TECHNISCHE UNIVERSITÄT MÜNCHEN
Lehrstuhl für Produktentwicklung

Object-Oriented Graph Grammars
for Computational Design Synthesis

Bergen Helms

Vollständiger Abdruck der von der Fakultät für Maschinenwesen der
Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. rer. nat. Dr. h.c. Ulrich Walter
Prüfer der Dissertation: 1. Univ.-Prof. Kristina Shea, Ph.D., Eidgenössische
                           Technische Hochschule Zürich, Schweiz
                        2. Univ.-Prof. Dr.-Ing. Martin Eigner, Technische
                           Universität Kaiserlsautern
                        3. Univ.-Prof. Dr.-Ing. Udo Lindemann

Die Dissertation wurde am 20.09.2012 bei der Technischen Universität München
eingereicht und durch die Fakultät für Maschinenwesen
am 06.02.2013 angenommen.

ABSTRACT

Conceptual design is an early design phase that is acknowledged as particularly critical. Its goal is the determination of the product's essential characteristics that meet the given requirements. Conceptual design is characterized by high uncertainty resulting from lacking knowledge about the future product that makes it difficult to evaluate design quality and to systematically explore the set of solutions. Computational Design Synthesis (CDS) aims at supporting conceptual design through formalization and automation of knowledge-intensive design tasks. However, CDS still has little acceptance in industry due to the high effort required for knowledge and task formalization, the limited scope of application, the lack of reuse of existing paper-based design knowledge, the lack of modeling standards and tool integration, and the low maturity of software tools. The potential of CDS to increase development efficiency and innovative power motivates addressing these problems. This thesis contributes to the goal of increasing the applicability of CDS in every day design practice with the development and implementation of a hybrid knowledge representation, termed object-oriented graph grammar, and an approach to automatically formalize engineering knowledge from design catalogs.

Object-oriented graph grammars enable the computational synthesis of product architectures on multiple levels of abstraction, i. e. Function, Behavior and Structure. This hybrid knowledge representation allows to capture declarative knowledge in a port-based metamodel and to formulate generic, procedural design rules in a graph grammar. The object-oriented graph grammar approach is implemented in the modular, open-source and platform-independent software booggie. A formal language definition represents the foundation for tool integration through model transformation. A complementary approach is developed to characterize and formalize physical effects contained in design catalogs. Through an automated analysis of the equation structure, abstraction ports are assigned to physical effects and represent valid mappings between functions and physical effects.

Through the hybrid knowledge representation of object-oriented graph grammars, advances in terms of efficiency and effectiveness of the knowledge formalization are achieved. These contributions are validated through the synthesis of a solution space of automotive hybrid powertrains. The impact of evolving engineering knowledge on the solution space is shown and noteable solutions are identified through the search for specific solution characteristics. The computational generation of aircraft cabin layouts validates the practical usability of object-oriented graph grammars as implemented in booggie in an industrial case study. The automated, equation-based formalization of physical effects makes paper-based engineering knowledge available for CDS. The advantage to computationally reusing this knowledge is validated with the formalization of the physical effects of two design catalogs and a separate software prototype that searches suitable physical effects for a given function.

The contributions achieved in this work support the efforts to bring CDS approaches into use in every day design practice. Besides an increase of the software maturity, future work should include the integration of object-oriented graph grammars and the automated assignment of abstraction ports in one implementation. The extension of the synthesis of product architectures towards parametric synthesis and design evaluation using simulation should be addressed as well. Further, the incorporation of logical reasoners could enable the solution of logical port-matching problems and using model transformation, the transformation to other modeling languages, e. g. SysML, could be realized.

ACKNOWLEDGMENTS

The following publications are part of the work presented in this thesis:

Helms, B.; Shea, K.; Hoisl, F.: A Framework for Computational Design Synthesis Based on Graph-Grammars and Function-Behavior-Structure. In: ASME 2009 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference, IDETC/CIE 2009, pp. 841–851. San Diego, USA 2009.

Helms, B.; Shea, K.: Object-Oriented Concepts for Computational Design Synthesis. In: 11th International Design Conference, DESIGN 2010. Dubrovnik, Croatia 2010.

Helms, B.; Schultheiß, H.; Shea, K.: Automated Assignment of Physical Effects to Functions Using Ports Based on Bond Graphs. In: ASME 2011 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference, IDETC/CIE 2011. Washington DC, USA.

Helms, B.; Shea, K.: Computational Synthesis of Product Architectures Based on Object-Oriented Graph Grammars. Journal of Mechanical Design 134 (2012) 2, pp. 021008-1 – 021008-14. ISSN: 10500472.

CONTENTS

1 Introduction

The development of products is an iterative decision-making process aimed at creating an artifact that satisfies a customer need. Typically, product development processes start with the identification of a need that is analyzed, clarified and expressed as a design problem and pass through several phases. In these phases tasks are carried out that either focus on gaining insights about the nature of the design problem or on synthesizing (sub-)solutions to solve design (sub-)problems. The goal is a detailed description of a product such that it can be manufactured according to the customer needs.

The established engineering design[1] literature (EDER & HOSNEDL, 2008; EHRLENSPIEL, 2009; FRENCH, 1999; PAHL ET AL., 2007; VDI, 1987) divides the product development process into the four phases depicted in Figure 1-1.
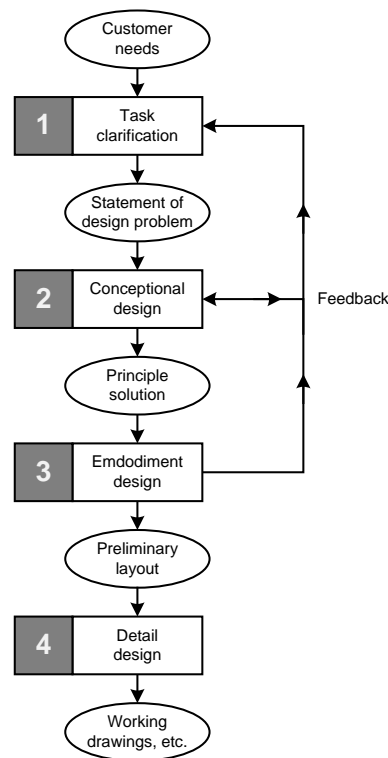
Figure 1-1: Phases of the engineering design process, adapted from (FRENCH, 1999, p. 2)

The *task clarification* phase is targeted at gaining clarity about *customer needs* and transforming them into a structured formulation of a design problem expressed as a set of requirements. It comprises two tasks: problem analysis and problem definition. The former aims to obtain information about the customer needs, hence identifying further requirements, details of the constraints and possible methods for successfully executing the design tasks (VDI, 1987). In the subsequent problem definition the design problem is expressed in the language of the problem solver, typically a designer or engineer, ideally without being biased by a previous solution or having a specific solution in mind.

---

[1]In the context of this work, the terms *product development* and *engineering design* are used synonymously.

Based on the problem statement, the *conceptual design* phase transforms this statement into a solution concept. This is achieved by iteratively specifying the product architecture[2] on various levels of abstraction. PAHL ET AL. (2007) call the outcome of this phase a *principle solution*. In the *embodiment design* phase designers define the preliminary construction layout that most often involves the creation of "arrangement drawings" (FRENCH, 1999, p. 3), commonly also termed as *preliminary layout*. Finally, within the *detail design* phase, the final product parameters are defined. In the engineering design domain this involves primarily the definition of the product geometry and the final definition of the product properties that determine the required product behavior. As these phases influence each other in a non-serial manner, interdependencies between working steps and results arise. Hence, they are not to be seen as isolated working steps, but rather as steps through which the designer proceeds iteratively, as depicted in Figure 1-1.

The importance of task clarification has been acknowledged and led to the establishment of a separate discipline: requirements engineering. Typical tasks in requirements engineering comprise the identification of stakeholders, gaining an understanding of the customer needs and the identification, analysis, prioritization, tracking and validation of requirements (YOUNG, 2003).

Among the remaining three design phases, the conceptual design phase is acknowledged as particularly critical. It offers the greatest scope for significant enhancements (FRENCH, 1999, p. 3) and decisions made in this phase impact all subsequent design phases (CHAKRABARTI & BLIGH, 1994). The wide scope of design options poses significant challenges for designers because it is here where "engineering science, practical knowledge, production methods, and commercial aspects need to be brought together, and where the most important decisions are taken" (FRENCH, 1999, p. 3). The result of the conceptual phase, the principal solution, has a major impact on the costs of the entire product life cycle as it significantly determines the product's essential characteristics. Hence, a major portion of the total costs are defined here while the actual cost occurrence are still low at this early phase (EHRLENSPIEL ET AL., 2007, p. 166), as shown in Figure 1-2.
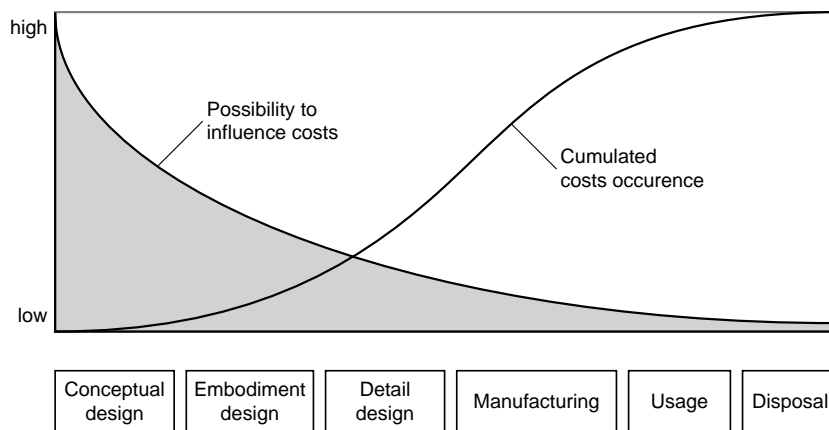
Figure 1-2: Possibility to influence costs vs. cumulated cost occurrence during the product life cycle, adapted from (EHRLENSPIEL ET AL., 2007, p. 11)

The conceptual design phase is characterized by high uncertainty resulting from vague, preliminary information and knowledge about the potential product in question. Hence, decisions having a substantial

---

[2]For a discussion of definitions of the term *product architecture*, refer to Section 2.2.4.

influence on phases further downstream in the development process are in turn influenced by this uncertainty and risk (Ponn & Lindemann, 2011, p. 293). This lack of knowledge makes it difficult to evaluate design quality and to assess the implications of these decisions. It is not without reason that this early phase is often referred to as the "fuzzy front end" (Cagan & Vogel, 2001, p. 3).

Designers often do not consider and investigate a wide range of solutions for the given design problem. Besides lack of knowledge, reasons include a bias to previous solutions or just the impossibility of iterating manually through multiple alternative solutions within a reasonable amount of time (Chakrabarti & Bligh, 1994). External factors also exist, such as the competitive pressure in global markets. Designers constantly have to increase their development efficiency and effectiveness.

There is consensus in the engineering design literature that the results of the conceptual design phase, and, consequently, the quality of the product to be developed, particularly profit from a systematic approach (Ehrlenspiel, 2009; French, 1999; Koller, 1994; Pahl et al., 2007; Ulrich & Eppinger, 2008). Chakrabarti & Bligh (1994) state that such a systematic procedure should incorporate a solution-independent description of the design task, a solution synthesis that is not biased towards previous and alternative solutions and an assessment of solution alternatives that is based on objective evaluation criteria.

The later design phases already strongly benefit from computational support. For example, computer-aided design tools are widely used in everyday engineering practice. The same is true for a wide range of simulation tools that are available for specific domains, e. g. in structural mechanics or fluid dynamics. In contrast, computational applications for supporting engineering design in the conceptual phase are primarily based on spreadsheet applications and simple modeling tools that create images rather than reusable, formal[3] models. Eigner et al. (2012) state that there is a lack of cross-disciplinary IT-support for system modeling in the conceptual design phase and emphasize the great need for research in that field. Research efforts in this direction with the goal of automating design tasks are subsumed under the field of research termed as *Computational Design Synthesis* (CDS). The focus of research in this area is typically on specific aspects rather than general approaches for the synthesis of creative solution concepts (Antonsson & Cagan, 2001a; Chakrabarti et al., 2011). However, CDS research provides a variety of promising answers to prevailing problems of the conceptual design phase, such as uncertainty and lack of knowledge. This potential is motivation for the research that is presented in this thesis.

1.1 Motivation and problem description

Due to the nature of the challenges facing product developers in the conceptual phase, e. g. the fuzzy front end and increasing competition, approaches from the research field of Computational Design Synthesis (CDS) offer promising opportunities. A goal of CDS is to iteratively and (semi-)automatically generate a range of solutions for given design tasks. Formal methods for computational design synthesis assist designers in developing better products faster, through rapid generation of spaces of feasible, optimized, and if appropriate, simulation-driven designs. CDS supports the creative step in the conception of products (Antonsson & Cagan, 2001b). Cagan et al. (2005) state that CDS is ideally "invoked in situations in which the human designers are often at a loss of what avenues to pursue, or the best method of achieving a solution requires the generation and evaluation of countless alternatives". Synonymously to

---

[3]Within the scope of this thesis, the definition of a *formal model* is used in accordance with the VDI guideline 3681 (VDI, 2005): A "Formal description method has a mathematical basis and a completely defined syntax as well as a clear semantic interpretation."

Computational Design Synthesis the term *Formal Engineering Design Synthesis* (Antonsson & Cagan, 2001b) is also used.

According to Chakrabarti et al. (2011), the importance and relevance of CDS for the conceptual design phase is established in two ways: First, computational approaches support the investigation of new directions in the design process as they are not biased by previous solutions and can potentially generate and evaluate a high number of solutions. Second, due to the possibility to automate design tasks, computational approaches can decrease the tedium in routine design tasks. Through that automation, the error rate can be reduced and more time is left for the designer to attend to creative tasks.

One can then conclude that CDS methods are a promising approach to address the described challenges of the conceptual design phase. Also from a more general and strategic perspective, the benefits of CDS offer the potential to increase the innovative capacity of manufacturing companies. This is the key point for securing their future viability in the context of an increasingly global competitive pressure (Cooper & Edgett, 2005, p. 3) and, consequently, to develop better products faster. The application of CDS approaches can contribute to an improvement of development efficiency leading to an acceleration of the development process. In terms of development effectiveness, an increase of the innovative capacity results in the development of better products. These are the essential aspects for securing success in the marketplace (Spath, 2001). Another challenge is the increase of reactivity in the product development process. Given its high susceptibility to cyclic influences, such as modified customer requirements (Berkovich et al., 2009), changes in laws (Langer & Lindemann, 2009) and evolving knowledge, e. g. due to new technologies (Helms & Shea, 2012), the ability to flexibly react to such influences is considered highly valuable (Christensen, 2006). Computational support, especially in the conceptual design phase, can address this point by enabling the quick generation and adaptation of designs.

Based on statements stemming from research and academia, it can be said that the promises of CDS address essential challenges in the conceptual design phase that are considered crucial for the development of competitive products. From an industrial perspective, the barrier that may prevent manufacturing companies from investing in CDS should be significantly low when considering the notably large challenge of constantly raising innovative capacity. This presumption is substantiated by market researchers (Infiniti Research, 2011) who state that with increasing market competition the need for design and analysis software that assures product reliability, durability and quality is also increasing. Companies have to accelerate the design process to launch new products, with fewer physical prototypes and at lower costs.

