapter 6.

3. Consistency in the Engineering Development Process

3.1. Definition of Consistency

The word "**consistency**" comes from the Latin noun "*consistentem*" and the verb "*consisto*" with the initial sense of "to stand still, stand, halt, stop, take a stand, post oneself". A new meaning for the word "**consistency**" – "agreeing with" was attested in 1646 and has been preserved until present. The definition of consistency in the Merriam-Webster dictionary [Merriam-Webster, 2003] also confirms this meaning: "*[Consistency is] agreement or harmony of parts or features to one another as a whole; Ability to be asserted together without contradiction*".

Although the word *consistency* is used in many domains – cognitive sciences, psychology, design, software engineering, etc. – it usually does not have a well defined meaning, but a rather implicit one. It is typically used to denote, as in the Merriam-Webster definition, a harmony between different parts of something or between the behaviors of a thing in different contexts. An example is in the graphical user interface design, where people expect the same terms, metaphors and controls to mean exactly the same thing across the whole user interface. This is one of the conditions for a user interface to be "consistent". However, there is no well defined and broadly accepted definition of what consistency for a user interface means.

Nevertheless, there are domains in which the term consistency is well defined. For example, in formal logic, consistency is an attribute of a logical system that is so constituted that none of the propositions deducible from the axioms contradict one another.

In this section, I will give a formal definition of consistency that can be used to automatically check whether two or more design models of different aspects of the same product are consistent with one another. The final goal of the design process is to create a product that is a physical realization of both design models. The consistency of the design models is a prerequisite for the existence of a physical realization of the product that is described by the two design models.

In order to define the consistency between engineering design models, I will use and extend the definition of consistency between different specifications as stated by the Reference Model for Open Distributed Processing (RM-ODP) [RM-ODP/3, 1996]. Design models can be considered as realization in the model building phase, and as specifications for the realization of

the physical prototypes. The RM-ODP framework has been chosen, since it already defines the consistency between specifications of different viewpoints for the same realization, which is a similar problem to the one that this work is addressing – consistency between different design models of the same product.

This chapter is structured as following: Section 3.2 gives a brief overview of RM-ODP and its definitions of viewpoints and consistency and show their formalization using first order logic. Section 3.3 discussed the application of the RM-ODP concepts to the engineering development process. Section 3.4 will illustrate a particularity of the design process for the engineering domain and will discuss the two roles that the design models play. Section 3.5 identifies the challenges encountered when dealing with multiple viewpoint of the same product. And finally, Section 3.6 will give a formal definition of consistency between design models using concepts from distributed first order logic.

3.2. Reference Model for Open Distributed Processing

After solving the problem of heterogeneous systems interconnection with the Open System Interconnection (OSI) model, the international standardization bodies faced another challenge concerning distributed systems. Rather than just interconnecting, the systems needed to communicate with one another in a standardized way. In order to be successful, the distributed system community needed solutions for the integration and interoperability of distributed applications in an open-service market, which required that application components communicate through standardized interfaces [Farooqui et al., 1995]. The ISO issued in the mid-nineties the Reference Model for Open Distributed Processing (RM-ODP) as a basis for the provision of these needs. The Open Distributed Processing (ODP) framework was intended to provide a standardized framework for an open distributed environment that spans heterogeneous systems.

The reference model for ODP prescribes a model for the development of complex distributed systems without committing to a certain domain. RM-ODP can be applied in various domains, like telecommunications, intelligent networks, software engineering, etc.

RM-ODP can be seen as a general specification theory for distributed systems and applications. The development process for distributed systems is only implicit in RM-ODP and has some similarities with the development process described in systems engineering. [Bab and Mahr, 2005] describe the implicit RM-ODP development in the following way: *Requirements* describe the capability that the system should have, for example the operation and use of the system in an environment. The system to be implemented is considered to be a *realization* of several interrelated specifications. The realization is the result of several *refinement* and *translation steps* applied to the specifications. Specifications are classified in different *view-*

points. RM-ODP also defines the relationships that may exist between the specifications (such as *compliance* and *consistency*), or between a specification and a realization (*conformance*).

I will apply the viewpoint concept and the relationships defined by RM-ODP to the engineering development process in order to define the consistency between design models. The next sections give a short overview of the viewpoints and consistency in RM-OPD. Section 3.2.3 tries to give a more formal definition of these relationships using First Order Logic.

3.2.1. Viewpoints

In order to manage the complexity of an open distributed system, RM-ODP supports the separation of concerns by employing the concept of viewpoints. Viewpoints are different perspectives on the system seen from different roles. RM-ODP defines five viewpoints together with a set of concepts, structures and rules for each of them: the *enterprise view* – concerned with the user needs of the system, the *information viewpoint* – describing the semantics of information content, information flows and information processing, the *computational viewpoint* – deals with the functional decomposition of the system by hiding from the designer the underlying distributed platform, the *engineering viewpoint* – delivers the infrastructure required to support distributed nature of the system, and the *technology viewpoint* – which identifies the choices of technology for implementation purposes.

The viewpoints concept allows the separation of a complex specification in manageable pieces, each focused on a certain aspect of the system. In this way, the complexity of specifying a distributed system is reduced, enabling different developers to concentrate only on one perspective of the system, rather then on one single integrated system.
Viewpoints can be understood as different projections of the system correspondent to the different interest and concerns of the viewer. The viewpoint concept allows the designers to focus on a particular problem, using a familiar vocabulary or language and established design methodologies for that particular viewpoint.

Viewpoints may also represent a certain system on different levels of abstraction. For example, the engineering viewpoint is the most abstract representation of the system, while the technology viewpoint is the most concrete one. Even if each viewpoint has its own perspective on the system, its own requirements and vocabulary, they still have some common ground, which is the actual system to be designed. To successfully achieve a realization, its specifications have to be consistent with one another.

The separation of concerns is also an important characteristic of the engineering development process. However, the separation of concerns in engineering development process is made according to different criteria than in RM-ODP (see Section 3.5).

3.2.2. Consistency

Consistency is a relation between specifications, as opposed to conformance which is a relationship between a specification and a realization. RM-ODP makes several statements about the consistency in different parts of the specification [Bowman et al., 1996].

One necessary condition for two specifications to be consistent is that ***they do not impose contradictory requirements***. This seems a natural condition, since it is hard to imagine that a realization (a system) may exist that is required in a specification to assign 4 as the value of a parameter, while the same parameter value is prescribed to be 5 in the other specification. However, the contradictions between the requirements are not always easy to detect, especially if the specifications, and hence their requirements, use different languages and there is implicit background knowledge that should also be considered.

The definition of consistency between two specifications stated in RM-ODP is the following: "*Two specifications are consistent if and only if it is possible for at least one example of a product (implementation) to exist that conforms to both specifications*". This definition is a much stronger one than simply checking for contradicting requirements. It is clear that if the imposed requirements are contradictory, than no common realization can be achieved, but this definition also states that the realization should also be implementable.

In order to ensure consistency between specifications in different viewpoints, RM-ODP suggests using statements of correspondences between terms and language constructs relating one viewpoint specification to another [RM-ODP/3, 1996]. The correspondences are statements asserting that some terms or structures in a specification correspond to other terms or structures in another specification. They may relate specifications in the same language or in different languages. RM-ODP stresses the importance of the correspondence rules, also called consistency rules, by stating that these relationships between the viewpoints ensure that they are specifying a single system, rather than being completely independent documents [RM-ODP/1, 1998].

The consistency rules are attached to the link between the key terms in the two specifications and can be used to check certain kinds of inconsistencies. One type of consistency checking is to use the consistency rules to transform a specification S_1 from language L_1 to language L_2 of another specification S_2, and then to compare the two specifications using the same language. However, very often the correspondences between the key terms in specifications are provided implicitly by using the same names or notations [RM-ODP/1, 1998].

The consistency rules, as defined by RM-ODP, enable a simple syntactic consistency check, which is often supported by the development tools. However, the consistency rules do not caption the background knowledge or assumptions that are made in a certain viewpoint or specification, which makes the semantic consistency much harder to check.

The semantic or conceptual consistency of the viewpoints deals with the meaning of the statements in the viewpoint specifications that should at least not contain contradicting statements.

In order to check the semantic consistency, we need an explicit formal representation of the meaning in the specifications and then use logical reasoning to check for inconsistencies.

3.2.3. Formal Definitions of the Specification Relationships

For the scope of conformance assessment (which determines whether a product fulfills its requirements), RM-ODP defines two types of relationships for specifications. The first type refers to the relationship between a specification and a real implementation – conformance – and the second type describes the relationships between specifications – compliance, consistency and validity. In this section I will give their definitions as stated in RM-ODP and then give more precise, formal definitions in first-order logic that insure a precise and unambiguous semantics for each term in the domain.

The main concepts used in the RM-ODP development process and their interrelationships are as following: Products must fulfill certain requirements which are stated in one or several specifications. A product or a realization is conformant to the specification, if it fulfills the requirements stated in the specification. A graphical representation of the relations between specifications that I will further detail in the section is shown in Figure 3.1.

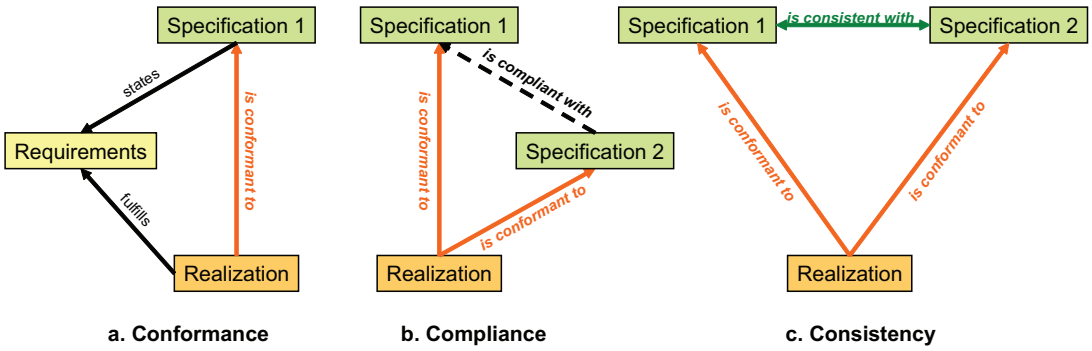

Figure 3.1.: Definition of conformance, compliance and consistency in RM-ODP.

The following notations are used for the rest of the section: Unary predicate names that are used to denote concepts in the domain are written with capital letters (e.g., *Requirement(x)*), binary predicates that are used to denote binary relationships are written with small letters (e.g., *states(x,y)*), variables in predicates are written in small letters (e.g. *x*, *s*, *req*) and if no quantifier is specified, the universal quantifier is assumed.

I will use three first-order unary predicates and a function with the following meanings, to represent the main concepts in the domain needed to define consistency between specifications.

Specification(x)	–	x is a specification
Realization(x)	–	x is a realization
Requirement(x)	–	x is a requirement
Requirements(x)	–	a function returning the set of requirements for specification x

Consider Specifications to be the set of all specifications for a product. A specification prescribes a set of requirements. I will use the notation *Requirements(x)* to denote the set of requirements of specification x, where x is from the set Specifications. For example, a requirement for a car could be that its weight should be less than 1300 kg. At this point, the formal representation of the content of the requirement (e.g., "*weight < 1300*") is not investigated. If a specification S_1 contains only this requirement, than the *Requirements(S_1)* would be a set containing only this requirement.

The *states* predicate is a binary predicate that relates a specification to a requirement in its Requirements set. The *states* predicate is true for each requirement in the requirements set prescribed by a specification.

$$\textit{Specification(s)} \wedge \forall \textit{req}(\textit{req} \in \textit{Requirements(s)} \Rightarrow \textit{states (s,req)}) \tag{3.1}$$

Each specification states at least a requirement.

$$\textit{Specification(s)} \Rightarrow \exists \textit{req}(\textit{Requirement(req)} \wedge \textit{states(s,req)}) \tag{3.2}$$

The *states* predicate has the following property: if something states A and it also states B, then it also states A and B.

$$\textit{states(x,a)} \wedge \textit{states(x,b)} \Rightarrow \textit{states(x,a}\wedge\textit{b)} \tag{3.3}$$

For example, if a specification states that a car should have the weight less than 1300 kg and it also states that the car should have four doors, it means that the specification actually states the conjunction of the two requirements: A car should have a weight less than 1300 kg and should have four doors.

A realization is obtained at the end of the development process by iterative refinement steps on the specifications. A realization has to fulfill the requirements prescribed by the specifications. The binary predicate *fulfills(r,req)* is true if the realization r fulfills the requirement *req*. The formal checking of the fulfillment relationship is not discussed here. The *fulfills* predicate also has a similar property to the *states* predicate: if something fulfills req_1 and it also fulfills req_2, it implies that it fulfills their conjunction. (Of course, in the case that the requirements are not contradictory, which is assumed here).

$$\textit{fulfills(r,req}_1\textit{)} \wedge \textit{fulfills(r,req}_2\textit{)} \Rightarrow \textit{fulfills(r,req}_1 \wedge \textit{req}_2\textit{)} \tag{3.4}$$

Definition 1 *A realization is **conformant** to a specification if and only if it fulfills the requirements defined by the specification.*

This definition can be formally written by using a binary predicate, *conformant(x,y)* which is true if realization x is conformant to specification y.

$$\begin{aligned} conformant(r,s) \Leftrightarrow\ & Realization(r) \wedge Specification(s) \wedge \\ & (\forall req(states(s,req) \Rightarrow fulfills(r,req))) \end{aligned} \tag{3.5}$$

In the previous example, if the specification of the car contained only one requirement, that the weight of the car should be less than 1300 kg and the physical product car has indeed the weight less than 1300 kg, this means that the car is a conformant realization of the specification.

Definition 2 *A specification is **valid** if there is at least one realization that is conformant to it.*

The formal definition of validity is given by the *valid* predicate:

$$valid(s) \Leftrightarrow Specification(s) \wedge (\exists r(Realization(r) \wedge conformant(r,s))) \tag{3.6}$$

A specification may not be valid in the case that the prescribed requirements are in conflict to each other, or in the case that the requirements are over-specified. For example, a specification may contain two requirements, the first one stating that the transmission ratio of a gear should be greater than some value, and the second one imposing a geometrical constraint, which is in conflict with the first requirement. In this case the specification is invalid, because a conformant physical realization of the specification could never be realized.

Definition 3 *Specification 2 is **compliant** with Specification 1 if Specification 2 fulfills the requirements stated by Specification 1.*

The formal definition of compliance can be given by using a binary predicate $compliant(s_1,s_2)$ which is true when s_1 is compliant with s_2:

$$\begin{aligned} & Specification(s_1) \wedge Specification(s_2) \wedge \\ & (\forall req(Requirement(req) \wedge (states(s_1,req) \Rightarrow fulfills(s_2,req))) \Rightarrow compliant(s_1,s_2) \end{aligned} \tag{3.7}$$

This also implies that a realization that is conformant to Specification 2 is also conformant to Specification 1 [RM-ODP/2, 1996].

$$compliant(s_1,s_2) \Rightarrow (\forall r(Realization(r) \wedge conformant(r,s_1) \Rightarrow conformant(r,s_2))) \quad (3.8)$$

Compliance can be seen as a subset relationship between specifications. For example, if Specification 1 states that a car should be able to carry up to 4 persons, and Specification 2 states that a car should be able to carry up to 2 persons, we can say that Specification 1 is compliant with Specification 2, because every car that carries 4 persons can also carry 2 persons.

Definition 4 *Two specifications are **consistent** with one another if and only if there is a realization that is conformant to both specifications.*

This can be formalized in the following way:

$$\begin{aligned} consistent(x,y) \Leftrightarrow\ & Specification(x) \wedge Specification(y) \wedge \\ & (\exists z(Realization(z) \wedge conformant(z,x) \wedge conformant(z,y))) \end{aligned} \quad (3.9)$$

Consistency between specifications is crucial when several specifications prescribing different aspects of the same product exist, which is often the case for complex product development. In order to achieve at the end of the development process a physical product (i.e., a realization), which conforms to both specifications, the specifications have to be consistent with one another.

As a consequence of this definition we can also infer that two specifications are **consistent** if and only if they do not impose contradictory requirements. In order to prove this, suppose that Specification 1 prescribes a Requirement 1 and Specification 2 prescribes Requirement 2, which contradicts Requirement 1. For example, Requirement 1 states that "*The car should have no gas emissions*" and Requirement 2 states that "*The (same) car should run only with gas fuel*". If the two specification are consistent, it means that a realization exists that conforms to both specifications; therefore, it will have to fulfill both Requirement 1 and Requirement 2. However, this is not the case, because the two requirements are contradictory and cannot be fulfilled at the same time in a system. This means that there is no realization that can conform to both of them, and consequently the two specifications are not consistent with each other. The relationships between a realization that conforms to two specifications is illustrated graphically in Figure 3.2, where the definition of conformance is expanded according to Definition 1.

In order to define the formal relationship between the concepts depicted in Figure 3.2, I will consider two specifications for the same product, s_1 and s_2 with their corresponding requirement sets, $Requirements(s_1)$ and $Requirements(s_2)$ and a Realization r that is conformant to both specifications s_1 and s_2.

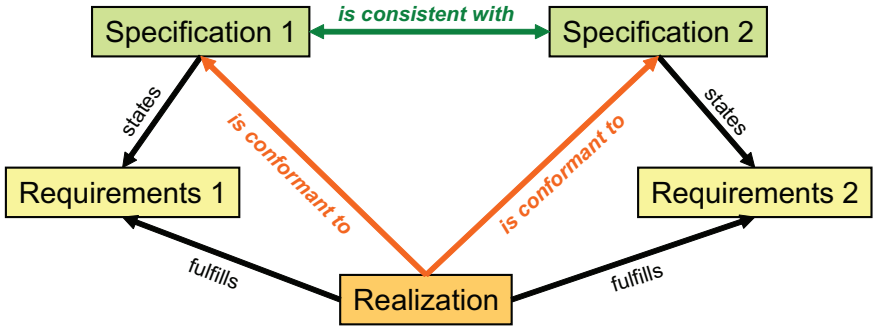

Figure 3.2.: Consistency between two specifications. A realization has to fulfill the requirements of both Specification 1 and Specification 2.

By expanding the definition of conformance and consistency according to Definition 1 and 4, the consistency between two specifications s_1 and s_2 can defined as follows:

$$\begin{aligned} consistent(s_1,s_2) \Leftrightarrow\ & Specification(s_1) \wedge Specification(s_2) \wedge (\exists r(Realization(r) \wedge \\ & (\forall req_1(states(s_1,req_1) \Rightarrow fulfills(r,req_1))) \wedge \\ & (\forall req_2(states(s_2,req_2) \Rightarrow fulfills(r,req_2))))) \end{aligned} \tag{3.10}$$

According to the definition from above, two specifications are consistent, if there exists a realization that fulfills the requirements stated in both specifications. From this definition it also becomes clear that if the specifications state contradictory things, then the specifications are not consistent and it is not possible to build a product that fulfills all the requirements from the two specifications.

There is also a close correspondence of the specification relationships in traditional logic, as noted in [Bab and Mahr, 2005], which investigates the use of reference points for the conformance assessment. If realizations are considered to be models or structures and specifications to be a set of sentences, then the conformance relationship would correspond to the satisfaction of the sentences in the model. Consistency is in this case the existence of a model that satisfies two specifications (that is, it satisfies both sets of sentences corresponding to the specifications).

3.3. Applying RM-ODP Concepts to the Engineering Development Process

RM-ODP has been specifically defined to be domain-independent, which allows a straightforward mapping from the concepts in RM-ODP to concepts in the engineering product develop-

ment process. The mapping between the terminologies of the two domains for a viewpoint is shown graphically in Figure 3.3. Nevertheless, there are certain characteristics of the development process in the engineering domain that do not have a direct correspondence in RM-ODP, but they are not discussed here.

RM-ODP defines several viewpoints under which a system to be designed can be specified. This allows decomposing a complex specification into several manageable pieces, each of them describing the system from a different perspective. The corresponding concept in product development is the splitting of development work into different development branches, in which each branch is concerned with a certain aspect of the product. Both in RM-ODP and in product development, each viewpoint or development branch has a system specification for that particular viewpoint that prescribes what the system should perform as a set of requirements. The outcome of the design process should be a realization, in the case of RM-ODP, and a product in the case of product development

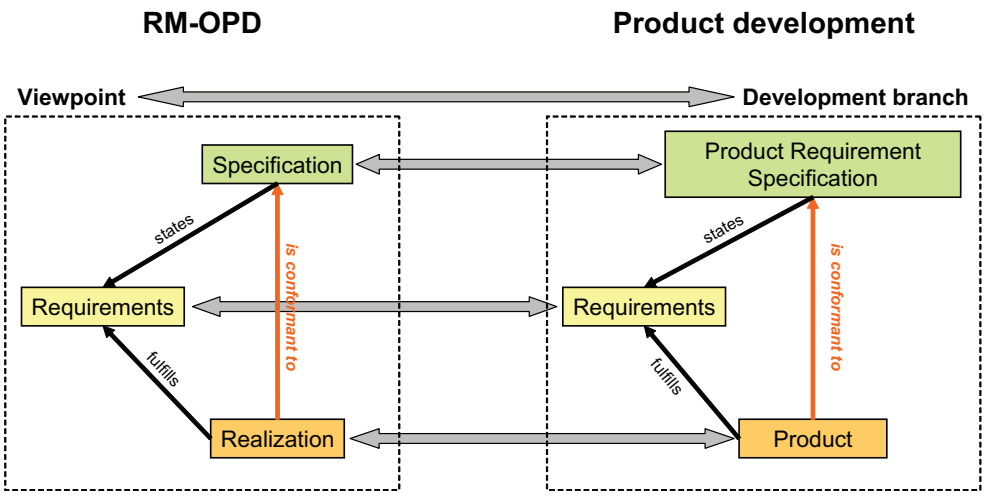

Figure 3.3.: The correspondence between terms in RM-ODP and terms in product development for a viewpoint.

The goal of the product development process is to build a product that fulfills the requirements. This is a highly iterative process. At different development stages the assessment of properties of the built models is made (see Chapter 2, Section 2.1). This is equivalent to the conformance testing used in RM-ODP. The conformance testing in product development is made first on a computer model of the product, then parts of the model are replaced in a stepwise manner with physical components until conformance is tested on a fully physical prototype (see Chapter 2, Section 2.6).

3.4. Two Roles of a Design Model

In RM-ODP, the difference between a specification and the realization or the real implementation is clear. The realization is the end product of the development process, in which specifications are refined in iterative steps and results in a concrete implementation. In the engineering development process, this limit between being a specification and an implementation is blurred. In the following section, I will describe the two roles that the design models play: as a virtual realization and as a specification. The two roles of the design models are shown graphically in Figure 3.4.

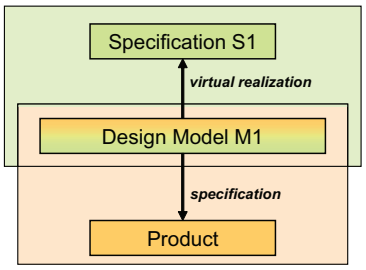

Figure 3.4.: The two roles of the design model: as a virtual realization for its specification and as a specification for the product itself.

3.4.1. Design Model as Virtual Realization

One of the goals of the engineering development process is to minimize the costs of development. Instead of building several physical prototypes, which is a very cost-intensive process, the modern engineering process will build in the early development stages only virtual prototypes. They are computer models, which act as a substitution for the real products. After being verified and validated, the computer models are used in model analysis, in which the desired product properties and behaviors are investigated.

The verification and validation activities, which are part of the assurance of properties phase as described in Section 2.2.1, ensure that the properties of the design model are evaluated on the basis of the requirements list. However, certain type of requirements can be checked only in the model analysis phase. For example, in order to check the safety requirements, the behavior of the system has to be assessed for various scenarios in which failures occur in different components of the system.

Model analysis serves different goals: The behavior and other characteristics of the design models can be investigated before first building a real prototype, and various designs can be compared with each other in order to determine the best design solution. For this use case, the

design models are treated as realizations, on which the conformance testing – in the RM-ODP terminology, is done.

I will use the expression **virtual realization** to refer to the realization role that the design model plays: it is a *realization* from the point of view of the specification, and it is *virtual*, since it is only a computer model, not a real implementation. However, the fact that it is treated as a real implementation qualifies it for the name *virtual realization*.

There are different methods used in the verification and validation (V&V) of the virtual realization with regard to the specification. One of the main methods used for property assurance and model analysis is simulation. The goal of the V&V is to determine whether the model is plausible and correct, whether it satisfies the prescribed requirements and if it models adequately the real system (if one exists). There are cases in which a real system already exists and a design model is created in order to analyze the possible behaviors of the system (model analysis) or in the cases when a new variant of the product has to be built by adapting an existing system. If a real system already exists, parameters in the design model are set by measuring technical values on the real system in the model building phase. If no real product exists, then the parameters in the models are set according to the requirements and a plausibility check determines whether the model can correspond to a real system. This requires a lot of empirical knowledge of realistic technical variables and physical behavior [VDI 2206, 2004].

However, it must be noted, that the requirements in the specification may be changed, if the testing process reveals that the requirements are over-constrained, contradicting, or impossible to realize physically or technologically.

3.4.2. Design Model as Specification

After being verified and validated, the design models are used in model analysis. The model analysis may point out unknown facts or things that must also be considered in the design of the system, which where not evident at the beginning of the development process. This may lead to the modification of the initial set of requirements. This is a highly iterative process, and there may be several steps in which new requirements are identified and the design model needs to be changed, followed again by model analysis, and so on, until an adequate design model has been achieved.

At the point in which the design model has reached the desired degree of maturity, it can be used as a specification for the real implementation of the product. The implementation is also done in a stepwise manner, in which parts of the model are replaced with real devices [Notteboom, 2003]. In each step, the property assessment (that is, the conformance checking) is done. Some of the methods for the conformance testing are model-in-the-loop (MIL), software-in-the loop (SIL) and hardware-in-the-loop (HIL). The testing methods are discussed in more detail in [Notteboom, 2003; Prenninger and Pretschner, 2005; Sargent, 1998].

3.4.3. Reconciling the Design Model Roles

With the introduction of the two roles, model as *virtual realization* and model as *specification*, the question of conformance must be reconsidered. Conformance was defined in RM-ODP as a relationship between a specification and a realization, which holds if the realization fulfills all requirements stated by the specification. This model applied very well for the situations in which there was a clear separation between the specification and the realization. However, in the engineering development process, the role of the design model is two-fold: It can be seen as a virtual realization or as a specification depending on the phase of the development process.

In its role as virtual realization, the design model is tested against the requirements stated in the specification. In Figure 3.5, if the virtual realization fulfills the *specification requirements*, then it is said to be conformant to the specification. The *specification requirements* in this case, are considered to be the requirements that are relevant to the purpose of the design model and which can be tested within the design model with the available methods. Models are always built for a certain purpose, and cannot be used to answer questions about the system that lie outside of their scope. This assumption about the *specification requirements* is used throughout this section.

After the model building phase, the model analysis phase follows. Model analysis investigates the behavior and properties of the system, and may reveal new requirements as a result of the analysis and testing process. If the new requirements are in conflict with the *specification requirements*, then the latter will be adjusted. There are two other cases: The new requirements are just refinements of the *specification requirements*, or they may be just new facts that don't directly relate to the *specification requirements*.

After the verification and validation phase, the design model will serve as a specification for building the realization, which in this case would be the real physical system. The design model in its role as a specification will be compliant to the specification from which it was built, since it fulfills all the requirements in the specification. In a second step, the realization (the real implementation) will be tested for conformance against the *design model requirements*. Conformance of the realization can also be checked against the initial specification or other specifications.

The two roles of the design models can also be expressed using first order logic axioms.
I will use a new unary predicate, *DesignModel* that is true if the argument denotes a design model object. The *DesignModel* is both a Specification and a Realization and has their characteristics. Therefore,

$$DesignModel(dm) \Rightarrow Specification(dm) \tag{3.11a}$$

$$DesignModel(dm) \Rightarrow Realization(dm) \tag{3.11b}$$

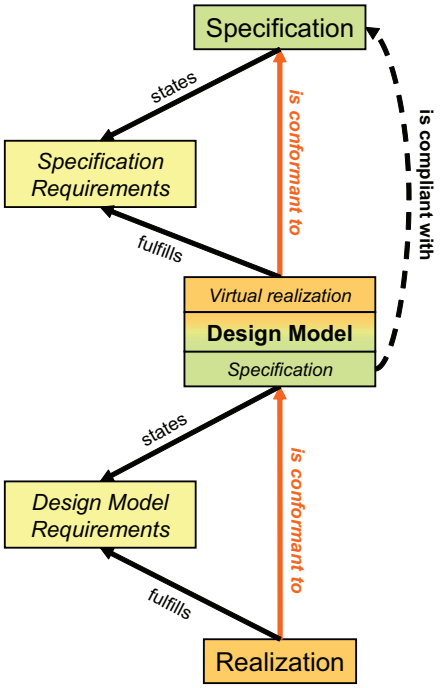

Figure 3.5.: The design model with its two roles: as a virtual realization and as a specification.

The converse relationship is not true (a realization or a specification is not necessary a design model).

In the role as virtual realization, the design model is conformant to the specification. By applying Definition 1, the following axiom is obtained:

$$\begin{aligned} conformant(dm,s) \Leftrightarrow\ & DesignModel(dm) \wedge Specification(s) \wedge \\ & (\forall req(states(s,req) \Rightarrow fulfills(dm,req))) \end{aligned} \tag{3.12}$$

In the role as specification, the design model is compliant with the specification, which means that every realization that is conformant to the design model is also conformant to the specification:

$$\begin{aligned} Specification(s) \wedge DesignModel(dm) \wedge compliant(dm,s) \Rightarrow \\ (\forall r(Realization(r) \wedge conformant(r,dm) \Rightarrow conformant(r,s))) \end{aligned} \tag{3.13}$$

In the development process, the realization must conform to the design model. This can be formalized using the conformance definition in the following way:

$$\begin{aligned} conformant(r,dm) \Leftrightarrow\ & Realization(r) \wedge DesignModel(dm) \wedge \\ & (\forall req(states(dm,req) \Rightarrow fulfills(r,req))) \end{aligned} \tag{3.14}$$

Following from the fact that the design model is compliant to the specification and that the realization is conformant to the design model, it can be inferred that the realization will also be conformant to the specification, which is the goal of the development process.

$$\begin{aligned} & Specification(s) \wedge DesignModel(dm) \wedge Realization(r) \wedge \\ & \quad compliant(dm,s) \wedge conformant(r,dm) \Rightarrow conformant(r,s) \end{aligned} \tag{3.15}$$

In the following section, I will analyze the consistency concept for the situation in which several specifications for the same product exist that describe different aspects of the system in different viewpoints.

3.5. Dealing with Multiple Viewpoints

RM-ODP defines five viewpoints under which a system can be described, starting from a very general one (enterprise viewpoint) and concluding with the most concrete one (technological viewpoint). In the engineering development process, there are also several viewpoints and specifications for a system. In order to avoid confusion between the terminology in RM-ODP and the engineering development process, I will use the expression “engineering viewpoint”, or simply “viewpoint”, to refer to a development branch for which a specification exists and that will be used to build a design model for that viewpoint as part of the engineering development process. When I will refer to a RM-ODP viewpoint, I will use the specific name “RM-ODP viewpoint”.

In Figure 3.6, on the vertical axis, the specifications are describing the same aspect of a system at different levels of abstraction; the ones at lower levels are refinements of the ones at higher levels. On the horizontal axis, the specifications are describing different aspects of a system at the same level of abstraction.

Even if the development process in engineering and in RM-ODP follow the same principles, there are different ways of classifying viewpoints in the two domains. For the purpose of this work, I have associated a viewpoint to a development branch in the engineering development process. Development branches are identified according to the specifics of the development process. For example, one classification is in mechanical, hydraulics, electric and electronics

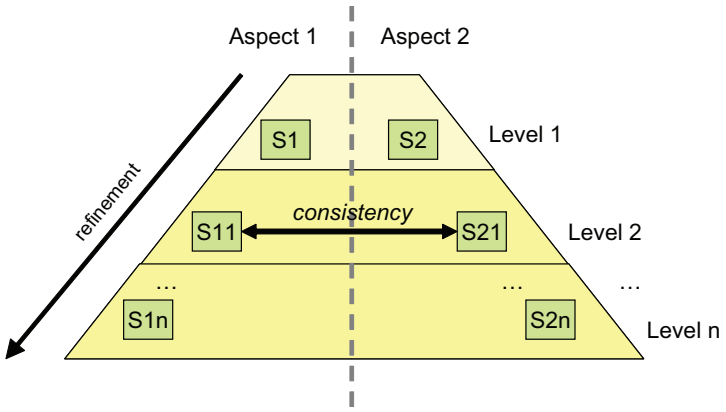

Figure 3.6.: Refinement and levels of abstractions for different specifications.

branches. Another classification might be in functional and geometrical viewpoints. No matter what classification is chosen, there are some shared characteristics of the classifications: the design models in the development branches are describing different aspects or parts of the same system and are built by different engineering teams in a parallel fashion.

In the engineering development process, the consistency is usually assessed between design models at (approximately) the same level of abstraction. This situation is also depicted in Figure 3.6. However, it is also possible to check the consistency of design models at different levels of abstraction, if this is supported by the formal representation and underlying consistency checking algorithm.

The supporting mechanism for checking consistency between different design models is the definition of explicit mappings or relationships between corresponding terms in the languages of design models. In the next sections, I will define formally the consistency between design models. In Chapter 6, I will discuss in more detail how the mappings between the design models are represented and how they can be composed together in order to check the consistency of compositional design models.

Taking into consideration that the design models are also specifications for the product itself, we may apply the consistency definition for two design models in two different viewpoints. According to the Definition 4, two design models are consistent if there exist at least a product (real implementation) that conforms to both of them. This state of affairs is also shown graphically in Figure 3.7.