Naturally, the question arises why the promising application of CDS research is not widespread in the industrial context or, more generally speaking, why everyday engineering design is not widely supported by applications of CDS research. In the remainder of this section, research issues are formulated and illustrated addressing the principal reasons for the weak dissemination of CDS approaches. The preceding depiction of the potential added value of CDS application motivates to face these research issues and leads to the formulation of research goals in the following section.

Research Issue 1. Inefficiency of knowledge formalization

It is common for CDS approaches to build on rule-based knowledge formalization. This means that the complexity of the design problem and the scope of application have a direct impact on the size of the rule set. For example, for transforming a function-based product description into a component-based product description, Sridharan & Campbell (2004) manually defined 69 rules and Kurtoglu et al. (2010)

derived a set of 170 rules from a repository of designs. The formulation of these design rules is time consuming and often involves the integration of expert knowledge, which can become a substantial cost factor. Further, large rule sets raise issues in terms of keeping the knowledge base consistent and avoiding contradictions between rules. When using CDS to automatically generate design candidates, often the design problem itself and the knowledge of how to tackle the design problem change due to various influences, such as modified requirements, changes in laws and evolving knowledge about technologies. The inflexibility of knowledge bases induces a considerable effort to keep the knowledge base up to date and might even require fundamental changes in the structure of the knowledge. Knowledge bases are often conceptualized as closed, monolithic repositories. As a result, a modularization of the knowledge base to promote reuse of formal knowledge is not supported (TOMIYAMA ET AL., 1989).

Another issue that falls under this category is the lack of systematic support for the process of knowledge formalization (CHAKRABARTI ET AL., 2011; McKAY ET AL., 2012). In general, the existence of formalized knowledge is assumed. Hence, static knowledge is captured up-front while the method and tool is under development. Where that knowledge comes from and how users of CDS approaches could modify, extend or even create formal knowledge bases is often not considered. The research field of knowledge engineering focuses on methods for integrating knowledge into computer systems. The application of research results from this field, as for example developed within the EU project MOKA[4] (STOKES & MOKA CONSORTIUM, 2001) could ease the time-consuming process of knowledge formalization. The problems of effort-intensive knowledge formalization, inflexibility and lack of support for systematic formalization pose a high barrier for applying results of CDS research in everyday engineering practice in both academia and industry. This is especially insufficient due to the temporal nature and connectivity of distributed knowledge throughout different engineering domains and company divisions.

Research Issue 2. Limited scope of application of CDS approaches

Several methods for formal, computational design synthesis have been successfully developed during the last few decades (ANTONSSON & CAGAN, 2001b; CHAKRABARTI, 2002; CHAKRABARTI ET AL., 2011). However, most approaches are often limited to a narrow engineering viewpoint of a synthesis task. Although general design methods and procedures are well established in academia and industry (EDER & HOSNEDL, 2008; EHRLENSPIEL, 2009; KOLLER, 1994; PAHL ET AL., 2007; ULRICH & EPPINGER, 2008), domain independent computational implementations for design synthesis are rare (ERDEN ET AL., 2008). Nevertheless, they have been in demand since 1987 as described in the VDI guideline 2221 (Systematic Approach to the Development and Design of Technical Systems and Products) issued by the Association of German Engineers (VDI, 1987). The trend towards mechatronic products, composed of mechanics, electronics and software, and towards product service systems, additionally composed of service aspects, underlines the need for general, computational synthesis methods. However, existing solutions are not multidisciplinary, not applicable in the early conceptual design phase and not sufficiently intelligent (EIGNER ET AL., 2012).

Another aspect is that design methods supporting the creation of innovative solutions typically transform product models from a solution-neutral design representation, e. g. a function structure, into a concrete design representation, e. g. a component structure. Hence, multiple levels of abstraction are required, as further elaborated in Section 2.2. Complexity is thus induced into the synthesis process as modeling elements on multiple levels of abstraction have to be considered along with their various relations.

Research Issue 3. Lack of reuse of existing, paper-based design knowledge and methods

---

[4]**M**ethodology and tools **O**riented to **K**nowledge based **A**pplications

Paper-based design methods take advantage of vast amounts of previously documented knowledge in the form of design catalogs. For example, when seeking physical effects to fulfill a certain function, the catalogs of elementary physical effects provided in paper-based design catalogs are of great value (KOLLER & KASTRUP, 1998; ROTH, 2001; PONN & LINDEMANN, 2011). Further, various design methods, such as functional modeling (PAHL ET AL., 2007), are widely taught in the academic engineering design education and should represent a common practice of how to proceed when tackling design tasks. Use of these knowledge sources and approaches to align the computational design process towards the human, i. e. paper-based, design process does not prevail in CDS applications. This leads to the necessity to formalize the knowledge from first principles instead of reusing knowledge. This increases the effort required for establishing CDS systems. Often, it is difficult for the user of CDS systems to understand the problem solving process and to be able to assess solution quality and validity. This is due to the fact that CDS approaches in many cases differ significantly from the known paper-based method. Hence, an intuitive comprehensibility of the computationally realized synthesis process is not given.

Research Issue 4. Lack of modeling standards and tool integration

The assessment of designs that are computationally synthesized often involves the integration of specialized tools, e. g. for simulating the design's physical behavior. Especially with regards to the development of complex products, this often requires the simulation of systems in multiple engineering domains. Research efforts are being made to support multidisciplinary modeling and exchange of models among disciplines and tools. This has led to the development of general, standardized modeling languages, such as SysML. These modeling languages increasingly gain significance, especially in the industrial context. However, only few CDS approaches are based on these and benefit from their advantages.

While there are numerous CDS approaches that integrate specific simulation tools, there still exists the need for general approaches for interfacing with other tools and to describe synthesized designs in a standard format. These issues lead to the difficulty to integrate synthesis approaches in tool chains and development processes, particularly when it comes to design problems that span multiple engineering domains. A CDS approach typically represents a special solution instead of a building block in the broader development process.

Research Issue 5. Low maturity of software tools

The goal of software development within CDS research is typically not the development of mature software for industrial application but the validation and illustration of synthesis methods based on software prototypes. This is primarily justified by the fact that elaborate software engineering is not within the scope of CDS research. It can be observed that in publications in this research field the development of software is not considered a key contribution. It is more the case that software prototypes are seen as enablers to show strengths and weaknesses of the actual synthesis methods. The immaturity of CDS software is – from a scientific perspective – comprehensible. However, this issue hinders the expansion of CDS into an industrial application context. It also restricts the exchange of CDS tools among researchers and obstructs research towards a common software platform for CDS.

1.2 Objectives and expected contributions

CHAKRABARTI ET AL. (2011) recently stated that the CDS research community at large strives to push CDS approaches toward "use in everyday design practice". The superordinate objective of this thesis is to contribute to this effort. However, tackling this objective in its entirety presents a scope far too extensive

for this thesis. Emphasis is put here on the advancement of computational support for the conceptual phase and this work specifically targets an enhancement of the representation foundation for CDS; the particular importance of representation for CDS is discussed in Section 2.1.

The conceptual design phase is a decisive part of the design process during which new product concepts and innovations are envisioned and transformed into physical configurations that meet the design requirements. The objective of enhancing computational support for this phase implies the following assumptions:

- In such an early phase of the engineering design process, geometry plays a minor role. Instead, the composition of the design, e. g. in terms of its function or component structure, is the focal point. Such product descriptions are termed *product architectures* and build on design representations that are introduced in Section 2.2. Graphs are a natural means to describe product architecture and are computationally well supported.

- Paper-based methods for conceptual design, such as functional modeling, concentrate on the composition of modeling elements instead of representing their inner workings. This kind of black box approach is also used here. Where necessary, the black boxes are enriched with parameters that, in general, exhibit a higher degree of granularity.

A graph-based overview of the scientific reasoning of this thesis is depicted in Figure 1-3.
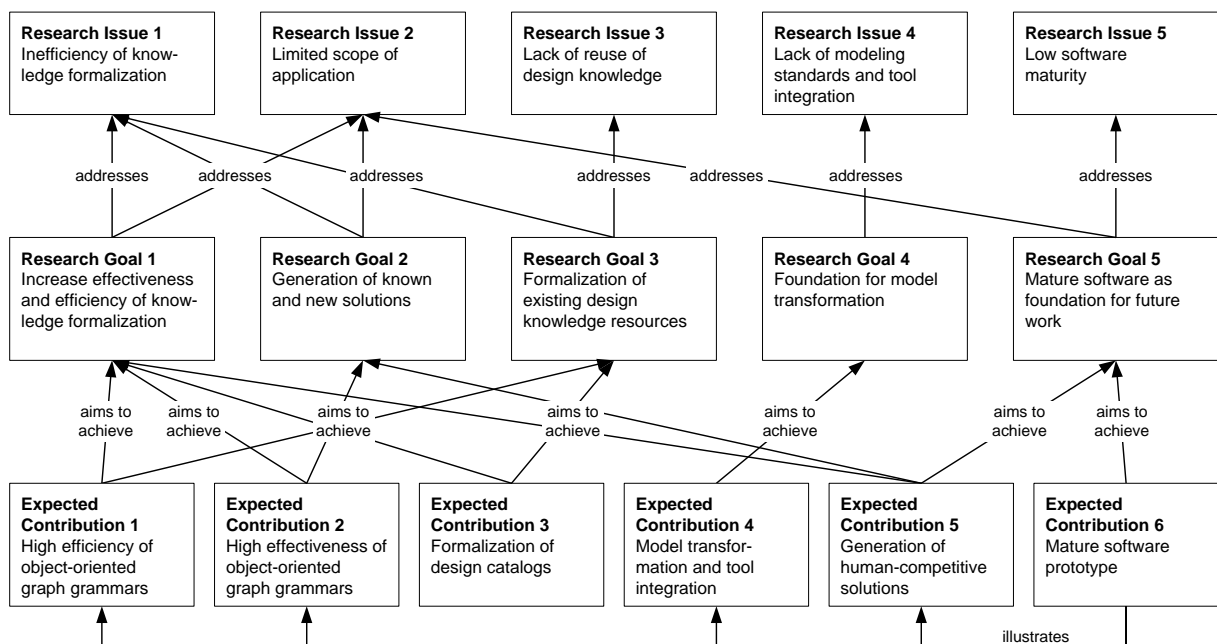
Figure 1-3: Overview of the argumentation structure of this thesis

Research Goal 1. Enhancement of knowledge representation to increase effectiveness and efficiency of knowledge formalization for CDS

The formal representation of knowledge and of the design that is in the process of being synthesized build the foundation for any CDS approach, which is further elaborated in Section 2.1. This is the main area of contribution of this research. The assumption is that CDS approaches at large profit from advances in making representations more effective and more efficient. *Efficiency* considers the effort that is required

to formalize knowledge and maintain the knowledge base; this addresses Research Issue 1 (inefficiency of knowledge formalization). *Effectiveness*, in turn, targets the suitability of the representation to formalize design problems and appropriate problem solving strategies within the relevant scope of application and is targeted at Research Issue 2 (limited scope of application).

Research Goal 2. Computational generation of known and new solutions based on the developed knowledge representation

The ultimate goal of formalizing design knowledge is to automate the creation of design candidates. Typically in the first step, known solutions are generated to validate the computational synthesis process. This provides the foundation for expanding towards synthesizing solutions using advanced search and optimization techniques, which is not within the scope of this thesis. The suitability of the knowledge representation developed in this research to cope with the synthesis of known and new solutions is demonstrated. To do so, a constrained random-walk search approach is developed. Additionally, it shall be shown that the developed knowledge representation is generally suited as a foundation for synthesis approaches that concentrate on more elaborate search and optimization algorithms.

Research Goal 3. Formalization of existing design knowledge resources as input for CDS knowledge bases

Making knowledge available for CDS approaches that is already expressed in the form of design catalogs, design methods or guidelines has the potential to increase the efficiency of the knowledge formalization process and, hence, contributes to addressing Research Issue 1 (inefficiency of knowledge formalization). However, this goal primarily addresses Research Issue 3 (lack of reuse of design knowledge).

Research Goal 4. Methodological foundation for the transformation between modeling languages

CDS approaches often have to interface with other tools and methods, e. g. for the evaluation of a design candidate. Beyond considering this primarily as an implementation issue that focuses on the control of external tools, e. g. for starting a simulation run, specific models have to be generated that contain the required information. A methodological foundation is needed that captures the knowledge about how to transfer between model representations. Such a transformation method could also be used to transfer models to standardized model languages, such as SysML. Thus, this research goal addresses Research Issue 4 (lack of modeling standards and tool integration).

Research Goal 5. Development of mature, expandable, open-source software that can serve as a foundation for future CDS research

The achievement of the previous research goals requires demonstration based on a software prototype. This research also aims to contribute to improved dissemination of CDS methods in an industrial context by providing a robust software foundation. To achieve this, mature, existing open-source software is reused. The focus on a graph-based representation supports a variety of open-source software solutions for the generation, representation, visualization and analysis of graph-based models. This requires that the software being developed in this research be issued under an open-source license. To enable future research to be based on this software platform it is important that other researchers can expand on it with their own pieces of software. Consequently, this research goal addresses Research Issue 5 (low software maturity)). As the software prototype aims to be independent of a specific application domain, it also addresses the problem of limited scope of application in Research Issue 2 (limited scope of application).

The scientific endeavor to achieve these research goals is based on the hypothesis that a knowledge representation is required that combines a model-based and a rule-based representation. The choice of such a hybrid representation is supported with analogies drawn from the development of expert systems in the area of artificial intelligence, as discussed in Section 2.3 that gives an overview of knowledge representations.

The envisioned support for the conceptual design phase can be built on the grammar-based synthesis of graph-based product architectures. However, conventional graph grammar approaches only cover rule-based knowledge formalization. The expansion towards including model-based aspects is achieved based on object-oriented modeling techniques. This results in the development of a hybrid knowledge representation that is termed *object-oriented graph grammars*.

The following contributions are expected in the scope of this thesis:

Expected Contribution 1. Object-oriented graph grammars are demonstrated to be an efficient knowledge representation approach. This means (1) the knowledge base remains manageable even when complex model structures are to be generated, (2) the knowledge formalization process is intuitively understandable, (3) the formalization of evolving knowledge is supported and (4) the computational efficiency of executing object-oriented graph grammars and, consequently, of the solution generation is high enough for the realization of elaborate search and optimization algorithms. Further, object-oriented graph grammars provide the representational foundation to capture the knowledge that is formalized from paper-based design catalogs.

Expected Contribution 2. Object-oriented graph grammars prove to be a suitable representational means to effectively capture declarative and procedural knowledge. Thus, knowledge for a wide range of applications and for multiple levels of abstraction can be formalized in an effective way. Based on this knowledge representation, the generation of known and new solutions is possible.

Expected Contribution 3. In conceptual design, functions are used to represent abstract product descriptions. During the design process, suitable physical effects are mapped to them. The ability of physical effects to realize functions is contained within the model-based part of the hybrid knowledge representation. The knowledge for this mapping is automatically derived from paper-based design catalogs.

Expected Contribution 4. The developed knowledge representation supports the formal specification of modeling languages and execution of rule-based transformations between them. This serves as a strong basis for the realization of model transformation approaches and interfacing with required tools.

Expected Contribution 5. The expected achievements in terms of efficiency and effectiveness of the knowledge representation in conjunction with a mature software prototype allows for the generation of human-competitive solutions, including known and new solutions, for industrial design problems.