The drawback of this definition is that conformance testing between the real product and a design model is not done using formal methods, but rather by measurements and other testing techniques on the real product (see Section 2.4.2). This means that a real prototype has to actually exist in order to do the conformance assessment. However, building a real prototype

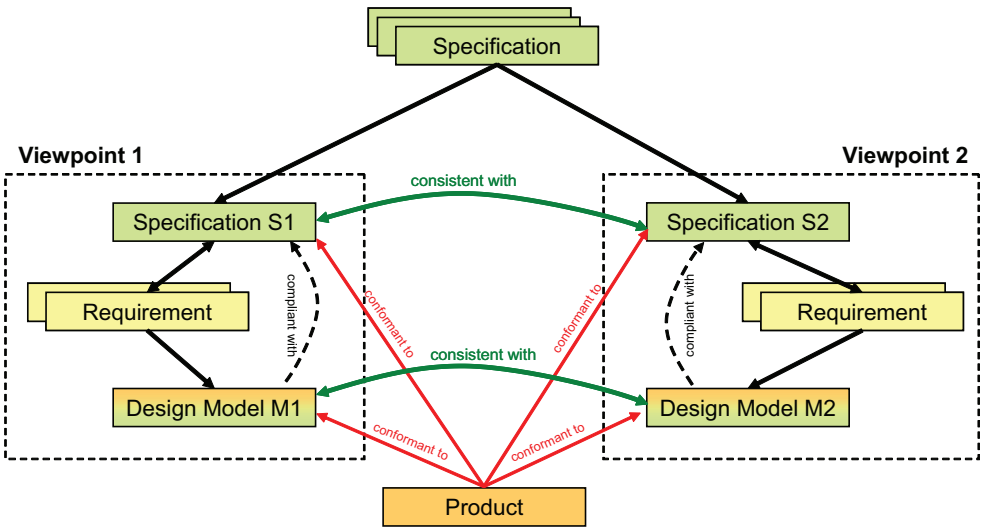

Figure 3.7.: The conformance and consistency relationships between two viewpoints.

is a something that the modern development process tries to delay as much as possible due to the high costs involved. The goal is to do most of the testing on a virtual realization, i.e., on a computer model, in a virtual experiment, and to refine and enhance the design model until it achieves the desired degree of maturity. Once the design model is considered to reflect the real product with certain accuracy and outputs the desired behavior, the building of the physical prototype may be started. Even if the final conformance testing is done on the real prototype, there are certain checks that can be done in the early development stages on the formal design models to rule out undesired or inconsistent models.

As this work is concerned with checking in a formal way the consistency between different design models, I will also assume that each viewpoint has its own formal language in which it is specified. Both the requirements and the design models are specified in the same formal language. However, different viewpoints may use different formal languages. In the case that the requirements are not specified in the same language as the design model, I will consider that there is a transformation of the requirements to sentences in the language of the design model.

There are several obstacles to checking formally the consistency of design models in two (or more) viewpoints. One of them is that the viewpoints have their own conceptualization of the product according to their own perspective of the world. A second obstacle is the fact that they typically use different formal languages to represent the concepts in the domain. Also, if the design models are at different levels of abstraction, they will represent the same object at different levels of granularity. In the following sections, I will discuss briefly the obstacles

that need to be taken into consideration when defining consistency between design models.

3.5.1. Differences in the Viewpoints

Each viewpoint has its own conceptualization of the product that is specific to its purpose. A conceptualization is an abstract, simplified view of the world that we wish to represent for some purpose [Gruber, 1993]. For example, a geometrical viewpoint will be concerned with representing the geometrical properties of the product and will operate with concepts like coordinates, points, other geometrical primitives, area, etc. and will not be concerned with the behavior of the system. A functional viewpoint will ignore the geometrical features and form of the system and will operate with components, ports, interactions, states, etc. Even if the specifications and their corresponding design models operate with a different conceptualization of the system, they still have a common ground, which is that in the end they represent the same system under different aspects.

The key to consistency is to find the correspondences between the design models and to make them explicit, in this way enabling automatic and formal checking of consistency.

The correspondence knowledge may often be implicit, for instance using the same name for corresponding terms in two specifications. A very common situation in collaborative engineering is when international teams are working together and each develops models using names and conventions from their own language. At certain milestones they need to integrate the models into a common design. This process is very tedious and requires a lot of manual work and the knowledge of the experts who have developed the models. These obstacles are caused by the fact that there is no common vocabulary used in the different specifications and design models. Even if there were one, there is no commitment to use the same terms with the same meaning. Meaning of terms remains often implicit in the tools and in the experts' background knowledge.

Finding the correspondences and making them explicit is an important task if we intend to check the consistency between different design models in an automated and formal way. However, finding the correspondences between design models that reflect two different conceptualizations of a product is a difficult problem, due to the fact that there might be different types of mismatches that might occur between them, which I will discuss in the following.

Implicit Knowledge and Domain Assumptions

Implicit knowledge and domain assumptions make up a significant part of the knowledge in a domain. Usually this type of knowledge is also hard-coded in the implementation of the tools. A novice engineer will have difficulties developing a model without knowing the assumptions made in the domain and in the implementation of the tool. To illustrate the implicit knowledge and domain assumptions, I will give an example from a functional modeling tool, which has

as representation entities, components that have ports through which they can communicate with the environment and with other components. Ports are interfaces of the components to the environment, and have a certain type, like electric port, information port, hydraulic port, mechanical port, etc. There is an assumption in the domain and also implicitly in the tool, that only ports of compatible types may be connected with each other. Connecting an electrical port and a hydraulic port results in an error, which is not discovered until the model is compiled and simulated.

Gruber [Gruber, 1993] proposed ontologies, defined as explicit specification of a conceptualization, as a mean to make explicit such domain assumptions and for enabling the knowledge sharing between different applications or software agents. Noy [Noy and McGuinness, 2001] also noted that making the domain assumptions explicit by using ontologies enables changing the domain assumptions when the knowledge about the domain changes.

Overlapping Conceptualizations

An important facet of collaborative engineering is that each team is concerned with only one aspect of the system. Each viewpoint represents only the part of the world that is of interest for the particular task to solve and will leave out any unnecessary information. Hence, the design models will also be based on different conceptualizations, which are not completely independent of each other. The common overlapping fragment of the viewpoints' conceptualizations ensures the fact that the viewpoints are actually describing the same system. It might be the case that different names in the viewpoints are used to denote the same physical object. However, at the conceptualization level, the correspondent of the two names will be the same object. This common object is part of the overlap between the two conceptualizations.

For instance, one specification may state that the weight of the motor should be less than 200kg – this being a constructive restriction, while the other specification may state that the engine should have a power of 165kW – a functional restriction. One specification uses the term "*engine*" and the other one the term "*motor*" and implicitly they refer to the same physical object that has a weight less than 200kg and a power of 165kW [Figure 3.8]. This means that the two terms refer to a common object in the conceptualization of the two viewpoints, and hence to the same physical object in the physical world.

Such correspondence must be made explicit, if the consistency of two viewpoints needs to be checked formally. In UML the correspondences between the different UML views are made implicitely by using the same name of a class across the different views. However, it is very often the case, such as also in engineering development, that the views contain redundant information (for example, different UML views specify the cardinality of the same relationship). Egyed [Egyed, 2000] has identified these redundancies to be the main cause of inconsistencies between the views. In engineering, a similar situation occurs, when two design models

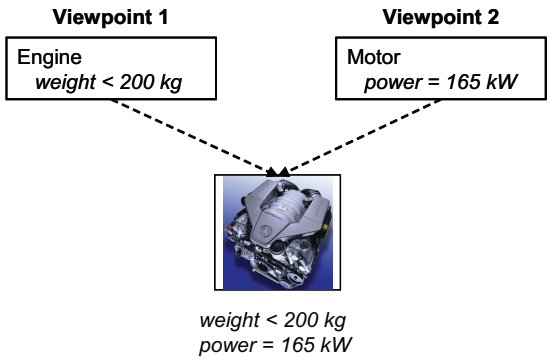

Figure 3.8.: Two terms in different specifications refer to a common object in the conceptualization of the two viewpoints.

specify the weight of the car with different values.

Different Terminologies

Different viewpoints use typically different terminologies to describe a certain aspect of a product. Each development branch has established development methods supported by specific tools and languages. Even more, development teams are international and use language-specific names for the concepts in the design models. The most common differences in terminologies are related to the usage of synonyms and homonyms. I will give in the following examples for each of these cases.

The first example comes from a very common situation in a collaborative development environment, in which a geometrical and a functional model of a car have been developed in parallel by different teams. The geometrical model has been developed using a CAD tool in German. The corresponding functional model has been developed using an English model library that uses English names for the concepts. Both the models describe the same clutch object, but use different names to denote it: The geometrical model used the term "*Kupplung*", while the functional model used the term "*Clutch*".

Another situation is one in which the same term is used in two specifications to mean different things. For example, in two manufacturing planning specifications, the term "*resource*" may be used with different meanings: In the first specification, "*resource*" is used to denote the car being assembled, while in the second view, "*resource*" is used for the machine (robot) that actually assembles the car.

These two types of mismatches are called by Klein *terminological mismatches*; the first one being a *synonym* mismatch and the second one a *homonym* mismatch [Figure 3.9]. Klein

[Klein, 2001] investigates in his paper also other types of mismatches that might appear between different conceptualization of a domain.

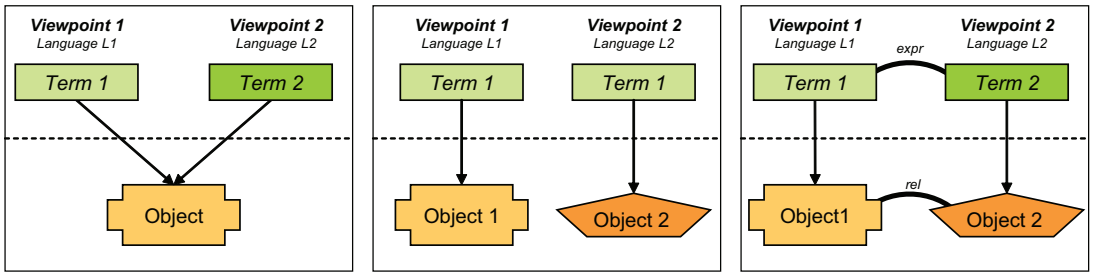

Figure 3.9.: Different terms in different languages may mean the same thing (left picture). The same term may refer to two different objects (center picture). It might be also the case that terms in different languages are related by some expression (right picture).

There are also more subtle dependencies between terms in different models resulting from the dependencies between their domains that are often left implicit. For instance, geometry of a system determines its behavior, so it is natural that there will also be relationships between the geometrical and functional components in the corresponding specifications. In a geometrical model, the mass of a component may be specified, while in a functional model, the mass is irrelevant, but the moment of inertia of the component is needed for simulation. There is a mathematical relationship that allows the computation of the moment of inertia using the mass and other geometrical characteristics of the component. However, this relationship is only implicit, but it is needed in order to put into correspondences the two models and ensure that they are describing the same physical system.

Granularity Mismatches

It is very often the case that the viewpoint languages represent objects with different granularities. For example, one viewpoint might represent the engine as a whole, while another viewpoint also represents the components of the engine and the relationships among them. In the first viewpoint, the engine is seen as a black box and only the interfaces of the engine to the environment are of import, while in the second viewpoint, the engine is a white box in which the internals of the engines are important and the way that the behavior of the engine is composed out of the behavior of its components.

In Figure 3.10, System 1 corresponds to System 2. They represent the same physical system from different viewpoints and at different levels of granularity. The two systems may be related at the level of their interface points in which certain parameters or behavior of the systems are exposed.

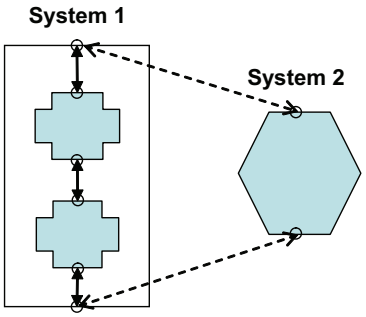

Figure 3.10.: Correspondences between models with different granularity. Circles represent interface points of the systems. The dotted arrow represents the correspondences between the interface points of different models.

However, it might be the case that variables in the System 2 interface are dependent not only on the variables exposed by the external interfaces of System 1, but also on variables of the internal components of System 1. In this case, we need a way to "navigate" the internal structure of System 1 and to address the internal components. In order to relate the models in the two viewpoints, it is necessary to be able to represent also 1-to-n and n-to-1 correspondences between parts of the models. For example, viewpoint 1 could represent the engine as a whole which has a certain weight, while viewpoint 2 represents the engine as a decomposition of several components, each having a specific weight. In order to check whether the weight of the engine in the two viewpoints is the same (as it should be) we need to compare the weight of the engine in the first viewpoint with the sum of the weight of engine components in the second viewpoint. In order to do the sum operation, we need a way to represent, address and navigate the components of the engine in second viewpoint.

3.6. Consistency of Engineering Design Models

In the following I will first give the definition of some terms needed to define in a formal way consistency in engineering design. I will discuss two types of consistency: internal consistency of a design model and consistency among design models in different viewpoints. The definitions will take into account the challenges identified in the previous sections. I will consider that the each viewpoint has associated a first order language in which the elements of a viewpoint are represented. A first order language is typically expressive enough for the representation of the knowledge required in the engineering design domain.

Since the term *model* is overloaded, I will use the term *design model* to refer to the representation of a design model in a formal language. And I will use the term *logical model*[1] to refer to

[1] A logical model is an interpretation that satisfies all the formulas in the design model

a first order model (or structure) of the design model as defined in [Barwise and Etchemendy, 1999].

3.6.1. Domain Knowledge

The domain knowledge contains general rules of the domain of interest. Design models are used in the context of the domain in which they are defined. For example, some design models will need to apply the laws of thermodynamics, which are not part of the design model itself, but are general rules in the thermodynamics domain, which are just used in the specification of the design model.

The *domain knowledge DK* is represented as a set of axioms in the language of the viewpoint that makes explicit the knowledge and assumptions of the domain. It defines a vocabulary of predicates, functions and constants, which is called the *ontology* of the domain [Russell and Norvig, 2003]. The axioms encode the rules that always hold in the domain and that make clear the meaning of the terms (to a certain extent). An example of a domain ontology is the PhysSys ontology [Borst, 1997] that describes the domain knowledge required to make simulation models for physical devices.

While the domain ontologies can be very general, like for example, representing the general theory of systems, they may also be specialized and used as a library of model fragments for the domain. If we would imagine a domain ontology for physical system, a design model is an instantiation of the model fragments that are interconnected with each other.

A possible representation of a domain of systems may contain the unary predicates `System(x)` and `Part(x)` denoting that `x` is a system and a part respectively. A binary predicate `hasPart(x,y)` encodes the fact that `x` has a part `y` and has the property that it is transitive. Another axiom of the domain might be that only parts of compatible types may be connected together. By identifying the general domain rules and by representing them as axioms, we ensure that the design models, which are a specific instantiation of the concepts in the domain, reflect the reality more faithfully. From the logical point of view, the additional domain knowledge axioms ensure a more precise semantics of the terms in the ontology. In Chapter 5, I will discuss in more depth a domain ontology used in the representation of design models.

3.6.2. Design Model

As I have already mentioned in previous section, if the domain ontology is used as a library of model fragments, then a *design model DM* encodes a specific instantiation of the concepts in the domain. It is represented as a set of sentences in the language of the viewpoint by using the vocabulary (ontology) defined in the domain knowledge. For the systems domain example given before, a design model might be a simple system that contains only a subsystem and three parts. The representation of this design model is:

```
System(mySimpleSystem)
System(subsystem1)
Part(part1)
Part(part2)
Part(part3)
hasPart(mySimpleSystem, subsystem1)
hasPart(mySimpleSystem, part1)
hasPart(mySimpleSystem, part2)
hasPart(subsystem1, part3)
```

Since the design model is an instantiation of the domain knowledge, it will obey the axioms defined in the domain knowledge. For example, if the `hasPart` predicate has the transitive property defined in the domain ontology, then we can infer that `part3` is also a part of `mySimpleSystem`. I will describe in more detail the representation of design models in Chapter 5.

3.6.3. Viewpoint

A *viewpoint* describes the perspective of a certain development branch on the product. From the formal point of view, a viewpoint is a tuple that contains a specification, a design model and domain knowledge. Each viewpoint has associated a first order language L. The notation for a viewpoint is $V = \langle S, DK, DM \rangle$. The specification S contains a set of requirements on the design model. Even if the specification is defined in another language, I will consider that there is a transformation from that language to the first order language of the viewpoint.
[Ciocoiu and Nau, 2000] has used the term *logical rendering* to describe the transformation of sentences from some declarative language *Lang* to a first order language L. This is achieved by applying a logical rendering function to a set of sentences in *Lang* having as result a set of sentences in the first order language L, which is called the *logical image* in L of the sentences in *Lang*. For example, a sentence in a declarative language *Lang* `(partOf ?x ?y)` can be translated in a first order language L as `partOf(x,y)`. In the following I will consider that the specification is a set of requirements that have already been logically rendered in the first order language of the viewpoint in a way which preserves the meaning of the original sentences in the specification. The different types of requirements that are part of a specification are discussed in detail in Chapter 5, Section 3.1.

3.6.4. Development Context

I define a **development context** to be a set of viewpoints that participate in a parallel development process. The viewpoints in a development context represent a product under different perspectives, which are related in some way in the development process. One example on how

viewpoints might be related in the development process is that the corresponding design models need to be synchronized or checked for consistency with each other at certain milestones. The notation for a development context is $\{V_1, \ldots, V_n\}$. In the case that there are only two viewpoints, the development context is denoted as $\{V_1, V_2\}$.

3.6.5. Internal Consistency

Definition 5 *A design model DM in a viewpoint V is said to be* ***internally consistent*** *if there exists at least one logical model of* $DM \cup DK \cup S$.

This can also be written as: *DM* is **internally consistent** if $\exists M$ s.th. $M \models DM \cup DK \cup S$, where the entailment $\models$ is the first order logic entailment.

This definition states that a design model is locally consistent if it is possible to find a first order logic structure (logical model) that satisfies the sentences that are describing the design model (*DM*), the axioms of the domain knowledge (*DK*) and also the sentences representing the requirements on the design model. The logical model is called a *local model* and defines the *local semantics* of a design model. A local model is a first order structure that is defined as a tuple $\langle dom, \mathfrak{I} \rangle$ where *dom* is the universe of discourse of the model and $\mathfrak{I}$ is an interpretation function.

3.6.6. Consistency among Design Models

In the following, I will define formally the consistency of two design models. However, there is no limitation to only two design models and the subsequent definitions can be applied to any number of design models. I have done this simplification in order to keep the definitions clear and to stay close to the motivating scenario.

As I have already pointed out in Section 3.5, the consistency between design models in different viewpoints can be defined in a similar fashion as consistency between specifications: Two design models are *consistent* if there exists a common realization that is conformant to both of them. Assessing the consistency requires the presence of an implementation against which conformance can be checked. A conformant implementation of the design models is rarely available. Checking the consistency of the design models at the formal level still brings many benefits. Even if it cannot guarantee that the common realization will be implementable, it can detect possible conflicts and inconsistencies in the design models and will prevent the building of prototypes that are known to be "wrong" or impossible to realize; for example, in the case that certain values of parameters in the models are in conflict with each other.

Intuitively, two design models are consistent if it is possible to build a common design model that is valid, that is, it is at least free of contradictions. This has a close correspondent in formal logics as also mentioned by [Bab and Mahr, 2005]: If we consider each model to be

a set of sentences, then two design models are consistent if there exists at least one common logical model of the two sets of sentences. However, there are several impediments that need to be tackled before consistency can be checked in a formal manner. First, we should take into consideration that the design models are defined in different formal languages with different semantics. Second, the correspondence relationships that link two design models together must also be expressed in a formal way and be used in consistency checking. Third, the domain knowledge is usually implicit, but plays an important role when bringing together design models with different world views.

In order to address these issues and to give a more precise definition of the consistency between design models, I will use the work done by Ghidini and Serafini on *Distributed First Order Logic* (DFOL) [Ghidini and Serafini, 1998]. DFOL makes the assumption that in a distributed knowledge representation system, knowledge is represented by a set of heterogeneous subsystems, that are not completely independent from each other, each of them representing their knowledge about the world from a certain point of view. DFOL provides the means to represent and reason about the relationships between terms of different first order languages with different semantics. [Ghidini and Serafini, 1998] defines the semantics of DFOL and the logical consequence for this semantics. It also describes a calculus for DFOL that is sound and complete w.r.t. the DFOL logical consequence.

DFOL defines the notion of a model of the distributed knowledge representation system as a tuple containing a set of local models of each language and a binary relationship r_{ij} relating the objects in the universe of discourse for each pair of languages L_i and L_j. L_i and L_j, are not necessarily disjoint: The same formula may appear in both of them, but may have different meaning.

For the case of two languages, L_1 and L_2, a model of the distributed system is $\langle\{S_1, S_2\}, \{r_{12}\}\rangle$, where S_1 and S_2 are a set of possible models for L_1 and L_2 on dom_1 and dom_2 respectively. The r_{12} relationship is called a *domain relationship* and relates objects from the domains of discourse. It is a directed relationship and a subset of $dom_1 \times dom_2$. If a tuple $\langle o_1, o_2\rangle$ is in the relationship r_{12} (where o_1 is from dom_1 and o_2 is from dom_2), it means that from the perspective of the second viewpoint, o_1 represents the same object as o_2. The converse may not be true: From the perspective of viewpoint one, o_2 may not represent the same object as o_1. However, this may often be the case.

Figure 3.11 depicts an example: At the representation (or language) level, two symbols *syst1* in L_1 and *s3* in L_2 have a local interpretation in the corresponding domain of discourse given by the interpretation functions $\mathfrak{I}_1$ and $\mathfrak{I}_2$ respectively. Hence, *syst1* is interpreted as *o1* in dom_1 and *s3* is interpreted as *o5* in dom_2. At the domains level, there is a domain relationship r_{12} defined as: $r_{12} = \{\langle o1, o5\rangle, \ldots\}$. This means that from the perspective of the second viewpoint, *o1* in dom_1 represents the same object as *o5* in dom_2. This is also shown by the arrow going from $\langle o1, o5\rangle$ to a real world thing, like for example an engine, which is represented by both *o1* and *o5* from the subjective perspective of viewpoint two. As a consequence, at the

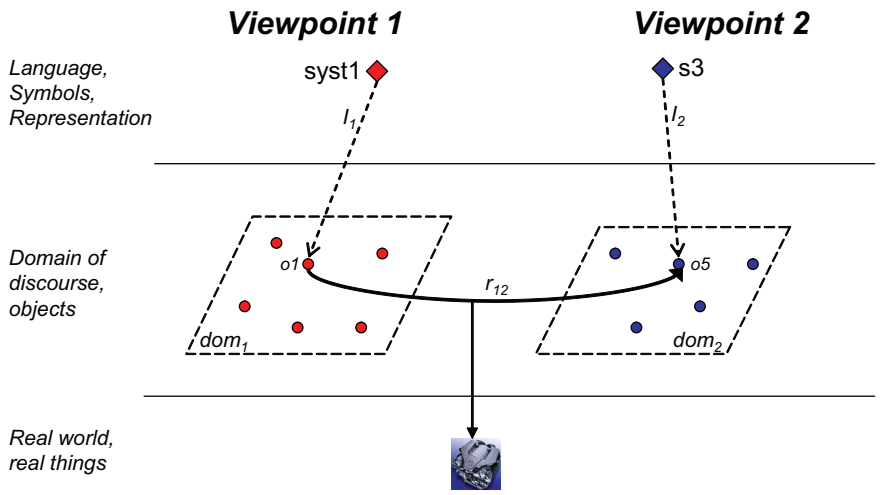

Figure 3.11.: The meaning of the symbols in the languages (e.g., *syst1* and *s3*) is given by the interpretation functions ($\mathfrak{I}_1$ and $\mathfrak{I}_2$), which relate symbols in the language to objects in the domain of discourse (dom_1 and dom_2). The domain relationship r_{12} states the fact that objects *o1* and *o5* are the same from the perspective of the second viewpoint.

language level, *syst1* and *s3* may be mapped to each other from the perspective of viewpoint two.

DFOL defines two types of constraints on DFOL models: the *domain constraints* and the *view constraints* [Ghidini and Serafini, 1998]. Domain constraints describe the containment relationship among the domains of discourse. A view constraint relates formulae over the two languages. In the following, I will use the notation *L:p* to represent the fact that *p* is a formula of the language *L*.

A *view constraint*, also called *mapping*, has the form $L_1 : \phi(x_1,\ldots,x_n) \rightarrow L_2 : \psi(x_1,\ldots,x_n)$ and captures the fact that from the perspective of viewpoint two, the set of tuples of objects of dom_1 which satisfy $\phi(x_1,\ldots,x_n)$ in L_1 corresponds to a set of tuples which satisfy $\psi(x_1,\ldots,x_n)$ in L_2. For example, if language L_1 has a predicate *hasPart(x,y)*, which represents the fact that *x* has a part *y*, and language L_2 has a predicate *besteht_aus(x,y)* with the same semantics as the first predicate, we can define a view constraint:

$$L_1 : hasPart(x,y) \rightarrow L_2 : besteht\_aus(x,y)$$

For this case we can also define the inverse constraint, since this correspondence holds from both the viewpoint one and viewpoint two perspectives:

$$L_2 : besteht\_aus(x,y) \rightarrow L_1 : hasPart(x,y)$$

An *interschema constraint* from viewpoint 1 to viewpoint 2 is defined as a tuple $IC_{12} = \langle DC_{12}, VC_{12} \rangle$, where DC_{12} represent the domain constraints and VC_{12} the view constraints from viewpoint 1 to viewpoint 2. The notion of the satisfiability of interschema constraints in a distributed model $M = \langle \{S_1, S_2\}, \{r_{12}\} \rangle$ is also defined in DFOL.

I will use in the following the semantics defined by DFOL adapted for the design models domain in order to define the consistency between design models.

I will consider two viewpoints, $V_1 = \langle S_1, DK_1, DM_1 \rangle$ represented in first order language L_1, and $V_2 = \langle S_2, DK_2, DM_2 \rangle$ represented in first order language L_2.
The consistency of two design models is defined with respect to the interschema constraints between L_1 and L_2. Since the interschema constraints are directional (from L_1 to L_2), it also makes the consistency notion directional.

Definition 6 *Considering two viewpoints* $V_1 = \langle S_1, DK_1, DM_1 \rangle$ *and* $V_2 = \langle S_2, DK_2, DM_2 \rangle$ *described in first order languages* L_1 *and* L_2 *respectively, and a set of interschema constraints* IC_{12}*, we say that the design model* DM_1 *is* ***consistent*** *with design model* DM_2*, if there exists a common logical model of* $(DK_1 \cup DM_1)$*and*$(DK_2 \cup DM_2)$ *that satisfy the interschema constraints* IC_{12} *as defined by DFOL.*

This notion of consistency does not require that the design models are also internally consistent, i.e. that they also satisfy locally the requirements in the specification of the viewpoints.

Definition 7 *Considering two viewpoints* $V_1 = \langle S_1, DK_1, DM_1 \rangle$ *and* $V_2 = \langle S_2, DK_2, DM_2 \rangle$ *described in first order languages* L_1 *and* L_2 *respectively, and a set of interschema constraints* IC_{12}*, we say that the design models* DM_1 *and* DM_2 *are* ***consistent with each other****, if the* DM_1 *is consistent with* DM_2 *and* DM_2 *is consistent with* DM_1*.*

It is often the case in the engineering development process, that the communication between the engineering tools is done only in one direction (from tool 1 $\rightarrow$ tool 2) as defined in the workflow of the development process. For this case it is enough to ensure that the corresponding design models are consistent only in the direction required by the communication. However, the goal of the development process is to obtain a product that is conformant to both design models. This implies that the design models need to be internally consistent (i.e., they fulfill the requirements in the specification) and also they are consistent with each other.

In Chapter 5, I will discuss in detail the representation of the design models by using an engineering domain ontology. Chapter 6, will deal with the representation and composition of design models mappings and how they can be used to ensure the consistency of design models in the development process.

4. Ontologies in Engineering Modeling

This section proposes ontologies for the formal representation of design models to support the development of higher quality design models and to enable knowledge sharing and reuse. Section 4.1 gives a Systems Engineering perspective on the development process and identifies *systems* and their parts to be the building blocks of engineering design models. Section 4.3 proposes *Components*, *Connections* and *Constraints* as general ontologies for modeling design models. Section 4.4 introduces the *Requirements* ontology that may be used to represent requirements and their attachment to systems and parts along the development process. Section 4.5 presents the *Constraints* ontology which may be used to restrict model parameters and to define relationships between them. Section 4.5.3 shows how the engineering ontologies may be used together to build an application ontology.

4.1. A Systems Engineering View on the Engineering Development Process

The goal of the engineering design process is to build complex systems that fulfill the customer's requirements. The methodology used for developing systems starting from requirements up to the realization phase is called Systems Engineering (SE). Systems are used as abstractions to divide a complex problem in several manageable pieces. For example, a car may be viewed as a system comprised of different subsystems, like powertrain, body, chassis, etc.

In order to have a more formal approach for supporting different phases of the design process, we need to identify the main concepts involved in the design process and tasks to be solved. In the following section, I will briefly describe the systems engineering approach, which defines the main terminology used in design. Then, I will identify the main components building up a system and other concepts that I will represent formally using ontologies.

4.1.1. Systems Engineering

Systems engineering deals with the methods necessary for developing and implementing complex systems [Stevens et al., 1998]. The definition of Systems Engineering given by the Inter-

national Council on Systems Engineering (INCOSE[1]) is the following: "*Systems Engineering is an interdisciplinary approach and means to enable the realization of successful systems. It focuses on defining customer needs and required functionality early in the development cycle, documenting requirements, then proceeding with design synthesis and system validation while considering the complete problem...*"

Systems engineering has been very successfully applied in software engineering. Following this success, systems engineering has also been adopted as the basis for the mechatronic development process [VDI 2206, 2004] to respond to the increasing product complexity in a challenging market situation. It is current practice, that activities in the system engineering process – for example, requirements specification, analysis, design, verification, validation, etc. – use different modeling languages, techniques and tools. The Object Management Group (OMG[2]) recognized the need for a general-purpose modeling language for describing systems that also provides a unified graphical representation. Several commercial and non-commercial organizations have joined efforts to respond to this need, which resulted in a general modeling language for systems engineering, SysML [2006]. SysML has a graphical notation based on a UML 2.0 [2005] profile and provides a simple, yet powerful language for specification of complex systems. For example, it has constructs for representing requirements, systems, composition and interconnection of systems, allocation of requirements to systems, behavioral diagrams, etc. However, even if SysML provides a basic language and a graphical notation for systems engineering, it lacks a formal semantics.

I will model the engineering systems using ontologies, which provide a precise way for defining the semantics of the terms. I will reuse part of the terms defined by SysML and give formal definitions for them.

4.1.2. System Models

The centerpiece of engineering design is the system concept. According to the ISO Standard ISO/IEC 15288 [2002], a system is a combination of interacting elements organized to achieve one or more goals. In engineering, computer models of systems are built for different purposes: understanding a system better, investigating its properties, or even to build it.

Neelamkavil gives a very illustrative definition of a model, which is "*a simplified representation of a system intended to enhance our ability to understand, predict and possibly control the behavior of a system*" [Neelamkavil, 1987]. A more detailed description about systems is given in Section 2.3.3.

A system is defined by its parts and connections. Connections between components can be made only in the interaction points called *ports* or *connectors*. SysML denotes the connectors

[1] http://www.incose.org
[2] http://www.omg.org

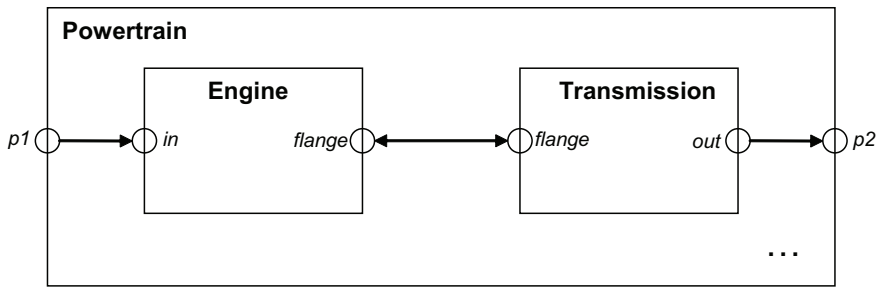

Figure 4.1.: An example of a simple system.

as *ports* as they are derived from the UML 2 notion of *ports*. They support building modular systems out of reusable components with clearly defined interfaces. SysML also makes a very important distinction between the definition of a component and its usage. The definition of a component is given by the declaration of its parts and their connections, whereas its usage is given by the role that the component plays as part of another component.

An example of a system is given in Figure 4.1, which depicts the components and connections of a powertrain subsystem of a car. Only two components of the powertrain system are shown for the sake of simplicity. In this example, a powertrain system is composed of two components – an engine and a transmission – and contains three connections.

I will use the following notation to explain the example.

`Component` – to denote a component. *Example*: `Powertrain` or `Engine`.

`Component.port` – to denote a port of a component. *Example*: `Engine.in`.

`connected(Component1.port1, Component2.port2)` – to denote that `port1` of `Component1` is connected to `port2` of `Component2`.
Example: `connected(Engine.flange, Transmission.flange)`.

The system `Powertrain` is defined by its parts: `Engine` and `Transmission` and the set of connection between them:

```
{connected(Powertrain.p1,Engine.in),
 connected(Engine.flange,Transmission.flange),
 connected(Transmission.out,Powertrain.p2)}
```

The connection notation has been inspired from Modelica [Modelica Specification, 2005] – an object-oriented language for modeling physical systems. The semantics of the connection is

related to the type of ports that it connects. Modelica supports two ways of handling connected ports. If the variables in the ports are *across variables*, then they are set equal in the connected ports (for example, voltages in two connected electrical pins). If the variables are of type *flow* (like in the case of electrical currents), then Kirchhoff's law is applied, meaning that the sum of the variables in a node is equal to zero.

4.2. Engineering Modeling with Ontologies

Engineering modeling is the activity of building models of engineering systems to be used in solving different tasks in product development, such as design, analysis, diagnosis, etc. Engineering modeling can be seen as a form of design, which is composed of several subtasks [Top and Akkermans, 1994]. Building an engineering model became a difficult process with the increase of product complexity. Engineers rely more and more on the support of model editors to help them build high quality design models.

Most models that are built in the modern development process are simplified representation of a system in a computer language. They are used to investigate the properties of the system, to understand better the system behavior, or they may even serve as a specification for building physical prototypes of the product (see Section 3.4.2).

Very often, modeling is done in a compositional fashion [Falkenhainer and Forbus, 1991]: General model fragments are assembled together to form the structure of a new model. In the case that the internal structure of the general components is not changed, the modeling process can be seen as a configuration task [Breuker and Van de Velde, 1994].