Expected Contribution 6. The illustration and validation of the previous expected research contributions (except Expected Research Contribution 3 (Formalization of design catalogs) that is validated based on a separate software implementation) results in the development of a software prototype. The integration of sophisticated open-source libraries and the use of established software development paradigms lead to a mature, modular and expandable software framework. The software requirements that have to be met for the illustration and validation purposes are presented in Section 5.1.

1.3 Thesis structure

Figure 1-4 shows the structure of the thesis linking the argumentation structure to the sections.

This work aims to develop computational support for the conceptual design phase. The research issues presented in this chapter provide the motivation for this project. Chapter 2 aims to support the assumptions and research goals presented in the introduction. After a general depiction of Computational Design Synthesis highlighting the importance of representation (Section 2.1), the relevant representational foundations of the two research areas, artificial intelligence and engineering design, are discussed in Section 2.2 and Section 2.3: design representations and knowledge representations. While the former serves the purpose of introducing the levels of abstraction and deriving requirements for knowledge representations for CDS, the latter corroborates the approach of developing a hybrid knowledge representation from the perspective of knowledge-based systems and artificial intelligence. Both sections introduce terms and definitions providing the conceptual foundation for the following chapters. The current state of the art in CDS is explored in Section 2.4. The scope of application of related work is described using primarily the levels of abstraction of design representations. The description of how these approaches formalize knowledge for design synthesis links back to the section about knowledge representations. In the conclusion of Chapter 2, previous research efforts are structured according to the dimensions of design representation and knowledge representation. The aim is to discuss the expected contributions in the context of research gaps in previous work.

These three chapters can be considered as a three-stepped approach describing the methods developed:

- Chapter 3 introduces object-oriented graph grammars for the computational synthesis of graph-based product architectures using a Function-Behavior-Structure representation. This method is validated through the synthesis of product architectures of automotive hybrid powertrains. The generation of the solution space is shown and solutions with interesting characteristics are discussed. Individual solutions and the impact of evolving engineering knowledge are discussed.

- The method presented in Chapter 4 is targeted at making the physical effects contained in design catalogs available for CDS approaches. For this purpose, abstraction ports are introduced that represent the valid mapping between functional operators and physical effects. For the automated assignment of abstraction ports, a method is presented that analyzes the equation structure of physical effects. The assignment of abstraction ports is validated through the formalization of the physical effects of two design catalogs and the development of a software prototype for the search of suitable physical effects for a given function.

- Chapter 5 describes the software platform booggie[5] (brings object-oriented graph grammars into engineering) that implements object-oriented graph grammars and selected software components. The formal foundation of this software, the booggie modeling language, is presented that enables the realization of model transformation and tool integration approaches. The automated generation of aircraft cabin layouts aims to validate the practical usability of the developed software platform in an industrial context (Section 5.5).

An overall discussion of the research contributions, limitations and potential future work is presented in Chapter 6. This thesis ends with a final conclusion.

---

[5]Project website: http://booggie.org

1. Introduction

1.1 Motivation and problem description	Research Issue 1	Research Issue 2	Research Issue 3	Research Issue 4	Research Issue 5	

1.2 Objectives and expected contributions	Research Goal 1	Research Goal 2	Research Goal 3	Research Goal 4	Research Goal 5	
	Expected Contribution 1	Expected Contribution 2	Expected Contribution 3	Expected Contribution 4	Expected Contribution 5	Expected Contribution 6

1.3 Research methodology and thesis structure

2. Computational design synthesis for supporting conceptual design

2.1 The importance of representation for CDS

2.2 Overview of design representations

2.3 Overview of knowledge representations

2.4 Related work

2.5 Conclusions

3. Synthesis of product architectures based on object-oriented graph grammars

3.1 Design representation: Function-Behavior-Structure

3.2 Knowledge representation: Object-oriented graph grammars

Contribution 1   **Contribution 2**

3.3 Validation: Synthesis of hybrid powertrains

3.4 Discussion and future work

4. Automated allocation of physical effects to functions using abstraction ports

4.1 Method context

4.2 Bond graph elements – the foundation for abstraction port types

4.3 Assignment of abstraction ports to functions   **Contribution 3**

4.4 Assignment of abstraction ports to physical effects

4.5 Validation: Formalization of design catalogs

4.6 Discussion and future work

5. Development of the software prototype booggie

5.1 Software requirements   **Contribution 5**   **Contribution 6**

5.2 Formal definition of the booggie modeling language (bgML)   **Contribution 4**

5.3 Software architecture

                                                             Contribution 5   **Contribution 6**

5.4 Implementation of selected (sub-)components

5.5 Validation: Synthesis of aircraft cabin layouts   **Contribution 5**   **Contribution 6**

5.6 Discussion and future work

6. Discussion and future work

6.1 Research contributions

6.2 Limitations and potential for future work

7. Conclusion

Figure 1-4: Overview of the thesis structure

2 Computational design synthesis for supporting conceptual design

According to Simon (1996) *design* is an activity leading to "possible worlds satisfying specific constraints". Further, design can be seen, from a psychological perspective, as an activity that involves creativity to achieve the goal of the creation of a new product (Pahl et al., 2007, p. 1). Consequently, *engineering design* is the performance of design activities in the field of engineering. Depending on the technological nature of the product to be designed, engineering design activities require a sound knowledge foundation in "mathematics, physics, chemistry, mechanics, thermodynamics, hydrodynamics, electrical engineering, production engineering, materials technology, machine elements and design theory as well as knowledge and experience of the domain of interest" (Pahl et al., 2007, p. 1). According to the systematic approach of the design guideline VDI 2221 (VDI, 1987) and the design methodology of Koller (1994), design activities can be grouped in three categories:

- *synthesis* as "the creative step itself: the conception and postulation of possibly new solutions" (Antonsson & Cagan, 2001b), but also the modification and enhancement of an existing solution,

- *analysis* to determine the new solution's properties,

- *evaluation and decision* to guide the design process through assessing the solution's quality with regards to the achievement of the constraints and design goal.

The purposeful combination of these activities determines the design strategy that is carried out in the course of the design process. In design literature, methods, procedural models and best practices are used to support the various design activities (Eder & Hosnedl, 2008; Ehrlenspiel, 2009; Koller, 1994; Pahl et al., 2007; Ponn & Lindemann, 2011; Ulrich & Eppinger, 2008; VDI, 1987). By putting emphasis on the term synthesis – as in Formal Engineering Design Synthesis – the goal of generating new solutions is underlined but necessarily involves activities of the other two categories. The term *formal* refers to the computational implementation of the design process requiring a high degree of structure and rigor (Antonsson & Cagan, 2001b).

Recently, the term *Computational Design Synthesis* (CDS) gained more acceptance and became the common denomination for this relatively young field of research. The constant increase of computational power promoted computational solving of engineering design tasks. This multidisciplinary field of research unites advances in research in computer science, especially from the field of artificial intelligence, and the systematic methods of engineering design to accomplish complex design and product development tasks. Thus, CDS research investigates the complex interplay between "representation, generation, and search of a design space" in search for new solutions to design problems (Cagan et al., 2005).

In the first section of this chapter, an overview of CDS is given by presenting general frameworks and models for CDS which stress the importance of representation. The representation is the aspect of CDS where the two disciplines of computer science and engineering design have an important overlap. The

following two sections are dedicated to these two viewpoints. While Section 2.2 reviews design representations and addresses the question *What needs to be represented to solve engineering design problems?*, Section 2.3 reviews knowledge representations and focuses on answering the question *How can knowledge be represented to computationally solve design tasks?*. The latter also serves the purpose of presenting developments in the history of expert systems to draw conclusions for the further development of research in CDS. Further, these two sections provide the term definitions and theoretical background for the following sections. As schematically depicted in Figure 2-1, the reviewed design representations and knowledge representations constitute two axes of a framework according to which the related state of art in CDS is structured. Thereby, common characteristics, strengths and weaknesses of the reviewed approaches can be discerned that are related to the applied representations. Both, the conclusions drawn out of this review and the analogy from the history of expert systems corroborate the approach detailed in the subsequent sections.
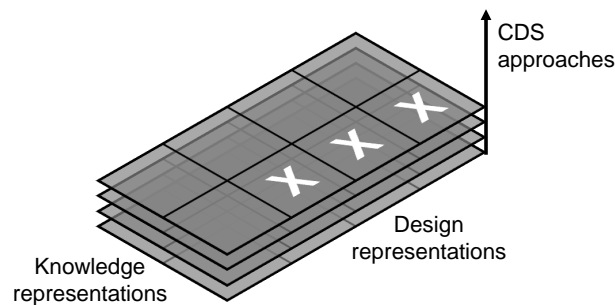
Figure 2-1: Review framework for CDS approaches based on design representations and knowledge representations

2.1 The importance of representation for CDS

To classify and consolidate research in CDS, generic frameworks have been proposed, two of which are presented in Figure 2-2. SHEA & STARLING (2003) developed a framework to support the creation of performance-based parametric CDS tools. It is built on a loop of the phases *investigate*, *generate*, *evaluate* and *mediate*. This framework emphasizes the use of production systems for the representation of "vast design languages" (SHEA & STARLING, 2003). The application of this framework is illustrated using structural and mechanical design problems. Nevertheless, structuring CDS approaches according to the aforementioned four phases is also valid for other areas of application.

Similarly, the framework by CAGAN ET AL. (2005) also contains a loop made up of the three phases *generate*, *evaluate* and *guide* which are based on a *representation*. These three phases define a search process that is initiated through the formulation of a design problem by the human designer such that it can be understood by the computational system. The final design is presented to the human designer who can interpret the solution and potentially reformulate the design problem accordingly.

Though using a different notation, the two frameworks are essentially comprised of the same steps that are required for computationally solving engineering design tasks. These are briefly described in the following:

- Investigation (STARLING & SHEA) or Representation (CAGAN ET AL.): The aim of this phase is to define the representation for the computational design synthesis task. To achieve this, the class
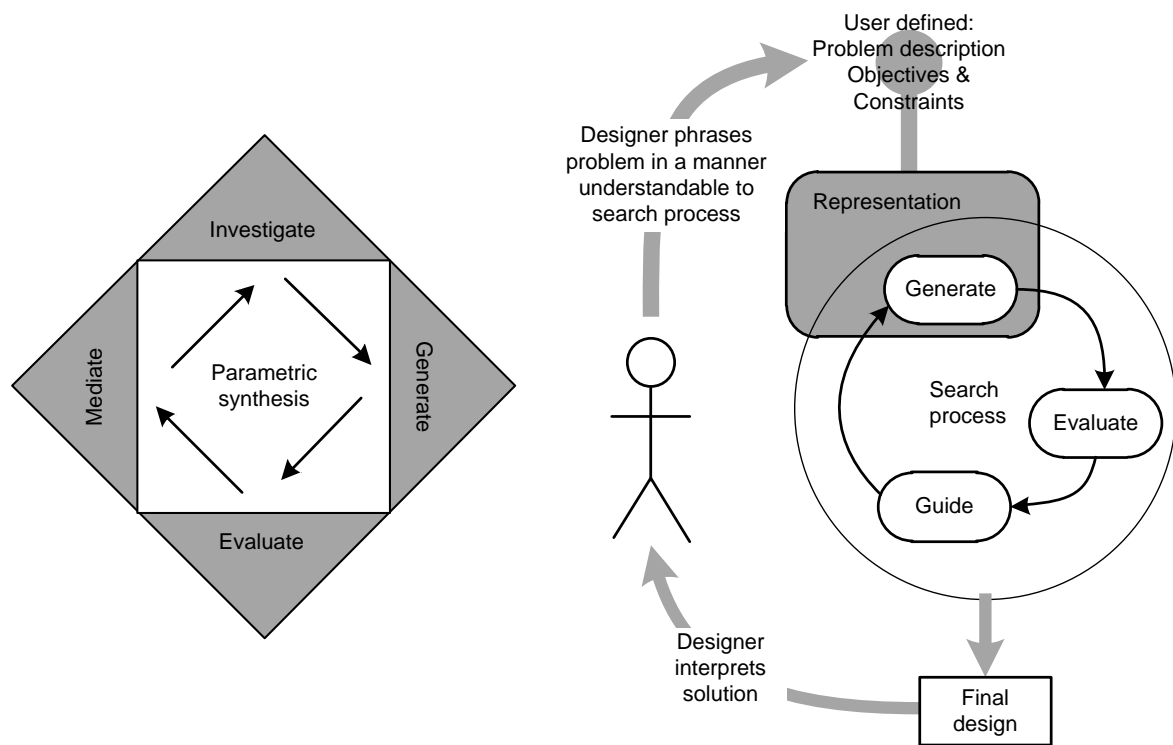
Figure 2-2: Generic CDS-frameworks based on (Shea & Starling, 2003) and (Cagan et al., 2005)

of design problem needs to be investigated, e. g. based on prior solutions or known methods and strategies, to identify an appropriate representation. Since this preparatory work builds the foundation for all upcoming operations, it is assumed that this work is primarily done manually.

- Generation (Starling & Shea and Cagan et al.): In the generative part, new solutions are computationally synthesized based on the defined representation. This is accomplished by balancing the use of randomly created naïve solutions and the use of complex problem solving knowledge. The more a representation allows for the combinatorial creation of designs, i. e. a high interconnectivity between the elements, the more applicable are search and optimization methods.

- Evaluation (Starling & Shea and Cagan et al.): To judge the quality of a computational design, methods have to be employed that separate promising solutions from poor ones. In cases where a high number of evaluations have to be carried out, it is preferable to integrate this task in the computational loop; otherwise, it is conceivable that the assessment or even partial assessment of the design can be done by the user.

- Mediation (Starling & Shea) or Guidance (Cagan et al.): After having evaluated the design's performance, the next step is to provide feedback to the system on how to proceed. Whether the next synthesis steps are triggered by using search methods, e. g. simulated annealing, or knowledge-based engineering methods, e. g. using an inference engine, depends directly on the representation of knowledge.

Both frameworks emphasize the representation as an essential and fundamental part of realizing CDS methods and tools. Shea & Starling (2003) propose the investigation of the problem domain through the exploration of prior and recent design problems and appropriate design methodologies. Building blocks

and their interactions, i. e. the representation, are identified leading to an increased understanding of the problem domain.

The representation determines the set of suitable search and generation techniques and, consequently, has a high impact on the overall efficiency of CDS approaches and tools. The required level of detail, the scope of application and the applied design strategy, or methodology, are determined in this first phase. Cagan et al. (2005) underline the importance of the representation and identify the definition of a suitable representation as the major challenge in CDS.

From this general perspective on CDS research, the claim that has been made in the hypothesis in Section 1.2 (p. 6), regarding the importance of the representation for an increase of CDS usability, is substantiated. Hence, tackling an enhancement of the foundation of CDS approaches, namely the representation, seems to be a promising approach towards contributing to the objective of CDS research at large of bringing CDS approaches into everyday engineering practice as stated by Chakrabarti et al. (2011). The following two subsections are dedicated to the foundations of CDS representations. In contrast to the aforementioned general frameworks, the discussion of representation for CDS in this thesis is subdivided into design representation and knowledge representation.

2.2 Overview of design representations

Engineering design problems are typically characterized by their high complexity. This is increasingly owed to the multiplicity of engineering domains that are combined in today's products, e. g. mechanics, electronics and software. Pahl et al. (2007) describe this complexity as a high number of components having a high degree of interdependency. Further, the required multidisciplinarity in design teams, which may be even geographically and temporally distributed, is another complexity driver (Szykman et al., 2000). Hence, the design of complex systems in such a complex environment requires a systematic approach for decomposing a problem into manageable sub-problems and sub-components. Depending on the degree of interdependence, the design of these sub-systems can be carried out to some extent in parallel (Simon, 1996).

The VDI guideline 2221 (VDI, 1987) describes a general, systematic problem-structuring approach, see Figure 2-3. It integrates the *decomposition* of a problem into sub-problems and, further, into individual problems and the synthesis of a solution through the composition of individual solutions into sub-solutions and, finally, into an overall solution. The procedure of decomposition is of vital importance in engineering design and can be seen in multiple variations. Pahl et al. (2007) for example apply the decomposition task to functional modeling by breaking down an overall function into subfunctions aiming at a reduction of complexity, see Figure 2-6. Such a structured, decomposition-based procedure provides a strong foundation for computationally tackling design problems due to these advantages (VDI, 1987):

- "the recognition of sub-problems by revealing patterns and relationships"

- "the discipline to proceed systematically"

- "the development of alternative solutions"

- "the adoption of familiar and well-tried sub-solutions"

- "the introduction of a rationally organized division of labor"
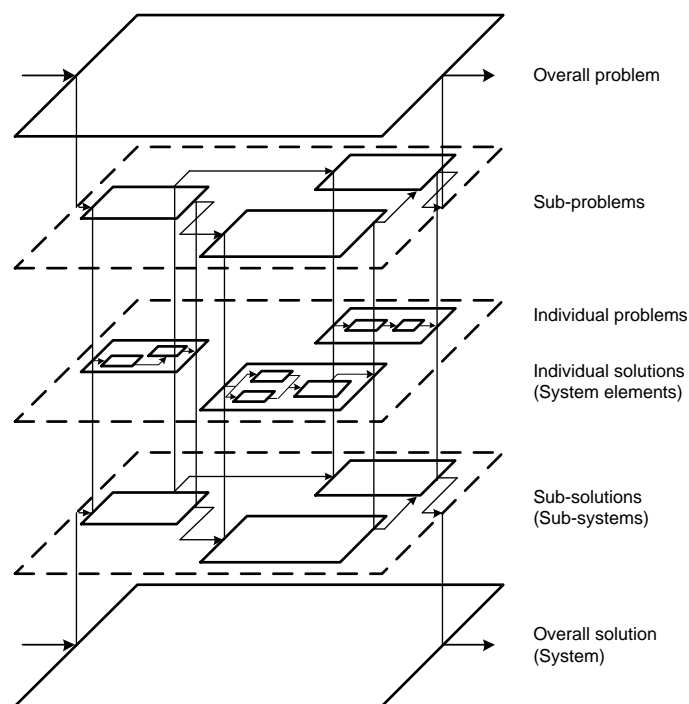
Figure 2-3: Decomposition of problems and synthesis of systems based on VDI guideline 2221 (VDI, 1987)

Nevertheless, WHITNEY (1996) states that there is a limit to problem decomposition in engineering design. Splitting design problems into subproblems induces new interdependencies between them and runs counter to the objective of reducing complexity. This is, among other issues, the reason why engineering design problems cannot be solved with the same tools and methods as successfully applied in VLSI[6] design, e. g. for microprocessors.

In the classic engineering design literature (PAHL ET AL., 2007; ULRICH & EPPINGER, 2008; EDER & HOSNEDL, 2008; EHRLENSPIEL, 2009; KOLLER, 1994), design is often described as a top-down transformation of a functional product definition into a component-based product model. Besides decomposition, the transition of an abstract description to a more concrete description (termed *concretization* according to (PAHL ET AL., 2007)) is another key element of this transformation. PONN & LINDEMANN (2011) consider *decomposing* and *concretizing* as orthogonal tasks in the design process according to the "Munich Model of Product Concretization". Together with *diversifying*, these three tasks (and their respective inverse tasks *composing*, *abstracting* and *restricting*) define a three-dimensional solution space in which various design actions take place aiming at the fulfillment of the requirements represented in the requirements space, see Figure 2-4. Generally speaking, these three tasks result in the generation and modification of product models. Through *concretization*, different *levels of abstraction* arise. Different levels of *granularity* are achieved through the decomposition of an element into its sub-elements. *Diversification* can be seen as the creation of alternative solutions while keeping the level of abstraction and the level of granularity constant.

Another view on the design process is the one by ANDREASEN & HEIN (1987, p. 63) who describe it as a sequence of selection and evaluation actions resulting in an incremental definition of "design character-

---

[6]**V**ery **L**arge-**S**cale **I**ntegration
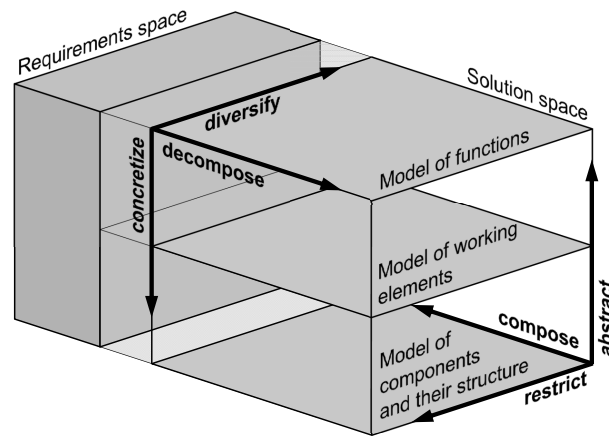
Figure 2-4: Munich Model of Product Concretization, adapted from (Ponn & Lindemann, 2011), translation based on (Hepperle et al., 2009)

istics", i. e. concretization. These actions take place on different levels of abstraction and are depicted as a multi-layered pyramid representing the "design degrees of freedom" as in Figure 2-5. This depiction of the design process is also termed "device pyramid of design" (Muller, 2007). Stankovic (2011) interprets the navigation through this design process from a computational or formal perspective as a "tree structured state space search process". To find a satisfying solution, the human designer is confronted with the challenge of stepping forward and backward and considering promising options. Thereby, models of different levels of abstraction and levels of granularity are generated and modified. Various modeling approaches in the design literature aim at supporting the systematic passage through this search process and additionally serve as means for communication and cooperation. In Figure 2-5, these levels of abstraction are mapped to the phases of engineering design as introduced in Section 1.
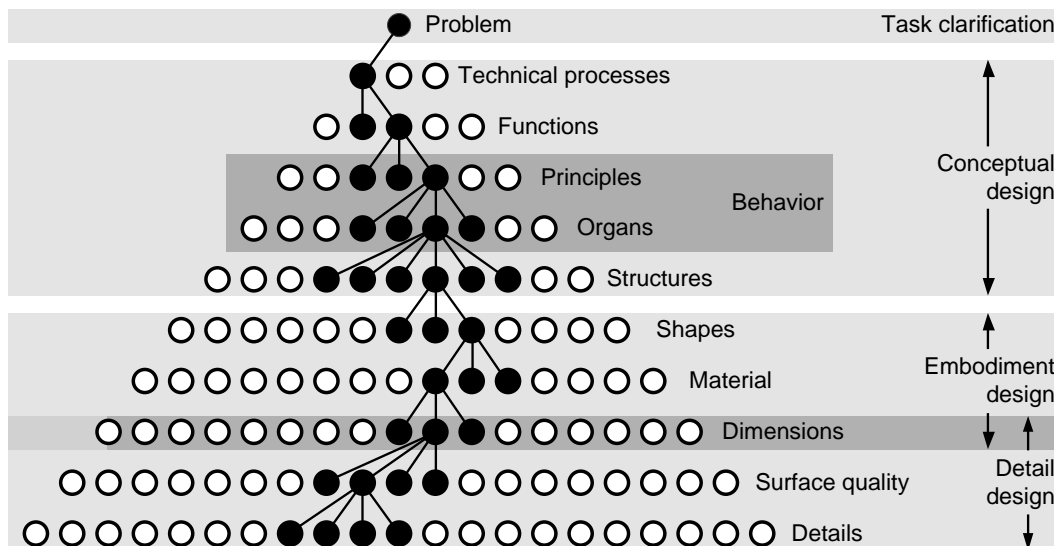
Figure 2-5: Design degrees of freedom based on (Andreasen & Hein, 1987, p. 64) and mapping to the phases of the engineering design process as depicted in Figure 1-1

A large part of the research on design methodology over the last few decades concentrated on the development of modeling approaches for supporting these design tasks. This leads to a variety of modeling paradigms which are subject to ongoing discussions. This section serves the purpose of introducing the

key elements for creating models for supporting conceptual design. Modeling elements and modeling approaches are structured according to the levels of abstraction where they can be located. Further, different grades of granularity of modeling elements are introduced that are typically interrelated through the design actions of decomposing or composing. This research focuses on supporting the conceptual design phase solely considering graph-based modeling approaches. This section introduces the key concepts of design representations and, by that, establishes one dimension of the review framework (cf. Figure 2-2) for comparing the state of the art in CDS in the following section.

As indicated in Figures 2-4 and 2-5, three generally accepted levels of abstraction can be identified: Function, Behavior and Structure (FBS). While Function and Structure are common terms in all modeling approaches, Behavior is, according to (PAHL ET AL., 2007), used to include "Physical effects" and "Working principle". In turn, PONN & LINDEMANN (2011) use "Models of working elements" as Behavior in the Munich Model of Product Concretization (Figure 2-4). GERO (1990) published a formal description of FBS using transformations that are based on set theory. They define the levels of abstraction and the transition actions between them. In a later paper (GERO, 2004), he descriptively paraphrases them as: *what the design object is for* (Function[7]) , *what the design object does* (Behavior[7]) and *what the design object is* (Structure)[7]. Another FBS (Function-Behavior-State) representation is the one by UMEDA & TOMIYAMA (1995) that supports functional decomposition, the allocation of physical effects (termed physical concepts) and their embodiment into working principles (termed physical features). The behavior is considered as changes of state of the physical features caused by physical phenomena.

ANDREASEN & HEIN (1987) claim that another, even more abstract, level is located above functions: technical processes as depicted in Figure 2-5. They play a lesser role in the overall design literature but are an essential part of the Theory of Technical Systems (EDER & HOSNEDL, 2008). This design theory sees the design process from a formal and rigorous perspective and the grammar-based CDS approach of STANKOVIC (2011) builds on that representation foundation.

2.2.1 Function

In the vast majority of the design literature, functional modeling is the entry point to the conceptual design phase (EHRLENSPIEL, 2009; KOLLER, 1994; PAHL ET AL., 2007; PONN & LINDEMANN, 2011; ULRICH & EPPINGER, 2008; VDI, 1987). The goal of functional modeling is to create an abstract, solution-independent description of the product to be developed. Thereby, bias from previous solutions is avoided and a vast exploration of the solution space is guaranteed. A functional model is hence an abstract representation of the system's tasks regardless of any solution; it "shows how the general goal of a system is achieved" (ERDEN ET AL., 2008).

One view on *functions* is that they describe input/output-relations as verb-noun pairs, e. g. "increase speed" or "harvest potatoes" (EHRLENSPIEL, 2009; PAHL ET AL., 2007). The verb can be considered as an operator manipulating an object or a stream of objects that is designated by the noun. Design literature does not provide one uniform definition for function. While PONN & LINDEMANN (2011) define functions as verb-noun pairs, PAHL ET AL. (2007) consider the verb itself as the function being associated to a flow. To prevent confusion in this research, the terms *(functional) operator* and *(functional) flow* are used to represent functions.

---

[7]The terms *function*, *behavior* and *structure* can easily be misunderstood. The specific terms being levels of abstraction within FBS are capitalized (*Function*, *Behavior* and *Structure*), whereas these terms in the general sense, e. g. in "overall function" or "data structure", are written in lower case letters.

An *overall function* models the task of the entire system and it can be decomposed into its subfunctions. A complex design task is thus divided into simpler subproblems resulting in a decrease of the grade of granularity, as depicted in Figure 2-6. Regarding this as the common foundation of functional modeling, various subtypes can be differentiated. They primarily differ in the way functions are interrelated.
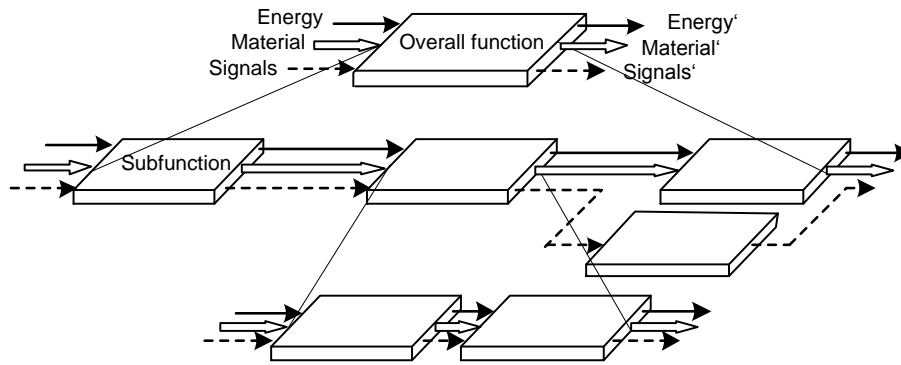
Figure 2-6: Decomposition of an overall function into subfunctions, based on (Pahl et al., 2007)

Flow-oriented functional modeling focuses on the exchange of energy, material and signal between functions and uses a designated arrow notation. Hence, the noun represents a flow whereas the verb models an operation of that flow. In the recent decades, various sets of verbs and nouns were proposed as elementary building blocks for functional modeling. E. g. Pahl et al. (2007, p. 35) claim that all functional operators can be reduced to a set of "generally valid functions": "change, vary, connect, channel, store". However, other researchers define other fundamental sets of functions. An effort to consolidate previous research on functional modeling resulted in the definition of the Functional Basis (Hirtz et al., 2002b) that consists of two taxonomies defining functional operators and flows arising from an analysis of the function sets of Pahl et al. (2007), Hundal (1990) and Altshuller (1984). While Pahl et al. (2007) only explicitly represent the flow types energy, material, signal, the Functional Basis defines more detailed flow types, e. g. translational mechanical energy. This step towards alleviating incompatibility within functional modeling has been issued as technical note by the American National Institute of Standards and Technology (NIST) (Hirtz et al., 2002a) and is presented in Appendix 9.3.

Figure 2-7 shows a flow-oriented functional model of a potato harvesting machine that uses the basic notation of Pahl et al. (2007). Figure 2-8 depicts a metamodel that was derived from this notation defining the modeling elements and their appearance in a formal representation. The extension of the flow type hierarchy according to the Functional Basis' flow taxonomy is exemplarily depicted with "...".

All of the following modeling approaches are characterized through a rough metamodel as shown in Figure 2-8 that gives a condensed view of the modeling syntax. The following stereotypes are used for defining the appearance of the modeling constructs based on a graph-based representation:

- A *«Node»* signifies an element type that can be used within a model.

- *«Edge»* represents types of connections that can be used to relate modeling elements.

- *Abstract* element types are written in italics. These cannot be instantiated and act as container for properties from which regular element types can inherit.