The generic model fragments are stored in model libraries, which grow in size over time. Very good model management methods are needed in order to support the model building process. Classification of models in different hierarchies play an important role in navigating and searching for adequate model fragments. Models in a library have to be annotated with different kind of information, like, modeling assumption (for example, "friction between elements is disregarded"), meta-data (like author, creation date, version), or other type of documentation (like design rationale). Change history also plays an important role in design.

Besides the model management aspects, other types of knowledge support the engineer in the modeling task. For example, consistency checking of a design model can be done in an automated fashion, if general rules or constraints of the domain are specified explicitly in a formal manner.

All these modeling support activities are possible if the underlying representation of models is rich enough to capture all necessary types of knowledge. Ontologies offer a rich and computer-interpretable representation of a domain that can be used to support an improved modeling process.

I will give a short introduction of ontologies in Section 4.2.1. In Section 4.2.2, I will discuss why ontologies are needed and the benefit of using ontologies in engineering modeling. Section 4.2.3 gives an overview of the Protégé knowledge model which will be used in building the engineering ontologies.

In Sections 4.3 to 4.5 I will describe three ontologies that have been used to support the model building process. They enable model reuse and improve the quality of design models by supporting automated and formal consistency checking. The ontologies have also been employed for model exchange between different engineering design tools. Sections 4.4 and 4.5 describe the *Requirements* and *Constraints* ontologies.

4.2.1. What is an Ontology?

The word "**ontology**" was first used in philosophy. It has its roots in the Greek words "*on*" (genitive "*ontos*") – "being", and "*logia*" – "writing about, study of". In philosophy, ontology is a theory about the nature of existence.

There are many definitions of the term ontology in Artificial Intelligence. The most broadly used one is given by Gruber [1993]: "*An ontology is a formal specification of a conceptualization*".

A conceptualization is an abstract simplified view of a domain that describes the objects, concepts and relationships between them that hold in that domain. All applications and knowledge-based systems are using some kind of implicit or explicit conceptualization of the domain they are working on. The set of objects that are formally represented is called the universe of discourse. The *formal specification* implies the existence of a representational vocabulary in which the objects in the universe of discourse and their relationships can be formally represented. Axioms may be used to constrain the interpretation of the terms in the vocabulary.

Borst [1997], who investigated the reuse of engineering ontologies, extends the definition given by Gruber by emphasizing that in order to be useful, an ontology should be reusable and shared across several applications. His definition of ontology is: "*An ontology is a formal specification of a shared conceptualization*".

Studer and his colleagues [Studer et al., 1998] expand the former definition by identifying another criterion for a useful ontology. In order to be reused, an ontology should make the domain assumptions explicit. Their definition of ontology is:"*An ontology is a formal, explicit specification of a shared conceptualization*". I will adopt this definition of ontologies for the scope of this work.

Some of the benefits of using ontologies are that they define a common understanding of a domain, so that they can be used to support inter-human and inter-organizational communication; and being machine-processable, they support the semantic interoperation of different software systems.

4.2.2. The Need for Ontologies

Developing an ontology is not a goal in itself. The effort spent in building an ontology has to pay off in the end. I will enumerate in the following some of the benefits of using ontologies in engineering modeling.

Share common understanding of a domain. One of the main benefits of using ontologies is that they enable knowledge sharing between humans and software applications by providing a common understanding of a domain [Gruber, 1993; Musen, 1992; Noy and McGuinness, 2001]. When several software applications commit to an ontology, it is guaranteed that they will use a term with the same meaning as specified in the ontology. For instance, if the ontology specifies a term "*Resource*" and defines it as a consumable thing, all software applications that commit to the ontology will use the same meaning of the term. This facilitates application integration and interoperability. Ciocoiu has investigated how ontologies may be used for integrating engineering applications [Ciocoiu et al., 2001].

Share terminology. A consequence of sharing a common ontology between applications is that they will also share the terminology defined in the ontology. Ontologies provide a declarative, machine readable representation that enable an unambiguous communication between software agents. An example of such an ontology is *EngMath* [Gruber and Olsen, 1994], which has been used as a common vocabulary for engineering mathematics in agent communications.

Make domain knowledge explicit and reason with it. Ontologies are used to make explicit the knowledge of a domain and to expose it as a formal representation. By having a formal representation, the domain knowledge can be used in formal algorithms for solving different tasks. For example, the configuration of a system may be automated, if all the required knowledge (such as the available components and the domain constraints) is explicitly represented.

Enable reuse. Ontologies expose implicit knowledge that has been previously hidden in domain assumptions or in the implementation of an application. There are several benefits for exposing this knowledge. First, the domain assumptions and implementation may be checked for correctness. Second, reuse of existing models is facilitated because they become visible to the external world and they can be searched for. For example, a library of physical models may be represented with the help of an ontology. To each component in the library information is attached that describes the function of the component (for example, a pipe has the function "transport liquid"). If this information is represented explicitly, an engineer may be able to search the library for component models that fulfill a certain function, rather then designing a new one.

Improved consistency. Ontologies give a precise semantics to its terms by employing formal axioms that constrain the possible interpretations of the meaning of terms. In this sense, an ontology is the statement of a logical theory [Gruber, 1993]. By representing the knowledge of a domain as a logical theory, it is possible to reason about it. For example, using an ontology of physical systems, which contains axioms about valid connections types between components, it is possible to use logical reasoning on a design model (represented using terms from the ontology) to check for inconsistencies in the model.

4.2.3. The Representation Formalism

There are several knowledge representation formalisms (KRF) which may be used to specify an ontology, such as frame-based systems, description logics, rule systems, etc. They differ in the expressiveness of the language (for example, some may allow the representation of defaults, while others will not); in the supported inference mechanisms (for example, subsumption reasoning, constraints checking); in the assumptions about the world (closed world vs. open world assumption); they have different tool support, etc. A description of the different ontology representation languages and ontology editors is given in [Gómez-Pérez et al., 2004].

Requirements on the Representation Formalism

In order to choose an appropriate representation formalism for the engineering ontologies developed in this work, I took into consideration the characteristics of the engineering models that should be represented in the ontology and the types of tasks to be solved.

The requirements on the representation formalism are presented below.

1. *Structured knowledge*. The engineering design models are very structured by nature. Therefore, the representation formalism should allow a structured representation that is natural to the domain experts.

2. *Scalability*. The models tend to be very large and complex. This means that the KRF should be able to handle large-scale models.

3. *Representation of templates* (general vs. concrete knowledge). The engineering models are usually built as templates that are instantiated in different concrete models. The KRF should support the representation and instantiation of model templates.

4. *Model reuse*. General models may be refined in other models. This means that the KRF should support inheritance.

5. *Complex relationships*. Complex relationships are used in a design model, like part-of, connections, and other types of relationships. These relationships must be expressed both for the general models (templates or classes) as well as for concrete models. The KRF should also support the representation of relationship types and reification[3].

6. *Constraints*. Constraints are a common ingredient of design models. The KRF should be able to represent mathematical constraints, such as $weight < 1500$.

7. *Consistency rules*. The KRF should be able to represent general rules that insure that a design system is correct. For example, certain components in a system must be connected in an appropriate way (we cannot connect an electrical port to a mechanical port).

8. *Query answerability*. The engineers should be able to ask queries about a concrete model (like, all the parts of this system) and also about the template model ("What type of engines are allowed in this car").

The knowledge representation formalism that has been chosen for specifying the engineering ontologies was a frame-based formalism, Protégé [Noy et al., 2000], which is compatible with the Open Knowledge-Base Connectivity (OKBC) protocol [Chaudhri et al., 1998]. OKBC is a common communication protocol for frame-based systems. The knowledge model of Protégé is described in the next section. Protégé[4] is a free, open source ontology editor and knowledge-base framework developed by Stanford Medical Informatics.

The Protégé knowledge model

In a frame representation system, a frame is the principal building block of a knowledge model, which represents an entity in the domain of discourse. The Protégé knowledge model [Noy et al., 2000] is built up from classes, slots, facets and instances.

Classes and Instances A class is a set of entities. An entity is said to be an *instance* of a class, and the class is known as the type of the instance. An instance may have multiple types. Classes are organized in taxonomic hierarchies formed by the *subclass-of* relationship. For example, an *ElectricalEngine* is a subclass of a more general class *Engine*. According to the semantics of subclass-of relationship in the knowledge model, all instances of *ElectricalEngine* are also instances of *Engine*. A class may have multiple superclasses (i.e., multiple inheritance is allowed). This enables modeling different classifications of the concepts in a domain. Classes are used as templates for building instances. For example, a class

[3] Reification is transforming an abstract thing into a concrete one. Reification is discussed in more detail later in this chapter.

[4] http://protege.stanford.edu/

Car serves as a template for an instance *MyCar*. Values may be assigned to the properties of instances that conform to the definition of the properties at the class level.

Slots Slots represent the properties of the entities in the domain. For example, a slot *weight* may describe the weight of a person. Slots are first order entities: They are themselves frames and can exist independently of their attachment to classes. In this case they are called *top-level slots*. Constraints are attached to the definition of slots that limit the values that the slot may take. The properties of the slots that can be constrained are: the value type (e.g., String, float, boolean, Instance, etc.), cardinality (e.g., single, multiple), minimum and maximum value for numerical slot, and so on. When a top-level slot is attached to a class, it becomes a *template slot* of the class, which is inherited to subclasses. When an instance of a class is created, its template slots are also instantiated and become *own slots* of the instance and may take values specific for that instance. For example, a class *Car* that has a template slot *weight* of value type float is instantiated. For the instance, *MyCar* of class *Car*, it is possible to specify a concrete float value for its own slot *weight*.

Facets Facets describe properties of slots. They are used to define constraints on allowed slot values. Examples of facets are the cardinality of a slot – that constrains how many values the slot may have, the range of a slot – that constrains the valid type of values for the slot, the minimum and maximum values for a numeric slot, and so on.

Metaclasses A metaclass is a class whose instances are themselves classes. A metaclass is a template for building classes, in the same way that classes are templates for building instances. Classes are also instances of metaclasses. Metaclasses are very useful in defining properties of the classes themselves, rather then properties of the instances of the classes. A very common use case for metaclasses is to provide synonyms for class names, or to provide names of classes in different languages. Metaclasses may also be used to add information that applies to all instances of a class, and therefore can be seen as a property of the class (e.g., average working hours for an employee).

Constraints, Axioms and Rules As many other frame representation systems, the Protégé knowledge model is very expressive. However, there are limitations on the things that can be expressed in the language. This includes complex types of constraints, rules or axioms of the domain. For example, it is not possible to represent constraints that range over different slot values, or to express the transitivity of a relation. These axioms have to be expressed with other methods. One approach is to use the Protégé Axiom Language[5] (PAL) to express the additional axioms. PAL is implemented as a variant of KIF [Genesereth and Fikes, 1992]

[5] http://protege.stanford.edu/plugins/paltabs

and can be used for constraint checking of the knowledge base. It is also possible to use a more powerful logical engine, such as $\mathcal{F}$LORA-*2*. $\mathcal{F}$LORA-*2* is a dialect of Frame-logic [Kifer et al., 1995] (F-logic) that is implemented on top of XSB Prolog[6] – a Logic Programming and Deductive Database system. By extending a Protégé knowledge base with additional axioms, that can be interpreted by a logical reasoner, we obtain a powerful framework for building intelligent applications.

Axiomatization of the Protégé Knowledge Model

In this section, I will give a partial axiomatization of the knowledge model constructs needed to define the semantics of the terms in the ontologies defined in the next sections. The OKBC specification [Chaudhri et al., 1998] already gives a starting point for defining the semantics of the knowledge model.

I will define the semantics of the terms using predicates and axioms in first order logic. The primitive predicates and functions are given in the following together with their meaning:

Cls(cls) – *cls* is a class

Slot(slot) – *slot* is a slot

Domain(slot) – a set of classes representing the domain of *slot*

Range(slot) – a set of classes representing the range of *slot*

RangeAtCls(cls, slot) – a set of classes representing the range of *slot* at class *cls*

instanceOf(inst,cls) – *inst* is an instance of class *cls*

hasTemplateSlot(cls,slot) – *slot* is attached to class *cls* as template slot

hasOwnSlotValue(inst,slot,value) – the instance *inst* has the value *value* for the own slot *slot*

A class is a subclass of another class if all instances of the subclass are also instances of the superclass. For example, all *ElectricalCar*s are also *Car*s, if *ElectricalCar* is defined as a subclass of *Car*.

$$\begin{aligned} subclsOf(subcls,cls) \Leftrightarrow\ & Cls(subcls)\ \wedge\ Cls(cls)\ \wedge \\ & \forall inst\ (instanceOf(inst,subcls) \Rightarrow instanceOf(inst,cls)) \end{aligned} \tag{4.1}$$

[6] http://xsb.sourceforge.net/

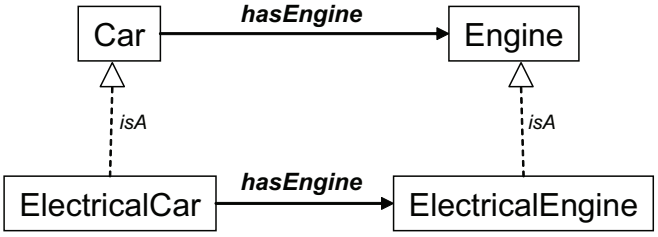

Figure 4.2.: The slot *hasEngine* attached to class *Car* is customized at the level of *ElectricalCar* subclass. The range of the *hasEngine* slot at *ElectricalCar* is restricted to *ElectricalEngine*.

Attaching a top-level slot to a class is equivalent to adding the class to the domain of the slot.

$$\begin{aligned} hasTemplateSlot(cls,slot) \Leftrightarrow Cls(cls) \ \wedge \ Slot(slot) \ \wedge \\ cls \in Domain(slot) \end{aligned} \tag{4.2}$$

By attaching a top-level slot to a class, the range of the slot, and other slot properties as well, may be customized at class level. However, there is a restriction that the properties of the slot at class must be a refinement of the corresponding slot properties at top-level. The following axiom captures this restriction for the range of a slot.

$$\begin{aligned} hasTemplateSlot(cls,slot) \Rightarrow Cls(cls) \ \wedge \ Slot(slot) \wedge \\ RangeAtCls(cls,slot) \subseteq Range(slot) \end{aligned} \tag{4.3}$$

The template slots attached to classes are inherited to subclasses, in which the ranges (and also other slot properties) of the template slots may be further refined. An example is given in Figure 4.2. The range of *hasEngine* slot defined at class *Car* is restricted at the level of class *ElectricalCar* to *ElectricalEngine*. This means that instances of electrical cars may only have as values for the *hasEngine* slot instances of electrical engines.

$$\begin{aligned} subclsOf(subcls,cls) \ \wedge \ hasTemplateSlot(cls,slot) \Rightarrow \\ RangeAtCls(subcls,slot) \subseteq RangeAtCls(cls,slot) \end{aligned} \tag{4.4}$$

A subslot is a specialization of a slot, in the same way in which a subclass is a specialization of a class. For example, *hasEngine* is a subslot of *hasPart*. In this way, we can build hierarchies of slots that help us represent the relationships in the domain more precisely. The subslots inherit all properties of a slot (like, domain, range, cardinality and other constraints), that can

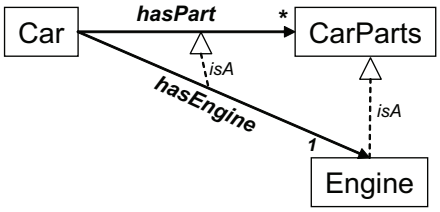

Figure 4.3.: *hasEngine* is a subslot of *hasPart*. It constrains the range to *Engine* and the cardinality to one.

be further restricted in the subslot. However, the slot constraints cannot be relaxed at the level of the subslot.

Figure 4.3 shows an example of a subslot *hasEngine* of slot *hasPart*. *hasEngine* inherits the domain (*Car*), the range (*CarParts*) and the cardinality (multiple) from the superslot *hasPart*. However, these properties may be further constrained at the level of the subslot. For example, the range of *hasEngine* has been restricted to only objects of type *Engine*. If we want to express the fact that a car has exactly one engine, we can restrict the cardinality of the *hasEngine* subslot to one.

Subslots are refinements of the superslots. The same restriction as for the inheritance of template slots to subclasses applies also for subslots, both at top-level and at class level. Below, is the axiom that ensures that subslots of a slot are refinements of the parent slot, meaning that both the domain and range of a subslot must be refinements of the ones of the parent slot.

$$\begin{aligned} subslotOf(subslot,slot) \Leftrightarrow\ & Slot(subslot)\ \wedge\ Slot(slot)\ \wedge \\ & Domain(subslot) \subseteq Domain(slot)\ \wedge \\ & Range(subslot) \subseteq Range(slot) \end{aligned} \tag{4.5}$$

According to the semantics of subslots defined by [Chaudhri et al., 1998], each value of a subslot is also a value of the slot.

$$\begin{aligned} subslotOf(subslot,slot)\ \wedge\ & hasOwnSlotValue(inst,subslot,value) \Rightarrow \\ & hasOwnSlotValue(inst,slot,value) \end{aligned} \tag{4.6}$$

In the previous example, if we assert that an instance of a *Car*, say *Car1 hasEngine Engine1*, the *Engine1* is also added as a value to the slot *hasPart* for the instance *Car1*.

4.3. Modeling Systems

This section describes the modeling of engineering system. A system is defined by its components and the connection between the components. Components are represented in the *Components* ontology which is described in Section 4.3.1. In order to define the part-of decomposition of a system, a modeling pattern has been developed, which is described in Section 4.3.2. Connections between system components are modeled in the *Connections* ontology which is described in Section 4.3.3.

4.3.1. Components

The *Components* ontology represents the components and their part-whole decomposition. A component is a fundamental concept that may be used to represent objects from different domains. The ontology contains abstract concepts and is intended to be very general in nature, so that it can be specialized for different domains. In this work, the *Components* ontology has been used to represent engineering systems and their decompositions. However, components do not necessarily represent physical systems. They may also represent abstract things, such as software modules, functional components, or requirements.

A component is a *StructuralThing* that is a general concept that encompasses all concepts used to describe the structure of something. A component may be atomic, meaning that it does not have any parts. In this case, I will denote it as *AtomicComponent*. If a component is composed of other parts, it is a *CompositeComponent*. A *CompositeComponent* may contain other *CompositeComponent*-s or *AtomicComponent*-s. The taxonomy of *Component* is shown in Figure 4.4.

In the example, given in Section 4.1.2, the *Powertrain* is a system that contains two other components, *Engine* and *Transmission*, which may themselves be systems.

In order to give a clear semantics to the concepts defined in this ontology, I will use axioms in first order logic. A class is a set of instances and it may be represented as a unary predicate. A slot is a binary relationship between two objects and can be represented as a binary predicate. For instance, the fact that x is an instance of a class *Component* may be represented as *Component(x)*. And the fact that x has a part y may be represented with a binary predicate *hasPart(x,y)*. In the following, I will consider that all variables in a first order logic axiom are universally qualified, if no quantifier is used.

A *Component* is a *StructuralThing*, formally:

$$Component(x) \Rightarrow StructuralThing(x) \tag{4.7}$$

The *hasPart* predicate has as a domain and range *Component*.

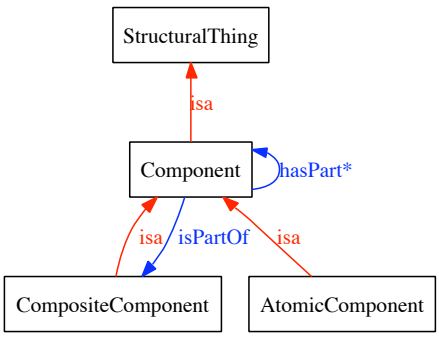

Figure 4.4.: The components taxonomy. A component may be composite or atomic. Composite components may contain other components, whereas atomic components may not.

$$hasPart(x,y) \Rightarrow Component(x) \ \wedge \ Component(y) \tag{4.8}$$

hasPart is used to represent the direct parts of a component. For this reason, it is not transitive. However, another transitive predicate *hasSubpart(x,y)* may be used to compute the transitive closure of the parts of a component.

Other properties of the *hasPart* predicate is that it is irreflexive and antisymmetric.

$$\neg hasPart(x,x) \tag{4.9}$$

$$hasPart(x,y) \Rightarrow \neg hasPart(y,x) \tag{4.10}$$

The axioms of the *hasSubpart* predicate are the following:

$$hasPart(x,y) \Rightarrow hasSubpart(x,y); \tag{4.11}$$

$$hasSubpart(x,y) \ \wedge \ hasSubpart(y,z) \Rightarrow hasSubpart(x,z) \tag{4.12}$$

An *AtomicComponent* is a component that is not further decomposed, meaning it cannot contain other components.

$$\begin{aligned} AtomicComponent(x) \Leftrightarrow Component(x) \ \wedge \\ \neg\exists y \, (Component(y) \ \wedge \ hasPart(x,y)) \end{aligned} \tag{4.13}$$

A *CompositeComponent* is a *Component* that has at least one part, i.e., it is not atomic.

$$\begin{aligned} \mathit{CompositeComponent}(x) \Leftrightarrow \mathit{Component}(x) \wedge \\ \exists y \left(\mathit{Component}(y) \wedge \mathit{hasPart}(x,y)\right) \end{aligned} \tag{4.14}$$

In many cases it is desirable to represent also inverse relationship of a relationship. Inverse relationships are important in the modeling process, for example, they support an easier navigation along relationships (going back and forth); they may be used in consistency checking of the knowledge base, or even in making certain kind of inference more efficient.

The inverse relationship for *hasPart* is *isPartOf*. This means that if *Car1* (instance of *Car*) *hasPart Engine1* (instance of *Engine*), then it may be inferred that *Engine1 isPartOf Car1*.

$$\mathit{isPartOf}(x,y) \Leftrightarrow \mathit{hasPart}(y,x) \tag{4.15}$$

There are many cases, in which more information about the part relationship needs to be expressed. An example is the case when we want to restrict the cardinality of a certain part of a composite component, such as specifying that a car must have exactly four wheels. The general *hasPart* relationship does not allow us to represent cardinalities or other constraints for specific classes from its range. We need a more precise representation of the relationships to capture this restriction.

This can be realized in the Protégé knowledge model with subslots. A subslot *hasWheel* of *hasPart* (with domain *Car* and range *Wheel*) can be restricted to have cardinality four at class *Car*. This ensures that an instance of a car must have for the *hasWheel* slot four instances of *Wheel* as values (which will become also values of *hasPart*). Subslots are a mean to represent qualified cardinality restrictions (QCR) in the ontology.

Subslots are also very useful, if we need to reference a specific part of a component. In the example given in Section 4.2.3, it is very easy to reference the engine that is part of a certain instance of a car: It is the value of the *hasEngine* subslot of the car instance. In a simple notation, the engine of a car, can be addressed as: *Car.hasEngine*. If the *hasEngine* slot wouldn't have been defined, and all the parts of the car would have been values of the general *hasPart* slot, then finding out which is the engine instance of a car would have been more difficult. Subslots can be used whenever addressing or referencing a certain part is important. This applies not only for instance level, but also for classes. At class level, a part is represented as a template slot attached to the class. It is possible to reference at class level a certain role by using a subslot (*hasEngine* is a role that an *Engine* component plays in a *Powertain* class).

Subslots of *hasPart* provide a way to model more precise part-of relationships. There are, however, some shortcomings when using subslots, that must be considered and dealt with in the modeling. The semantics of subslots does not prevent us from adding another instance of *Engine* directly to the *hasPart* slot. In this way, we may end up with a *Car* that has two

engines, although our intention, when introducing the *hasEngine* subslot, was to model the fact that a car may have only one engine. In order to capture the desired semantics, we need to add more axioms to the ontology that will rule out incorrect models.

I have mentioned previously that we may use subslots of *hasPart* in order to define more constraints on the relationship between a component and its parts. However, the way subslots are defined in the Protégé knowledge model and also by the previous axioms does not prevent us from representing unintended models (as in the case I have described before, in which we may end up with a car with two engines, although only one was allowed). In order to deal with such situations, we need a modeling pattern that is in a way similar to the value partitions modeling pattern defined for description logic ontologies by [Rector, 2004; Rector et al., 2004].

4.3.2. Part-Whole Modeling Pattern

I will use the example in Figure 4.5 throughout this section to explain the part-whole modeling pattern. We want to represent the fact that a car has an engine, a transmission, and other parts for which we want to impose specific constraints (like, for example, that it has exactly four wheels) and also other parts that are not constrained in anyway. The range of the *hasPart* slot is *Component*. The subslots, *hasEngine*, *hasTransmission* and *otherSubslot* have as ranges partitions of the superslot domain (not necessarily disjoint partitions). The unconstrained part of the range is covered by direct values of the *hasPart* slot. There are several reasons, why we may want to use unconstrained values of slots. One of them is that the model is not yet completely specified. This may happen in a library of models, where abstract or underspecified models are important: They may be further specified in subclasses of the model. Another reason when subslots are not needed, is the case when it is not important to make a differentiation between the values of the slot. For example, the engineer may not care about specific properties of wheels of the vehicle, and instead of naming each wheel in a separate subslot, it will use a relationship *hasWheel* that has as values all wheels of a car. However, there are also cases in which using subslots makes sense. Subslots describe actually the role that a component plays in the context of the class where it is attached. SysML denotes this as the *usage* of a component in another component. When building models with a fairly small number of elements (for example, a simulation model as opposed to a part list of a car), it is feasible to describe the components of the model and their usage at a very detailed level. For example, the interconnection of components in a containing component plays an important role for simulation. It is crucial to be able to name and reference the subcomponents in an unambiguous way. This can be realized by using subslots.

This modeling pattern may be applied for the cases in which:

1. A slot has a range over a set of classes.

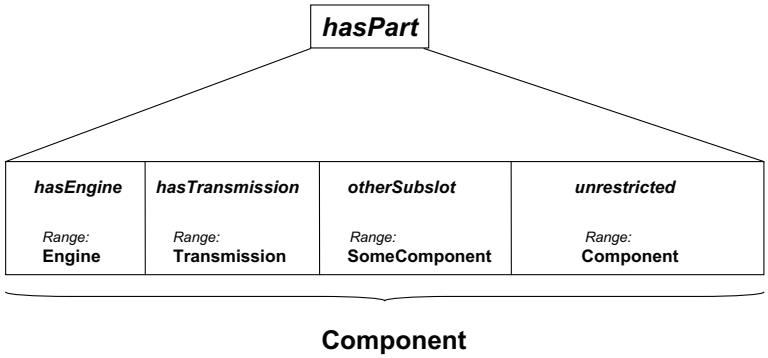

Figure 4.5.: The slot *hasPart* has as range the class *Component*. The subslots *hasEngine*, *hasTransmission* and *otherSubslot* have as ranges subsets of the class *Component*. The unrestricted portion of the range is taken by direct values of the slot *hasPart*.

2. It is important to add extra constraints for some partition of the range.

3. It is important to be able to address or reference a certain value or set of values of the slot.

Consider that a slot S has a definition: *Slot(S)* with the domain and range *Domain(S)* and *Range(S)*. The modeling pattern assigns a subslot for each portion of a range that fulfills items 2. and 3. The set of subslot is $\{S_1, S_2 \ldots S_n\}$. Each subslot S_i, has a range $Range_i$ and a domain $Domain_i$, where $i = \{1,2 \ldots n\}$. The ranges and domains of the subslots are subsets of the range and domain of the superslot, according to their definition.

$$\begin{aligned} Domain(S_i) &\subseteq Domain(S) \\ Range(S_i) &\subseteq Range(S) \qquad \forall i = \{1,2 \ldots n\} \end{aligned} \tag{4.16}$$

Both the domain and the ranges of the subslots are constrained. For instance, suppose that a class *Car* has a slot *hasPart* with the top-level domain *Component* and range *Component* attached to it. If we want to specify that the car has exactly one engine, we will create a subslot of *hasPart*, called *hasEngine* and we will set its domain to *Car* and its range to *Engine*.

This design pattern has been applied in the modeling of the *hasPart* relationship in the *Components* ontology. For some cases, in which it was clear that the models where complete, I have introduced additional constraints: 1. All the parts of the components were specified by a subslot. And as a consequence: 2. It is not allowed to add a direct value to the *hasPart* slot. All the values of the *hasPart* slot are the values of its subslots. This additional constraint is enforced by an additional axiom:

$$\begin{aligned} \textit{hasValue(inst,slot,value)} \Rightarrow \exists \textit{subslot}\ (&\textit{subslotOf(subslot,slot)} \wedge \\ &\textit{hasOwnSlotValue(inst,subslot,value)}) \end{aligned} \tag{4.17}$$

Another recommended modeling practice is to restrict the range of the *hasPart* slot for each class, where it is attached, to the union of the ranges of the subslots at that class. For example, if a class *Powertrain* has only two parts, engine and transmission, that are the ranges of two subslots, then the range of the *hasPart* slot at class *Powertrain* should be restricted to the set $\textit{Engine} \cup \textit{Transmission}$.

If S is a slot attached to class C and $\{S_1, S_2 \ldots S_n\}$ are subslots of S also attached to C, then the range of the slot S at C is the union of the ranges of the subslots at class C.

$$\textit{RangeAtCls(C,S)} = \bigcup_{i=1}^{n} \textit{RangeAtCls(C,}S_i\textit{)} \tag{4.18}$$

Example The example that was shown at the beginning of the section in Figure 4.1 can be modeled by extending the *Components* ontology. The class hierarchy and the relationship between the classes is shown in Figure 4.6. The *StructuralThing*, *Component* and *CompositeComponent* are part of the *Components* ontology. The class *System* has been introduced as a subclass of *CompositeComponents* to represent the notion of a system as engineers understand it. The *Powertrain* class inherits the *hasPart* slot from the class *Component*. Two subslots *engine* and *transmission* of the slot *hasPart* have been introduced with the domain *Powertrain* and the ranges *Engine* and *Transmission* respectively. The other two subslots *p1* and *p2* represent also parts of the *Powertrain* and have the range *Flange*. The way components can be connected to each other will be discussed in the following section. The domain of *hasPart* slot has been constrained to the union of *Engine*, *Transmission* and *Flange*.

Figure 4.7 shows an instantiation of the *Powertrain class*. It can be seen in the figure that the value of the subslots are also values of the superslot, i.e., *hasPart*. If an application would query the ontology about the parts of *Powertrain_1*, the ontology would return {*Engine_1*, *Transmission_1*, *Flange_1*,... *Flange_6*}. In Figure 4.7 only the direct parts of the components are shown.

4.3.3. Connections

A system is not only defined by the parts it contains, but also by its behavior. The behavior of a system is determined by the interactions of its parts. The interactions are abstracted as connections between the components of a system and form the topology of the system. In engineering systems, the connections between parts represent usually flows of stuff (energy,

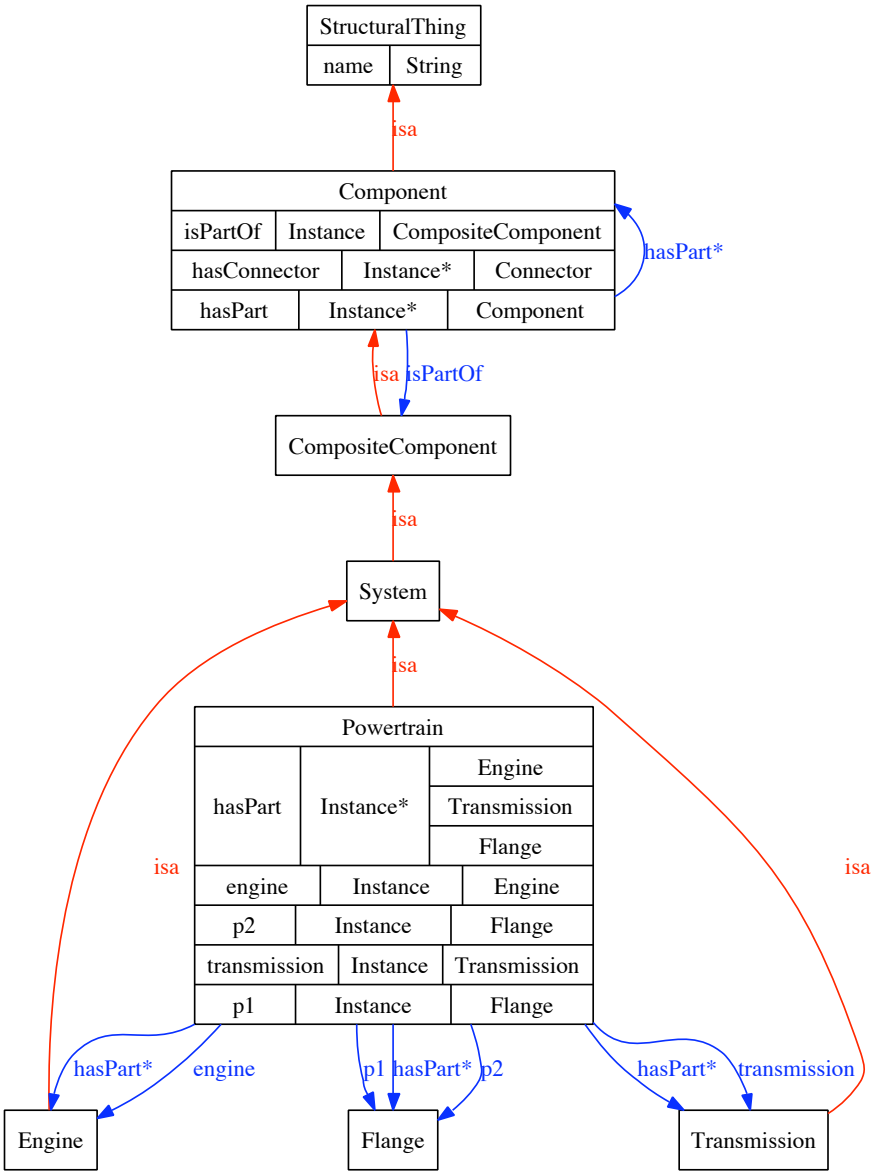

Figure 4.6.: The class hierarchy of components and their relationships. A box represents a class which contains the name of the class on the first row and the slots of the class represented with their value type (such as, Instance) and their range. For example, the *transmission* slot at class *Powertrain* has the range *Transmission*.