- *Inheritance* relationships are represented with a white arrowhead targeting towards the more general concept.
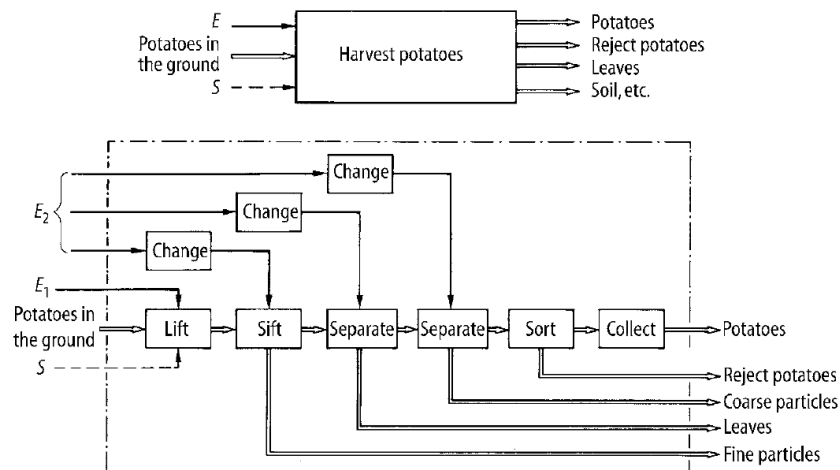
Figure 2-7: Functional model of a potato harvesting machine representing the overall function decomposed into subfunctions (PAHL ET AL., 2007, p. 276)

- A *«Frame»* can be used to group elements of the former categories.

- In the case that there is a standardized visual representation of modeling elements, their depiction is included in a gray box.

- Relations between stereotypes are indicated by a short description including the reading direction (►) and the associated multiplicity. The characterization of a relation with an edge type is indicated with this symbol: ●–.

For a detailed description of this notation for describing metamodels, refer to HOLT & PERRY (2008).
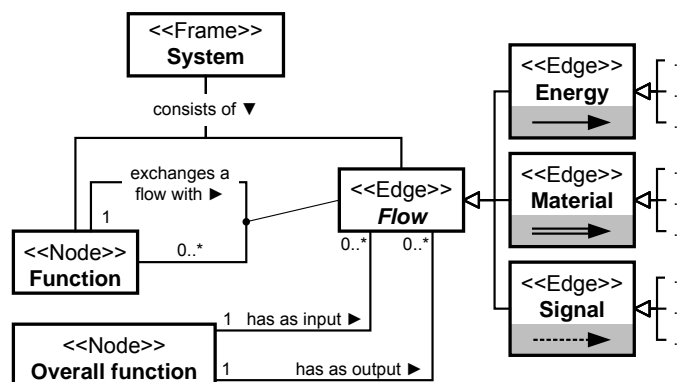
Figure 2-8: Metamodel defining the modeling syntax for flow-oriented functional modeling

The *relation-oriented functional modeling* approach differentiates two types of functions: *useful functions* and *harmful functions* that can be connected based on three relation types: *is necessary for*, *causes*, *is introduced to avoid* (PONN & LINDEMANN, 2011). Based on these modeling elements, four functional patterns can be modeled as depicted in Figure 2-9. This modeling approach supports the investigation of the logical structure of a system to analyze failure modes or problematic operating conditions. The definition of the modeling domain of relation-oriented functional modeling is depicted in Figure 2-10

Flow-oriented functional modeling is the more widely applied modeling approach in the design literature.
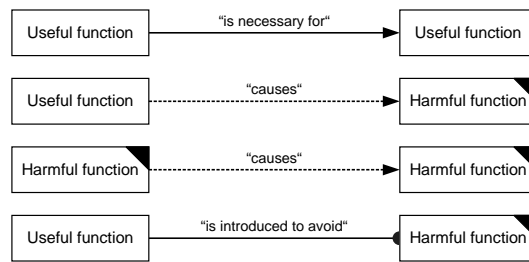
Figure 2-9: Patterns of relation-oriented functional modeling, translated from (PONN & LINDEMANN, 2011, p. 75)
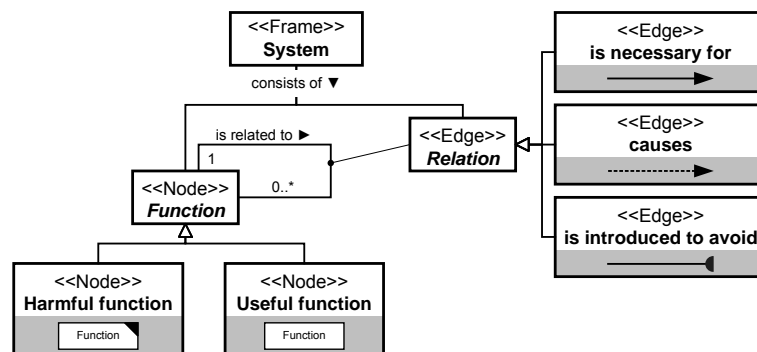
Figure 2-10: Metamodel defining the modeling syntax for relation-oriented functional modeling (the metamodel notation is described on p. 20)

Various modeling subtypes can be differentiated and are comprehensively discussed by ERDEN ET AL. (2008). The dissemination of relation-oriented functional modeling is much lower. However, it has some significance within the TRIZ[8] design methodology as detailed in (HERB, 2000). What is notable is that relation-oriented functional modeling can be seen as a simplification and a problem-oriented adaptation of flow-oriented functional modeling. Design literature, e. g. (PONN & LINDEMANN, 2011), explicitly encourages the designer to adapt modeling approaches – and by that the viewpoint on the system under consideration – according to the needs of specific design situations. This is easy to accomplish for simple, paper-based models, but, the adaptation of computational modeling tools is either not supported or requires expert know-how in programming or metamodeling. In the area of formal modeling, the formal definition of customized modeling languages is termed *domain-specific language* (DSL). DSLs are used to build the conceptual foundation for the development of customized modeling tools, e. g. when using the Eclipse Modeling Framework (EMF)[9]. While being widespread in software engineering, an increasing acceptance of DSLs in engineering design can be observed such as in the work of KERZHNER & PAREDIS (2009).

It can be stressed that the decomposition of functions, as it is described in widespread design literature, is not explicitly defined within the modeling syntax, cf. the metamodels in Figures 2-8 and 2-10. This is remarkable as the act of decomposing a function into its subfunctions is one of the essential concepts of functional modeling. In their review paper on functional modeling, ERDEN ET AL. (2008) address this issue in detail and provide a comprehensive discussion of functional modeling approaches.

---

[8]Russian acronym for Theory of Inventive Problem Solving
[9]Project website: http://www.eclipse.org/emf

2.2.2 Behavior

This level of abstraction describes how a system accomplishes the tasks required in the functional model. The abstract, function-based description is transformed into a physical description of the system. This is achieved by identifying suitable "solution principles" and combining them into a "principal solution" (VDI, 1987). This preliminary embodiment of the system is typically modeled using sketches. Especially in the mechanical engineering domain, embodiment features are associated with geometry or movement, which is more conveniently represented in a drawing. This is why no commonly accepted or standardized modeling approaches can be found in design literature as it is the case with the approaches for modeling functions. Nevertheless, a common terminology evolved over the last decades that is primarily influenced by PAHL ET AL. (2007) and designates elements of different grades of granularity on this level of abstraction.

Physical effects are equation-based representations of the physical laws defining the behavior of physical quantities. They concentrate on one specific equation disregarding side-effects or other unintended or subordinate effects. This idealized description links directly to function and contains the key mechanism for the realization of function. Physical effects can also be combined to fulfill a function (PAHL ET AL., 2007, p. 38). Design catalogs, such as issued by KOLLER (1994) and ROTH (2001), provide a large source of knowledge and methodological support for the identification of suitable physical effects. The embodiment of physical effects requires the assignment of materials and geometric properties. Doing so for one or the combination of more physical effects results in the definition of a *working principle*.
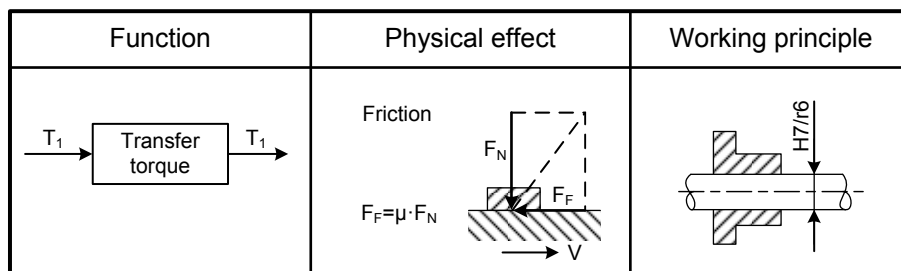
Figure 2-11: Example for a function that is fulfilled by a physical effect that is implemented with a working principle, adapted from (PAHL ET AL., 2007, p. 39)
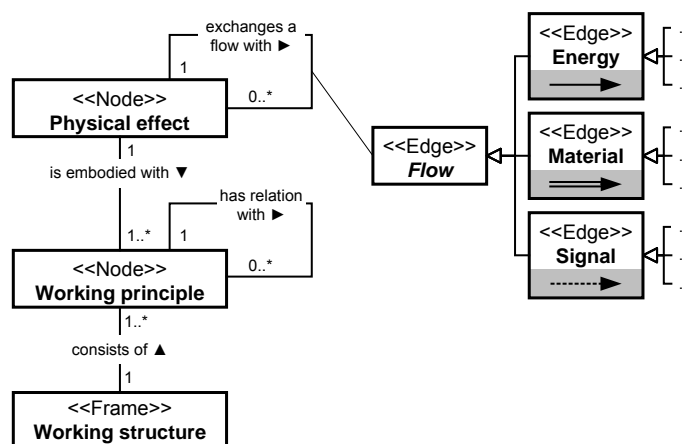
Figure 2-12: Generic metamodel defining the modeling syntax for Behavior modeling according to the textual description of (PAHL ET AL., 2007) (the metamodel notation is described on p. 20)

The example in Figure 2-11 shows the correlation between a function, a physical effect and the embodiment in a working principle: The function of transferring torque can be fulfilled based on the physical law for Coulomb friction relating the frictional force F_F to the normal force F_N through the friction coefficient μ. Consequently, the transfer of torque necessarily requires the application of a normal force. The working principle is based on a clamping hub that applies a normal force through the ISO fit H7/r6. The interaction of this working principle with other working principles constitutes the working structure that builds the foundation for the development of concrete components. The metamodel in Figure 2-12 defines these interrelationships according to the description of PAHL ET AL. (2007) in a generic representation. However, there is no established modeling standard for Behavior and various alterations of this representation can be found. Especially the relations between working principles, here generically defined as "has relation with", can be expanded to capture more specific spatial, energetic or material-related relationships and constraints.

It can be observed that, if the design task does not require the development of new components, this level of abstraction is disregarded. In this case, a direct mapping from functions to already existing components is achieved. KOMOTO & TOMIYAMA (2012) describe a conceptual design process that explicitly takes this point into consideration. GERO (1990) termed this mapping "catalogue lookup" and does not consider this as part of designing.

2.2.3 Structure

According to GERO (2004), this level of abstraction "describes the components of the object and their relationships". However, designing the detailed geometry of components based on working structures is a complex and time-consuming process that is not carried out in the conceptual phase. Consequently, this level of abstraction does not contain that kind of detailed information. In the scope of this research, the term *component* should rather be seen as a division of the working structure into "realizable modules" (VDI, 1987) that is later on further detailed to be manufactured.

Design literature provides no general indication of how to model such component structures. Nevertheless, ULRICH (1995) and the VDI guideline 2221 (VDI, 1987) stress the great necessity to define relations between components as early as the conceptual design phase. Even if this definition has to be revisited and adapted in later design phases, this preliminary layout of the system "is particularly important in the case of complex products, as it facilitates the efficient distribution of design effort. It also helps with the identification and solution of embodiment design problems" (VDI, 1987).
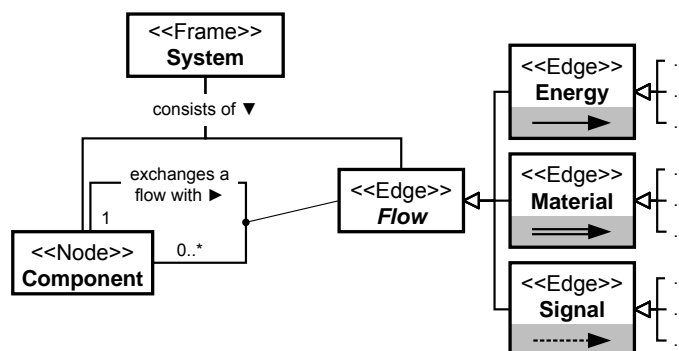
Figure 2-13: Generic metamodel defining the modeling syntax for component-based Structure modeling
(the metamodel notation is described on p. 20)

The need for representing the relations between components while having almost no information about their inner workings calls for a block-oriented representation, treating components as black boxes. Further design activities turn them into white boxes. Nevertheless, there are numerous types of relations between the components that might be relevant for consideration on this level of abstraction. The fact that components ultimately embody the functions defined earlier makes it beneficial to use the same taxonomy of flows used in functional modeling, consisting of energy, material, signal flows and their respective subtypes as defined in the Functional Basis (HIRTZ ET AL., 2002b). Moreover, geometric relations might be of interest, e. g. for addressing packaging problems. Here too, taxonomies are at hand defining geometric relations, as for example in the work of KOMOTO & TOMIYAMA (2010), which provides a class hierarchy of spatial relations. A more holistic solution has been elaborated by LIANG & PAREDIS (2004) in their work on the definition of a port ontology. It comprises a structured definition of various flow types, geometric relations, relative movements between components and logical relations.

2.2.4 Reflections on design representations

The modeling approaches presented in this section each consider one level of abstraction. Modeling constructs that could create a mapping between the mentioned levels of abstraction could not be identified in the description of how to use these modeling techniques nor in application examples of the reviewed literature. Also composition/decomposition-relationships have not been considered. For example, no explicit representation of the dependency of an overall function with its subfunctions is provided according to the functional modeling approach of PAHL ET AL. (2007).

Within this thesis the term *model* is understood as defined in the VDI guideline 3681 (2005): "Depiction of a system or process in another conceptual or concrete system, which is obtained on the basis of the application of known legitimacies, an identification or assumptions and which displays the system or the process with sufficient accuracy with respect to selected questions." *Product models* represent a specific type of model that focus on the depiction of products or future products. It is assumed in this thesis that product models are data models that only cover one level of abstraction. This assumption is backed up by various examples of product models in the design literature, e. g. function models or CAD models in (PONN & LINDEMANN, 2011, p. 21). The vocabulary and modeling logic of product models is defined in a *design representation* that is, using the method presented in this thesis, formally described as a *metamodel*. The INTERNATIONAL ORGANIZATION FOR STANDARDIZATION (2004) defines this term as a "data model that specifies one or more other data models".

Models that represent products on multiple levels of abstraction are termed *product architectures* (ULRICH, 1995). They are typically composed of the Function and the Structure level. Or, as ULRICH (1995) defines it: "(1) the arrangement of functional elements; (2) the mapping from functional elements to physical components; (3) the specification of the interfaces among interacting physical components". Also in later publications, this definition is supported by ULRICH & EPPINGER (2008). Other researchers extend this view of the product by adding working principles (ANDREASEN, 1992; KOMOTO & TOMIYAMA, 2010). STONE ET AL. (2000) additionally consider modules of functional models from which modular component models on the Structure level are identified. Other approaches can be found that support manual modeling on different levels of abstraction but domain independent implementations that synthesize such models computationally are rare (ERDEN ET AL., 2008). CRAWLEY ET AL. (2004) emphasize the importance of developing support for exploring product architectures on multiple levels of abstraction such as Function, Behavior and Structure. This issue is taken into account in this thesis and motivates the definition of the a design representation for graph-based product architectures as detailed in Section 3.1.

2.3 Overview of knowledge representations

According to Luger (2005, p. 37) the task of any representation scheme is to grasp the essential characteristics of a problem domain and make that description accessible to a problem-solving procedure. He claims that the main criteria for assessing the suitability of knowledge representations are their expressiveness and their efficiency. A good trade-off between these two dimensions has to be found. Computational restrictions or the complexity of the problem might require increasing efficiency to the detriment of the expressiveness. However, this must not lead to "limiting the representation's ability to capture essential problem-solving knowledge." The importance of representations for CDS and the fact that it represents a core challenge has been described in Section 2.1.

In the following sections, fundamentals of common knowledge representations are given. Knowledge representations can be clustered in three categories:

- *Rule-based knowledge representation:* Rules capture knowledge in a procedural manner. They can be seen as IF-THEN-relations representing the transition of an initial situation[10] to a modified situation. Often, rules contain heuristics describing expert knowledge for problem solving.

- *Model-based knowledge representation:* This category comprises a set of representations (frames, constraints, description logics) that are either built on one another or evolved in parallel. Model-based representations capture knowledge declaratively. They represent in theoretical models how a situation is or should be and rely on reasoners to perform actions for the transition.

- *Case-based knowledge representation:* Learning from problem-solving in previous situations, case-based reasoning approaches aim at solving problems based on analogies. They make use of procedural and declarative knowledge representations.

2.3.1 Rule-based knowledge representation

Rules represent a good compromise for formalizing knowledge in a comprehensible, yet formal, representation (Beierle & Kern-Isberner, 2008, p. 72). Rules are conditional sentences consisting of a condition, or premise, *A* and a conclusion, or consequence, *B*. A rule *R* can be represented in the form:

$$R : \quad \text{IF } (A \text{ is TRUE}) \text{ THEN } B$$

Depending on the nature of *B*, two types of rules can be differentiated: *Rules of inference* represent the logical connection between a condition *A* and a conclusion *B* and are usually displayed by separating them with a horizontal line (Boolos et al., 2007, p. 169):

$$R : \quad \frac{A}{B} \quad \text{e. g.} \quad \frac{\text{The fuel tank is empty}}{\text{The combustion engine will not operate}} \tag{2.1}$$

Production rules, in turn, also consist of a condition *A*, but the consequence *B* is formulated as an operation to be executed if the condition is true. As they allow for the identification of a specific situation in *A* and define the measures to be taken in *B*, they are also referred to as situation-action rules (Russell

---

[10]The term *situation* is used here to describe a system at a specific state. Knowledge-based methods can be applied to perform the transformation from an initial to a final situation. This definition is adapted from (Russell & Norvig, 2003, p. 329).

& Norvig, 2003). The condition *A* is termed *left-hand side L* whereas the consequence, or action, is termed *right-hand side R*. Production rules are displayed with an arrow pointing from *L* to *R*:

$$R: \quad L \rightarrow R \quad \text{e. g.} \quad \text{The fuel tank is empty} \rightarrow \text{Switch to electrical mode of driving} \qquad (2.2)$$

In both cases, the condition can be composed of sub-conditions being connected by logical AND and logical OR operators. Therefore, the conclusion *B* is true or, in other words, the action *B* applies when all conditions connected by a logical AND and at least one of the conditions connected by a logical OR evaluate to true.

The conclusion *B* in equation 2.1 is the result of a logical deduction returning either TRUE or FALSE. In contrast, production rules encode knowledge on how to proceed when the situation in *L* arises. This reasoning does not necessarily have to be logical. The condition of the rule in *L* is a pattern that determines when that rule can be applied to a problem instance. A production rule's right-hand side *R* contains imperative statements representing actions for modifying, replacing and adding data structures such as strings or graphs and defines by that the associated problem-solving step.

Using knowledge formalized in a rule-based representation for computational problem solving, is the underlying principle of early *expert systems*. Based on the application of inference rules, early approaches concentrated on answering questions, rather than on making decisions by modeling human problem-solving behavior. For example MYCIN, developed in the early 1970s by Shortliffe et al. (1975), was developed to diagnose bacterial infections based on a set of about 600 inference rules. The main conceptual shortcoming was the lack of a theoretical model of the application domain out of which rules could have been deduced. Knowledge had to be acquired and formalized by interviewing experts, who in turn acquired their knowledge from textbooks, other experts and through their experience (Russell & Norvig, 2003, p. 23). Consequently, the knowledge acquisition step was an expensive procedure. Another – even earlier – expert system was DENDRAL, developed in the late 1960s by Lindsay et al. (1993) for analyzing mass spectrographic data. DENDRAL infers the structure of molecules from their chemical formula and mass spectrographic data for applications in organic chemistry. DENDRAL was one of the first programs to achieve expert level problem-solving performance. However, the representational formalisms used in DENDRAL are highly specific to the domain of chemistry and could not be transferred to other application areas. Nevertheless, the approach was so successful that descendants of the system are used in chemical and pharmaceutical laboratories throughout the world (Luger, 2005, p. 23). Later generations of expert systems built on hybrid knowledge representations by integrating other paradigms for knowledge representation. Especially model-based knowledge representations were developed.

While the term *Expert System* is generally used for systems using knowledge for problem solving, the term *Production System* encompasses a subset of expert systems and is used in a more specific sense (see overview in Figure 2-14). Production systems are based on a rule-based representation as well, but solely consist of production rules. The first generation of expert systems was based on that kind of rule-based representation. Hence, first generation expert systems and production systems are considered as equivalent. Production systems are composed of a rule set, a working memory and an inference engine. They are characterized as follows:

1. The *rule set* consists of production rules each defining a single chunk of procedural problem-solving knowledge.
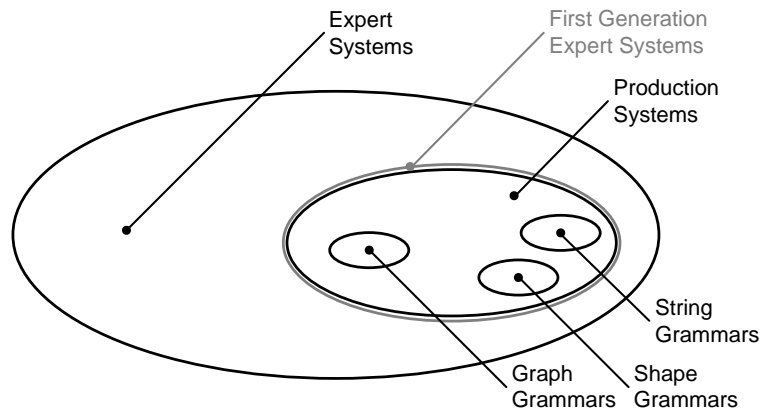
Figure 2-14: Overview of expert systems, production systems and grammars

2. The *working memory* contains the current state of the data structure in a reasoning process. This description can be a string, a shape or a graph (termed *host graph*) and represents the pattern that is matched against the condition's part *L* of a production rule to select the problem-solving action *R*. If the condition of a rule is matched by the contents of working memory, the action associated with that condition is executed. Actions of production rules are specifically designed to alter the contents of working memory.

3. The *inference engine* is a control structure that determines the logic of rule execution for problem-solving. Luger (2005, p. 200) describes a simple recognize-act cycle: The working memory is initialized with the description of the initial problem. The actual state of the problem-solving is kept in the working memory as a set of patterns that are matched against the conditions of the production rules. This results in a subset of the production rules, called the *conflict set*. The rules' left-hand sides match the patterns in working memory. Thereby, the production rules in the conflict are enabled. One of the production rules in the conflict set is selected for resolving a conflict and is fired. To fire a rule, its action is performed, resulting in a change of the contents in working memory. After the selected production rule is fired, the cycle continues with the modified working memory. The process stops when the content of the working memory does not match any rules' left-hand side conditions anymore. Based on the approach of rule sequences (presented in Section 3.2.2), various problem-solving strategies, such as the recognize-act cycle of Luger (2005) but also search or optimization algorithms, can be implemented.

Dym & Levitt (1991, p. 14) add to these three components *input/output facilities* to enable the interaction with a user and a *knowledge acquisition facility* for acquiring further knowledge, either from experts or automatically from libraries or databases. They state that these five components form the basic structure of knowledge-based systems.

Production systems can be further distinguished according to which data structures they apply, the way they are defined, the interpretative mechanism used to apply production rules and the resulting type of data structure (Gips & Stiny, 1980). This characterization is based on the term *grammar* that was first introduced by Chomsky (1957) in the sense of a mathematical concept for the generation of sequences of symbols, such as strings, shapes or graphs. As the method presented in this thesis builds on the foundation of graph grammars, a deeper insight into this specific type of production systems is given in Chapter 3.

The advantages of rule-based knowledge formalization are:

- The underlying principle of rules are widely applied in natural language in conditional sentences (BEIERLE & KERN-ISBERNER, 2008, p. 73). Hence, "production rules are a natural and intuitive way for representing heuristic knowledge" (DYM & LEVITT, 1991, p. 151). This is important in domains that rely on heuristics for problem solving, e. g. engineering design.

- Human experts tend to formulate knowledge in rule-form (BEIERLE & KERN-ISBERNER, 2008, p. 73).

- The reasoning mechanism is simple, flexible and therefore robust (RUDE, 1998, p. 72). Further, the separation of the representation and application of knowledge "enables an iterative development process" (LUGER, 2005, p. 310) and facilitates debugging of the knowledge base.

- Purely rule-based systems are efficient in applications with limited scope. In such application areas they have proven to be "extremely useful" (LUGER, 2005, p. 310).

In contrast, the disadvantages of a rule-based knowledge formalization are:

- The extraction of knowledge from the human expert and the formulation of rules is an expensive process. Often, rules acquired from experts "are highly heuristic in nature, and do not capture functional or model-based knowledge of the domain" (LUGER, 2005, p. 311). They need to be generalized to be applicable in a wider scope.

- Maintenance of extensive rule sets is very difficult, (KLÄGER, 1993; LEEMHUIS, 2004). PUPPE (1990) illustratively names deeply intertwined rule sets "rule spaghetti".

- The reasoning logic for solving complex problems that involves the concatenation of procedural rules is hard to grasp for the human user (RUDE, 1998, p. 72).

- The knowledge in manageably small rule sets tends "to be very task dependent. Formalized domain knowledge tends to be very specific in its applicability" (LUGER, 2005, p. 311).

2.3.2 Model-based knowledge representations

Frame-based and object-oriented representation

The term *frame* was first introduced by MINSKY (1975). The intention was to capture knowledge about a problem domain in organized data structures. More specifically, frames allow for the explicit representation of implicit connections of information and support by organizing knowledge into more complex units reflecting the organization of objects in the domain (LUGER, 2005). Frames can be considered a specific implementation of semantic networks, which provide an earlier scheme for representing knowledge in graph-based, yet less structured, manner (SRIRAM, 1997, p. 123). Frames also use graphs as the underlying data structure that contain a fixed set of relation types and realize the key features of object-orientation. This representation can be seen as an attempt to handle the overwhelming amount of information in large semantic nets without decreasing their high expressiveness (DYM & LEVITT, 1991, p. 140). As a result, the disadvantage of ill-defined names and edges can be partially avoided. An overview of graph-based structures for knowledge representations is given in LEHMANN (1992).

In the following, essential concepts of this knowledge representation are depicted using a frame description of a car as an example, Figure 2-15. Note: This is just a small sample of a more comprehensive
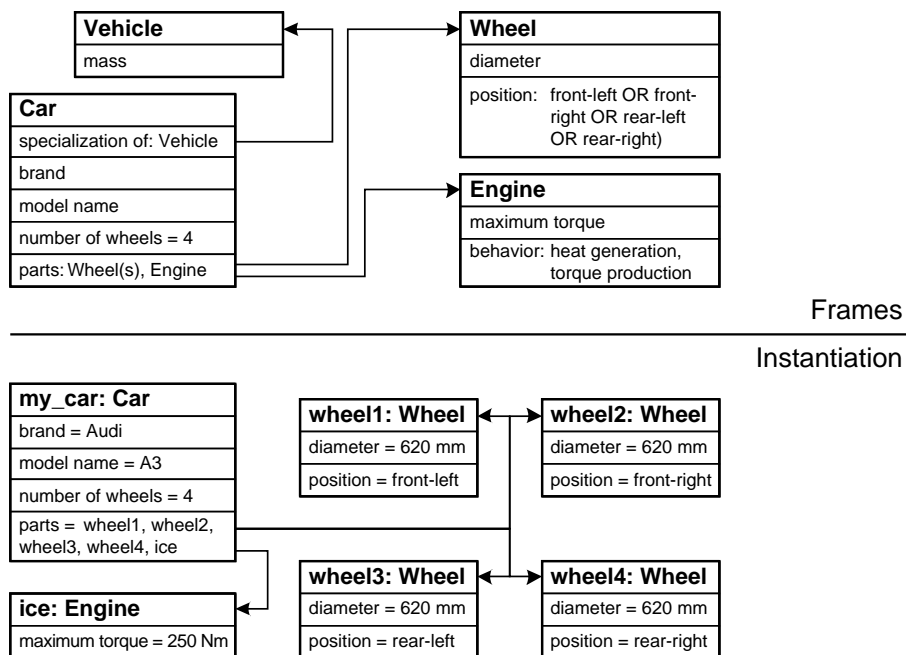
Figure 2-15: Part of a frame-based description of a car and example instantiation (representation schema adapted from (LUGER, 2005))

frame model. A frame is a form-like definition of an object, e. g. car, describing the object's characteristics by means of *slots*[11] , e. g. number of wheels. *Instances* can be derived from each frame. They assign concrete information to the characteristics of a frame's definition. Slots are attribute-value pairs (e. g., attribute: brand, value: Audi) and they can represent various kinds of information, (LUGER, 2005, p. 245), such as:

- *Frame identification information:* Information for the differentiation of individual frame instances, e. g. brand, model name.

- *Relationship between frames:* This kind of slot allows the representation of the dependency between frames. In this example two relation types can be distinguished:

  - *Generalization/specialization*: Vehicle is a more general frame than car. One could also say that car is a specialization of vehicle. This dependency is indicated by the specialization slot and the arrow pointing from the specialized term car towards the generalized term vehicle. Generalization/specialization relations constitute hierarchies that are commonly referred to as *is-a-hierarchies* or *hierarchies of inheritance*.

  - *Aggregation*: A car is, among other components, composed of wheels and an engine. This kind of relationship is represented by the slot parts while the slot number of wheels defines that exactly four wheels are required. This type of property defining the number of frames in a set that can be associated to a frame is termed *multiplicity* or *cardinality*. Aggregation relations constitute hierarchies that are commonly referred to as *part-of-hierarchies* or *hierarchies of aggregation*.

---

[11]In his paper of 1975, MINSKY used the term *terminal*. However, since then, the term *slot* has become established.

- *Descriptors of requirements for a frame:* These requirements must be fulfilled when a new instance of that frame is created. For example, a wheel has to have a specific position that is defined by the slot position and is set to front-right for wheel2.