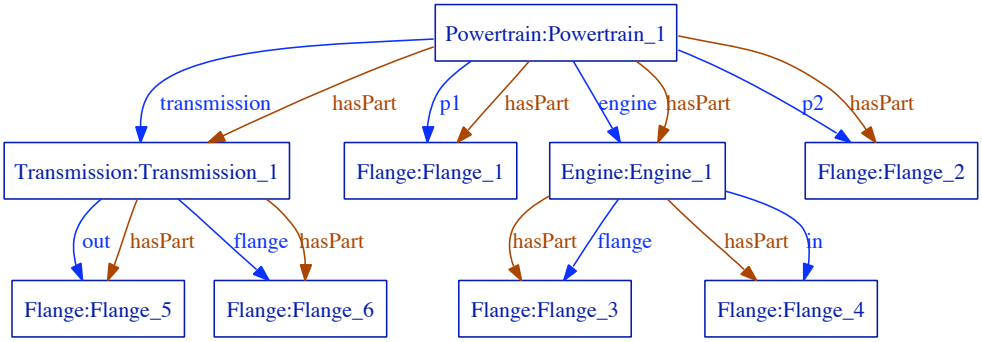

Figure 4.7.: Example instantiation of *Powertrain* class. The boxes represent instances. The name of the instances have as prefix the name of the classes. The links between the instances represent relationships between the instances. They are actually values of the slots.

matter or signals) between the components of the system [Pahl and Beitz, 1996]. The Modelica language [Modelica Specification, 2005; Modelica Tutorial, 2000] extends the connection concept with a new type of connection for *across variables* with the meaning that two across variables that are connected, are equal. SysML [SysML, 2006] defines two types of ports through which components may communicate: *Flow Ports* – that describe the flow of stuff – and *Standard Ports* – which are used to invoke services on components in a similar manner as in service oriented architectures.

The components own their ports or connectors and therefore the ports are part of the specification of the component. Connectors play a very important role in the design of systems, since they enable a modular and reusable design.

There are some challenges when representing the connection between components. First, we are dealing with two levels of abstraction for models: the class level, which describes a class of models, and the instance level, that describes a particular (instantiated) model. We will have to represent the connection information at class level and then apply it to the instance level in a consistent manner (i.e., the connection instantiation should conform to the connection definition at the class level). This will be discussed in next sections.

Second, the representation of connections should support different types of connections. One classification of connectors is in flow, across and services ports. The classification of connections is made according to the type of information that they transmit. We can distinguish between mechanical, electrical or hydraulic connections.

Third, there are specific rules on the compatibility between the connectors, that influence the validity of the connection. Only ports of compatible types may be connected together. For instance, connecting a fuel flow port with an electrical port should be prohibited in a valid model.

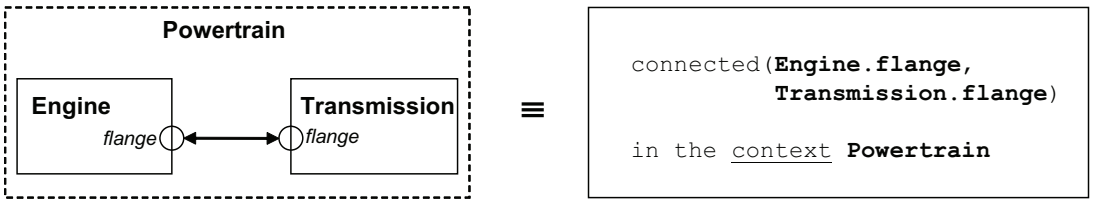

Figure 4.8.: A simple connection between the engine and transmission defined in the context of the powertrain (containing component). The flange port of the engine is connected to the flange port of the transmission.

Fourth, connections always appear in a context, which is defined by the component that contains the parts which are connected. In Figure 4.8, the connection between the engine and transmission is made in the context of the powertrain, which is the system that contains them.

I will use in the following, the simple example in Figure 4.8 to describe the challenges in the representation of connections and a modeling pattern for them.

The *Connections* Ontology

The *Connections* ontology describes the topology of a system, that is, the way in which components are interconnected with each other. Although it would have been possible to develop the *Connections* ontology independently from the *Components* ontology, I have chosen to include the *Components* ontology in the *Connections* ontology, because the part-whole modeling pattern is essential in the representation of the context of a connection. The main concepts in the ontology are the connectors that are the only points through which components can be interconnected, and the connections which are represented with the help of a reified concept for describing the actual connection between components (see Section 4.3.3). The topological individual which represents anything that can be connected is represented in the ontology by the concept *TopologicalIndividual*.

The ontology contains a general class *TopologicalThing*, that is a superclass of all the classes which are used to describe the topology of a system. A *TopologicalIndividual* is a subclass of *TopologicalThing* and is the superclass of the classes used for modeling topological entities and relationships. The *TopologicalIndividual* is an abstract class (i.e., it cannot have any direct instances), and it is intended to be refined in ontologies that include the *Connections* ontology. For example, the *Engine* class is also a *TopologicalIndividual* (modeled as subclass), since it may have connections to other systems or parts. A graphical representation of the *Connections* ontology is shown in Figure 4.9.

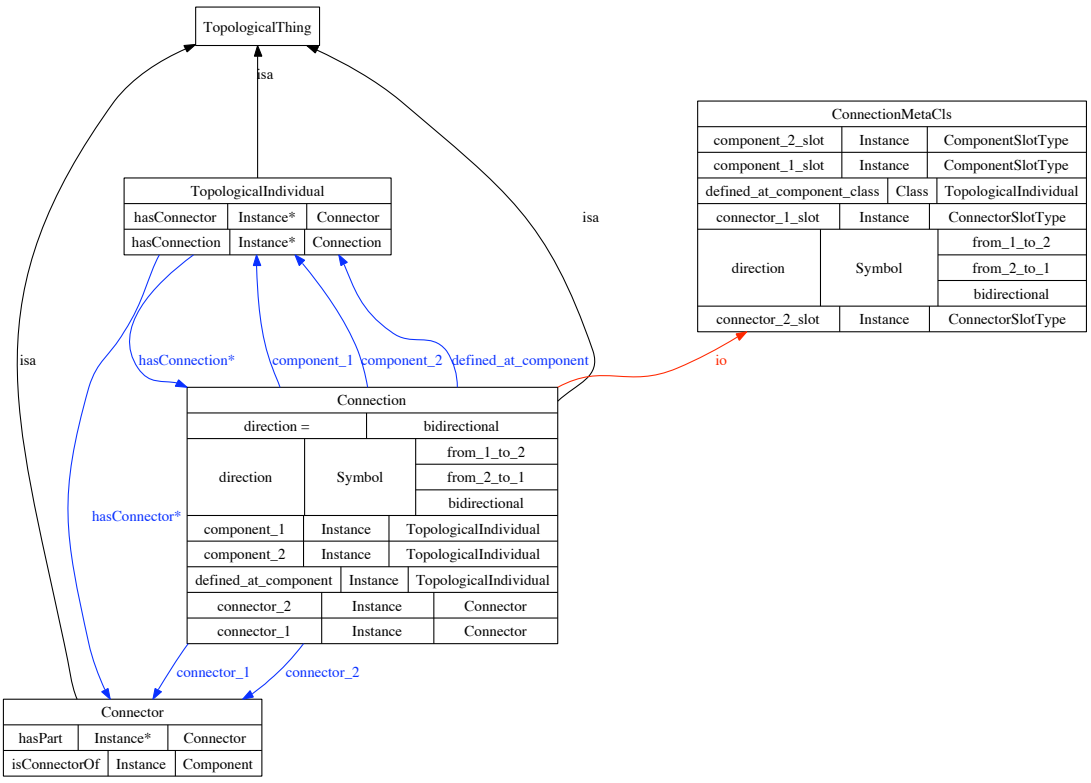

Figure 4.9.: The *Connections* Ontology.

The *Systems* Ontology

The *Systems* ontology gives a formal representation of systems and combines concepts from the *Components* and the *Connections* ontologies, shown in Figure 4.10.

Systems and their parts are topological individuals since they can have connections. Therefore,

$$System(x) \Rightarrow TopologicalIndividual(x) \tag{4.19}$$

$$AtomicPart(x) \Rightarrow TopologicalIndividual(x) \tag{4.20}$$

This is modeled in the ontology with the subclass relationship,i.e., *System* and *AtomicPart* are subclasses of *TopogicalIndividual*.

The connection between the *Systems* and the *Components* ontologies are given by the fact

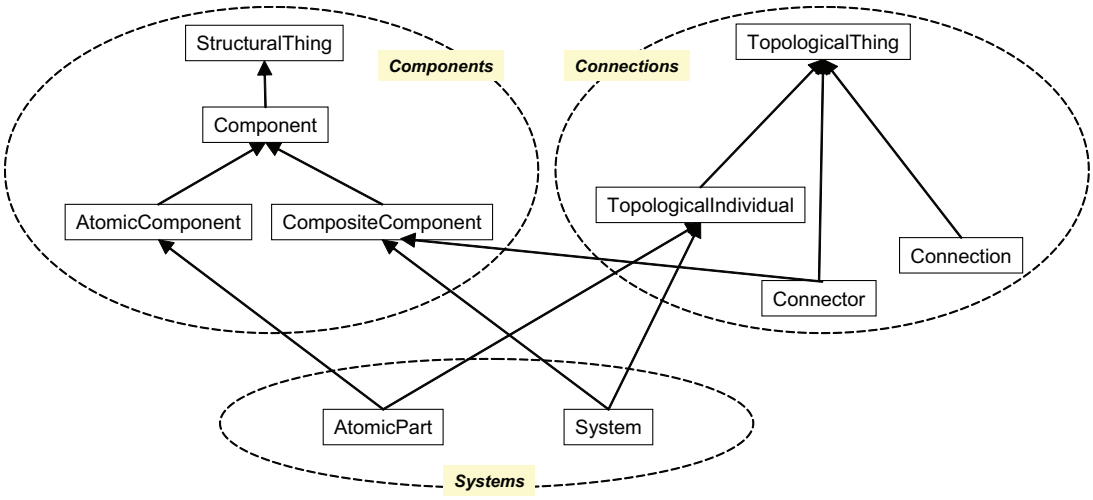

Figure 4.10.: The relationship between the *Components*, *Connections* and *Systems* ontologies.

that systems are composite components, while atomic parts are atomic components. This is formally represented as:

$$System(x) \Rightarrow CompositeComponent(x) \tag{4.21}$$

$$AtomicPart(x) \Rightarrow AtomicComponent(x) \tag{4.22}$$

By combining the *Components* and the *Connections* ontology, systems are defined as composite objects that can have connections to other components, while atomic parts are atomic components that may also be connected to other components.

Connectors

Connectors represent points of interaction between a component and the environment. Connectors are also known as ports in UML2 and SysML. Connectors are owned by the component and are part of the definition of a component. A component can be seen as a black box that exposes a certain behavior and communicates with the outer world (its environment) through the connectors. This type of modeling supports the reuse and modularization of system models: A component can be replaced with another one if it has the same interfaces (types of connectors) and has the same behavior. The internal implementation of the component is of no import to its environment. The way a component communicates with the environment is only through the connectors.

Connectors are composite components: They can themselves contain other connectors. In this way it is possible to represent more complex connectors, as for example, a serial port that contains several pins. This is realized by making the *Connector* a subclass of the *CompositeComponent* class from the *Components* ontology.

$$Connector(x) \Rightarrow CompositeComponent(x) \tag{4.23}$$

Connectors are attached to a *TopologicalIndividual* through the *hasConnector* slot or one of its subslots. The same modeling pattern like the one used for modeling parts (described in Section 4.3.2) may be used for modeling the attachment of connectors to components. This will bring several advantages: It will be possible to address or reference a certain connector of a component, and also to impose additional constraints on the connector (for example, refine the allowed types of connected components). The domain of the *hasConnector* slot is the *TopologicalIndividual* and the range is *Connector*.

$$\begin{aligned} hasConnector(ind,connector) \Rightarrow & TopologicalIndividual(ind) \wedge \\ & Connector(connector) \end{aligned} \tag{4.24}$$

The connectors of components are themselves parts of the components. Hence, the *hasConnector* – that is used to attach a connector to a component – is modeled as a subslot of the *hasPart* slot.

Connectors might be organized in hierarchies of connectors by means of the *is-a* relationship to describe different classification of the connectors. For the engineering domain, the connectors can be classified in mechanical, electrical/electronical and hydraulic connectors. This can be modeled in the ontology by subclassing the *Connector* class with more specific types of connectors.

Reifying Connections

A simple connect statement between two components can be represented by a binary predicate *connected(x,y)*. However, if we want to describe how two components are connected – for example, via which connectors or ports –, then we need to reify the predicate into a connection object. Reification is defined as making an abstract thing concrete, or in the context of first order logic – as turning a proposition into an object [Russell and Norvig, 2003; Sowa, 1999]. By reifying a relationship, it becomes an object that can be referenced and to which we can add properties. For example, a reified connection object may contain a property that describes the rationale of a connection or other documentation related to the connection. Reification is also a mean for representing n-ary relationships as described by [Noy and Rector, 2006; Dahchour and Pirotte, 2002].

For the purpose of this work, a connection is a reification of a connect statement between two components. A connection is defined by the two components that it connects and a port for each component through which the connection is made. Theoretically, this should not pose any challenges to the representation. The problem comes from the fact that we need to represent both the connection among classes of models (at the class level) and the connection among instantiation of these classes of models (a "real" model at the instance level). For example, the *Powertrain* class may be a general class used for representing powertrains. A powertrain contains typically (among other things) an engine and a transmission that are connected together: The engine produces a torque that is transferred and used by the transmission of the car to produce the motion of the wheels. The torque is the rotational force and is abstracted and represented as a *Flange* object. Two classes *Engine* and *Transmission* are used to represent the classes of the engine and transmission models. They both have a *flange* connector attached to them (as a template slot of type *Instance* and allowed classes *Flange*). Our goal is to represent that a powertrain model (i.e., at the class level) has as parts an engine and a transmission, which are connected together through the corresponding flange connectors. At the instance level, we want to enforce the constraint, that in a specific powertrain, represented by the instance *Powertrain_1* of class *Powertrain*, its parts, *Engine_1* – instance of *Engine*, and *Transmission_1* – instance of *Transmission*, may be connected together only by their *flange* connectors.

Connections at the class level A connection between two parts in a system has the form "*Part1.Connector1 is connected to Part2.Connector2*", and is represented as a reified relationship modeled as an instance of a metaclass *ConnectionMetaCls* in the *Connections* ontology. The definition and slots of the *ConnectionMetaCls* are shown in Figure 4.9. The connection relationships at class level are actually relationships between the roles that certain components play in a system class. At class level, we represent classes of systems that serve as templates for instantiated systems. Since the parts of the systems are represented as roles that the components play in a system (for example, *engine*), the connection are also made between the roles of the components. The connection classes will serve also as templates for building the connections between the corresponding component instances (see Section 4.3.3).

The metaclass *ConnectionMetaCls* has four template slots, *component_1_slot*, *component_2_slot*, *connector_1_slot* and *connector_2_slot* that hold as values at the instances of the metaclass (i.e., at class level) the roles that are connected in a connection. Figure 4.11 shows how the connection between an engine and a transmission in a powertrain can be modeled with the *Components* and *Connections* ontologies. The *Powertrain* class has two template slots, *engine* and *transmission* that represent the roles that the *Engine* and *Transmission* are playing in the containing system. In order to represent the fact that the roles *engine* and *transmission* are connected by their flange connector, an instance of the *ConnectionMetaCls*, called *EngineTransmissionConnection*, is created that has as own slot values for the template slots in the metaclass, the following values:

```
component_1_slot           -> Slot(engine)
component_2_slot           -> Slot(transmission)
connector_1_slot           -> Slot(flange)
connector_2_slot           -> Slot(flange)
defined_at_component_class -> Cls(Powertrain)
direction                  -> bidirectional
```

Some information that is not shown in Figure 4.11 is that the connection class also holds an own slot, *defined_at_component_class*, that specifies to which class the connection belongs to, that is, it defines the context of the connection. In the previous example, the connection is defined at class *Powertrain*. Another own slot, *direction*, defines the direction of the connection. The *direction* slot is defined as a symbol slot with the allowed values: *from_1_to_2*, *from_2_to_1* and *bidirectional*.

In case that the connection is done between a port of the system with a port of a part of the system (as in the case of connecting the *p1* port of the powertrain to the *in* port of the engine in Figure 4.1), the own slot corresponding to the system component is left empty (in the powertrain-engine connection, the *component_1_slot* does not have a value).

A system is defined by its parts and the connections that exists between its parts. The parts of a system are specified using the role template slots. A connection in a system is modeled at the class level as an allowed class (i.e., the range) of the template slot *hasConnection* defined at the *CompositeComponent* class. The way this is used for enforcing correct connections at instance level is described in next section.

Connections at the instance level The connections at the instance level should conform to the definition of the connections at the class level. For example, if at the class level, we have defined that an engine is connected to a transmission only by some port, then at instance level, we should not be allowed to create another type of connection between instances of the engine and transmission by a port, than the ones already defined at class level. In order to enforce this consistency constraint, we need a way to translate the connections between roles defined at the class level to templates of allowed connections which will be applied for instances. At the instance level, we can actually fill in the component and connectors roles involved in a connection with instances of component and connectors.

The connections at the class level are modeled using two pairs of the form *Part.Connector*, that describe a path in the part-whole hierarchy. For example, in the powertrain, *engine.flange* denotes the flange of the engine of the powertrain. These pairs are modeled as own slot values at the connection class as described in previous section. However, the own slot values are not inherited by the instances. In order to enforce the connection constraints at the instance level (for example, that the first component in a connection must be of type *Engine*), we need to create template slots with the constraints defined in the corresponding own slots. This is not

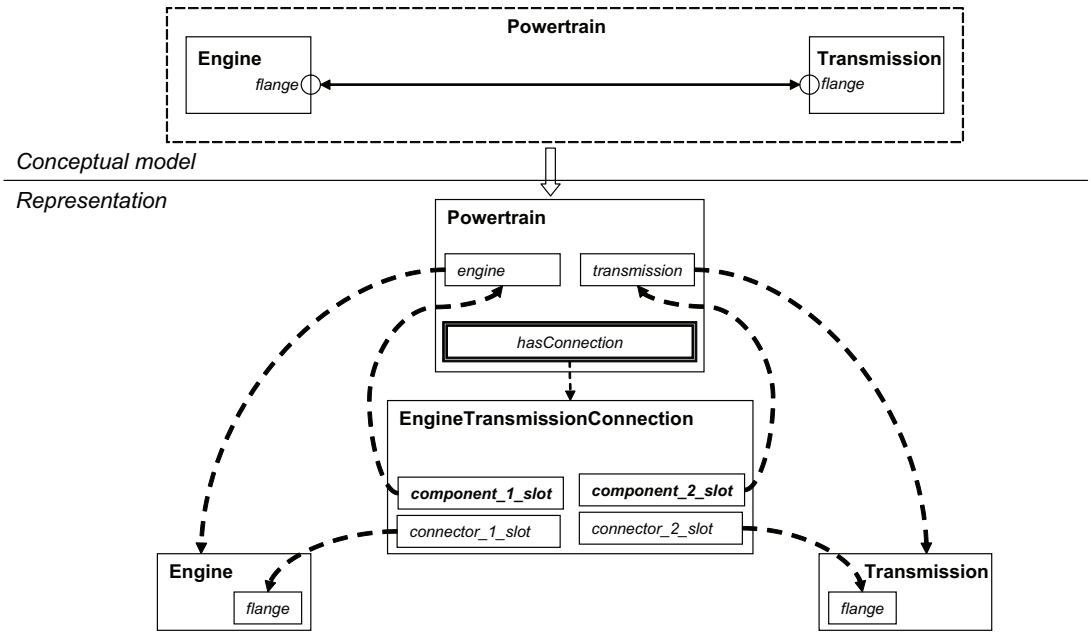

Figure 4.11.: Representation of the connection between the engine and transmission in a powertrain model with a reified relationship – *EngineTransmissionConnection*. The arrows from the connection class point to the values of the slots. The connection class represents the fact that the flange connector of the engine is connected to the flange connector of the transmission.

a typical way how instantiation of classes works in frame-based models, but this extension is needed in order to translate the constraints defined for connections at the class level to constraints for connections at the instance level.

The connection classes have already defined the template slots needed to represent connections for instances. They are: *component_1*, *connector_1*, *component_2* and *connector_2*. However, the ranges (i.e., the allowed classes) of these template slots are very general and need to be refined to the ranges of the corresponding own slots. The correspondence between the own slot and the template slots in a connection class is the following:

Own slot	***Template slot***
`component_1_slot`	`component_1`
`connector_1_slot`	`connector_1`
`component_2_slot`	`component_2`
`connector_2_slot`	`connector_2`

At instantiation time, the translation of the connection class constraints to connection instance constraints is done, by making the range of the template slot to be the range of the corre-

sponding own slot value. For example, in the engine-transmission connection, the following translation is done:

Own slot	*Value*
component_1_slot	engine
connector_1_slot	flange
component_2_slot	transmission
connector_2_slot	flange

Template slot	*Range*
component_1	Engine
connector_1	Flange
component_2	Transmission
connector_2	Flange

The constraints translation rules can be written using first order logic axioms. In the following axioms, the constants are written in double quotes (for example, "ConnectionMetaCls").

The *ConnectionMetaCls* is a class that has six template slots.

$$\begin{aligned}
&\textit{Cls}(\text{“ConnectionMetaCls”}) \wedge \\
&\textit{hasTemplateSlot}(\text{“ConnectionMetaCls”}, \text{“component_1_slot”}) \wedge \\
&\textit{hasTemplateSlot}(\text{“ConnectionMetaCls”}, \text{“connector_1_slot”}) \wedge \\
&\textit{hasTemplateSlot}(\text{“ConnectionMetaCls”}, \text{“component_2_slot”}) \wedge \\
&\textit{hasTemplateSlot}(\text{“ConnectionMetaCls”}, \text{“connector_2_slot”}) \wedge \\
&\textit{hasTemplateSlot}(\text{“ConnectionMetaCls”}, \text{“direction”}) \wedge \\
&\textit{hasTemplateSlot}(\text{“ConnectionMetaCls”}, \text{“defined_at_component_class”})
\end{aligned} \tag{4.25}$$

The *Connection* class is an instance of the *ConnectionMetaCls* and has also six template slots defined corresponding to the own slots.

$$\begin{aligned}
&\textit{Cls}(\text{“Connection”}) \wedge \\
&\textit{instanceOf}(\text{“Connection”},\text{“ConnectionMetaCls”}) \wedge \\
&\textit{hasTemplateSlot}(\text{“Connection”}, \text{“component_1”}) \wedge \\
&\textit{hasTemplateSlot}(\text{“Connection”}, \text{“connector_1”}) \wedge \\
&\textit{hasTemplateSlot}(\text{“Connection”}, \text{“component_2”}) \wedge \\
&\textit{hasTemplateSlot}(\text{“Connection”}, \text{“connector_2”}) \wedge \\
&\textit{hasTemplateSlot}(\text{“Connection”}, \text{“direction”}) \wedge \\
&\textit{hasTemplateSlot}(\text{“Connection”}, \text{“defined_at_component”})
\end{aligned} \tag{4.26}$$

The range of the *component_1* template slot has to be the same as the range of the value of the *component_1_slot* at the class that is a value of the *defined_at_component_class* own slot. The same applies for *component_2*, but the axioms is not repeated here.

$$
\begin{aligned}
&Cls(c) \wedge subclsOf(c,\text{“Connection”}) \wedge \\
&hasOwnSlotValue(c,\text{“is\_defined\_at\_component\_class”},systCls) \wedge \\
&hasOwnSlotValue(c,\text{“component\_1\_slot”},comp1Slot) \Rightarrow \\
&\qquad hasTemplateSlot(systCls,comp1Slot) \wedge \\
&\qquad RangeAtCls(c,\text{“component\_1”}) = RangeAtCls(systCls,comp1Slot)
\end{aligned} \tag{4.27}
$$

The range of the *connector_1* template slot is the same with the range of the connector of the *component_1* slot. The following axiom also holds for *connector_2*, but it is not repeated here.

$$
\begin{aligned}
&Cls(c) \wedge subclsOf(c,\text{“Connection”}) \wedge \\
&hasOwnSlotValue(c,\text{“is\_defined\_at\_component\_class”},systCls) \wedge \\
&hasOwnSlotValue(c,\text{“component\_1\_slot”},comp1Slot) \wedge \\
&hasOwnSlotValue(c,\text{“connector\_1\_slot”},conn1Slot) \wedge \\
&\forall comp1Cls \; (\\
&\quad comp1Cls \in RangeAtCls(systCls,comp1slot) \Rightarrow \\
&\qquad hasTemplateSlot(comp1Cls,\text{“connector\_1\_slot”}) \wedge \\
&\qquad RangeAtCls(c,\text{“connector\_1”}) = RangeAtCls(comp1Cls,conn1Slot) \\
&\quad)
\end{aligned} \tag{4.28}
$$

The information about where a connection is defined at class level is also preserved for the instance level. This means that the range of the *defined_at_component* slot has to be the same with the value of the *defined_at_component_class* own slot.

$$
\begin{aligned}
&Cls(c) \wedge subclsOf(c,\text{“Connection”}) \wedge \\
&hasOwnSlotValue(c,\text{“is\_defined\_at\_component\_class”},systCls) \Rightarrow \\
&\qquad RangeAtCls(c,\text{“defined\_at\_component”}) = systCls
\end{aligned} \tag{4.29}
$$

The connection class serves two purposes: It is a reification of a connect statement between the component classes by storing the information in the own slots, and it is also a reification of a connection relationship that serves as a template for connecting component instances, which is represented with the help of the template slots. The connection between instances is realized by making an instance of a connection class. This will only allow the “right” component instances to be connected together, since the ranges of the template slots at the connection class were already refined as described previously in this section. An example of the connection at class level and instance level is shown in Figure 4.12.

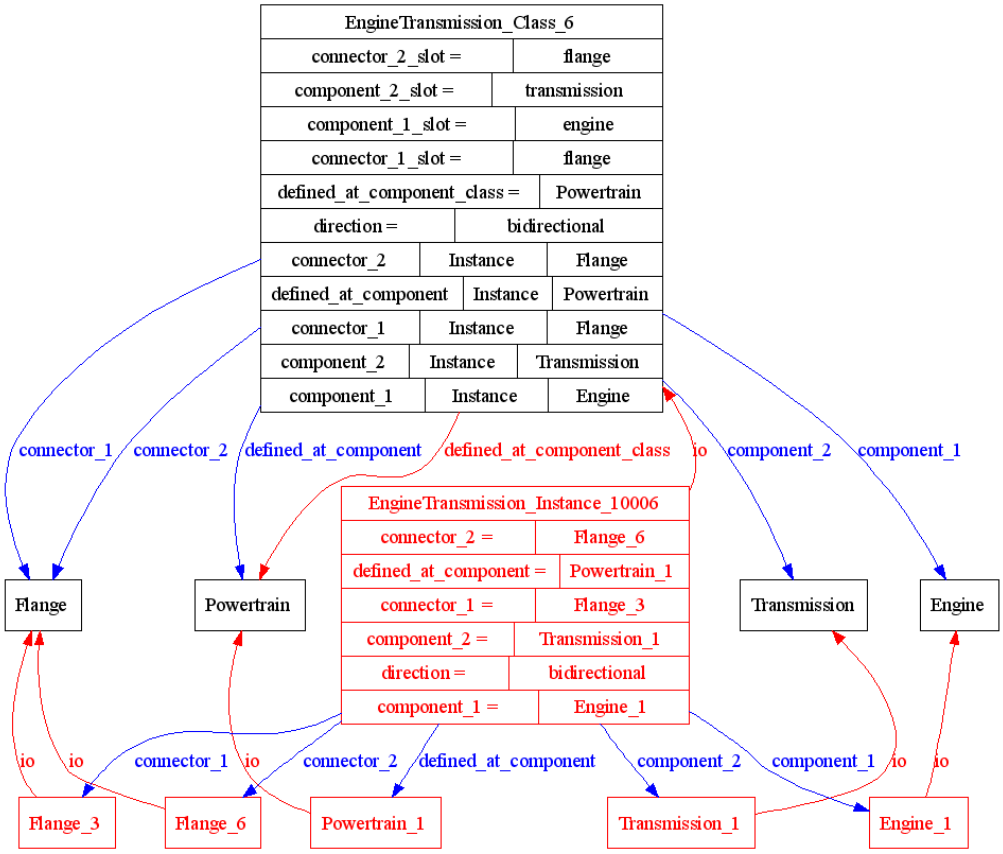

Figure 4.12.: The connection representation at class and instance level. The *io* arrow stands for "instance-of". Classes are shown in black color and instances in red color. The connection class, *EngineTransmission_Class_6* has own slots (that have the equal sign in them, like *component_1_slot*) and template slot that are shown with their type and range (for example, *component_1*). The *EngineTransmission_Instance_10006* is an instance of the connection class that is used to connect the component instances.

4.4. Modeling of Requirements

A requirement specifies a capability or condition that the system to be designed must satisfy [SysML, 2006]. Requirements are the driving force of the development process. They are acquired at the beginning of the development process and are used throughout the entire process in different development stages. Requirements are evolving from the initial phases, when they are very vague, ambiguous, informal to become more precise and formal at the later stages of the development.

Being able to trace the requirements evolution from the initial specification to the design is crucial for an optimal development process [Grabowski et al., 1996; Stevens et al., 1998; Kim et al., 1999; Lin et al., 1996]. Another important aspect is the attachment of requirements to systems and their parts. The documentation of requirements verification is also an essential aspect. A requirements modeling framework should support the attachment of test cases to systems and requirements. Certain types of requirements may be formalized and represented using mathematical expressions, which may be used to constrain different characteristics of the product. In these cases, the requirements may have mathematical constraints attached to them.

4.4.1. The *Requirements* Ontology

The *Requirements* ontology contains the main concepts needed for the representation of the requirements and of their properties. The *Requirements* ontology and the *Constraints* ontology are shown together in Figure 4.13.

The requirements are also composite objects, meaning that they may contain other requirements. In order to represent the part-whole relationship between the requirements, the ontology includes the *Components* ontology. The class *Requirement* is modeled as a subclass of the *CompositeComponent* class from the *Components* ontology. In this way it inherits the *hasPart* and *isPartOf* template slots, which are refined at class level to have as range the class *Requirement*. This means that a requirement may be composed of other requirements. For example, a top level requirement object, which might represent the customer specification, may be decomposed in requirements related to performance, ergonomics, safety, etc.

4.4.2. Classification of Requirements

There are different ways in which requirements can be classified. A first classification according to their origin is in initial and detailed requirements. The initial requirements are the ones stated by the customer. They are static in nature, and changing them requires a very big effort and involves renegotiating the contract. After the analysis process, the initial requirements are refined into more specific requirements, called detail requirements. They are dynamic in

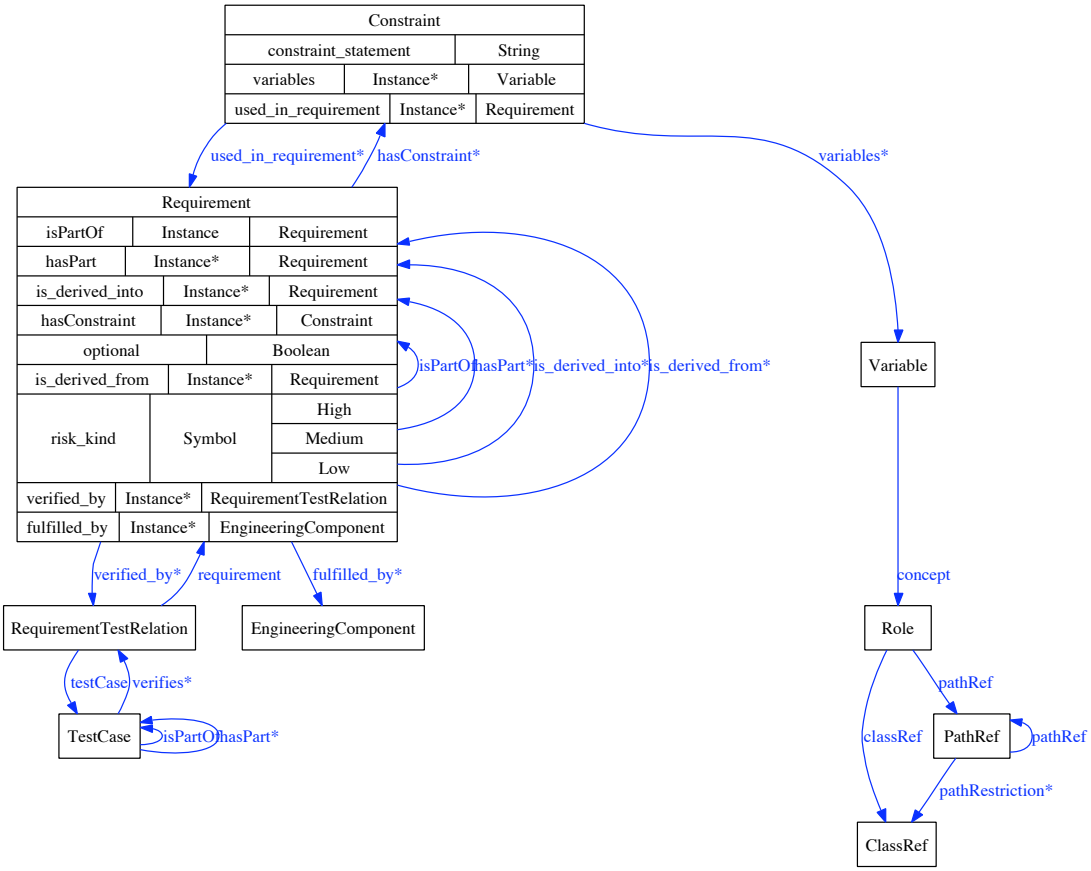

Figure 4.13.: The *Requirements* and *Constraints* ontologies.

nature, and may be often changed during the design process. The initial and detailed requirements are already available as classes in the ontology, because they are common to all design processes.

Another classification of requirements is according to the aspect of the product that they are constraining. They may be classified in cost, functional, safety, technological, ergonomical, etc. requirements. Different classification of requirements have been proposed by Pahl and Beitz [1996] and Rzehorz [1998]. These categories of requirements are modeled in the ontology as subclasses of the *Requirement* class. Multiple inheritance may be used if a requirement belongs to different categories.

Another important classification is in demands and wishes [Pahl and Beitz, 1996]. Some requirements must be fulfilled by the end product, while others are just “nice to have”, but not compulsory. This is modeled in the ontology with the help of a boolean template slot,

optional. If it is set to true, then the requirement is a demand, otherwise it is a wish.

4.4.3. Properties and Relationships of Requirements

The properties and relationships of requirements are shown graphically in Figure 4.13.

As I have mentioned in previous section, requirements may be composed of other requirements. This is modeled with the *hasPart* relationship, which has all the properties that have been defined in Section 4.3.1.