- *Procedural information on use of the structure described:* This is a feature that shows the proximity to programming languages. In particular the possibility of declaring and defining instance functions. It enables linking to procedural descriptions of a frame's characteristics, e. g. a simulation model for representing a specific behavior of a frame as for example heat generation or torque production in an engine. A simulation language, such as MODELICA[12], is well-suited to model the physical behavior of an engine. Hence, this concept allows for integrating representations that are better suited for specific purposes.

- *Frame default information:* Unless otherwise provided, these values are assigned by default to instances, e. g. the multiplicity of the wheels in the slot number of wheels.

- *New instance information:* Many slots are left unspecified, such as the maximum torque of an engine. Values have to be assigned upon creation of a new instance, e. g. maximum torque = 250 Nm for an internal combustion engine.

Frames are used as blueprints to define instances. In programming, for this concept the term *class* is commonly used to describe a data structure that can be instantiated to create class objects or class instances. To prevent confusion, the term *class* is used in the context of the development of the software prototype booggie in Chapter 5, whereas the term *type* designates blueprints of instances in the context of modeling and knowledge representation.

Frame-based representations support type inheritance. The slots and default values of a frame type are inherited across the subtype hierarchy. For example, the mass property of the vehicle frame automatically becomes a property of the car frame as well and all of its potential subtypes through inheritance. This implies that a system queries all supertypes for slots that have to be filled upon instantiation. Frames were the first knowledge representation explicitly defining the concept of inheritance as part of the representation syntax. Another important feature in terms of the expressiveness is the possibility to include procedural, or code-based, knowledge representations. Thereby, frames combine the declarative nature of knowledge representation with a procedural representation.

The use of frames can also be interpreted from a modeling perspective. The lower part of Figure 2-15, i. e. the instantiation of the frames, is a model that uses the element types defined in the upper part. In turn, the upper part can also be interpreted as a model; namely the type model of the instance model. Such types of models are commonly termed *metamodel* in accordance with the definition presented in Section 2.2.4.

The similarity of MINSKY's work to ideas that developed during the rise of object-oriented programming led to the accusation he was recycling prior work (RUSSELL & NORVIG, 2003, p. 366). Especially the concepts of default values, inheritance and separation between type- and instance-level are particularly important in programming as well. These concepts were already in use in the late 1960s when DAHL ET AL. (1970) developed the programming language SIMULA, which is considered the first object-oriented programming language. As the name suggests, SIMULA is a simulation modeling language primarily focused on the description of physical systems.

---

[12]Website of the MODELICA association: http://www.modelica.org

Research efforts towards structuring knowledge using frame-based, or object-oriented, representations can be found in the field of engineering design. The Core Product Model 2[13] (CPM2) (FENVES ET AL., 2008) captures product information, e. g. functions, behavior, geometry, material, to provide a template of a product model that is not specific to any application, software or product development process. Similarly, the goal of the MOKA[14] project (STOKES & MOKA CONSORTIUM, 2001) was to develop a general framework for structuring and representing engineering knowledge. The result was a two-sided modeling framework containing an informal model using natural language and a formal model using an object-oriented representation. Both, the formal MOKA representation and CPM build on class definitions using UML[15] that use the same concepts as frames. In both projects, the importance of incorporating multiple levels of abstraction within one product architecture is acknowledged. For a review of product modeling frameworks that focus on the exchange of product information, refer to (FENVES ET AL., 2005).

The advantages of frame-based representations are as follows:

- The formalism of inheritance allows an economical, hierarchical representation of element properties. Every attribute does not have to be repeated at every level but is inherited. Further, "a hierarchical classification is usually a fairly comprehensive classification since all rules for aggregation and distinction must be made a priori." Hence, a lot of knowledge about the problem domain, i. e. "entities, their attributes, and the important criteria" has to be gained in advance (KWASNIK, 1999).

- High level of expressiveness through the combination of declarative and procedural knowledge representation.

- Both, hierarchical dependencies (inheritance) and network-like dependencies (slots) can be represented. Thus, frames provide an efficient approach for organizing knowledge in a declarative representation.

- Complex objects can be represented as a single frame, rather than as a large network structure. This provides a natural and intuitively understandable representation (LUGER, 2005).

In terms of expressiveness, economy of notation and comprehensibility, frames are a very good compromise. However, the following issue remains as a disadvantage:

- Frames only provide means to define pointers to procedural descriptions within slots, such that, an additional representation, e. g. a programing language or a rule-based knowledge representation, is required to reach their full potential.

Constraint-based representation

Dependencies between objects can be described as a mathematical problem based on *constraints*. These are defined as conditions on objects' variables that have to be met for obtaining a valid solution. If multiple variables are interrelated, a constraint network is defined. Hence, the goal of solving a *constraint satisfaction problem* (CSP) is to find valid values for each problem variable.

---

[13]The initial Core Product Model was developed at the American National Institute of Standards and Technology (NIST) and further developed to a generic, abstract model in the CPM2.

[14]**M**ethodology and tools **O**riented to **K**nowledge based **A**pplications

[15]**U**nified **M**odeling **L**anguage

One of the earliest famous constraint satisfaction problems was the 8-queens problem, Figure 2-16. It is solved when eight queens are placed on a chessboard such that no queen can be attacked by another queen (RUSSELL & NORVIG, 2003, p. 66). This problem was formulated in 1848 by the chess player Max Bazzel and serves since the early days of constraint satisfaction research as a benchmark problem. It can be solved with a recursive backtracking algorithm (FREUDER & MACKWORTH, 2006). This problem was popular in recreational mathematics in the late 19th century (LUCAS, 1891) and led to the first approaches for solving CSPs computationally (LEHMER, 1957).

Figure 2-16: One possible solution for the 8-queens problem (TSANG, 1993)

FREUDER & MACKWORTH (2006) state that the "interest in constraint satisfaction developed in two streams":

The *language stream* focused on the development of languages for modeling, programming and solving CSPs. Including constraint-type statements in procedural programming languages, such as FORTRAN, that had already been proposed in 1964 by WILKES. In 1967 the logic programming language Absys was developed that built on a declarative representation (ELCOCK, 1990). A specific language for expressing CSPs as means for representing knowledge about electrical circuits is CONSTRAINTS; it has successfully been applied for synthesis tasks (SUSSMAN & STEELE, 1980) in electric circuit design. Another early application in the domain of engineering is the system called CARI. It represents the knowledge of the human expert to plan "the sequence of machining cuts of mechanical parts" (DESCOTTE & LATOMBE, 1981).

The *algorithm stream*, in contrast, was primarily influenced by the research community in the area of machine vision. Using a network-based representation, MONTANARI (1974) developed a formal framework for the representation and handling of constraints for picture processing. Representing the consistency of constraints in graphs and the development of algorithms for eliminating inconsistencies within these structures has been another achievement that emerged out of research on machine vision (MACKWORTH, 1977). In general, research in this stream focused on generic computational methods for solving CSPs, (e. g. (HARALICK ET AL., 1978; HARALICK & SHAPIRO, 1979, 1980)) rather than responding to the need to solve CSPs for specific applications.

The following conclusions can be drawn out of these two research perspectives on CSPs: The language stream indicates that this representation type is an appropriate way to formalize engineering knowledge for synthesis purposes. For design tasks involving systems' parametric properties, e. g. dimensioning of parameters, this representation is a natural fit. On the other hand, the algorithm stream shows potential for combining a graph-based representation with constraint modeling. Further, the strong focus on the

development of methods for solving CSPs led to a high maturity of computational toolkits; an overview is given in (Frühwirth & Abdennadher, 2006).

The combination of a mature modeling foundation together with robust computational solving methods makes the constraint-based representation a promising all-round package for modeling and solving parametric design problems.

According to Freuder & Mackworth (2006), a CSP P can be defined as a triple $P = (X, D, C)$ where X is a n-tuple of variables $X = (x_1, x_2, x_3, ..., x_n)$. Each variable x_i has a domain d_i of possible values contained in the n-tuple $D = (d_1, d_2, d_3, ..., d_n)$. C is a t-tuple of constraints $C = (c_1, c_2, c_3, ..., c_t)$ where each constraint c_j contains a subset of the variables X and specifies valid combinations for that subset according to D. The solution of a CSP is a n-tuple $A = (a_1, a_2, a_3, ..., a_n)$ where each a_i contains the valid value for a constraint x_i that does not violate any other constraints.

Often, constraints represent quantitative relations between parameters such as in the work of Münzer et al. (2012) where constraints are used for dimensioning automotive powertrain components. In the scope of this thesis, these constraints are termed *parametric constraints*. On the contrary, *topological constraints* represent topological relations between components. Rozenblit & Hu (1992) use the term *coupling constraints* and state that they "impose a manner in which components [...] can be connected together"[16]. In Section 3.2.1 the concept of *ports* is introduced allowing to clearly represent structural constraints. An example that illustrates the theoretical foundation for modeling constraints leading to the definition of ports can be found in the Appendix 9.1.

Rude (1998) states that constraint-based representations are a powerful approach for representing design inherent dependencies, which is being widely applied in combination with a frame-based or rule-based representation within the expert systems of the second generation. One of the key applications in engineering is the solving of geometric constraints in CAD systems (Hoffmann, 2005).

Advantages of constraint-based modeling and constraint solving are:

- Constraint modeling is a powerful declarative representation for parametric relations, such as geometric constraints in CAD systems.

- CSPs are a formal representation that is computationally well supported. Various professional software tools are available, e. g. Matlab[17].

- Constraints allow the definition of boundary conditions within which a solution has to be found without defining the solution itself. (Rude, 1998)

- Constraints can be used in a bidirectional manner as they represent equations that can be rearranged, e. g. $x = y - 1 \Rightarrow y = x + 1$. Hence, the user does not have to specify inputs and outputs; the right ordering of equations is achieved by the constraint solver (Leemhuis, 2004).

Disadvantages of this knowledge representation are:

- Representing topological constraints purely using constraint modeling is cumbersome and unintuitive. Ports can be seen as a simplification layer to overcome this shortcoming.

---

[16]Other common terms are *structural constraints* or *network structure constraints* such as used in (Wyatt et al., 2012)

[17]Company website: http://www.mathworks.de/products/matlab

- Constraint-based representations have a rather mathematical appearance. Engineers are used to more visual representations, which require an additional representation only for the graphical user interface, such as those implemented in current CAD systems.

- Constraints are rarely isolated. In most cases, they are concatenated in constraint networks, which can easily become confusing.

For a comprehensive overview on using CSP in engineering design, refer to BETTIG & M. HOFFMANN (2011).

Description logics and ontologies

While frames can be seen as one specific type of semantic networks, *description logics* (DL) evolved from them responding to the need to formalize the networks' meaning while keeping emphasis on taxonomic structure as an organizational principle (RUSSELL & NORVIG, 2003, p. 353). According to BAADER (2010), a DL-based knowledge base consists of two components: the *TBox* and the *ABox*. The TBox defines the set of vocabulary, i.e. terminology and their relationships. It can be seen analogously to the definition of types in frame-based representations. Assertions about instances of TBox-types are contained in the ABox. NARDI & BRACHMAN (2010) claim that the knowledge about terminology in the TBox is considered to remain static. Knowledge that is specific to instances and is contained in the ABox is seen as time-dependent or "subject to occasional or even constant change". In this context, types are termed *concepts*, relationships are termed *roles* and instances are termed *individuals*. An illustrative example in the Appendix 9.2 presents the key elements of DL.

The first system that built on DL was KL-ONE (BRACHMAN & SCHMOLZE, 1985). It was developed in the late 1970s and introduced most of the language features that were further explored in subsequent research. While the application of KL-ONE targeted the modeling on the concept level, i.e. TBox, CLASSIC was developed at AT&T (BORGIDA ET AL., 1989) to support software engineering of large software systems. In the domain of medicine, expert systems showed the potential of knowledge-based methods but also their limitations coping with complex problem domains with a solely rule-based representation. The challenge of constructing and maintaining large-scale knowledge bases, that contain up to hundreds of thousands of concepts, led to the development of such specialized DL systems, for example GALEN (RECTOR & NOWLAN, 1994). The need to combine multiple knowledge sources made language standardization necessary (NARDI & BRACHMAN, 2010). PATEL-SCHNEIDER & SWARTOUT (1993) proposed a language specification approach to facilitate knowledge sharing, which resulted in the DL standard language KRSS.

The use of DL approaches for reasoning on XML documents laid the foundation for research efforts in capturing knowledge contained in the World Wide Web (NARDI & BRACHMAN, 2010). The goal was to "support intelligent applications that can better exploit the ever increasing range of information and services accessible via the web" and led to the development of the Ontology Web Language (OWL) (HORROCKS ET AL., 2010). GRUBER (1993) defined *ontologies* as "an explicit specification of a conceptualization. The term is borrowed from philosophy, where an ontology is a systematic account of Existence. For knowledge-based systems, what 'exists' is exactly what can be represented." OWL proved that description logics are a suitable means for the representation of ontologies. Due to its high usability and useful reasoning techniques, OWL has been recommended by the World Wide Web Consortium as the language for the Semantic Web (HORROCKS ET AL., 2010). The expressiveness of OWL and the availability

of convenient editing tools, such as PROTÉGÉ (Gennari et al., 2003), led to a wide range of application, also in the domain of engineering design.

Commercial applications in the area of product configuration have reached a high level of maturity, e. g. for the online configuration of personal computers at Dell (McGuinness, 2010). Based on empirical studies, Ahmed et al. (2005) developed the design ontology EDIT (Engineering Design Integrated Taxonomy) for representing the development process, design rationale and product functions. EDIT is used for analyzing and indexing text-based documents to support search queries in the product development process. An overview of research projects for supporting engineering design using ontologies is given in (Kim et al., 2008).

Advantages of description logics and ontologies can be summarized as follows:

- Description logics combine the advantages of semantic networks and frames, regarding the expressiveness of the knowledge representation.

- Description logics are not bound to any visual appearance. Depending on the use case, form-based or graph-based visualizations can be used.

- Large knowledge-bases are already available. Most of them are based on standard languages and can hence be combined and reused. CYC for example is an ontology that aims to collect general knowledge and supports a knowledge base for semantic data mining (Foxvog, 2010). The Ontology Server[18] of the Knowledge Systems Laboratory (KSL) at Stanford University supports collaborative editing and creation of ontologies.

- Modeling topological constraints is inherently supported by DL. Dedicated solvers, e. g. SAT-solvers, are available for solving those kinds of constraint satisfaction problems and identifying models that satisfy the TBox' terminology (Horrocks, 2010).

Disadvantages of description logics and ontologies include:

- Using mature software tools, the definition of an ontology is intuitively feasible with little training. However, the formulation of reasoning queries requires a deep understanding of the query language and expert knowledge of ontologies.

- Natively, description logics do not support the link to procedural rules, such as frame-based representations. Such an extension has been proposed and the DL languages CLASSIC and LOOM support the integration of procedural rules (Baader et al., 2010). However, recent implementations concentrate on the further development of DL-solvers instead of their integration with systems for procedural rule execution, e. g. graph transformation systems.

- Modeling parametric constraints is not a native feature of DL but is feasible. Nevertheless, the integration of CSP languages and solvers is still subject to current research. An approach for mapping simple CSPs from an ontology-based representation to CSP-solvers has been elaborated by Mao et al. (2008).