The semantics of the *hasPart* decomposition of a requirement is that the containing requirement is satisfied if the contained requirements are satisfied. This ensures that the meaning of the containing requirement is preserved by its decomposition.

I will use the unary predicate *Requirement(r)* with the meaning that r is a requirement. This is modeled in the ontology as an instance r of class *Requirement*. The decompositions of the requirement r is represented in the values of the slot *hasPart*, let's say $\{r_1, r_2 \ldots r_n\}$. The fulfillment of a requirement by a system s is represented by a predicate *fulfills(s,r)*. We can state that:

$$fulfills(s,r_1) \wedge fulfills(s,r_2) \wedge \ldots \wedge fulfills(s,r_n) \Rightarrow fulfills(s,r) \tag{4.30}$$

The *optional* property is used to represent whether the requirement must be fulfilled by the product, or if it is only optional, i.e., a wish requirement. Another property *risk_kind* is used to represent the risk level of the characteristics described by a requirement. For example, a functional requirement about the braking distance on wet surface will be labeled as "high risk". This is an indicator for system designers and testers that this requirement must be fulfilled in any case and that failing to fulfill it may pose serious risks in using the systems. For some cases, the requirements have another property, the *weighting* that shows how important the requirement is in the design. In case that the requirement cannot be fulfilled (for example, because it is in conflict with another requirement), this property is used to select which requirement may be relaxed in order to obtain a consistent design.

The *is_derived_into* relationship is used to trace the evolution of the requirements. At the first stages of the development, only the customer requirements are specified. They are then further refined in different types of requirements. For this case, it is said that the latter requirements are derived from the source requirements. Usually, the *is_derived_into* relationship links different levels of abstraction between the requirements. The *is_derived_from* is the inverse relationship of *is_derived_into*.

A very important relationship is the *is_fulfilled_by*, which is used to link a requirement to an object that fulfills it. In case of systems design, the relationship has as range a system or part of a system that fulfills the requirement. For example, the minimal braking requirement mentioned earlier is fulfilled by the braking subsystem of a car.

The fulfillment of requirements is tested with diverse methods in different stages of the development process. The goal of the process is to achieve a product that fulfills all requirements. For this reason, it is crucial to keep track whether a requirement has been tested or not and to store the result of the testing. Using this information, the degree of maturity of a design might be evaluated. The relationship between requirements and test cases is modeled with a reified relationship, *RequirementTestRelation* that contains more information than a simple link between the two classes. For example, the verdict of the test is a property of the reified relation, which represents whether the requirement has been successfully tested or not. The possible values of this property are the one proposed by SysML and UML2: *pass*, *fail*, *inconclusive*, *error*. Other properties of the reified relationship might be the tester of the requirement, and other meta-information, like for example, the date when the test has been done, or a natural language description of the test. The reified relationship contains two roles modeled as slots, *requirement* and *test* which link the corresponding test and requirements. The test cases can be further refined as subclasses of the *TestCase*. SysML proposes as four types of tests: *Analysis*, *Demonstration*, *Inspection* and *Test*.

4.5. Modeling Constraints

Constraints are an important part of the design models. They may be used to represent mathematical relationships between model elements. For example, a constraint might specify that the weight of a car should be less than 1500 kg. Another type of constraint can be used to model the physical behavior of a system, like for example, the Newton's second law of motion, $F = m \cdot a$. They may also be employed to model costs or for optimization purposes. Constrains restrict the solution space and are related to the quality of the product.

Constraints are attached at class level and applied at instance level. They may be used to constrain the properties of different type of classes. For example, they are attached to requirements and represent the mathematical translation of the requirement (if possible). In the example given before, it is easy to imagine that a car system has a weight property, say w. The formal representation of this requirement is straightforward: $w < 1500$ (if we consider that the unit of measure, kg, is also represented in a way in the design model). There are however, requirements that cannot be expressed in a formal way, either because the design model does not contain all necessary information, or the state of affairs that is constrained by the requirement is so complex that it cannot be formalized. In this case, checking the fulfillment of the requirement is done by testing.

Nevertheless, the constraints play an extremely important role in the design process, especially in the initial phases, when the preliminary design is made [Leemhuis, 2004]. In the first stages, only a coarse layout of the design and approximate parameter values are given. In the next stages, the design model is incrementally refined under consideration of the requirements and constraints defined in the model.

4.5.1. Constraints

The constraints are modeled in the *Constraints* ontology, shown in the right hand-side of Figure 4.13. A constraint instance contains a constraint statement given as a mathematical relation, like $w < 1500$. It also contains variable declarations that link the variables used in the constraint statement to elements of the design model. For instance, the variable w corresponds to the weight property of the car class. Variables are represented using paths of roles. The declaration of the variable w would be: *Car.weight*. There is no restriction on the length of the variable paths, which are made along the relationships (i.e., slots) defined in the model.

The constraint statement has been modeled as a simple String. However, this leaves space for many mistakes in editing the constraints. For example, it is possible to define a constraint statement and not to declare all variables in the statement. In order to overcome this shortcoming and to support the exchange of constraints between different engineering tools, a more formal approach to modeling the constraints statements can be chosen. For instance, Gruber and Olsen [1994] developed an ontology for mathematical modeling in engineering. The ontology includes concepts for scalar, vector, and tensor quantities, physical dimensions, units of measure, functions of quantities, and dimensionless quantities and has been explicitly designed for the knowledge sharing purpose.

A constraint has also attached to it a documentation that describes in natural language the content of the constraint. It might also contain a relationship to the element to which the constraint has been attached. This is important for the interpretation of the variable paths. For example, if a constraint is directly attached to a *Car* class that has a *weight* property, then the declaration of the w variable can be just *weight* because the constraint is interpreted in the context of the *Car* class and the *weight* property is local to the *Car* class. In Figure 4.13, the slot *used_in_requirement* points to the requirement to which the constraint has been attached. The inverse relationship is the *hasConstraint* at class *Requirement* that describes the attachment of a constraint to a requirement.

4.5.2. Role Paths

The variables of a constraint are declared as paths along the relationships in the model. A general declaration of a variable has the form:

$$v := Class.Rel_1[Restr_1].Rel_2[Restr_2] \ldots Rel_n[Restr_n].property \tag{4.31}$$

The definition of the path starts with a *Class* that specifies the context for the constraint. For example, the *Class* might be *Car*. Rel_i represent the relationships (i.e., slots) along which the path is defined. And $Restr_i$ represents a class restriction on the path that is applied to Rel_i. The definition of the path ends with a property to which the variable is bound.

An example of a variable declaration in the context of a *Car* class is:

$$tc := hasPart[Body].hasPart[PetrolTank].tankCapacity \tag{4.32}$$

The interpretation of the variable *tc* starts at class *Car* (because the constraint is attached to class *Car*), then from the values of the *hasPart* of the instances of *Car* only those are chosen which are of type *Body* (because of the class restriction), then for each instance the values of *hasPart* which are instances of *PetrolTank* are chosen and then the value of the property *tankCapacity* is the value of the variable *tc*. Usually the paths specify unique values, but there are cases in which the value of the variable is a set. In this case the operator applied to the variable must be a set operator.

The interpretation of the variable declaration is described formally below.

Suppose the variable declaration has the form:

$$v := Class.s_1[R_1].s_2[R_2]\ldots s_n[R_n].s_{n+1} \tag{4.33}$$

All relationships and also the final property in the path are represented by slots. The class restrictions are represented by a set of classes.

The interpretation of the path is done for a certain instance, denoted by *startInst*, which must be of type *Class*. The value of the path until step i, will be denoted by $Path_i$. I will use a helper function, *ownSlotValues(inst,slot)* which returns the set of values of slot *slot* at instance *inst*. The following holds:
$v = ownSlotValue(inst,slot) \Leftrightarrow hasOwnSlotValue(inst,slot,v)$

The value of the variable v is computed in a recursive way and is equal to the $Path_{n+1}$:

$$\begin{aligned} Path_0 &= \{startInst\},\ R_0 = \emptyset,\ R_{n+1} = \emptyset; \\ Path_i &= \bigcup_{x \in Path_{i-1}} ownSlotValue(x,s_i) \setminus \{i \mid instanceOf(i,R_i)\}; \\ v &= Path_{n+1} \end{aligned} \tag{4.34}$$

The concept needed for representing constraints (like, variables, role paths, class references) are modeled in the *Constraints* ontology. See Figure 4.13.

The path in declaration of variable should be a valid one. This means that it should be possible to navigate all through the path from the start to the end along the relationships in the order that they appear in the declaration. This imposes restrictions on the domain and ranges of the slots appearing in the path.

This restriction can be formalized in the following way:

$$\begin{aligned} &\textit{hasTemplateSlot(Class, } s_1\textit{)}; \\ &\exists \textit{inst} \in \textit{Path}_i,\ \textit{instanceOf(inst,cls)} \ \wedge\ \textit{hasTemplateSlot(cls,}s_{i+1}\textit{)} \end{aligned} \tag{4.35}$$

4.5.3. Using the Engineering Ontologies

I will refer to the *Components*, *Connections*, *Systems*, *Requirements* and *Constraints* ontologies as **engineering ontologies**.

The engineering ontologies have been develop as very general and abstract ontologies, so that they can be reused in several engineering applications. The ontological commitments are kept to a minimum, which is a desiderata in building ontologies [Gruber, 1993]. The engineering ontologies form a conceptual lego that can be assembled together to serve the purposes of different applications. Figure 4.14 shows such a situation.

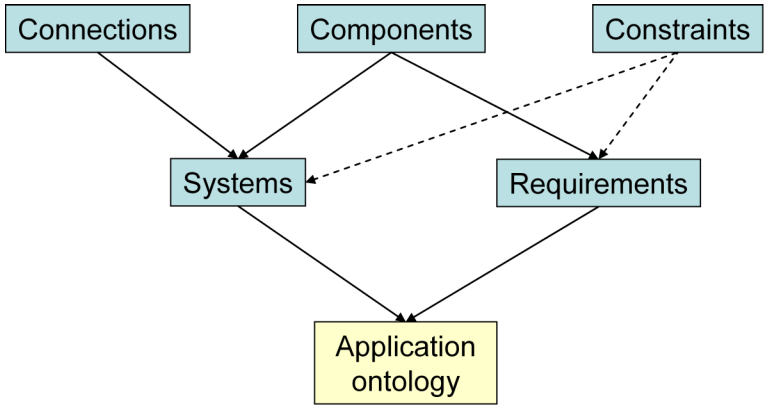

Figure 4.14.: Combining the engineering ontologies to build an application ontology.

An application that needs to model systems and requirements will include the *Systems* and the *Requirements* ontologies. Another application that only models systems and constraints may include only those ontologies without including the rest.

Chapter 5 will show how the engineering ontologies may be used as upper level ontologies that ensure a common vocabulary for design models from different viewpoints and for supporting the mapping process between them.

5. Semantic Mappings Between Engineering Models

This chapter defines an ontology mapping framework that supports different tasks in the design process, such as consistency checking and change propagation between design models. The central element of the mapping framework is a mapping ontology that stores the correspondences between design model templates, which are used at runtime to infer correspondences between concrete design models.

The chapter is organized as follows. Section 5.1 gives an overview of the role of ontology mappings in different scenarios. Section 5.2 describes the mapping framework that is used in solving different design tasks. The Mapping ontology is described in Section 5.3. Section 5.4 shows by means of examples how the mapping framework can be used to define and execute mappings between design models.

5.1. Roles of Ontology Mappings

The area of ontology mapping is a subject of active research in different disciplines, like databases, peer-to-peer architectures, information integration, service-oriented architectures, etc. Rather than simply exchanging data on the syntactic level as current data integration approaches, research in ontology mapping is investigating solutions for exchanging the meaning of information between different software applications. Sharing the same vocabulary between different applications, or even between people, does not guarantee that the same term or word is used with the same meaning by all parties involved. Klein [2001] identified several classifications of data source heterogeneity. Wache et al. [2001] gives a high level classification in structural and semantic heterogeneity. Structural heterogeneity is caused by the fact that information systems are storing their data in different structures, whereas semantic heterogeneity deals with the content of the information and its intended meaning. A solution for the semantic heterogeneity between different software systems is to use ontologies as a way to make explicit the meaning of the terms in a domain [Wache, 2003].

Ontology mappings are especially important whenever meaning of information needs to be exchanged between software systems. In the following I will illustrate briefly the role of ontology mappings in different tasks.

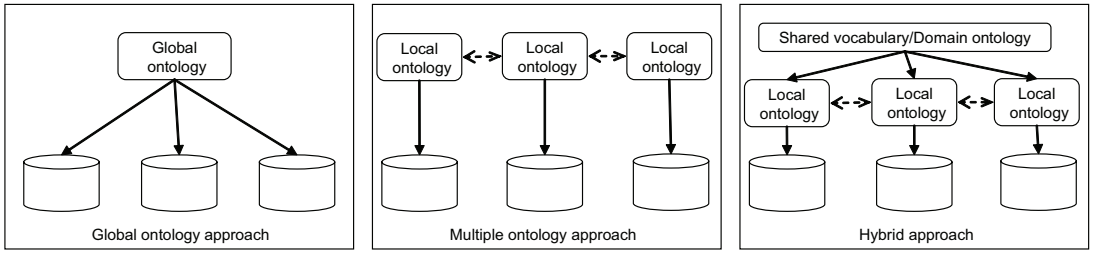

Figure 5.1.: Three ontology-based integration approaches. Adapted from [Wache, 2003].

5.1.1. Semantic Information Integration

Enterprises information systems store huge amount of data in different data sources (databases, intranets, documents, web pages, etc.). The main challenge lies in providing the right information to the right people at the right time. Many commercial solutions assist the information integration, however only at the syntactic level of the data, which cannot cope with the challenging requirements on the modern information systems. One of the key application of ontologies is to facilitate the semantic interoperability and information integration [Uschold and Gruninger, 2004] .

Wache [2003] identified three main approaches used in semantic integration showed in Figure 5.1. The mappings involved in these approaches are mappings between information sources and ontologies or between different ontologies. The three approaches are:

- Global ontology approach
- Multiple ontology approach
- Hybrid approach

The **global ontology approach** uses one ontology to which all information sources relate. It is similar to the global-as-view (GAV) integration used in databases. This approach is suitable for situations in which the information sources represent the same viewpoint on a domain. However, the scalability of this approach is problematic, because the global ontology and the mappings between the information sources and the global ontology need to change whenever a new information source is added.

In the **multiple ontologies approach**, each information source is described by its own ontology, and ontology mappings are used to put into correspondence concepts in the different local ontologies. The advantage of this approach is that new information sources can be added without influencing the other local ontologies or the mappings between them. However, this

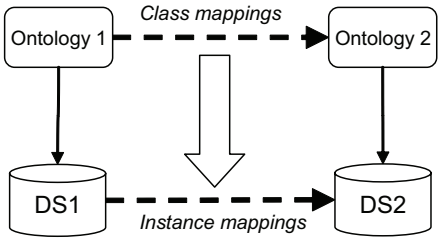

Figure 5.2.: Data migration using ontology mappings. The class mappings are applied at instance level to transform the data from one representation into another.

approach can be difficult to realize, due to the lack of a common vocabulary between the local ontologies, which can make the mapping discovery and definition very hard.

The **hybrid ontology approach** overcomes the disadvantages of the first two approaches. A common domain ontology serves as a common vocabulary for the local ontologies. In this way, the local ontologies will share a common vocabulary and the meaning of the terms. The local ontologies may extend the concepts in the domain ontology. The correspondences between the local ontologies is made by the mappings, which are easier to discover and maintain due to the shared concepts defined in the domain ontology. This approach has also been used in this work to enable the synchronization and consistency checking between design models.

Another approach of using an ontology for interoperability between software applications is to use it as an *interlingua* ontology [Uschold and Gruninger, 1996]. Instead of designing n^2 point-to-point translators between n applications, using an inter-lingua ontology, only n translations have to be built. For this case, mappings are defined between the local ontologies and the inter-lingua ontology. This approach has been implemented in [Gruninger and Kopena, 2005; Ciocoiu and Nau, 2000].

5.1.2. Data Migration

Ontology mappings may also facilitate data migration, which has as a purpose the transformation of data from the representation of one data source to the representation of another data source. Figure 5.2 shows this approach.

Mappings between the local ontologies of the data sources are usually defined between the classes of the two ontologies. For example, if the ontologies define the class *Car* and *Vehicle* respectively, and it is known that cars are vehicles, then a mapping can be defined from class *Car* in the source ontology to class *Vehicle* in the target ontology that will allow the transformation of all instances of *Car* in instances of *Vehicle*.

Ontology mappings and instance transformation enable the reuse of problem solving methods (PSM) on different domains. The mappings allow the transformation of the domain instances

into method instances on which different PSMs, like classification, configuration, parametric design, etc., may be applied [Crubézy and Musen, 2004].

Data migration stores persistently the results of the transformation in the target format. There is another more flexible approach that does not require the persistent storage of the instances, but still allows the interoperability of different systems. Ontology mappings may be used for *query rewriting* in an integration framework. The common use case is that a user asks a query that is evaluated in several data sources. Mappings between the ontologies of the data sources facilitate the query rewriting at runtime and the aggregation of the results. This integration approach has been implemented in the OBSERVER framework [Mena et al., 1996].

5.1.3. Ontology Management

One of the strength of ontologies is that they enable knowledge sharing and reuse between different applications. However, there are cases in which an application requires concepts from several ontologies. A solution to this is to merge the ontologies into a new ontology that will be used by the application. The merging of ontologies is also supported by the definition of ontology mappings [Noy and Musen, 2000].

As the ontology-based technologies reached a certain degree of maturity, managing their life cycle in an efficient way became crucial. Just like software, ontologies also evolve in time. Keeping track of versions and changes in the ontologies is an important task. However, existing version control systems cannot be used on ontologies, because of their syntactic (text-based) operation methods. The differences between several versions of the same ontology must be made at the level of concepts, together with their associated properties and relations. Currently, there are several tools supporting structural diffs applied on ontology versions, such as PromptDiff [Noy and Musen, 2002] or SemVersion [Völkel and Groza, 2006]. Ontology mappings may be used to document the relationships between different versions of a concept.

5.2. A Mapping Framework for Model Interoperation

The motivating scenario in Section 2.1 shows a number of use cases in the collaborative development process, which involve design models of the same product built in different engineering tools. The use cases show that during the development process, different operations and tasks that operate on models need to be done, like model synchronization, change propagation and consistency checking. All these operations require that the models "understand" each other, even if they are represented in different modeling languages and have different views and conceptualization on the modeled product. The tools themselves do not have the capability to deal with the semantic heterogeneity of the models. A supporting framework needs to be developed that will facilitate the operations and tasks mentioned before at the semantic level.

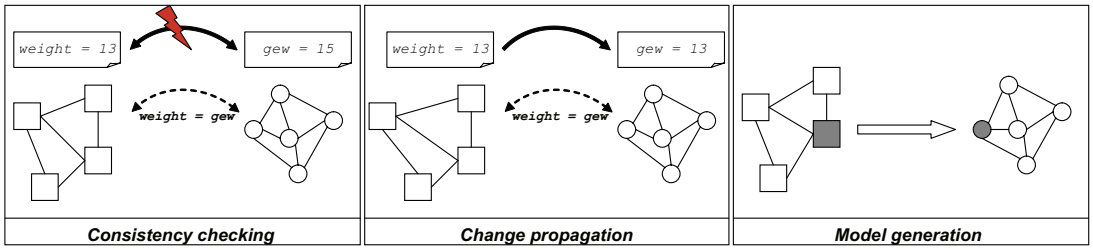

Figure 5.3.: Three tasks that should be supported by the interoperation framework.

I will start by analysing the requirements of such an interoperation framework in Section 5.2.1 and in the rest of the section, I will describe its design.

5.2.1. Requirements to the Interoperation Framework

The design process is highly iterative (see Section 2.2.1). The interoperation framework should support the repetitive design tasks, which are enumerated in the following and shown in Figure 5.3.

Consistency checking Design models in different tools are representing the same product from different viewpoints. Even if they are developed independently and for different purposes, they still have some common ground that ensures that they are representing the same product in the end. Usually there are dependencies between the design models at the conceptual level. For example, the geometry of a product influences its behavior, so it is natural that there will be some mathematical correspondences between parameters in a geometrical model and a functional model of the same product. Consistency checking insures that the correspondences between the models are not violated.

Change propagation Design models evolve very quickly, especially in the initial phases of the development. It is crucial that the design models of a product are permanently synchronized, otherwise they may follow too different paths that risk to become incompatible over time. For this reason, change propagation between different models is essential in the design process. Change propagation should be done in a "soft" way: Changes from one model need to be integrated in the other model without regenerating the whole model, but rather by only integrating the diffs in the source model interpreted in the language of the second model. Even if the integration of the diffs is not done in an automatic way, it is a benefit for the engineers if they can obtain "delta reports" of the changes in the other model in the language of their own models.

Model skeleton generation In many cases, certain types of design models are developed first and some of their properties are used in building models for another viewpoint. For example, the physical structure of a product may be generated out of a functional model [Pahl and Beitz, 1996]. This means that having correspondences between the general system components facilitates the generation of a model skeleton for a viewpoint out of a model from another viewpoint. Complete model generation is often not possible, because it is usually impossible to specify complete correspondences between general components in different viewpoints. However, even the partial generation of a model brings benefits for the design process.

Other requirements on the interoperation framework are automation, flexibility, extensibility and reuse.

Automation Because of their repetitive nature, all the activities described above should be supported by automated or semi-automated methods.

Flexibility The interoperation framework should adapt easily to changes in the tools or design models. For example, if some correspondences between design models change, the mapping framework should be able to adapt to these changes in a natural way.

Extensibility It should be easy to extend the framework to support new viewpoints, if needed. For example, if new engineering tools are added in the design process, they should be easily integrated in the interoperation framework.

Reuse The interoperation framework should support the reuse in the design process, both at the level of design systems and at their components'. This is extremely important especially if the design is made with the help of model libraries.

5.2.2. The Mapping Framework

The mapping framework used to solve the tasks in the design process has been developed taking into considerations the requirements defined in Section 5.2.1. Its architecture is shown in Figure 5.4. The components of the mapping framework are:

- Design models – on which the different tasks are applied,
- Local ontologies – which represent the semantically enriched models of the design models,

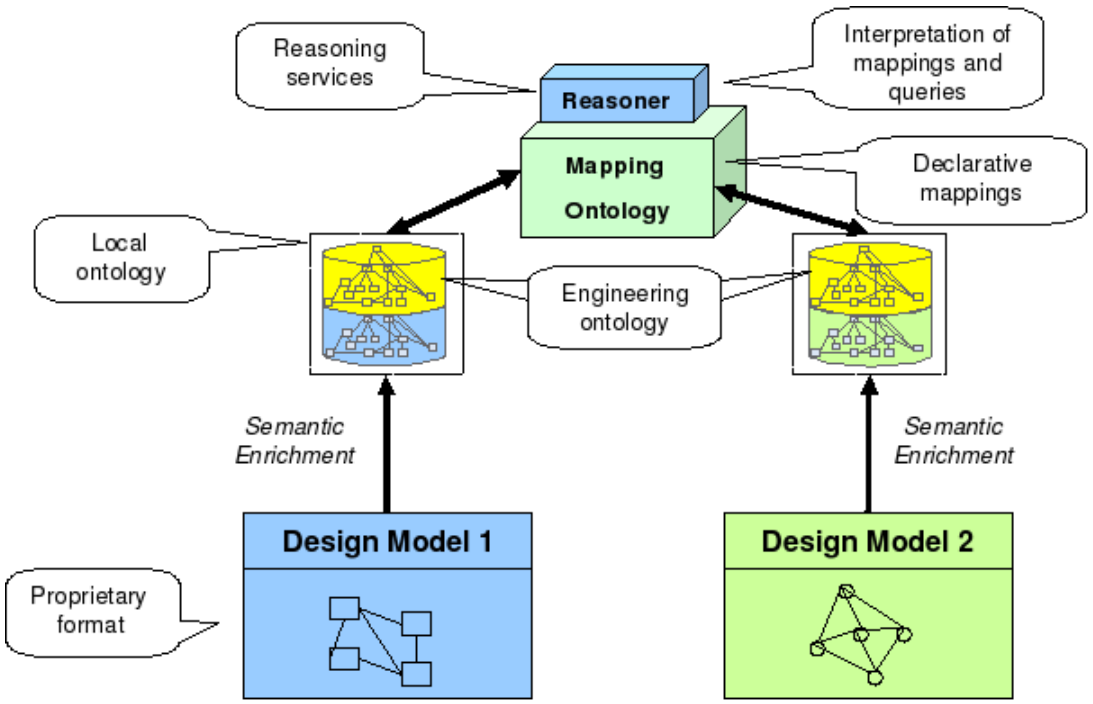

Figure 5.4.: The mapping framework.

- Engineering ontologies – used as a common upper ontology for the local ontologies,
- Mapping ontology – used to represent declaratively the mappings (correspondences) between the local ontologies,
- Reasoner – interprets the defined mappings at run time and supports the execution of the tasks.

In the following I will describe briefly each of the components of the mapping framework.

Design Models

The design models are the models as defined in the engineering tools. For example, an excerpt of a functional model and an XML representation of geometrical model of a planetary[1] are shown in Figure 5.5 side by side.

[1] Planetaries are gear-wheels used in gearboxes to change the gears in a car

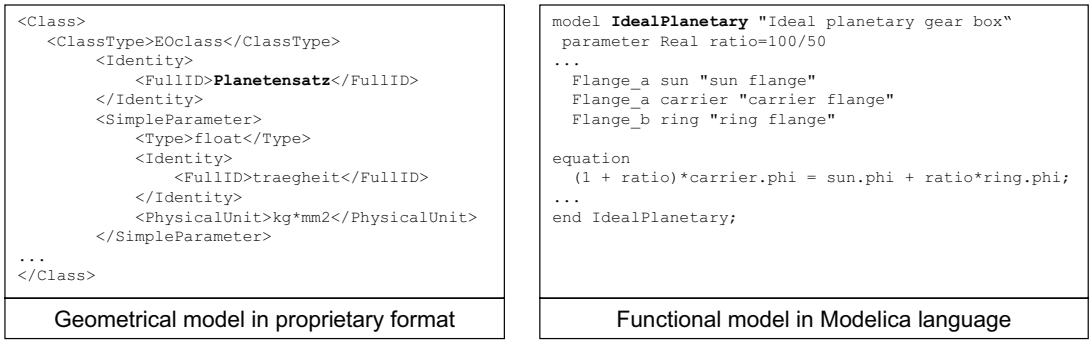

Figure 5.5.: Excerpts from two design models in different modeling languages side-by-side.

The design models that participate in the design tasks are represented typically in different modeling languages. In the previous example, the functional model is represented in an object-oriented language for modeling physical systems, Modelica [Modelica Specification, 2005], and the geometrical model is represented in an XML-based proprietary format defined in [Zimmermann, 2005].

Local Ontologies

A local ontology is the result of the semantic enrichment of a design model. The act of enriching, sometimes also called lifting [Maedche et al., 2002], transforms the design model represented in some modeling language into an ontology. This requires understanding the meta-model of the language and implies some transformations from the language meta-model to an ontology [Karsai et al., 2003]. In the example from Figure 5.5, the geometrical representation of the planetary uses a XML element – *SimpleParameter* – to represent the definition of a parameter of type float of the planetary[2]. This would be represented in the local ontology using a template slot of type float. The semantic enrichment can be done in different ways: either by annotating the design model elements with the semantic concepts from the ontology, similar to the approach taken by [Wache, 2003], or by transforming parts of the design model in classes and instances in the local ontology. I have chosen the second approach in order to allow logical reasoning to be performed on the concepts in the ontology. [Wache et al., 2001] gives an overview of the ontology-based integration approaches and discusses also the semantic enrichment. [Stojanovic et al., 2002] describes an approach for the semantic lifting of relational databases using frame logic.

Not all elements from the design models need to be translated in the local ontology, but only those elements which are relevant for the task that needs to be solved. For example, the

[2] "Planetensatz" (in German) means planetary

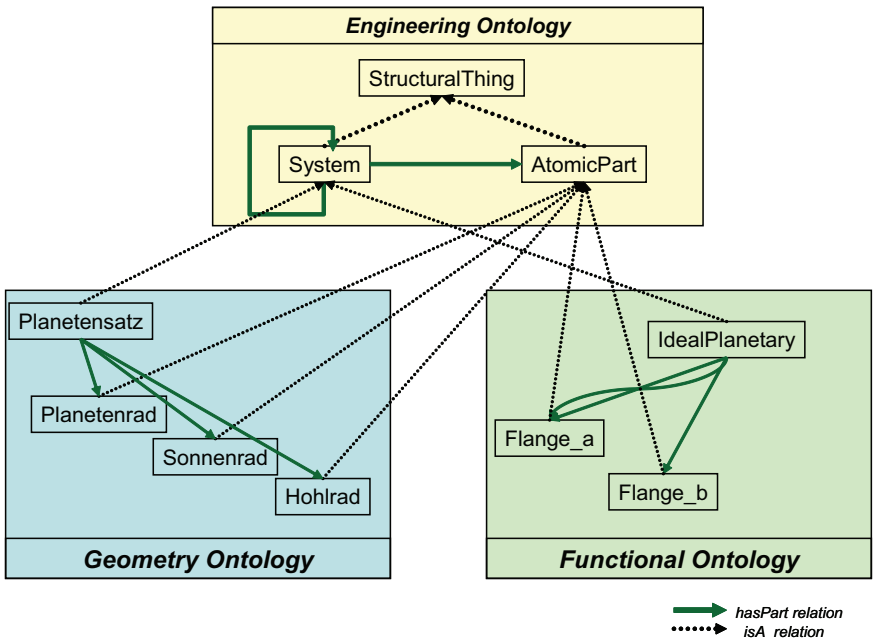

Figure 5.6.: The local ontologies (geometrical and functional ontologies) extend the *Engineering* ontology.

functional model represents the behavior of the systems using mathematical equations. In a change propagation task, in which data from a functional model is propagated to a geometrical model, it does not make sense to translate also the equations, because the geometrical tool cannot understand or process them.

The *Engineering* Ontology

The *Engineering* ontology is used as a common upper level vocabulary for the local ontologies. As in the hybrid integration approach, described in [Wache et al., 2001], the upper level ontology ensures that the local ontologies share a set of concepts with the same semantics. The local ontologies specialize the upper level concepts. This also ensures that the local ontologies will be comparable with each other. This brings many benefits if mappings between the local ontologies need to be discovered automatically. An example of the way local ontologies extend the *Engineering* ontology is shown in Figure 5.6.

The Mapping Ontology

The mapping ontology is used to represent in a declarative way the correspondences between the local ontologies. Having an explicit and well-defined representation of mappings enable reasoning about them, such as, checking whether two mappings are equivalent or mapping composition [Madhavan et al., 2002].

An explicit representation of mappings makes it also possible to model different types of mappings, like, renaming mappings, lexical mappings, recursive mappings, and so on. Park et al. [1998] proposes a classification of mapping types. However, the mapping ontology must be designed in such a way that it supports the tasks that must be solved. For instance, mappings between ontologies of engineering systems have to take into consideration the part-whole decomposition of systems and hence has to support the representation of paths in the part-whole hierarchy.

An example of the mappings between the functional and geometrical ontologies in the previous example is shown in Figure 5.7.

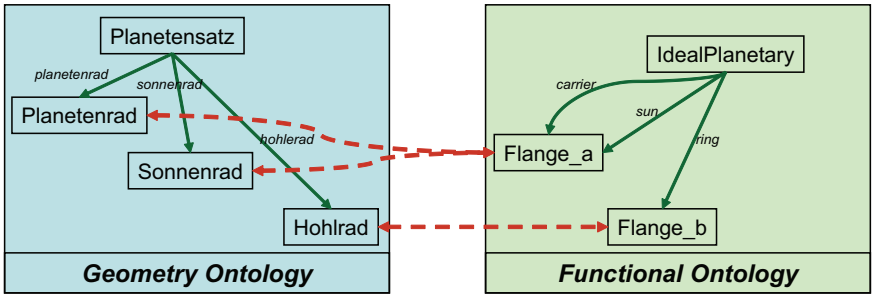

Figure 5.7.: Mappings between the local geometrical and functional ontologies. The classes *Planetensatz* and *IdealPlanetary* are mapped together. Also their parts are mapped together, *hohlrad* of the *Planetensatz* is mapped to the *ring* of *IdealPlanetary*. The mapping between the *Planetensatz* and *IdealPlanetary* also contain a constraint between the attributes of the classes in the form of a mathematical relationship.

The Reasoner

The reasoner is an important part of the mapping framework. The reasoner is used both at design-time of the mappings and at runtime to support the execution of the design tasks.

At design time, the reasoner may be used to verify the mappings and to see their effect in a testing environment. It can also provide suggestions for other mappings based on the already defined ones, or warn about missing mappings. At runtime, the reasoner executes the mappings and other operations, such as consistency checking, that are relevant for a particular task.

The reasoner used in the framework is $\mathcal{F}$LORA-*2*. $\mathcal{F}$LORA-*2*[3] (F-Logic translator) is a declarative object oriented language used for building knowledge intensive applications. It is also an application development platform [Yang et al., 2005]. $\mathcal{F}$LORA-*2* may be used for ontology management, information integration or software engineering.

$\mathcal{F}$LORA-*2* is implemented as a set of XSB[4] libraries that translate a language created by the unification of F-logic, HiLog [Chen et al., 1993] and Transaction Logic into Prolog code. $\mathcal{F}$LORA-*2* programs may also include Prolog programs. It provides strong support for modular software through dynamic modules. Other characteristics make $\mathcal{F}$LORA-*2* appealing for using it in reasoning with frame-based ontologies, such as support for objects with complex internal structure, class hierarchies and inheritance, typing, and encapsulation.

5.2.3. Template Mappings for Model Libraries

One of the requirements on the mapping framework is to support the reuse of components and of their mappings. The design process typically uses component libraries that contain general, parameterized descriptions of components, which may be instantiated to create components with specific parameter values.

The reuse of components in a model library reduces significantly the time needed for designing a new product. Components are selected from the model library and are assembled together to form a basic design for a new product. The quality of the design models is also increased by reusing already validated components. Another benefit is reducing the complexity of the product by modularization and by using standardized components. The library components are templates for building specific components and are modeled as classes in the library models ontology.

Similar to the component templates, I propose using **template mappings** that interrelate template components from different model library ontologies. The template mappings are instantiated and applied to a concrete design model, which is built out of instantiated template components from the library. Figure 5.8 shows two viewpoints (for example, a functional and a geometrical viewpoint) in a design setting. Each viewpoint contains a library of models that is represented as a taxonomy of model classes in a local ontology. The template mappings are defined at the level of the general components in the library. In the example

[3] http://flora.sourceforge.net/
[4] http://xsb.sourceforge.net/

given in Figure 5.7, the class *Planetensatz* from the geometrical library is mapped to the class *IdealPlanetary* in the functional library. The instantiation of template mappings will be discussed in details in Section 5.4.3

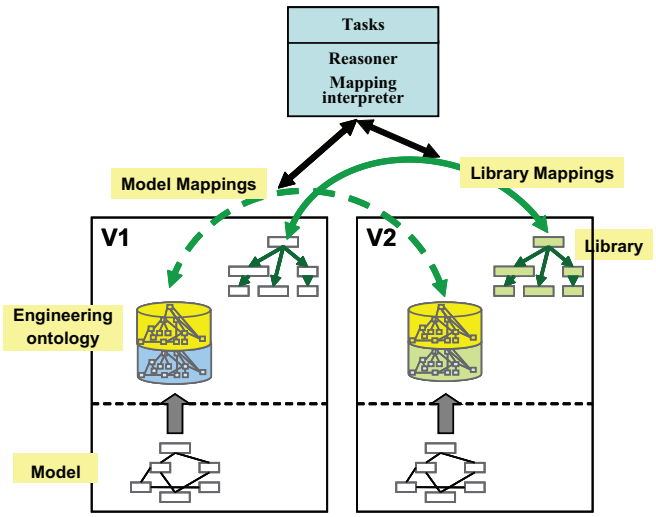

Figure 5.8.: Template mappings between components in the model libraries are used to infer mappings between specific instances of components.

5.3. The *Mapping* Ontology

This section defines the *Mapping* ontology, which is used to represent in a declarative way the correspondences between different ontologies. First, the requirements on the representation of the ontology are identified. Second, the modeling of the *Mapping* ontology is presented together with examples that demonstrates its usage.

5.3.1. Requirements to the *Mapping* Ontology

The *Mapping* ontology is the centerpiece of the mapping framework. Considering the tasks that need to be solved and the environment in which they will be employed, there are several requirements that the *Mapping* ontology should fulfill. They are enumerated in the following.

Structural mappings The design models are structured along the part-whole dimension: Systems are made of parts, which in turn may be systems. The mappings should be able to represent and reference parts of systems.

Context Some mappings are defined and valid only in a specific context. For example, two parts are mapped in the context of a mapping between the containing systems, which are also mapped to each other.

Multiple ontologies The *Mapping* ontology should be able to represent the mappings among several ontologies, not just between two ontologies.

Mapping rules Certain mappings contain additional rules, for example mathematical relations between properties in the two ontologies, which should hold in the context of a mapping.

Composability It should be possible to compose mappings and obtain a new mapping. For example, if all the parts of two systems are mapped, it is also possible that the systems themselves are mapped to each other.

Extensibility The *Mapping* ontology should be extensible with new types of mappings. This is useful for the case that new properties or constraints have to be added to the mappings. For instance, mappings may be classified in mappings between systems and mappings between atomic parts. The mapping between the system may contain an additional rule that ensures that all the parts of the system are also mapped to each other.

5.3.2. Representation of the *Mapping* Ontology

Several requirements expressed the need to handle composite structures and paths in these structures. The *Constraints* ontology defined in Section 4.5 has the right constructs for dealing with paths. It also provides the constructs for representing constraints, which is another requirement for the *Mapping* ontology. For this reason, the *Constraints* ontology is imported in the *Mapping* ontology. Mappings between ontologies are represented as instances of the *Mapping* class. The classes of the *Mapping* ontology are shown in Figure 5.9 and described in the following paragraphs.

Concept An instance of this class represents a concept that is the unit, which can be mapped to a concept in another ontology. Concepts may be simple classes (such as *Car*), or they may be attributes (i.e., template slots) attached to classes (for example, *Car.weight* might be mapped to *Vehicle.mass*). It is also possible to represent concepts that are reached by traversing slots paths as it is shown in Figure 5.10. Mapping the x component of the position can be done by storing in a concept both the start class and a slot path.

Mapping An instance of this class represents the mapping between two concepts originating in different ontologies. The slots *concept1* and *concept2* have as values the concepts that are mapped. Mappings may have dependent mappings, with the meaning that a

dependent mapping is valid only in the context of another mapping. For example, a mapping between the parts of a system are dependent on the mapping between the systems. A mapping may also have preconditions, meaning that the mapping is executed only if the preconditions hold. Mappings have also constraints attached to them, so that it is possible to model additional relationships between the mapped classes or between the dependent mappings. In the example from Figure 5.7, a mathematical relationship between properties of the parts of the system is attached to the mapping between the systems. A mapping may also be defined as reversible, which means that the mapping holds in both directions.

Namespace Since the mapping ontology may define mappings between an unspecified number of ontologies, identifying uniquely each of them is done with the help of namespaces. The *Namespace* class defines the attributes that uniquely identify a mapped ontology in the *Mapping* ontology. It contains as slots a logical URI and a physical URI originating from the location and name of the ontology file.

ClassRef Instances of this class are used to uniquely identify a class from a given ontology. It consists of a namespace and the name of the class. The namespace is used to identify the ontology, and the *className* identifies the name of the class.

PathRef The instances of the PathRef class define one element in a chain of elements that form the slot path of a concept. It contains the name of the slot, a class reference to define the path restriction, or the context the slot is defined in, and the next path reference in the chain of slot path.

5.4. Defining and Executing Mappings

The next step after defining the *Mapping* ontology is to learn how it can be used. This section shows with the help of examples how the mappings between systems can be defined and how the mapping algorithm is used to map instances by composing class mappings. It will also describe how instances between different ontologies may be merged in a virtual instance.

5.4.1. Mapping Systems and Components

The *Mapping* ontology allows the definition of mappings between component classes and their structure. In the example from Figure 5.7, the mapping of the *Planetensatz* class from the geometrical ontology to the *IdealPlanetary* class in the functional ontology will also contain (i.e., have as dependent) the mappings of the component parts (such as the mapping of *Planetenrad* to *Flange_a*). The mapping of a component usually defines also the mappings of the components parts, if they are known.

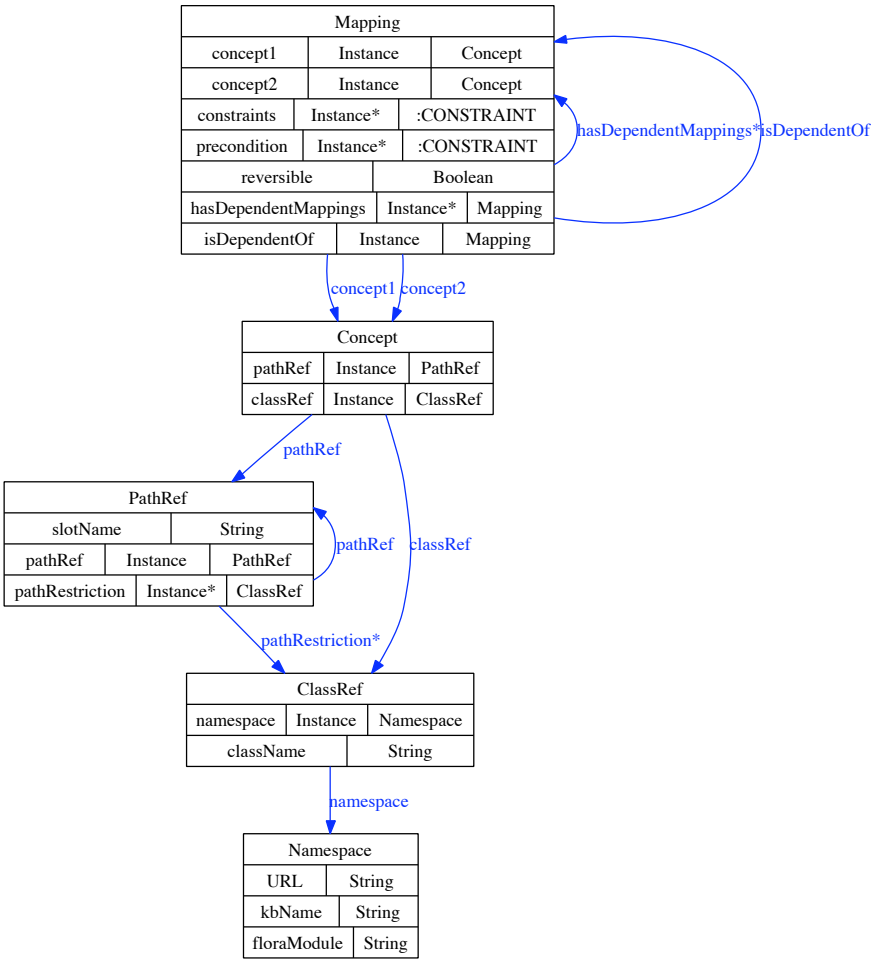

Figure 5.9.: The *Mapping* ontology is used to represent the mappings between concepts in different ontologies as instances of the *Mapping* class.

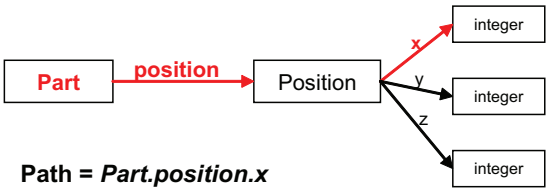

Figure 5.10.: Example of addressing a concept using a slot path: the x coordinate of a part can be addressed as *Part.position.x*.

I will use the following notations:;

- $Concept_1 \Longrightarrow Concept_2$ – to denote that $Concept_1$ is mapped to $Concept_2$ at class level. $\Longleftrightarrow$ is used for bidirectional mapping.
- $Concept_1 \longrightarrow Concept_2$ – to denote that $Concept_1$ is mapped to $Concept_2$ at instance level. $\longleftrightarrow$ is used for bidirectional mapping.
- $Elem.slot_1 \ldots slot_n$ – to denote a path starting from class or instance *Elem* and following the relationships defined by the slots.

In the example from Figure 5.7, the following mappings between the classes in the two ontologies have been defined:

$$Plantensatz \Longleftrightarrow IdealPlanetary \tag{5.1}$$
$$Planetensatz.sonnenrad \Longleftrightarrow IdealPlanetary.sun \tag{5.2}$$
$$Planetensatz.planetenrad \Longleftrightarrow IdealPlanetary.carrier \tag{5.3}$$
$$Planetensatz.sonnenrad \Longleftrightarrow IdealPlanetary.sun \tag{5.4}$$

The mappings 5.2, 5.3 and 5.4 are defined as dependent mappings of 5.1. This means that they cannot exist by themselves, but only in the context of the top-level mapping between the system classes. The dependent mappings define correspondences between roles in the mapped systems. Figure 5.11 shows the representation of the *Planetensatz.sonnenrad* $\Longleftrightarrow$ *IdealPlanetary.sun* mapping in the *Mapping* ontology.

Figure 5.7 shows three independent mappings:

$$Planetenrad \Longrightarrow Flange\_a \tag{5.5}$$
$$Sonnenrad \Longrightarrow Flange\_a \tag{5.6}$$
$$Hohlerad \Longrightarrow Flange\_b \tag{5.7}$$

The difference between a *dependent* mapping and an *independent* one is that the independent mapping is done between definition of classes, such as *IdealPlanetary* and *Flange_a*, while dependent mappings are done between the roles that parts (i.e., slots) are playing in a system (i.e., a class). In the previous example, *IdealPlanetary.sun* references the role that a *Flange_a* component is playing in the system *IdealPlanetary*.

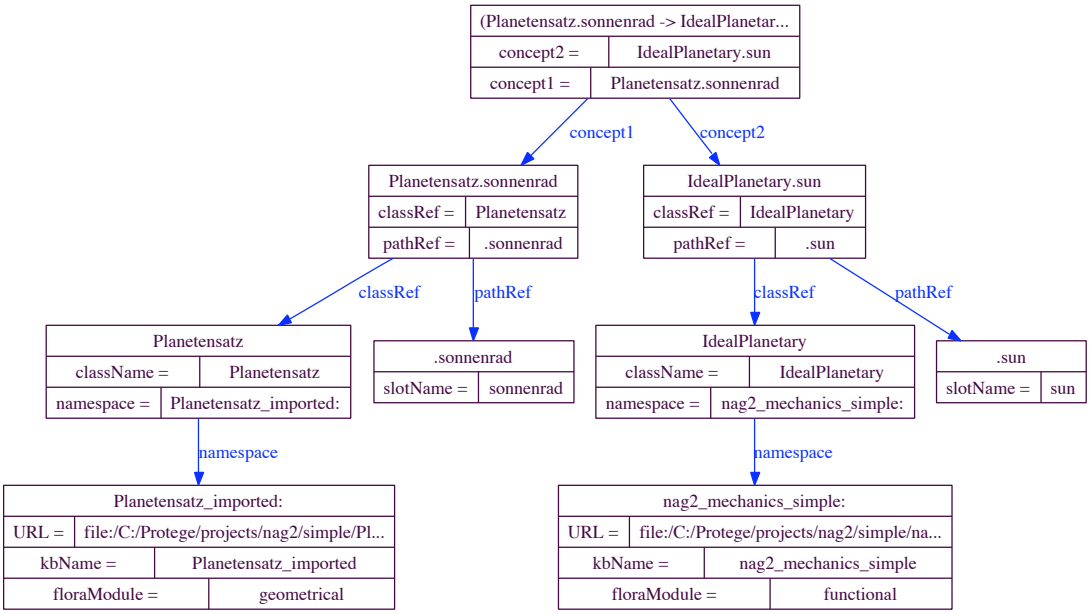

Figure 5.11.: An example of the mapping representation of *Planetensatz.sonnenrad* $\Longleftrightarrow$ *IdealPlanetary.sun*.

5.4.2. The Mapping Algorithm

The mapping algorithm has been specifically conceived to work with composite structures, such as the design models. The mapping algorithm takes as input one or more source ontologies, one or more target ontologies, a mapping ontology that contains the definition of the mappings between the source and target ontologies and two instantiated models from the source and target ontologies, respectively. The instantiated model is often part of the ontology. The output of the mapping algorithm is different based on the task that is being executed. For example, in consistency checking it will return true or false depending whether the two models are consistent with respect to the defined mappings and constraints, whereas in the change propagation task, it will set the values of target properties to the computed values from the source ontology.

However, there are some basic functionalities that the mapping algorithm offers and on top of them, other algorithms may be defined. The basic functionalities are described in the following sections. Figure 5.12 shows the mapping algorithm and its inputs and output.

The mapping algorithm is executed in $\mathcal{F}$LORA-*2*, which has been introduced in Section 5.2.2. In order to execute the algorithm, the mapping ontology and the two mapped ontologies as well are exported to the $\mathcal{F}$LORA-*2* representation. Each ontology is converted to a $\mathcal{F}$LORA-*2*

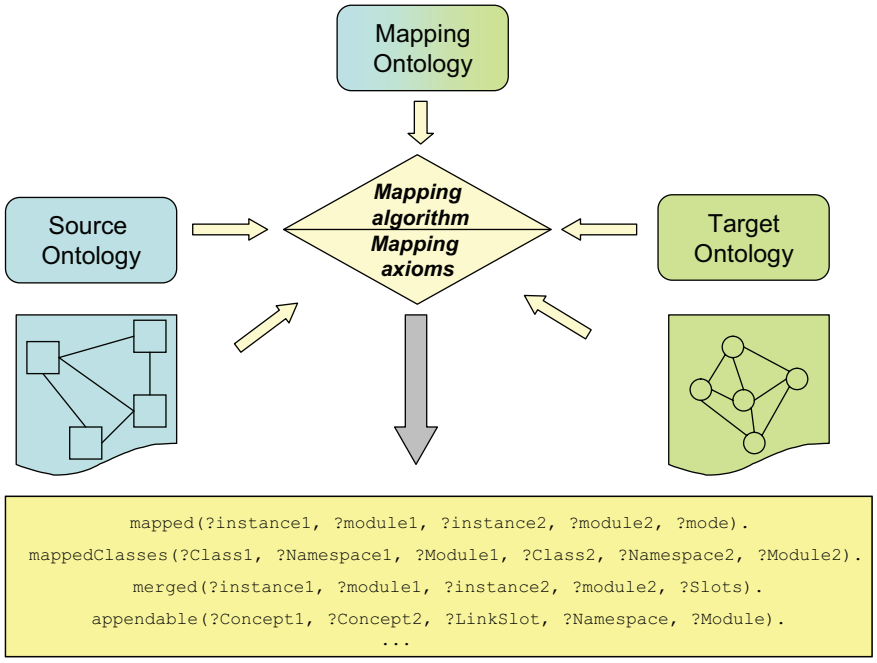

Figure 5.12.: The mapping algorithm with its inputs and output. The output represent the basic predicates that the algorithm offers.

file. In order to solve the possible name clashes (i.e., two classes have the same name in two ontologies), the ontologies are loaded into separate $\mathcal{F}$LORA-*2* modules[5]. The mapping ontology is loaded into the default module together with the mapping axioms. A full listing of the mapping axioms is given in Appendix A.

5.4.3. Mapping Instantiation

One of the basic functionalities of the mapping algorithm is to find mapped instances in the source and target ontologies based on the definition of mappings done at class level and on the mapping axioms. In order to define the semantics of mappings, I will use the first order logic predicates defined in Section 4.2.3 and additional predicates, with the following meaning:

- *mapped(A,B)* – which is true when instance *A* is mapped to instance *B*. (*A* and *B* are from different ontologies).

[5]The support for modules is one of the most powerful features of $\mathcal{F}$LORA-*2*. Modules are loaded in different memory spaces and do not share the namespace. However, it is possible to reference a module from another module. For further details see [Yang et al., 2005]

- *mappedCls(A,B)* – which is true when class A is mapped to class B.
- *path(A,B)* – which is true whenever there is a path in the ontology between elements A and B. The paths may be both at instance and at class level. The definition of paths in the ontology is given in Section 5.3.2.

Two instances are mapped if there exists a path in the source and target ontologies that have starting points which are already mapped to each other.

$$\begin{aligned} \textit{mapped(I1,I2)} \Leftrightarrow\ & \textit{instanceOf(I1,C1)} \wedge\ \textit{instanceOf(I2,C2)} \wedge \\ & \textit{mappedCls(C1,C2)} \wedge \\ & \textit{path(A,I1)} \ \wedge\ \textit{path(B,I2)} \ \wedge\ \textit{mapped(A,B)}. \end{aligned} \tag{5.8}$$

This axiom is used in the top-down mapping algorithm, which also needs as an input an initial mapping between two instances, called **entry point**. All the axioms of the top-down mapping algorithm are provided in Appendix A. The algorithm starts with the mapping of the entry points, which has been defined by the user. Then, it will map all the instances that are reached by following the paths defined by the mappings. If an end point is reached, then it will look for mappings that start in the end point, and it will try to compose the mappings by following the paths until no more paths can be found.

The top-down mapping algorithm has proved to be very suitable for mapping systems along their structure. It has been also shown that this mapping algorithm supports the mapping reuse by allowing the mapping composition. This is demonstrated by the following example.

The example uses the mappings already defined in the previous section, which are repeated here for convenience. The mappings show the correspondence between an *IdealPlanetary* system and a *Planetensatz* system, which are also mapped along their structures.

$$\textit{Plantensatz} \Longleftrightarrow \textit{IdealPlanetary} \tag{5.9}$$

$$\textit{Planetensatz.sonnenrad} \Longleftrightarrow \textit{IdealPlanetary.sun} \tag{5.10}$$

$$\textit{Planetensatz.planetenrad} \Longleftrightarrow \textit{IdealPlanetary.carrier} \tag{5.11}$$

$$\textit{Planetensatz.sonnenrad} \Longleftrightarrow \textit{IdealPlanetary.sun} \tag{5.12}$$

Two systems, *Gearbox* and *Getriebe*[6] defined in two different ontologies are mapped to each other. In order to define a complete mapping, their structure (i.e., parts) must also be mapped. Both systems have a similar structure made up of three identical gears which are connected

[6]Getriebe (German) means gearbox in English. A gearbox is used in a car to change gears.

to each other. The definition of the two systems is shown below. (The dashed arrow $\dashrightarrow$ applied to a path at class level symbolizes the allowed classes of a template slot, and applied at instance level it denotes the value of that instance holds for a property)

$$Getriebe.planetensatz1 \dashrightarrow Planetensatz \tag{5.13}$$
$$Getriebe.planetensatz2 \dashrightarrow Planetensatz \tag{5.14}$$
$$Getriebe.planetensatz3 \dashrightarrow Planetensatz \tag{5.15}$$

and, similar for the *Gearbox* class

$$Gearbox.frontPlanetary \dashrightarrow Planetary \tag{5.16}$$
$$Gearbox.backPlanetary \dashrightarrow Planetary \tag{5.17}$$
$$Gearbox.centerPlanetary \dashrightarrow Planetary \tag{5.18}$$

The template mappings between the two systems are the following:

$$Getriebe.planetensatz1 \longrightarrow Gearbox.frontPlanetary \tag{5.19}$$
$$Getriebe.planetensatz2 \longrightarrow Gearbox.backPlanetary \tag{5.20}$$
$$Getriebe.planetensatz3 \longrightarrow Gearbox.centerPlanetary \tag{5.21}$$

The template mappings between *Getriebe* and *Gearbox* only map their internal structure. In order to map the whole system with all their parts (also indirect parts), it is necessary to compose the mappings defined between the gearboxes and the ones defined between the planetaries classes. This is done automatically by the top-down mapping algorithm, in this way enabling the mapping of the whole systems and all of their parts. The mappings between the planetaries are reused three times in mapping the gearbox classes to each other.

This algorithm also improves the modularization of the mappings. It allows defining mappings between individual systems which may be composed at runtime to map complex systems.

The top-down mapping algorithm was not the only one implemented. A bottom-up and a hybrid mapping algorithms were also implemented, but they did not seem appropriate for mapping systems. However, the top-down mapping algorithm has proved to be applicable for mapping system ontologies. It is most suitable for the cases in which the relationships between the instance form a tree, rather than a graph. For this case, a hybrid mapping algorithm is suitable, but this will require further investigations.

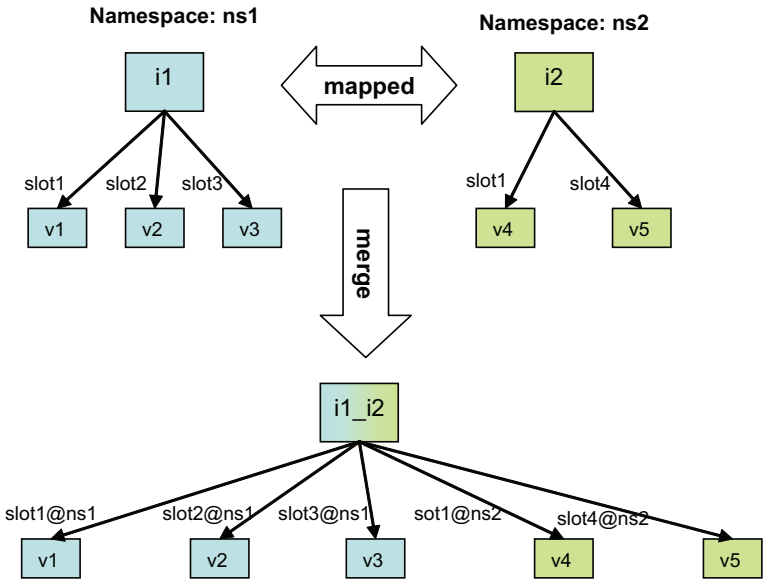

Figure 5.13.: Virtual instance merging. The merged instance has all the slot values of the mapped instance annotated with the namespace of the source instance.

5.4.4. Instance Merging

One common scenario is to have an aggregated view on two design models, without really merging them. This allows engineers to execute queries about properties of a system defined in another design model. For instance, a design model has a class *Car*, which is mapped to a class *Vehicle* from another design model. The second model defines the *weight* property of the class *Car*, which is missing from the first model. An engineer working on the first model might want to query for the weight of a car without integrating the two models. This can be realized by having a virtually aggregated model on which queries may be asked.

The virtual merging is supported in the mapping framework, by defining a predicate *merged* which returns a common instance of two instances defined in the two ontologies.

Merging two instances is done by first identifying the mapped instances, and then, for each mapped instance pair, a virtual instance is created, that has all the slot values attached to the two instances. This is shown in Figure 5.13.

The current version of merging mechanism does not perform a deep merging, but only a shallow merging. It is quite common for mapped instances to have slots with similar names. To avoid naming confusions, the namespace is used to annotate each slot of the merged instance.

5.4.5. Supporting Mapping Definition and Editing Process

The mapping framework also provides axioms that support the user in defining the mappings. For example, it provides a predicate, *appendableConcepts* that identifies which classes are likely to be mapped considering the mappings that have already been defined.

Another useful functionality is that the user can test at edit time the defined mappings. He can check what the effects of the mappings are on the instances and change them if they are not as expected.

The mapping framework also offers a set of predefined query templates that are very useful at edit time, like for example, getting the mapped instances, or the mapped classes, getting the possible mappings and so on.

The mapping framework is highly extensible. New mapping algorithms can be defined declaratively in $\mathcal{F}$LORA-*2* language and may be integrated at edit or run-time in the mapping framework. The consistency checking task has been implemented as axioms on top of the mapping axioms. In the same way, any other tasks that need to handle interrelated design models may be implemented.

5.5. Related Work

When trying to achieve the semantic interoperability between different ontologies, it is important to know the different types of mismatches that can occur between the ontologies. Klein [2001] and Chalupsky [2000] have analysed the different kinds of mismatches between ontologies. One category is meta-model level mismatches. They include syntactic mismatches, different languages, and different expressiveness of the languages. The second category are the ontology mismatches, which may be further split into conceptualization mismatches (between different conceptualizations of the same domain), and explication mismatches (which are mismatches in the way a conceptualization is specified).

Schema and ontology mappings can be used to support various kind of tasks. One of the most common one is information integration. In many cases an integrated view over a set of data sources has to be provided by some application. The user asks queries in a mediated schema, and the answers are given as an aggregation of the local answers of more data sources. The mappings are used in this scenario to define the relationships between the mediated schema and the local data source schema. There are several systems that use the centralized approach for information integration, like [Hammer et al., 1995] and [Madhavan et al., 2001]. For these cases the mappings are defined as views, either as global-as-view (GAV) or as local-as-view (LAV).

Very often, an application may need terms that are defined in different ontologies. Instead of building an ontology from scratch, it is more efficient (and cheaper) to create an ontol-

ogy by selecting and reusing portions of existing ontologies and composing them to form a new ontology. This approach has been taken by Mitra et al. [2001] which defines an algebra for ontology operations. A similar approach, but for models in general, is taken by Bernstein [2003], which defines the most common operations needed in model management, like: Match, Merge, Compose, etc.

An approach that is close to the one taken in this work is Prompt [Noy and Musen, 2000]. Prompt offers a toolset for mapping and merging ontologies, as well as doing structural diffs between different versions of an ontology. Prompt uses a mapping ontology to store the mappings between a source and a target ontology. Prompt is available as a plug-in in the Protégé ontology editor. MAFRA is another framework for defining mappings between ontologies on the Semantic Web [Maedche et al., 2002]. It employs several algorithms for mapping discovery, which are later stored in a bridging ontology.

Several surveys have investigated the ontology-based integration of information and the use of mappings for achieving interoperability, such as [Noy, 2004a; de Bruijn et al., 2004; Wache et al., 2001]. Different categories of tools have emerged from these surveys: tools for merging two ontologies (iPrompt [Noy, 2004b], Chimaera [McGuinness et al., 2000]); tools for defining a translation of ontologies (OntoMorh [Chalupsky, 2000]); mapping discovery tools (AnchorPrompt [Noy, 2004b], Glue [Doan et al., 2002], Observer [Mena et al., 1996], Mafra [Maedche et al., 2002]); and tools for mapping parts of ontologies based on an ontology algebra (Onion [Mitra et al., 2001]).

Another related research topic is defining the semantics of mappings. Madhavan et al. [2002] analyzed from a formal point of view the use of mappings for supporting the interoperability between different applications. He defined a general semantics for mappings between models in different formal languages. He also described the types of reasoning that can be done with the mappings, such as mapping composition and mapping inference.

6. Concept Validation

6.1. Introduction

This chapter presents two applications, which apply concepts from this work for improving activities in product design process.

Section 6.2 shows an application in requirements engineering, which has been implemented in a research project, Specification Driven Design (SDD), with a duration of 3 years. The goal of the project was to develop methods and tools that support the engineer in deriving an optimized design layout starting from a system specification, which is refined in iterative design steps. Checking and maintaining the consistency of the design layout throughout the design iterations was one of the main tasks in the iterative design process.

Section 6.3 presents a scenario from the collaborative development process in which different models of the same product where developed by several engineering teams in a parallel fashion. At certain milestones, the consistency of the two models had to be assessed and if necessary, changes from one model had to be propagated to the other model.

6.2. Improving the Requirements Engineering Process

This section shows how the requirements engineering process can be improved by using an ontology-based approach for the representation of requirements and of design models of technical systems that can be used in checking the consistency of a model in successive refinement steps.

6.2.1. Scenario Description

Requirements are the driving force of the development process and play an essential role in all development phases. After analyzing the customer requirements, a system specification is generated that is more detailed and structured than the user requirements. Out of the systems specification, a design layout is developed, which is optimized in successive refinements steps until a design solution is reached that fulfills all requirements.

One of the greatest challenge that must be confronted in this process is that requirements are usually stored as documents without any kind of formal representation. Design model parameters are constrained based on the requirements. It is a very hard task for the engineers to keep track of the dependencies between different parameters of the design model and to asses their consistency. The typical way of handling complex requirements and model parameters is to use Excel spreadsheets, which contain formulas over the parameters. In this way, it is possible to compute the values of some model parameters. However, this approach has serious limitations. If the parameters involved in different equations have circular dependencies, they cannot be computed in Excel. Another serious limitation is that it is impossible to track the history of design which is given by the successive refinement of parameter values.

The scenario that I will use to explain the application that has been built is in the context of the layout phase in the development of an engine. In the layout phase, the structure of the engine is already known, but optimizations of the parameters of the components are still performed. The goal is to obtain a design that is optimal with respect to some cost function and that does not violate any constraints.

In the layout scenario, the geometry of the motor has to be improved such that the compression ratio[1] of the engine will be increased. The compression ratio depends on a multitude of geometrical parameters of the engine.

Our task was to develop a requirement management system that will support:

- The stepwise refinement of the design parameters
- The consistency checking of the design based on the defined constraints
- In case that the design is inconsistent, meaning that at least two constraints are in conflict, the application should support the automatic solving of the conflict based on some cost function of the constraints
- Support the allocation of requirements to parts and the tracking the history of requirements
- Support for reuse of parts and requirements based on libraries

6.2.2. The Requirements Management System

Architecture

The architecture of the requirement management system is shown in Figure 6.1.

[1] The compression ratio is a very important parameter of the engine that describes the efficiency of the engine. The greater the value, the better the engine.

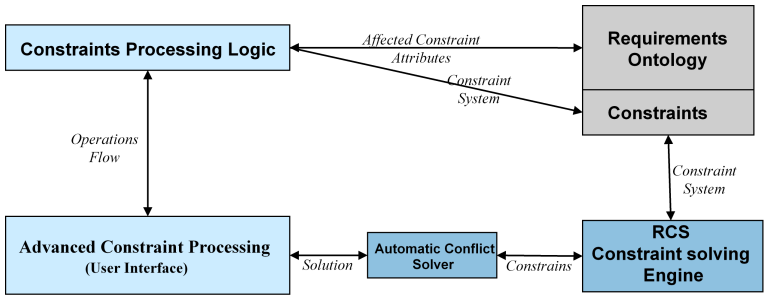

Figure 6.1.: The architecture of the requirement management system.

The *Requirements* ontology[2] provides the framework for modeling of requirements, for their allocation to parts of the systems, for attaching constraints to requirements and parts of the system and also keeping track of different versions of requirements. Systems and their components are represented as classes in the ontology. Classes have different attributes, which are modeled as slots. In order to handle the stepwise refinement of the attribute values, a new attribute type was developed, which is described in Section 6.2.2.

The constraint engine used in the architecture is the *Relational Constraint Solver* (RCS) [Mauss et al., 2002], that supports different types of constraints:

- linear equations and inequations, such as $2x + 3y < 0$. The coefficients may also be intervals with open or closed boundaries, e.g., $[2.8, 3.2)$.
- multivariate polynomial, trigonometric, and other non-linear interval equations, such as $x = sin(y)$, $x = sqrt(y)$.
- assignments of symbols to variables, such as $x = blue$
- disjunctions (*or*) and conjunctions (*and*) of the above relations, such as $x = red \vee x = blue$

The *Constraint Processing Logic* is responsible for the stepwise refinement of the design parameters. The *Automatic Conflict Solver* takes a conflict between two or more constraints as an input and generates a list of possible solution ordered by a cost function.

[2]The *Requirements* ontology presented in Chapter 4 has been developed based on the experiences gained from this project. The ontology used here also contains the definitions of components and constraints.

The Graphical User Interface

The graphical interface of the requirement management system (see Figure 6.2) has been implemented as a plug-in for Protégé[3] and provides support for:

- Viewing the attributes that are involved in a constraint system
- Viewing the constraint system
- Viewing the conflicting constraints (if the case)
- Editing the input values for any processing step and attribute
- Inconsistency detection and conflict solving

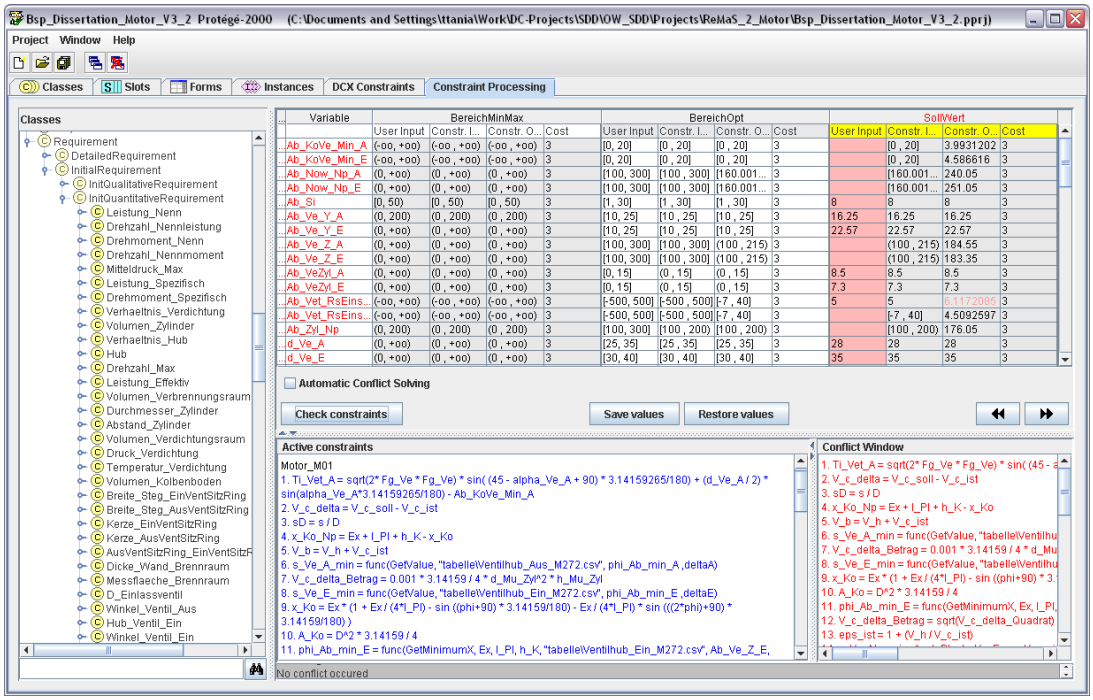

Figure 6.2.: The GUI of the requirement management system. The left part of the display shows the Requirements class hierarchy. The upper part shows a list of all variables involved in the constraint system. Each variable is associated to an attribute of a class. The bottom-left window shows a list of the constraints in the constraint system. The bottom-right panel shows a list of the constraints which are in conflict after executing the consistency checking.

[3] http://protege.stanford.edu

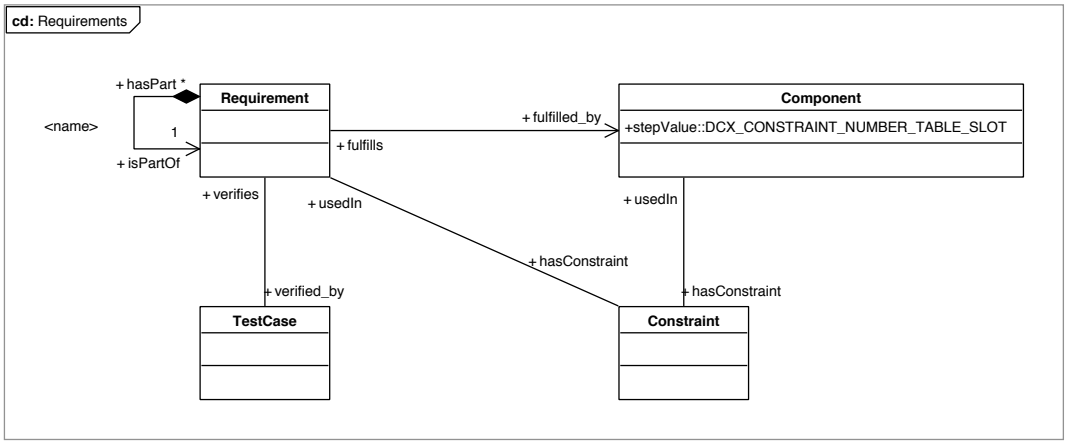

Figure 6.3.: An excerpt from the *Requirements* ontology.

The Requirements Ontology

The *Requirements* ontology is used to model the requirements and their interrelationships, such as *hasPart* – for the representation of the requirements decomposition – or *is_derived_into* – to represent the evolution of requirements. An excerpt from the *Requirements* ontology is shown in Figure 6.3. Constraints may be attached to requirements or to specific parts of a system, modeled as *Component*s.

The ontology supports the modeling of requirements and component templates by representing them as classes. A specific requirement can be modeled as an instance of a requirement class or as a subclass[4]. The *stepValue* attribute attached to the *Component* class is a complex attribute that stores the values of a simple attribute in different processing steps. The representation of the attribute is described in the next section.

By modeling requirements, components, constraints and their relationships explicitly, the ontology improved substantially the correctness of models. The engineers were not allowed to enter incorrect information, because the knowledge acquisition process was constrained and guided by the ontology. For example, an attribute of a requirement is the *status* describing the fulfillment of the requirement in the model, which can take only three predefined values (*passed*, *failed*, *not verified*) and which must have a value. When editing the requirements of a component, the engineer had to enter one of the allowed values for the *status* attribute. He was prevented from entering an invalid value for that attribute.

[4]The borderline between modeling something as a class or an instance is very fine. We have chosen to use the subclass modeling.

Managing the complex network of requirements and model parameters using this approach has proved to be much easier than it was with the existing approaches. Once the constraints were represented explicitly, the consistency of the model could be checked in an automated way by the constraint engine.

The modeling of the constraint attribute, which enabled the stepwise constraint processing deserves a special attention, and is described in the next section.

Representation of the Constraint Attributes

The attributes of a design model are refined in iterative steps until an optimal solution is reached. In each step the consistency of the design model is assessed. A design step is accomplished only if it is consistent. The processing of the next step cannot begin unless the previous step is consistent. Figure 6.4 shows an example of the iterative refinement steps. *BereichMinMax*[5], *BereichOpt* and *CatiaInput* are refinement steps, also known as *processing steps*. Each step is more restrictive than the previous one.

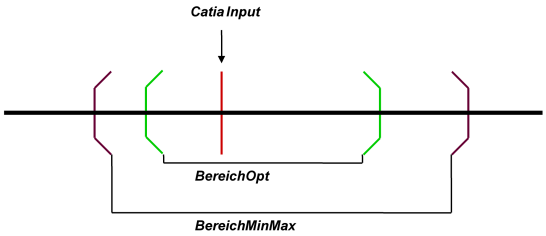

Figure 6.4.: Iterative refinement of the parameters values. Each step is more refined than the previous one.

In order to support a stepwise constraint processing, a new type of attribute[6] is defined, *:DCX_CONSTRAINT_NUMBER_TABLE_SLOT*, which stores information related to different processing steps. An attribute of this type is shown in Figure 6.5. The attribute has to store different values for different processing steps.

The attribute contains a number of processing steps (e.g., *BereichMinMax*, *CatiaInput*, etc.) and each processing step has several properties (e.g., *userInputValue*, *constrOutputValue*, etc.). For example, the processing step *BereichOpt* has for the property *userInputValue* the value $(150, 450)$.

A *processing step* corresponds to one of the steps in the constraint processing. In the example from Figure 6.5, there are four processing steps defined: *BereichMinMax*, *BereichOpt*, *CatiaInput* and *CatiaBack*.

[5] *Bereich* (in German) means *interval*.
[6] An attribute is a slot.

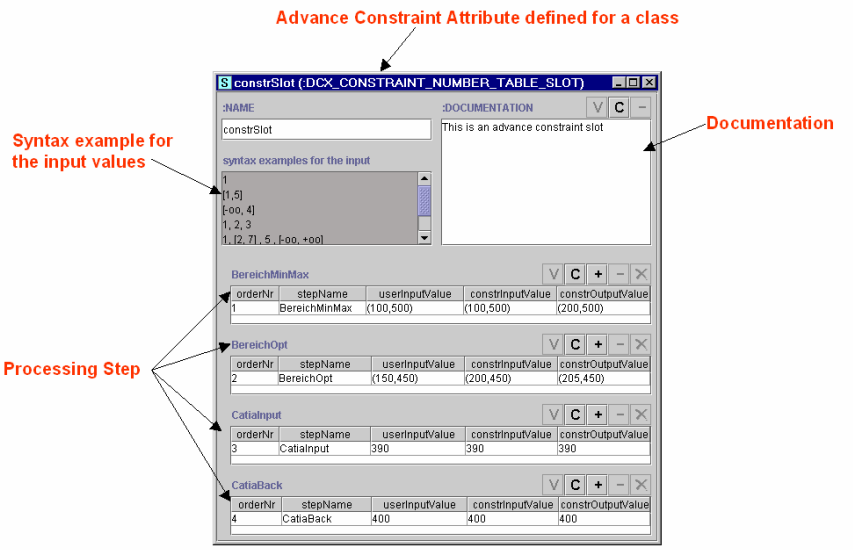

Figure 6.5.: Constraint attribute.

Each processing step has the following properties:

orderNr specifies in which order the steps are processed, e.g. *BereichOpt* has the $orderNr = 2$, this means that it is the second processing step.

stepName stores the name of the processing step.

userInputValue the input value given by the user for a step, e.g. for *BereichOpt*, the $userInputValue = (150, 450)$. This is the value that the user gives in as the "desired" value for a step.

constrInputValue is the input value that the constraint solver will get, i.e., the values which are used in the consistency checking. The *userInputValue* and the *constrInputValue* do not have necessariliy the same value. See next section for an explanation.

constrOutputValue is the output value computed by the constraint solver for a certain processing step.

priority is the priority for a processing step. (Not shown in Figure 6.5)

documentation is the documentation of a processing step represented as a string.

The next section explains how the properties of the processing steps are employed in the iterative refinement steps.

6.2.3. The Constraint Processing Logic

The constraint processing logic defines the flow of operations and data and describes how the information in one processing step affects the information in the next processing step. A rule is that each processing steps is more restrictive than its predecessor. Even more, the "real" input value of a step is the intersection of the user input value for that step and the output value of the predecessor step.

This can be formally described by:

$$constrInputValue_i = constrOutputValue_{i-1} \cap userInputValue_i$$

In this way, it is ensured that a processing step is more restrictive than its predecessor.

Each constraint attribute has a state that depends on which processing step is processed at a certain time. This is implemented by a counter that shows what processing step is currently active. For example, if the user is working in the processing step *BereichOpt*, then the current counter is 2, because *BereichOpt* is the second step of processing. After completing step two, the processing logic activates the next processing step (the one with *orderNr=3*).

The representation of the constraint attribute type is very flexible, so that it is straightforward to add a new processing step, a processing step property, or to change the name of a processing step.

6.2.4. The Automatic Conflict Solving

If the constraint system defined in the ontology is consistent, then a design solution for a particular processing step is obtained. However, if the resulting system of equations is inconsistent, then we have to relax (i.e., remove) some constraints in order to obtain a consistent state of the system.

The process of choosing what constraint to relax can be done manually by trying different alternatives until a solution is found. However, this process is very tiring and complex. Even if a constraint is relaxed, it is not guaranteed that the new constraint system is consistent.

In order to solve this problem, an algorithm for the automatic conflict solving has been developed. This algorithm provides the user with several conflict solving alternatives. It returns a list of constraints that can be relaxed in order to obtain a consistent system, sorted by the values of a cost function . The algorithm may also be optimized by different factors. The algorithm is presented in Appendix B.

6.2.5. Benefits of an Ontology-Based Approach

The requirements management system has been successfully employed in solving several design problems. The ontology-based approach has enabled the reuse of requirements and systems by defining a library of templates that can be instantiated in different design settings. The support for reuse is one of the major benefits that has been realized by using an ontology-based approach.

Another benefit was obtained by the structured information acquisition that prevents the user from entering incorrect information in a design model. The definitions of component templates are represented as a classes in the ontology. The classes serve as templates for building instances that represent a concrete design model. Therefore, all constraints defined in a template component (at class level) are checked for a concrete design model (at instance level). An intelligent model editing environment can interpret the constraints defined for the template components and prevent invalid input in a concrete design model.

Enabling consistency checking in iterative steps and documenting the history of parameter values was a huge improvement over the existing solutions. The constraint system for the scenario described in the previous sections had over 150 complex constraints between parameters of the design model, which were impossible to manage in Excel. The ontology-based approach helped in reducing the complexity of the product and to improve the design model quality by consistency checking in iterative refinement steps.

6.3. Maintaining the Consistency Between Different Design Models

6.3.1. Scenario Description

This scenario is part of the motivating scenario described in detail in Section 2.1. It shows how the consistency checking between different design models can be checked in a collaborative design process. For convenience, I will give in the following paragraph a brief description of the scenario.

The motivating scenario is taken from the development of an automatic transmission gearbox of a car. The automatic transmission is a very complex mechatronic component that contains mechanical, hydraulic, electrical and software parts, that need to interoperate in order to achieve the overall functionality of the product.

The development of the automatic transmission is made in a parallel fashion by different engineering teams. One team of engineers develops the geometrical model of the gearbox and another team develops the functional model of the same gearbox. The developed models are different in several ways. First, they are represented in different modeling languages. Second,

they have different conceptualizations of the gearbox: One models the geometrical characteristics of the product, while the other represents the gearbox as a composition of functional blocks coupled together. However, there are certain correspondences that interrelate the two design models. For instance, a parameter from the functional model, *ratio* of the gearbox is computed as a function of two parameters in the geometrical model. In the following, I will use the keyword *viewpoint* to refer to the geometrical or functional development branch.

The scenario requires that a framework should be implemented that will support the consistency checking between the two design models and the change propagation from one model to another.

I have developed a prototype based on the mapping framework described in Chapter 5. I will show in the following sections how the prototype may be used to check the consistency between the geometrical and functional model.

6.3.2. Mapping Between Library of Components

The design of the functional and geometrical models is made using model libraries. A model is composed by instantiating template components from the model libraries and by interrelating them. Figure 6.6 shows this situation.

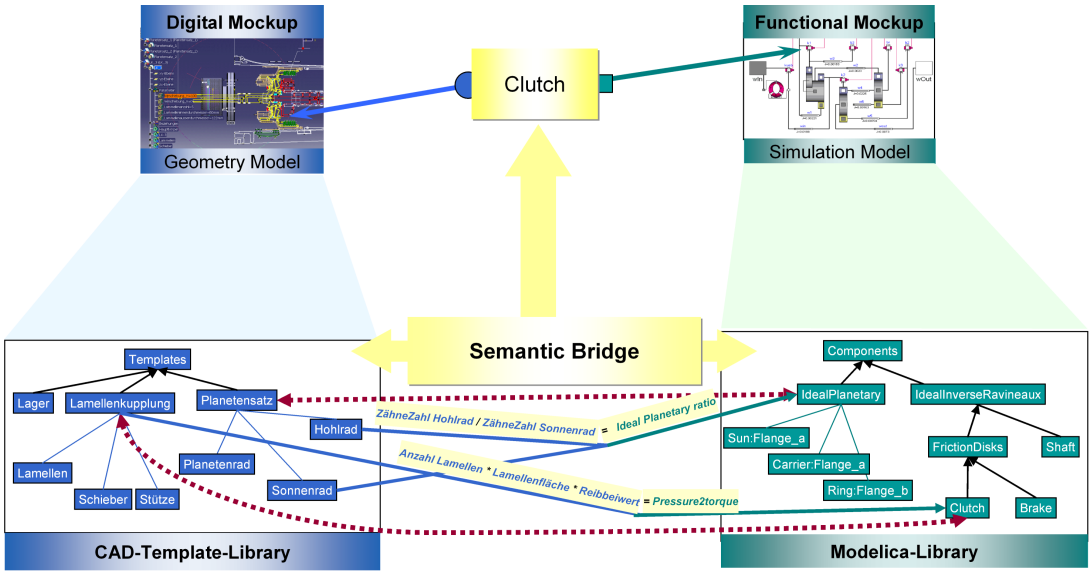

Figure 6.6.: Mappings between component libraries in two viewpoints.

Engineers build geometrical models by instantiating template components from a geometrical model library, while functional engineers, instantiate components from a functional model

library. The libraries and the design models are represented in different modeling languages. The names used for corresponding components in the two model libraries are different. The CAD library uses German names, whereas the functional library uses English names for the components.

However, the design models built from these component libraries still have one thing in common: They represent the same product from different perspectives. As a consequence, there will be correspondences between the structures and parameters of the models in the two libraries, shown in Figure 6.6. The correspondences (also known as mappings) are defined between components of the library. In this way it is possible to reuse them between all instances of corresponding templates.

The automatic transmission gearbox is a complex system composed of several parts, which are in turn decomposed in other subparts, and so on. The mappings are defined between template components. In order to map a system it is necessary that the mapping framework is capable of composing mappings between components of a system to infer the mapping of the system.

The mapping framework should support the following operations:

- The definition of mappings between library components in different viewpoints
- Reuse of template mappings for different instantiations of the design models
- Computing the mapping of systems out of the mappings of their parts
- Consistency checking between design models from different viewpoints based on the predefined mappings

6.3.3. Defining the Mappings

The mapping framework presented in Section 5.2 has been used to define the mappings between the design models.

In order to define the mappings, the design models in the two viewpoints have to be semantically enriched, as described in Section 5.2.2. The enrichment is done by exporting the data from the design models to instances of the local ontologies. The local ontologies already contain the class definitions corresponding to the template components in the library of models in the two viewpoints.

The local ontologies corresponding to the template libraries both include the *Components* ontology presented in Chapter 4. In this way, a common vocabulary for the two viewpoints is defined, which already provides a starting point for finding the correspondences between the ontologies. For example, a *Component* in one ontology is typically mapped to another

Component in the other ontology. The *Plantensatz*[7] class in the geometrical viewpoint is mapped to the *IdealPlanetary*.

The mappings are interrelating components from the two ontologies. Some examples are shown in Section 5.4.3. The mapping algorithm defined in Section 5.4.2 is used to compose the mappings between the components in order to map the "top-level" systems, such as the *Getriebe* and *Gearbox*. The top-down mapping algorithm, presented in Section refsect:Mappings:MappingAlg, was executed in $\mathcal{F}$LORA-*2* and mapped successfully the two systems and their parts.

6.3.4. Checking the Consistency

The mappings themselves are not a big help for the engineers. Their goal is to check the consistency of the design models based on the correspondence that have been defined. Figure 6.6 shows some of the correspondences between the parameters of the two models that need to hold if the two models are consistent.

The mathematical relationships between the parameters in the design models are modeled as constraints attached to the mappings between concepts in the ontologies. For example, the *ratio* of the *IdealPlanetary* is computed out of two parameters of the parts of the corresponding class *Planetensatz*. The *Planetensatz* has as parts a *Sonnenrad* and a *Hohlrad* which have each defined a number of teethes for the gears (in German, *ZaehnenZahl*). The constraint between the ratio and the number of teethes of the gears is shown in the GUI of the mapping editor in Figure 6.7 and it is defined below:

$$IdealPlanetary.ratio = \frac{Planetensatz.hohlrad.Zaehnenzahl}{Planetensatz.sonnenrad.Zaehnenzahl} \tag{6.1}$$

The constraint is defined in the context of a mapping and it is checked whenever the mapping is used. For example, this constraint is checked three times in the mapping between the *Getriebe* and *Gearbox* systems, because they contain three corresponding planetaries.

The constraint is exported as a predicate that uses the *mapped* predicate defined in Section 5.4.3 to find the instance of *Planetensatz* correspondent to an instance of *IdealPlanetary*. The predicate returns true if the constraint holds between the corresponding instances and false otherwise.

6.3.5. Mapping Between Three Ontologies

The mapping framework has also been used in another scenario, in which the geometrical and the functional design models had to be integrated with another data source containing

[7] *Planetensatz* in German means *Planetary* in English

documentation about best practices for the design of automatic gearboxes. The documentation of best practices was also defined as an ontology, which modeled the gearbox and its decomposition together with documentation of best practices attached to each component.

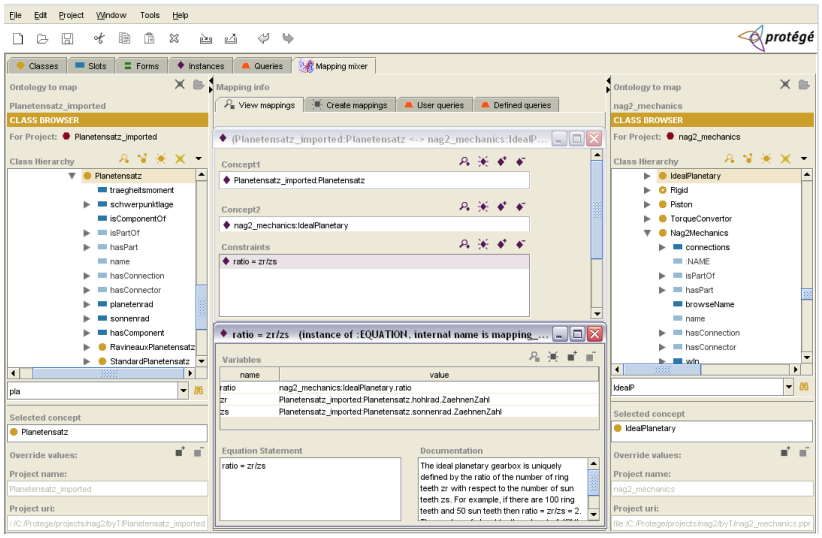

Figure 6.7.: The Mixer plug-in for Protégé used for mapping two ontologies. The mapping between the *Planetensatz* and *IdealPlanetary* are shown together with the attached constraint.

The scenario for this application was to present to the engineers an integrated view of the best practices together with the functional and geometrical views of the gearbox and the documentation of their correspondence (if the case). This has been realized in a web-based application that allowed the navigation along the system decomposition structure defined in the best practices ontology (see Figure 6.8). If geometrical or functional information was available from the corresponding viewpoints ontology, it was displayed in the web-based application, together with the correspondences between the components in the two viewpoints (if any defined). This was realized by a three-way mapping between the documentation, the functional and geometrical ontologies, which were used together in the mapping framework. The reasoner evaluated at runtime the mappings defined between the ontologies in order to display in the web application the information regarding the correspondences between geometrical and functional components.

6.3.6. Benefits of an Ontology-Based Approach

The benefits for the design process came both from modeling the technical systems using ontologies and from defining the correspondences between the design models explicitly, which contributed to improving the consistency of a design model and between the models.

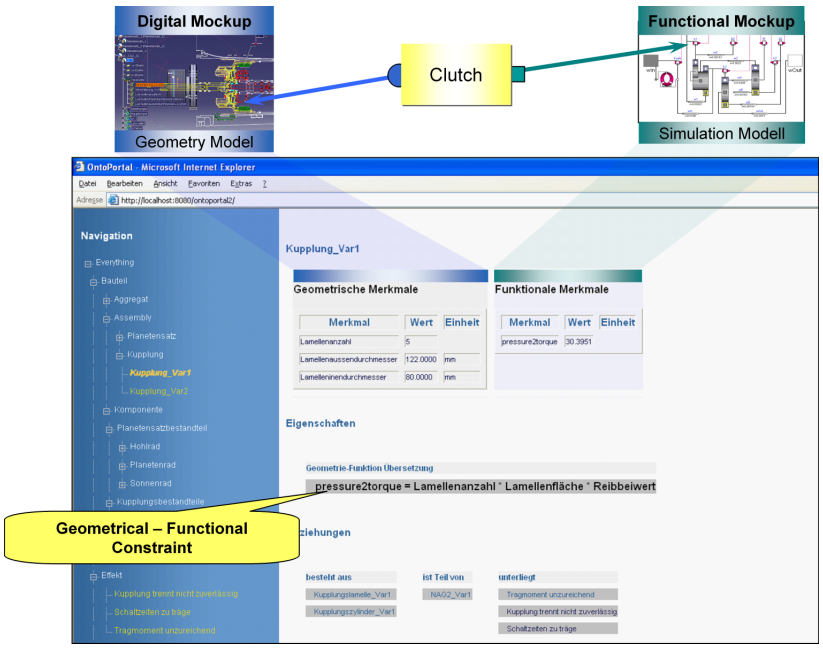

Figure 6.8.: A web-based application which shows the integration of three ontologies: a best-practice ontology, a functional and a geometrical ontology.

The ontology-based approach for modeling the functional and geometrical design models allows the definition of meta-rules that constrain correct models and which can be checked at model building time. For example, certain types of components cannot be connected together in a functional model. This type of constraint can be easily formulated as an axiom in the functional ontology.

The ontology-based representation also improved the model management process. For instance, it was straightforward to search for components in a model library that have a certain type of connectors by using a simple query. The design models can also be enriched with other type of semantic relationships that improve the documentation of the model. Different types of meta-data can be attached to components in the ontologies, such as, provenance, version, release status, etc., that are very important in the development process.

The explicit definition of the correspondence between the components in two design models allowed an automated consistency checking, which was not possible before. Although, the mapping scenario only showed how the mapping framework can be used for consistency checking and knowledge management, the defined mappings may also be employed for propagating changes from one model to another or generating a model skeleton from an existing model.

7. Conclusions and Future Work

The goal of this thesis is to bring a contribution to the collaborative model-based development by improving the design model quality and enabling the consistency checking between several design models. In order to achieve this goal, I have investigated the following three research questions:

Q1. How is the consistency of a design model and the consistency between models in different viewpoints defined?

Q2. How can the representation of the current design models be improved so that it will support a more efficient building of consistent design models?

Q3. How can the consistency between different design models be checked in an automated way?

Chapter 3 gives a formal definition of the consistency of design models by adapting concepts from RM-ODP. Chapter 4 proposes four ontologies for the representation of common patterns of design models in the engineering domain, such as part-whole relations, connections, constraints and requirements. Chapter 5 defines a mapping framework, which can be used in tasks that require the interoperation of two or more design models, such as consistency checking, change propagation and model skeleton generation. Chapter 6 shows practical use cases from the automotive industry in which the concepts have been implemented and validated.

7.1. Conclusions

In this section, I present the major conclusions and the contributions of this thesis. The discussion is structured around the three major topics that were introduced by the research questions.

The general conclusion of the thesis is that ontologies can be used to enhance both the quality of the design models and also the tasks that support the design process. This has been demonstrated in the use cases presented in Chapter 6. However, there are major obstacles in adopting ontologies on the large scale and in a productive setting that are illustrated below.

1. Useful ontologies will be large [Borst, 1997]. A good methodology for managing ontologies is needed, one that supports modularization and also different ways of interrelating ontologies.

2. The tools available for managing ontologies are not really scalable. This becomes a serious issue in editing and maintenance of large ontologies.

3. Ontologies need to be integrated seamlessly in the existing IT infrastructure. This has proven to be a challenge in many situations, because many engineering tools are not open and do not allow an easy integration with other tools.

4. Ontologies are hard to sell. This comes from the fact that *ontology* is a very abstract notion and it is hard for the engineers to understand it, because they typically think very concretely. Also, there are not many success stories that demonstrate clearly the benefits that ontologies bring in an application.

Nevertheless, ontologies **do** help in the engineering development process. The two use cases described in this thesis and other projects, which were not discussed here due to the space limits, demonstrate this. Ontologies have been used successfully for checking the consistency between different design models or to propagate the changes from one model in another. They have also been employed in a requirements management process and supported the task of iterative refinements of a specification until an optimal design has been reached.

In the following sections, I will sum up the conclusions for each of the investigated research questions.

7.1.1. Consistency Between Design Models

Consistency between design models is essential in a collaborative development process. Ignoring inconsistencies and incompatibilities between the design models causes huge correction costs and great time delays. In Chapter 3, I have investigated the consistency concept for design models from the perspective of the Reference Model for Open Distributed Processing (RM-ODP) which already defines the main concepts needed in a distributed development. RM-ODP defines the relationships that may exist between specifications and realizations, such as consistency, compliance and conformance.

In order to be able to reuse the RM-ODP relationships, I had to adapt the concepts in the engineering design process to the RM-ODP development concepts. Then, I could apply the notion of consistency between specifications to the engineering design models.

However, the engineering design process diverges at some point from the RM-ODP development process. Design models in engineering are built first out of system specifications that

have been refined from customer specifications. In this role, a design model is as a realization that can be checked for conformance against the specification. However, in later design phases, when the model has reached a certain maturity level, the design model changes its role and becomes a specification for building a real prototype.

I have also discussed the challenges that must be tackled when checking the consistency of two design models. The most important one is that the models have different conceptualization of the product according to their purpose and viewpoint on the world. I have identified different types of conflicts that might occur between the conceptualizations of two domains. I have proposed using ontologies for solving the conceptualization mismatches. This approach was elaborated in Chapter 4. After taking in consideration the different conceptualizations of the viewpoints, I gave a formal definition of consistency between design models using concepts from distributed first order logic.

7.1.2. Engineering Ontologies

The modern development process in the engineering domain is based on Systems Engineering, which defines the system development phases starting from requirements acquisition until product testing. The central components of this method are the systems. The systems engineering method is applied in all development branches in the collaborative design process.

This brings a common layer in the representation of the design models from different viewpoints. A design model will typically contain recurring representation patterns, such as part-whole relationships, connections, constraints and requirements. As a representation for the design models, I have proposed ontologies, that provide rich and formal descriptions of the concepts in a domain. I have discussed the need for ontologies and their benefits when employed in solving different engineering tasks.

I have defined five engineering ontologies, which I have called *Engineering* ontologies, in a frame based representation system. The ontologies provide a common representation for the recurring modeling patterns in the design models from different development branches. The *Engineering* ontologies are enumerate below:

- The *Components* ontology – defines the part-whole relationship and is used to represent the decomposition structure of systems.
- The *Connections* ontology – defines the topology of a system and describes how components of a system are connected to each other.
- The *Systems* ontology – used to describe systems by extending the *Components* and *Connections* ontologies.
- The *Requirements* ontology – used for representing requirements and their various relationships, as well as the relationships between requirements and systems.

- The *Constraints* ontology – used for defining constraints on different model elements, for instance, constraints on model parameters expressed as mathematical relationships.

I have also developed a generic part-whole modeling pattern, which is appropriate for representing the decomposition structures. The fact that the *Components* ontology defines at a very abstract level the part-whole relationship enables its reuse in different ontologies: In the *Connections* ontology – for the decomposition of connectors, and in the *Requirement* ontology – for the decomposition of requirements.

7.1.3. Mapping Framework

One of the contributions of this thesis is the development of a mapping framework, which allows the definition of correspondences between several ontologies. The framework has been specifically designed and optimized for the tasks to be solved in the design process, such as consistency checking and change propagation. An important aspect that has been considered in its design is the fact that the ontologies to be mapped describe design models from different development branches by using common modeling patterns. This is ensured by using the engineering ontologies as a common upper level for the ontologies to be mapped.

The mapping framework allows the definition of correspondences (mappings) between different ontologies. Since it has been specifically developed for mapping design model ontologies, it supports a mapping pattern that is very common for engineering design ontologies, i.e., mapping between paths in different ontologies.

The mappings are stored as instances in a *Mapping* ontology, which reuses concepts defined in the *Constraints* ontology by importing it. The *Constraints* ontology defines concepts for modeling constraints and paths along the structure of the concepts. I have also identified the requirements for the *Mapping* ontology, which has been later developed under consideration of these requirements.

I have defined the concept of *template mappings*, which describes a mapping between two general model library components. The mapping framework supports the mapping composition, so that complex systems may be mapped in a compositional way out of simple mappings between individual components.

The mapping framework may be used for performing consistency checking, change propagation between the models, or model skeleton generation. The second use case in Chapter 6 shows an example of consistency checking, in which the mapping framework has been implemented and validated.

7.2. Future Work

The concepts presented in this thesis have been shown to bring benefits in real world applications. However, there are still many aspects that can be improved and need further research and investigations. I will illustrate some of them in the following.

Extensions of the engineering ontologies The engineering ontologies provide basic modeling patterns for design model ontologies. One of them is a very general type of *part-of* relationship that is used in the decomposition of systems, connectors and requirements. However, in many cases the part-of relationship also has other properties than the ones defined in this work. For this case, other types of part-of relationships should be defined. It is also worth investigating what other modeling patterns are common for engineering design models and to propose ontological representations for them in a similar fashion as for the part-whole modeling pattern developed in this work.

Connections between different components are realized by connectors. The connectors may be composite components, which contain other connectors as well. The current representation of connectors does not allow to define how the internal structures of two composite connectors are related to each other. However, this seems to be useful in many practical cases. An extension of the *Connectors* ontology could provide this feature.

One of the most complex concept representations in the *Connections* ontology are the connections, which have to be represented both at class level (to model connections between classes of models) and at instance level (to represent the connections between instantiated design models). The connection instantiation has to be done by a reasoner using additional axioms, because the representation formalism did not support this type of instantiation. Other representation formalisms should be investigated that support the representation of concepts, which are at the same time a class and a property, like it is the case with connection objects.

Extending the mappings framework One necessary extension of the *Mapping* ontology is to support the additions of different kind of meta-information on the mapping between two concepts. For example, in a change propagation scenario, documenting the change rationale would bring many benefits to the engineers.

The *Mapping* ontology allows mappings on paths in the ontologies. This has proven to be very useful when mapping between systems that have a well defined structure. However, this mapping pattern will not work for all cases. Identifying new mapping patterns and the scenarios in which they can be applied is a future research direction.

The mapping framework supports the virtual instance merging, by merging all the properties of two mapped instances into a virtual instance. However, the current algorithm does only a

superficial merging, and it might be the case that an instance will have two equivalent properties attached. An algorithm for the deep merge is needed, which also takes into consideration the mapping between the properties of instances in the virtual merging.

The mapping algorithms can be very easily encoded in a declarative way in the form of $\mathcal{F}$LORA-*2* axioms. Different mapping algorithms have been developed in this thesis (top-down, bottom-up, hybrid), but only the top-down algorithm proved to be suitable for mapping system structures. It is worth investigating other types of algorithms and the use cases in which they may be applied.

A. Mapping axioms

```
%instance level mapping axioms

mappedConcept(C1, C2) :- _:'Mapping'[concept1 -> C1, concept2 -> C2].

%mapped(?instance1, ?module1, ?instance2, ?module2).
mapped(I1, M1, I2, M2) :- mapped(I1, M1, I2, M2, unidir).

%Allows one to specify the mode: top-down, bottom-up, or mixed
%mapped(?instance1, ?module1, ?instance2, ?module2, ?mode).
mapped(I1, M1, I2, M2, bidir) :-
        mappedTopDown(I1, M1, I2, M2, bidir).
mapped(I1, M1, I2, M2, unidir) :-
        mappedTopDown(I1, M1, I2, M2, unidir).
mapped(I1, M1, I2, M2, bidir) :-
        mappedBottomUp(I1, M1, I2, M2, bidir).
mapped(I1, M1, I2, M2, unidir) :-
        mappedBottomUp(I1, M1, I2, M2, unidir).
mapped(I1, M1, I2, M2, topdown) :-
        mappedTopDown(I1, M1, I2, M2, unidir).
mapped(I1, M1, I2, M2, bottomup) :-
        mappedBottomUp(I1, M1, I2, M2, unidir).


%mode allows forcing to only use top-down or mixed mode
%mappedTopDown(?instance1, ?module1, ?instance2, ?module2, ?Mode).
mappedTopDown(I1, M1, I2, M2, bidir) :-
        mappedConcept(Concept1:'Concept', Concept2:'Concept'),
        path(Ep1, I1, Concept1, M1),
        path(Ep2, I2, Concept2, M2),
        mapped(Ep1, M1, Ep2, M2, bidir).
mappedTopDown(I1, M1, I2, M2, unidir) :-
        mappedConcept(Concept1:'Concept', Concept2:'Concept'),
        path(Ep1, I1, Concept1, M1),
        path(Ep2, I2, Concept2, M2),
        mapped(Ep1, M1, Ep2, M2, topdown).
%This predicate mapps instances by using compound concepts. (see appendable\5)
mappedTopDown(I1, M1, I2, M2, bidir) :-
        mappedConcept(Concept1:'Concept', Concept2:'Concept'),
        path(Ep1, EndI1, Concept1, M1),
        path(Ep2, EndI2, Concept2, M2),
        mapped(Ep1, M1, Ep2, M2, bidir),
```

```
        appendable(Concept1, NextConcept1, LinkSlot1, _Namespace1, M1),
        appendable(Concept2, NextConcept2, LinkSlot2, _Namespace2, M2),
        mappedConcept(NextConcept1, NextConcept2),
        instanceSlotValue(EndI1, LinkSlot1, SecondEp1, M1),
        instanceSlotValue(EndI2, LinkSlot2, SecondEp2, M2),
        path(SecondEp1, I1, NextConcept1, M1),
        path(SecondEp2, I2, NextConcept2, M2).
mappedTopDown(I1, M1, I2, M2, unidir) :-
        mappedConcept(Concept1:'Concept', Concept2:'Concept'),
        path(Ep1, EndI1, Concept1, M1),
        path(Ep2, EndI2, Concept2, M2),
        mapped(Ep1, M1, Ep2, M2, topdown),
        appendable(Concept1, NextConcept1, LinkSlot1, _Namespace1, M1),
        appendable(Concept2, NextConcept2, LinkSlot2, _Namespace2, M2),
        mappedConcept(NextConcept1, NextConcept2),
        instanceSlotValue(EndI1, LinkSlot1, SecondEp1, M1),
        instanceSlotValue(EndI2, LinkSlot2, SecondEp2, M2),
        path(SecondEp1, I1, NextConcept1, M1),
        path(SecondEp2, I2, NextConcept2, M2).


%mappedBottomUp(?instance1, ?module1, ?instance2, ?module2, ?mode).
mappedBottomUp(I1, M1, I2, M2, bidir) :-
        mappedConcept(Concept1:'Concept', Concept2:'Concept'),
        path(I1, Ep1, Concept1, M1),
        path(I2, Ep2, Concept2, M2),
        mapped(Ep1, M1, Ep2, M2, bidir).
mappedBottomUp(I1, M1, I2, M2, unidir) :-
        mappedConcept(Concept1:'Concept', Concept2:'Concept'),
        path(I1, Ep1, Concept1, M1),
        path(I2, Ep2, Concept2, M2),
        mapped(Ep1, M1, Ep2, M2, bottomup).
%This predicate mapps instances by using compound concepts. (see appendable\5)
mappedBottomUp(I1, M1, I2, M2, bidir) :-
        mappedConcept(Concept1:'Concept', Concept2:'Concept'),
        path(StartI1, Ep1, Concept1, M1),
        path(StartI2, Ep2, Concept2, M2),
        mapped(Ep1, M1, Ep2, M2, bidir),
        appendable(PreviousConcept1, Concept1, LinkSlot1, _Namespace1, M1),
        appendable(PreviousConcept2, Concept2, LinkSlot2, _Namespace2, M2),
        mappedConcept(PreviousConcept1, PreviousConcept2),
        instanceSlotValue(LastInstance1, LinkSlot1, StartI1, M1),
        instanceSlotValue(LastInstance2, LinkSlot2, StartI2, M2),
        path(I1, LastInstance1, PreviousConcept1, M1),
        path(I2, LastInstance2, PreviousConcept2, M2).
mappedBottomUp(I1, M1, I2, M2, unidir) :-
        mappedConcept(Concept1:'Concept', Concept2:'Concept'),
        path(StartI1, Ep1, Concept1, M1),
        path(StartI2, Ep2, Concept2, M2),
        mapped(Ep1, M1, Ep2, M2, bottomup),
```

```
        appendable(PreviousConcept1, Concept1, LinkSlot1, _Namespace1, M1),
        appendable(PreviousConcept2, Concept2, LinkSlot2, _Namespace2, M2),
        mappedConcept(PreviousConcept1, PreviousConcept2),
        instanceSlotValue(LastInstance1, LinkSlot1, StartI1, M1),
        instanceSlotValue(LastInstance2, LinkSlot2, StartI2, M2),
        path(I1, LastInstance1, PreviousConcept1, M1),
        path(I2, LastInstance2, PreviousConcept2, M2).


%Collects all the mappings, and for each, the reunion of slots and values is computed
%merged(?I1, ?M1, ?I2, ?M2, ?Slots)
merged(I1, M1, I2, M2, Slots) :-
        mapped(I1, M1, I2, M2, _),
        findall([SlotName1, Value1, M1], instanceSlotValue(I1, SlotName1, Value1, M1), SlotList1),
        findall([SlotName2, Value2, M2], instanceSlotValue(I2, SlotName2, Value2, M2), SlotList2),
        append(SlotList1, SlotList2, Slots)@prolog(basics).

%merged(?I1, ?M1, ?I2, ?M2, ?Slot, ?Value, ?SlotModule)
merged(I1, M1, I2, M2, Slot, Value, M1) :-
        mapped(I1, M1, I2, M2),
        instanceSlotValue(I1, Slot, Value, M1).
merged(I1, M1, I2, M2, Slot, Value, M2) :-
        mapped(I1, M1, I2, M2),
        instanceSlotValue(I2, Slot, Value, M2).

path(EntryPoint, EntryPoint, Concept, Module) :-
        EntryPoint:Class[]@Module,
        Concept:'Concept'[tnot pathRef -> _, classRef -> ClassRef],
        classFromClassRef(Class, ClassRef).
path(EntryPoint, GoalInst, Concept, Mod) :-
        Concept:'Concept'[pathRef -> PathRef],
        recPathRef(EntryPoint, PathRef, GoalInst, Mod).

%recPathRef(?CurrInst, ?CurrPathRefInst, ?GoalInst, ?Module)
recPathRef(CurrInst, CurrPathRefInst, GoalInst, Mod):-
        CurrPathRefInst:'PathRef'[tnot pathRef -> _, slotName -> CurrSlotName],
        instanceSlotValue(CurrInst, CurrSlotName, GoalInst, Mod).
recPathRef(CurrInst, CurrPathRefInst, GoalInst, Mod):-
        CurrPathRefInst:'PathRef'[pathRef -> NextCurrPathRefInst, slotName -> CurrSlotName],
        instanceSlotValue(CurrInst, CurrSlotName, NextCurrInst, Mod),
        recPathRef(NextCurrInst, NextCurrPathRefInst, GoalInst, Mod).


instanceSlotValue(Instance, SlotName, SlotInstance, Mod) :-
        Instance[SlotName -> SlotInstance]@Mod.
instanceSlotValue(Instance, SlotName, SlotInstance, Mod) :-
        Instance[SlotName ->> SlotInstance]@Mod.
instanceSlotValue(Instance, SlotName, SlotInstance, Mod) :-
        Instance[SlotName -> InstanceList]@Mod,
        eqMember(SlotInstance, InstanceList).
```

```
classSlotAllowedValue(Class, SlotName, AllowedValue, Module) :-
        classDirectSlotAllowedValue(Class, SlotName, AllowedValue, Module).
classSlotAllowedValue(Class, SlotName, AllowedValue, Module) :-
        classSlotAllowedValueInherited(Class, SlotName, AllowedValue, Module).
classDirectSlotAllowedValue(Class, SlotName, AllowedValue, Module)
:-
        Class:_[SlotName => AllowedValue]@Module.
classDirectSlotAllowedValue(Class, SlotName, AllowedValue, Module)
:-
        Class:_[SlotName =>> AllowedValue]@Module.
classDirectSlotAllowedValue(Class, SlotName, AllowedValue, Module)
:-
        Class:_[SlotName => AllowedValuesList]@Module,
        eqMember(AllowedValue, AllowedValuesList).
classSlotAllowedValueInherited(Class, SlotName, AllowedValue,
Module) :-
        Class:_[SlotName *=> AllowedValue]@Module.
classSlotAllowedValueInherited(Class, SlotName, AllowedValue,
Module) :-
        Class:_[SlotName *=>> AllowedValue]@Module.
classSlotAllowedValueInherited(Class, SlotName, AllowedValue,
Module) :-
        Class:_[SlotName *=> AllowedValuesList]@Module,
        eqMember(AllowedValue, AllowedValuesList).

%This implementation of eqMember has never been tested, so it may not be good.
%eqMember(Instance, Instance).
eqMember(Instance, InstanceList):-
        member(Instance, InstanceList)@prolog(basics).

%class level

%subclassOf(?SubClass, ?Class, ?Module).
subclassOf(SubClass, Class, Module) :-
        SubClass::Class@Module.
subclassOf(SubClass, Class, Module) :-
        subclassOf(SubClass, IntermClass, Module),
        IntermClass::Class@Module.
%mappedClasses(?Class1, ?Namespace1, ?Module1, ?Class2, ?Namespace2, ?Module2).
mappedClasses(Class1, Namespace1, Module1, Class2, Namespace2,
Module2) :-
        _:'Mapping'[concept1 -> Concept1, concept2 -> Concept2],
        endsConcept(Concept1, Class1, Namespace1, Module1),
        endsConcept(Concept2, Class2, Namespace2, Module2),
        tnot Module1 = Module2.

%endsConcept(?Concept, ?Class, ?Namespace, ?Module)
%In this predicate, the fact that the class is defined in the specified module is
%checked before class is instantiated, because otherwise that statement fails. This
```

```
%way is slower, but works.
endsConcept(Concept, Class, Namespace, Module) :-
        Concept:'Concept'[tnot pathRef -> _, classRef -> ClassRef],
        ClassRef[namespace -> Namespace],
        Class::_[]@Module,
        classFromClassRef(Class, ClassRef).
endsConcept(Concept, Class, Namespace, Module) :-
        Concept:'Concept'[pathRef -> PathRef, classRef -> ClassRef],
        ClassRef[namespace -> Namespace],
        endsPathRef(PathRef, Class, Module).

%endsPathRef(?PathRef, ?ClassName, ?Module)
endsPathRef(PathRef, ClassName, Module) :-
        PathRef:'PathRef'[tnot pathRef -> _, slotName -> SlotName, pathRestriction ->> PathRestriction],
        classFromClassRef(PathRestrictionClass, PathRestriction),
        classSlotAllowedValue(PathRestrictionClass, SlotName, ClassName, Module).
endsPathRef(PathRef, ClassName, Module) :-
        PathRef:'PathRef'[pathRef -> NextPathRef],
        endsPathRef(NextPathRef, ClassName, Module).

%startsConcept(?Concept, ?ClassName, ?Namespace).
startsConcept(Concept, ClassName, Namespace) :-
        Concept:'Concept'[classRef -> ClassRef],
        ClassRef:'ClassRef'[namespace -> Namespace, className -> ClassName].

%Two concepts are appendable (unidirectional relation) if it is discovered that
%the ending class of the first concept it is similar to the start class of the second one.
%The two concepts share the same namespace, and classes are computed in the same module.
%appendable(?Concept1, ?Concept2, ?LinkSlot, ?Namespace, ?Module).
appendable(Concept1, Concept2, LinkSlot, Namespace, Module) :-
        Concept1:'Concept'[pathRef -> _],
        endsConcept(Concept1, EndClassName, Namespace, Module),
        classSlotAllowedValue(EndClassName, LinkSlot, StartClassName, Module),
        startsConcept(Concept2, StartClassName, Namespace).


%classFromClassRef(?ClassName, ?ClassRef).
classFromClassRef(ClassName, ClassRef) :-
        ClassRef[className -> ClassName].

%equalsClassRef(?ClassRef1, ?ClassRef2).
equalsClassRef(ClassRef1, ClassRef2) :-
        ClassRef1[className -> ClassName, namespace -> Namespace1],
        ClassRef2[className -> ClassName, namespace -> Namespace2],
        equalsNamespace(Namespace1, Namespace2).

%equalsNamespace(?Namespace1, ?Namespace2).
equalsNamespace(Namespace1, Namespace2) :-
        Namespace1['URL' -> URL, kbName -> KbName],
        Namespace2['URL' -> URL, kbName -> KbName].
```

B. The conflict solving algorithm

The algorithm uses the following variable names:

`solutions`: - the list of all found solutions so far; an element in the list is a list of the constraints that need to be relaxed to get the solution of the CS. The list is sorted according to a defined metric (or priority).

`candidates`: - the list of incomplete solutions that are waiting to be expanded; an element in the list is a list of the constraints that need to be relaxed to get a possible solution. The list is sorted according to a defined metric (or priority).

`currentCandidate`: - an element from candidates that is currently expanded

`constraintSystem`: - the constraint system to be solved

`maxDepth`: - the maximum depth until where the algorithm should search (the maximum number of variables to set free)

`cost`: - the cost of a solution or partial solution, computed by a metric function, or cost of a single constraint

```
algorithm SolveConflicts

input: constraintsSystem
output: solutions

{
solutions = empty
candidates = empty
currentCandidate = empty

while (size of solutions < maxNoSols) and (curentCandidate not empty) {

    currentCandidate = the first element of candidates (remove it from candidates)

    configure the constraintSystem to remove the constraints from currentCandidate

    solve constraintSystem

    if (constraintSystem is inconsistent) {
        newCandidates = the sets formed by expanding the currentCandidate
                        with the new conflicts of the constraint solver
```

```
        add in order of costs the elements of newCandidates to candidates, if candidates
        does not contain a superset of the element to be added, and if the depth of the
        solution does not exceed the maximum depth


    } else  // constraintSystem is consistent
    {
        add currentCandidate to solutions
    }
} //end while

} //end algorithm
```

List of Figures

Bibliography

Bab, S. and Mahr, B. (2005). Reference points for the observation of systems behaviour. In *Human Interaction with Machines Proceedings of the 6th International Workshop held at the Shanghai JiaoTong University, March 15-16, 2005*. Springer-Verlag.

Barwise, J. and Etchemendy, J. (1999). *Language, Proof and Logic*. Center for the Study of Language and Information, Stanford, CA, USA. Written in collaboration with Gerard Allwein, Dave Barker-Plummer, Albert Liu.

Bernstein, P. A. (2003). Applying model management to classical meta data problems. In *CIDR*.

Borst, P. (1997). *Construction of Engineering Ontologies for Knowledge Sharing and Reuse*. PhD thesis, Universiteit Twente.

Bowman, H., Boiten, E., Derrick, J., and Steen, M. (1996). Viewpoint consistency in ODP, a general interpretation. In Najm, E. and Stefani, J.-B., editors, *First IFIP International Workshop on Formal Methods for Open Object-Based Distributed Systems*, pages 189–204. Chapman and Hall.

Breuker, J. and Van de Velde, W., editors (1994). *CommonKADS Library for Expertise Modelling. Reusable Problem Solving Components*, volume 21 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, Amsterdam, The Netherlands.

Burr, H., Deubel, T., Vielhaber, M., Haasis, S., and Weber, C. (2003). CAx/engineering data management integration: enabler for methodical benefits in the design process. In *Proceedings of the 14. International Conference on Enigneering Design 2003 (ICED 03), Stockholm*, volume 32 of *Design Society*. The Design Society & the Royal Institute of Technology.

Chakrabarti, A. (2002). *Engineering Design Synthesis: Understanding, Approaches and Tools*. Springer-Verlag, 1st edition.

Chalupsky, H. (2000). Ontomorph: A translation system for symbolic knowledge. principles of knowledge representation and reasoning. In Cohn, A. G., Giunchiglia, F., and Selman, B., editors, *Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR2000)*, pages 471–482, San Francisco, CA. Morgan Kaufmann Publishers.

Chaudhri, V. K., Farquhar, A., Fikes, R. E., Karp, P. D., and Rice, J. P. (1998). *Open Knowledge Base Connectivity 2.0.3*.

Chen, W., Kifer, M., and Warren, D. S. (1993). HILOG: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230.

Ciocoiu, M. and Nau, D. S. (2000). Ontology-based semantics. In Cohn, A. G., Giunchiglia, F., and Selman, B., editors, *Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR2000)*, pages 539–546, San Francisco, CA. Morgan Kaufmann Publishers.

Ciocoiu, M., Nau, D. S., and Gruninger, M. (2001). Ontologies for integrating engineering applications. *Journal of Computers and Information Science in Engineering*, 1(1):12–22.

Crubézy, M. and Musen, M. A. (2004). Ontologies in support of problem solving. In *Handbook on Ontologies*, pages 321–342.

Daenzer, W. F. and Huber, F., editors (2002). *Systems Engineering: Methodik und Praxis*. Verlag Industrielle Organisation, Zürich, 11th edition.

Dahchour, M. and Pirotte, A. (2002). The semantics of reifying n-ary relationships as classes. In *ICEIS*, pages 580–586.

de Bruijn, J., Martín-Recuerda, F., Manov, D., and Ehrig, M. (2004). State-of-the-art survey on ontology merging and aligning v1. deliverable d4.2.1. Technical Report IST-2003-506826, EU-IST Integrated Project SEKT.

Doan, A., Madhavan, J., Domingos, P., and Halevy, A. Y. (2002). Learning to map between ontologies on the semantic web. In *Proceedings of the Eleventh International World Wide Web Conference*, pages 662–673, Honolulu, Hawaii. ACM Press.

DoD (2001). *Systems Engineering Fundamentals*. Department of Defene – System Management College. Defense Acquisition University (DAU) Press, Fort Belvoir, VA.

Egyed, A. F. (2000). *Heterogeneous view integration and its automation*. PhD thesis, University of Southern California. Adviser-Barry William Boehm.

Ehrlenspiel, K. (2003). *Integrierte Produktentwicklung*. Carl Hanser Verlag, München, Germany.

Falkenhainer, B. and Forbus, K. D. (1991). Compositional modeling: Finding the right model for the job. *Artificial Intelligence*, 51(1-3):95–143.

Farooqui, K., Logrippo, L., and de Meer, J. (1995). The ISO Reference Model for Open Distributed Processing: An introduction. *Computer Networks and ISDN Systems*, 27(8):1215–1229.

Fritzson, P. (2004). *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Press.

Gausemeier, J. (2005). From mechatronics to self–optimizing concepts and structures in mechanical engineering: new approaches to desig methodology. *International Journal of Computer Integrated Manufacturing*, 18(7).

Gausemeier, J., Grasmann, M., and Kespohl, H. D. (1999). Verfahren zur integration von gestaltungs– und berechnungssystemen. In *VDI–Berichte Nr. 1487*. Verein Deutscher Ingenieure (VDI), VDI-Verlag.

Genesereth, M. R. and Fikes, R. E. (1992). Knowledge Interchange Format, version 3.0 – reference manual. Logic Group Report Logic-92-1, Computer Science Department, Stanford University.

Ghidini, C. and Serafini, L. (1998). Distributed first order logics. In Baader, F. and Schulz, K. U., editors, *Frontiers of Combining Systems 2*, Berlin. Research Studies Press.

Gómez-Pérez, A., Fernández-López, M., and Corcho, O. (2004). *Ontological Engineering*. Advanced Information and Knowledge Processing. Springer-Verlag, 1st edition.

Grabowski, H., Rude, S., Gebauer, M., and Rzehorz, C. (1996). Modelling of requirements: The key for cooperative product development. In *Flexible Automation and Intelligent Manufacturing 1996, (Proceedings of the 6th International FAIM Conference May 13-15, 1996 Atlanta, Georgia USA)*, pages 382–389.

Gruber, T. R. (1993). Towards principles for the design of ontologies used for knowledge sharing. In Guarino, N. and Poli, R., editors, *Formal Ontology in Conceptual Analysis and Knowledge Representation*, Deventer, The Netherlands. Kluwer Academic Publishers.

Gruber, T. R. and Olsen, G. R. (1994). An ontology for engineering mathematics. In Doyle, J., Sandewall, E. J., and Torasso, P., editors, *Proceedings of the Fourth International Conference on Principles of Knowledge Representation (KR'94)*, pages 258–269. Morgan Kaufmann Publishers.

Gruninger, M. and Kopena, J. B. (2005). Semantic integration through invariants. *AI Mag.*, 26(1):11–20.

Hammer, J., García-Molina, H., Ireland, K., Papakonstantinou, Y., Ullman, J., and Widom, J. (1995). Information translation, mediation, and mosaic-based browsing in the TSIMMIS system. pages 483–483.

Harashima, F., Tomizuka, M., and Fukuda, T. (1996). Mechatronics – "what is it, why, and how?" an editorial. *IEEE/ASME Transactions on Mechatronics*, 1(1):1–4.

Hubka, V. (1984). *Theorie Technischer Systeme: Grundlagen Einer Wissenschaftlichen Konstruktionslehre*. Springer-Verlag, Berlin, 2nd edition.

IAM (2002). Networking the way to new systems. *International Automobile Management*, 1(1):10–13.

ISO/IEC 15288 (2002). *ISO/IEC 15288:2002 Systems engineering – System life cycle processes*. International Organization for Standardization (ISO) / International Electrotechnical Commission (IEC).

Karcher, A., Bender, K., and Fischer, F. (2001). KEIM – Kontinuierliches Engineering-Informationssystemmanagement im Product LifeCycle. *industrie MANAGEMENT*, 6.

Karsai, G., Láng, A., and Neema, S. K. (2003). Tool integration patterns. In *Proceedings of the Workshop on Tool Integration in System Development (TIS 2003) at ESEC/FSE 2003*, pages 33–38, Helsinki, Finland.

Kifer, M., Lausen, G., and Wu, J. (1995). Logical Foundations of Object-Oriented and Frame-Based Languages. *Journal of the Association for Computing Machinery*.

Kim, H. M., Fox, M. S., and Gruninger, M. (1999). An ontology for quality management – enabling quality problem identification and tracing. *BT Technology Journal*, 17(4):131–140.

Klein, M. (2001). Combining and relating ontologies: an analysis of problems and solutions. In Gómez-Pérez, A., Gruninger, M., Stuckenschmidt, H., and Uschold, M., editors, *Proceedings of the IJCAI-01 Workshop on Ontologies and Information Sharing*, pages 53–62, Seattle, USA.

Krause, F.-L. (1999). Erfolgreiche produktentwicklung – eine frage der software. *Konstruktion*, 51(3).

Krause, F.-L., Baumann, R. A., Kaufmann, U., Kühn, T., Leemhuis, H., and Ragan, Z. (2003). Computer aided conceptual design. In *Proceedings of the 36th CIRP international seminar on manufacturing systems, Saarbru?cken, Germany*.

Leemhuis, H. (2004). *Funktionsgetriebene Konstruktion als Grundlage verbesserter Produktentwicklung*. PhD thesis, Technische Universität Berlin.

Lin, J., Fox, M. S., and Bilgic, T. (1996). A requirement ontology for engineering design. *Concurrent Engineering: Research and Applications*, 4(4):279–291.

Madhavan, J., Bernstein, P. A., Domingos, P., and Halevy, A. Y. (2002). Representing and reasoning about mappings between domain models. In Dechter, R., Kearns, M. J., and Sutton, R. S., editors, *Eighteenth national conference on Artificial Intelligence*, pages 80–86.

Madhavan, J., Bernstein, P. A., and Rahm, E. (2001). Generic schema matching with cupid. In Apers, P. M. G., Atzeni, P., Ceri, S., Paraboschi, S., Ramamohanarao, K., and Snodgrass, R. T., editors, *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 49–58. Morgan Kaufmann Publishers.

Maedche, A., Motik, B., Silva, N., and Volz, R. (2002). MAFRA – a mapping framework for distributed ontologies. In Gómez-Pérez, A. and Benjamins, V. R., editors, *Knowledge Engineering and Knowledge Management. Ontologies and the Semantic Web, 13th International Conference, EKAW 2002, Siguenza, Spain, October 1-4, 2002, Proceedings*, volume 2473 of *Lecture Notes in Computer Science*, pages 235–250. Springer-Verlag.

Mauss, J., Seelisch, F., and Tatar, M. M. (2002). A relational constraint solver for model-based engineering. In *CP*, pages 696–701.

McGuinness, D. L., Fikes, R. E., Rice, J. P., and Wilder, S. (2000). An environment for merging and testing large ontologies. In Cohn, A. G., Giunchiglia, F., and Selman, B., editors, *Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR2000)*, pages 483–493, San Francisco, CA. Morgan Kaufmann Publishers.

Mena, E., Kashyap, V., Sheth, A. P., and Illarramendi, A. (1996). OBSERVER: An approach for query processing in global information systems based on interoperation across pre-existing ontologies. In *roceedings of the First IFCIS International Conference on Cooperative Information Systems (CoopIS'96)*, pages 14–25, Brussels, Belgium. IFCIS, The International Foundation on Cooperative Information Systems, IEEE Computer Society Press.

Merriam-Webster, editor (2003). *Merriam-Webster's Collegiate Dictionary*. Merriam-Webster, Springfield, Massachusetts, 11th edition.

Mitra, P., Wiederhold, G., and Decker, S. (2001). A scalable framework for interoperation of information sources. In *Proceedings of the First International Semantic Web Working Symposium (SWWS)*, pages 317–329.

Modelica Specification (2005). *Modelica® – A Unified Object-Oriented Language for Physical Systems Modeling. Language Specification*. Modelica Association.

Modelica Tutorial (2000). *Modelica® – A Unified Object-Oriented Language for Physical Systems Modeling. Tutorial*. Modelica Association.

Motta, E. (1998). *Reusable Components for Knowledge Models*. PhD thesis, Knowledge Media Institute, The Open University, Milton Keynes, UK.

Motta, E., Fensel, D., Gaspari, M., and Benjamins, V. R. (1999). Specifications of knowledge components for reuse. In *Eleventh International Conference on Software Engineering and Knowledge Engineering (SEKE '99)*.

Musen, M. A. (1992). Dimensions of knowledge sharing and reuse. *Computers and Biomedical Research*, 25(5):435–467.

Neelamkavil, F. (1987). *Computer Simulation and Modelling*. John Wiley and Sons Ltd., New York, NY, USA.

Notteboom, E. (2003). Multiple V-model in relation to testing. *Softwaretechnik-Trends*, 23(1).

Noy, N. and Rector, A. (2006). Defining N-ary relations on the semantic web. W3C Working Group Note, World Wide Web Consortium (W3C). http://www.w3.org/TR/swbp-n-aryRelations/.

Noy, N. F. (2004a). Semantic integration: A survey of ontology-based approaches. *SIGMOD Record, Special Issue on Semantic Integration*, 33(4):65–70.

Noy, N. F. (2004b). *Steffen Staab and Rudi Studer (editors). Handbook on Ontologies*, chapter 18. Tools for Mapping and Merging Ontologies, pages 365–384. International Handbooks on Information Systems. Springer-Verlag, 1st edition.

Noy, N. F., Fergerson, R. W., and Musen, M. A. (2000). The knowledge model of Protégé-2000: Combining interoperability and flexibility. In Dieng, R. and Corby, O., editors, *Knowledge Acquisition, Modeling and Management, 12th International Conference, EKAW 2000, Juan-les-Pins, France, October 2-6, 2000, Proceedings*, volume 1937 of *Lecture Notes in Computer Science*, pages 17–32. Springer-Verlag.

Noy, N. F. and McGuinness, D. L. (2001). Ontology development 101: A guide to creating your first ontology. KSL-Report KSL-01-05, Stanford Knowledge Systems Laboratory.

Noy, N. F. and Musen, M. A. (2000). PROMPT: Algorithm and tool for automated ontology merging and alignment. In *AAAI/IAAI*, pages 450–455.

Noy, N. F. and Musen, M. A. (2002). Promptdiff: A fixed-point algorithm for comparing ontology versions. In *Proceedings of The Eighteenth National Conference on Artificial Intelligence (AAAI-02)*, Edmonton, Alberta.

Pahl, G. and Beitz, W. (1996). *Engineering Design – A Systematic Approach*. Springer-Verlag, London, 2nd edition. Second Edition.

Park, J. Y., Gennari, J. H., and Musen, M. A. (1998). Mappings for reuse in knowledge-based systems. In *Proceedings of the 11th Workshop on Knowledge Acquisition, Modeling, and Management (KAW '98)*, Banff, Canada.

Patzak, G. (1982). *Systemtechnik – Planung komplexer innovativer Systeme. Grundlagen, Methoden, Techniken*. Springer-Verlag.

Prenninger, W. and Pretschner, A. (2005). Abstractions for model-based testing. *Electronic Notes in Theoretical Computer Science*, 116:59–71.

Rector, A. L. (2004). Representing specified values in OWL: "value partitions" and "value sets". World Wide Web Consortium (W3C) Working Group Note. http://www.w3.org/TR/2004/WD-swbp-specified-values-20040803/.

Rector, A. L., Drummond, N., Horridge, M., Rogers, J., Knublauch, H., Stevens, R., Wang, H., and Wroe, C. (2004). OWL pizzas: Practical experience of teaching OWL-DL: Common errors & common patterns. In Motta, E., Shadbolt, N., Stutt, A., and Gibbins, N., editors, *Engineering Knowledge in the Age of the Semantic Web, 14th International Conference, EKAW 2004, Whittlebury Hall, UK, October 5-8, 2004, Proceedings*, volume 3257 of *Lecture Notes in Computer Science*, pages 63–81. Springer-Verlag.

RM-ODP/1 (1998). *ISO/IEC CD 10746-1. Information technology: Open Distributed Processing Reference Model: Overview*. International Organization for Standardization (ISO) / International Electrotechnical Commission (IEC).

RM-ODP/2 (1996). *ISO/IEC CD 10746-2. Information technology: Open Distributed Processing - Reference Model: Foundations*. International Organization for Standardization (ISO) / International Electrotechnical Commission (IEC).

RM-ODP/3 (1996). *ISO/IEC CD 10746-3. Information technology: Open Distributed Processing - Reference Model: Architecture*. International Organization for Standardization (ISO) / International Electrotechnical Commission (IEC).

Roos, F. and Wikander, J. (2003). Mechatronics design and optimisation methodology – a problem formulation focused on automotive mechatronic modules. In *Mekatronikmöte 2003*.

Roth, K. (1982). *Konstruieren mit Konstruktionskatalogen. Band 1: Konstruktionslehre*. Springer-Verlag.

Rude, S. (1998). *Wissensbasiertes Konstruieren*. Shaker Verlag, Aachen.

Russell, S. and Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition.

Rzehorz, C. (1998). *Wissensbasierte Anforderungsentwicklung auf der Basis eines integrierten Produktmodells*, volume 3 of *Berichte aus dem Institut RPK*. Shaker Verlag.

Sargent, R. G. (1998). Verification and validation of simulation models. In *WSC '98: Proceedings of the 30th conference on Winter simulation*, pages 121–130, Los Alamitos, CA, USA. IEEE Computer Society Press.

Sowa, J. F. (1999). *Knowledge Representation: Logical, Philosophical, and Computational Foundations*. Brooks Cole Publishing Co., Pacific Grove, CA.

Stachowiak, H. (1973). *Allgemeine Modelltheorie*. Springer-Verlag, Wien.

Stevens, R., Brook, P., Jackson, K., and Arnold, S. (1998). *Systems Engineering: Coping with Complexity*. Prentice Hall PTR.

Stojanovic, L., Stojanovic, N., and Volz, R. (2002). Migrating data-intensive web sites into the semantic web. In *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing*, pages 1100–1107, New York, NY, USA. ACM Press.

Struss, P. and Price, C. (2004). Model-based systems in the automotive industry. *AI Mag.*, 24(4):17–34.

Studer, R., Benjamins, V. R., and Fensel, D. (1998). Knowledge Engineering: Principles and Methods. *Data and Knowledge Engineering*, 25(1-2):161 – 197.

SysML (2006). *OMG Systems Modeling Language (OMG SysML^{TM}) Specification*. Object Management Group (OMG).

Tiller, M. M. (2001). *Introduction to Physical Modeling with Modelica*, volume 615 of *The International Series in Engineering and Computer Science*. Kluwer Academic Publishers, Boston.

Top, J. L. and Akkermans, H. (1994). Tasks and ontologies in engineering modelling. *International Journal of Human-Computer Studies*, 41(4):585–617.

UML 2.0 (2005). *Unified Modeling Language: Superstructure – Specification v2.0*. Object Management Group (OMG).

Uschold, M. and Gruninger, M. (1996). Ontologies: principles, methods, and applications. *Knowledge Engineering Review*, 11(2):93–155.

Uschold, M. and Gruninger, M. (2004). Ontologies and semantics for seamless connectivity. *SIGMOD Record, Special Issue on Semantic Integration*, 33(4):58–64.

VDI 2206 (2004). *Design methodology for mechatronic systems*. Verein Deutscher Ingenieure (VDI), Berlin. VDI-Richtlinie 2206, Beuth Verlag, Berlin.

Völkel, M. and Groza, T. (2006). SemVersion: An RDF-based ontology versioning system. In *Proceedings of IADIS International Conference on WWW/Internet*, Murcia, Spain. IADIS, IADIS.

Wache, H. (2003). *Semantische Mediation für heterogene Informationsquellen*, volume 261 of *Dissertationen zur Künstlichen Intelligenz*. Akademische Verlagsgesellschaft, Berlin.

Wache, H., Vögele, T. J., Visser, U., Stuckenschmidt, H., Schuster, G., Neumann, H., and Hübner, S. (2001). Ontology-based integration of information — a survey of existing approaches. In Gómez-Pérez, A., Gruninger, M., Stuckenschmidt, H., and Uschold, M., editors, *Proceedings of the IJCAI-01 Workshop on Ontologies and Information Sharing*, pages 108–117, Seattle, USA.

Wielinga, B. J., Akkermans, H., and Schreiber, G. (1995). A formal analysis of parametric design. In Gaines, B. R. and Musen, M. A., editors, *Proceedings of the 9th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*, Alberta, Canada.

Yang, G., Kifer, M., Zhao, C., and Chowdhary, V. (2005). *Flora-2: User's Manual*.

Zimmermann, J. U. (2005). *Informational Integration of Product Development Software in the Automotive Industry*. PhD thesis, University of Twente.