---

[18]Website of the KSL Ontology Server Projects: http://www.ksl.stanford.edu/software/ontolingua/ontology-server-projects.html

2.3.3 Case-based knowledge representation

Humans that are faced with complex problems tend to draw analogies from previous similar solutions. They do not solely use fundamental knowledge but build on individual experience about previously solved cases. Problem-solving through analogy-based reasoning is one of the most important human inference methods (RUDE, 1998, p. 84). A fundamental prerequisite for case-based reasoning (CBR) is a knowledge base that contains knowledge about previous *cases*. A case consists of a problem definition together with a situation description in which the problem occurred and the solution that solved the problem.

In literature, slightly different processes for CBR reasoning can be found (CARBONELL, 1983; KOLODNER, 1992; RUDE, 1998; LUGER, 2005). Typically, they accomplish the following four tasks:

1. *Find previous cases for similar problems:* A previous case is relevant if its solution can be applied to a new problem. For identifying promising previous cases, solvers search for similar cases. To do so, they apply the same heuristic as humans would do. They identify the similarity based on common characteristics. If, for example, two patients have common symptoms and a similar medical history, it is likely that they have the same illness and respond similarly to the same medical treatment (LUGER, 2005, p. 306). To enable efficient search of suitable cases, the underlying structure of the case repository, i. e. the case-based knowledge representation, has to provide for this kind of indexing.

2. *Adapt previous solution:* The solution of a problem usually consists of a process that requires the execution of operations that transforms a problem state into a solution state. However, these operations might require modifications to be applicable in a new situation. "Analytic methods, such as curve fitting the parameters common to stored cases and new situations can be useful; for example, to determine appropriate temperatures or materials for welding" (LUGER, 2005).

3. *Apply solution:* Although the solution has been adapted, the application of the solution in a new situation can raise issues that were unknown in previous cases. This may trigger iterations of steps 1 and 2.

4. *Evaluate solution and save case:* Once a problem has been solved in a new situation, a new case can be stored in the case repository. To retain valuable information for future use, it is important to keep track of the success or failure of a solution. This requires an update of the knowledge base indices. "There are methods that can be used to maintain indices, including clustering algorithms (FISHER, 1987) and other techniques from machine learning (STUBBLEFIELD & LUGER, 1996)" (LUGER, 2005).

CBR can be seen as a cyclical process of learning from past experience to solve a problem, learning from that new experience to solve future problems and so forth. The link to (machine) learning is the clear distinguishing factor from rule- and model-based representations; this approach is also referred to as case-based learning. The major advances arose in the machine-learning community since CBR systems can complement expert systems through learning (AAMODT & PLAZA, 1994). They allow to continuously acquire knowledge by gathering and storing more cases. Potential information sources are historical records or the observation of current operations reducing the effort for the human expert to externalize expert knowledge (LUGER, 2005, p. 308).

LUGER (2005) states that design is an obvious application area for CBR as "aspects of a successfully executed artifact may be appropriate for a new situation, and diagnosis, where the failures of the past often recur". In the work of GOEL ET AL. (1996) in the late 1980s, the representation of components and their composition is used for the indexing of previous designs to provide assistance for future designs, and has been implemented in the software KRITIK. It combined case-based reasoning for proposing new solutions with model-based reasoning for the verification of a new solution. A recent application of this approach is the construction of structural models from unlabeled 2D line drawings (YANER & GOEL, 2007).

An early industrial realization of CBR has been developed by BARLETTA & HENNESSY (1989) with their system CLAVIER at Lockheed Corporation. At that time there was no theoretical model available to predict the behavior of composite components during the baking process. Through the assessment of a human expert, experiences with previous composite parts were recorded in a knowledge base. CLAVIER proposed the subdivision of complex geometries to prevent an unsatisfying baking process.

For a detailed introduction to case-based design including the representation of design cases, indexing and retrieving design cases and the range of paradigms for adapting design cases refer to (MAHER ET AL., 1995).

The advantages of CBR can be summed up as follows:

- The similarity to the human problem-solving process makes this approach intuitively understandable (RUDE, 1998, p. 86).

- The ability to include knowledge about historical cases enables the exploitation of existing knowledge sources such as maintenance reports or working protocols. The tedious and expensive formalization of expert knowledge can be avoided. Hence, CBR provides an approach to learn from previous cases for problem solving but also for updating the knowledge base based on the experience of new cases (LUGER, 2005, p. 311). In CBR systems, problem solving and learning are seen "as two sides of the same coin" (AAMODT & PLAZA, 1994).

- CBR does not require an upfront model of the problem domain. Instead, it "allows a simple additive model for knowledge acquisition". However, this poses great challenges to an appropriate representation of cases, a useful retrieval index, and a case adaptation strategy (LUGER, 2005, p. 311).

The following disadvantages of CBR can be observed:

- The identification of suitable indexing criteria is difficult. It primarily determines the ability to find similar cases and hence the quality of solutions proposed. LUGER (2005) states that this "can offset many of the advantages CBR offers for knowledge acquisition".

- Cases do not include any deeper theoretical model of the problem domain, such as those that model-based approaches would. Hence, the knowledge representation itself does not support reasoning about cases and support explanation. This might lead to the misinterpretation of cases and hence to the application of unsuitable solutions (LUGER, 2005).

2.3.4 Reflections on knowledge representations

Retrospectively, a trend towards *integrated* – or, according to (Luger, 2005), *hybrid – knowledge representations* can be observed. This trend becomes obvious when considering the evolution of the term *expert system.*

Rule-based representation is the oldest technique for formalizing domain knowledge. Hence, in the 1960s and 1970s the term expert system was used synonymously with production systems, i. e. rule-based expert systems. The lack of expressiveness and of an underlying model of the problem domain limited the scope of these systems to narrow applications. For a comprehensive review of limitations of first generation expert systems, refer to (Clancey, 1985; Partridge, 1987).

Second generation expert systems attempted to overcome these limitations by combining rule-based and model-based representations. It can thus be said that expert systems emancipated themselves from purely rule-based systems. In a later definition by Feigenbaum (Barr & Feigenbaum, 1982), expert systems are generally seen as systems for problem-solving: "An intelligent computer program that uses knowledge and inference procedures to solve problems that are difficult enough to require significant human expertise for their solutions." That is, regardless of the type of knowledge representation, an expert system is a computer system mimicking the human problem-solving behavior.

Case-based reasoning systems are not subsumed under the notion of expert systems, but they are seen as a suitable complement for expert systems. They offer the potential to include the functionality of learning from past experience. Further, CBR allows for the simplification of knowledge acquisition for expert systems. Often, the search for previous solutions is less tedious than the acquisition of theoretical or heuristic knowledge for constructing knowledge bases (Althoff et al., 1989).

In closing it can be stated that the combination of knowledge representations in hybrid representations enables finding better trade-offs between their strengths and weaknesses. Hence, more complex and difficult problems in a larger scope of application can be solved (Rozenblit & Hu, 1992). In particular, Dym & Levitt (1991, p. 151) stress the relevance and suitability of a combination of rules, frames and object-oriented principles for synthesis tasks in design. The significant advantages offered by this kind of hybrid representation combining model-based and rule-based knowledge representations, as discussed by Luger (2005, p. 313), motivate this direction to deliver the expected contributions 1 (high efficiency of object-oriented graph grammars) and 2 (high effectiveness of object-oriented graph grammars). For this reason, within the remainder of this thesis, CBR is not further regarded in detail. However, it shows great potential for extending this work towards learning or self-improving design synthesis support.

2.4 Related work

The previous two sections introduced the key concepts of design representations and knowledge representations. These are the two dimensions based on which the related state of the art in CDS is structured. The research is grouped according to underlying knowledge representations while indicating the levels of abstraction and modeling elements that are used for the design representation. Emphasis is placed on the scope of application versus the applied knowledge representation to identify the gaps in previous work and to obtain a deeper understanding of the problem descriptions from Chapter 1.

The terms previously introduced are used in the subsequent sections. In cases where there are deviations regarding the terminology, the terms used in the respective approaches are indicated in parentheses.

According to the scope of this thesis, the investigated CDS approaches support the conceptual design phase through the computational generation of graph-based, or at least graph-like, product models.

2.4.1 Rule-based CDS approaches

The use of rule-based systems for solving architectural design problems can be traced back to STINY & MITCHELL (1978) who developed a production system that built on shape grammars and generated a set of visual layouts of ground plans of Palladian villas. Two years later, GIPS & STINY (1980) provided a uniform characterization of production systems and grammars highlighting their commonalities and differences, which had a great influence on subsequent research in CDS. Accordingly, graph grammars are production systems that are based on vocabularies of graph symbols, i.e., nodes, edges, and their combinations. They are able to define the evolution – or synthesis – of graph-based product models representing the involvement of engineering knowledge to synthesize design solutions.

Based on the work of REDDY & CAGAN (1995), a grammar-based approach for the synthesis of truss structures was developed by SHEA (1997) and later on incorporated in the software tool eifForm (SHEA ET AL., 2005). In conjunction with finite element simulation, stochastic search methods seek for an optimal solution of two- or three-dimensional truss structures. The rule set defines the valid operations for the search method to modify the truss topology and aims to minimize stress, buckling and displacement while maximizing usage and aesthetic requirements. The formulation and application of rules is depicted in Figure 2-17. Although this approach is categorized as a shape grammar application, the graph-like nature of rules and synthesized truss structures is obvious: truss members are considered as edges and the joints between them are represented as nodes.

(a) Definition of topology modification rules   (b) Example rule applications
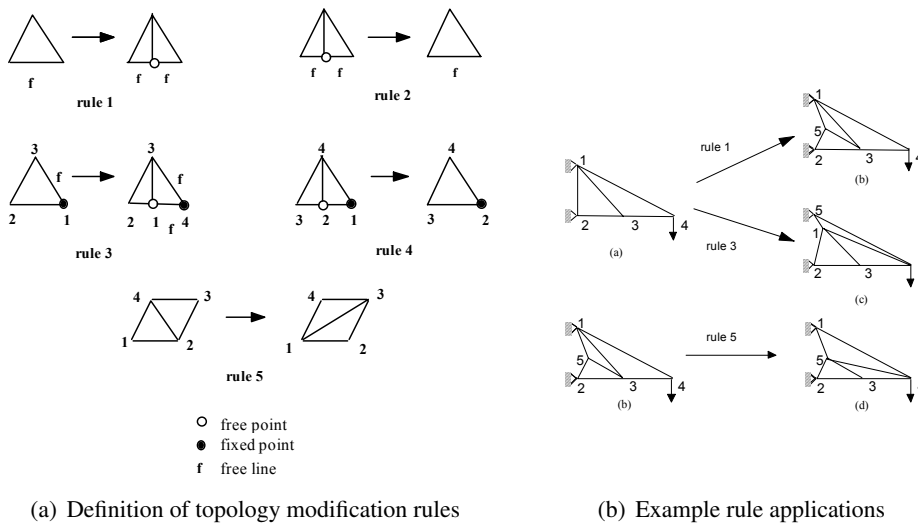
Figure 2-17: Synthesis of planar truss structures (SHEA, 1997)

The work by STARLING & SHEA (2005) also synthesizes graph structures containing geometric properties. The application area is the synthesis of gear systems that integrates a simulation-driven evaluation of the systems. A simulation model in the physical modeling language MODELICA is generated that is simulated in the software tool DYMOLA[19] and provides for the assessment of the physical behavior of synthesized solutions. The rule set contains the knowledge about valid operations of adding, removing and modifying

---

[19]Company website: http://www.3ds.com/Dymola

gears and shafts. It is subdivided into C(reate)-Rules for adding and removing components and P(erturb)-Rules for parametrically modifying existing components. In the underlying graph-based representation, shafts are represented as nodes while gear pairs are modeled as edges. This work initially focused on the synthesis of clockwork mechanisms and has been further developed as a plugin for the commercial gearbox design tool RomaxDesigner[20] (Lin et al., 2009). The knowledge for the gearbox synthesis was formalized in nine rules distinguished by six topological and three parametric rules. This shows the general applicability of CDS approaches within a focused scope of application in an industrial context.

MEMS are micro-electro-mechanical systems. The generation of MEMS designs using a shape-grammar-based representation has been developed by Agarwal et al. (2000). Using a graph-grammar-based representation, Bolognini et al. (2007) automatically generate microresonators under the consideration of the two design criteria of resonant frequency and device size. The synthesis method is called CNS-Burst and integrates a generate-and-test algorithm (Burst) with a graph-based design representation termed Connected Node System (CNS). The synthesis process is based on five general modification operators. Although these operators were hardcoded in the source code of the software prototype, they are interpreted as graph transformation rules as they procedurally represent the modification of a graph-based structure. Their application in a synthesis process results in the generation of systems as depicted in Figure 2-18. MEMS are multi-domain systems comprised of structural, mechanical and electrical characteristics. This has been taken into account by providing links to multiphysics simulation for quantitative evaluation of design performance throughout the synthesis process.
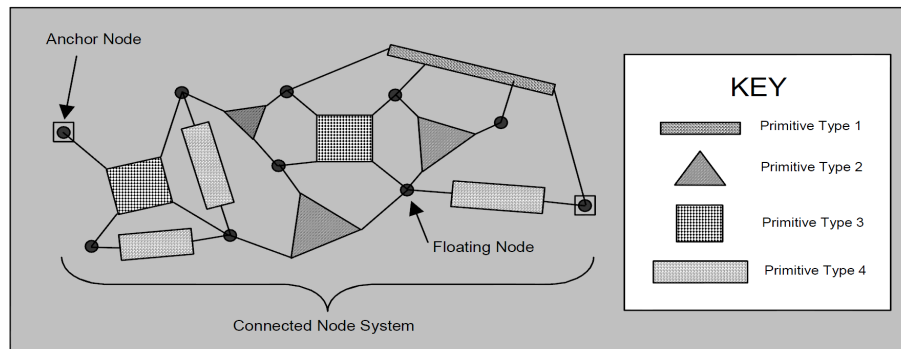
Figure 2-18: Example of a connected-node system (Bolognini et al., 2007)

The previous approaches are mostly based on the combination of graph (or graph-like) grammars, automated simulation and multi-objective search. However, each application uses a different computational representation, all including behavior and structure levels to some degree. Each is focused on a specific scope of application. The integration of the behavior level is achieved through mapping to simulation tools, rather than synthesizing models that explicitly contain behavior elements, e. g. physical effects. Only little expressiveness of the knowledge representation is required as the variety of the modeling elements is small and does not require concepts like inheritance or generalization to define the modeling domain. In the respective approaches, nodes correspond to only one specific type of element (e. g. joints, shafts, resonator primitives) similarly with the edges (e. g. trusses, gear pairs). The knowledge representation of Bolognini et al. (2007) slightly touches on the integration of model-based knowledge formalization. Primitive types are defined that encapsulate common characteristics, possible interaction points (defined as specific port nodes) and are instantiated in the particular MEMS models.

---

[20]Company website: http://www.romaxtech.com/RomaxDesigner

The integration of model-based aspects in knowledge formalization is taken further by KERZHNER & PARE-DIS (2009) who use the formal modeling language SysML to define a hierarchy of components, including structure, dynamics behavior, cost and the components' compatibility. This definition is considered a *domain-specific language* (DSL) that is built on top of the standard language SysML. Instead of using this DSL for the formalization of procedural engineering knowledge in graph transformation rules, the metalanguage MOF is used within the external tool MOFLON. This is disadvantageous in the sense that a mapping between these languages has to be defined to assure that the grammar complies with the modeling elements defined in the DSL. This approach is successfully applied to generate fluid-power circuits, where the application of the four design rules uses a random selection of rules. The generated layouts are based on the standard language SysML.

The graph grammar implementation GraphSynth is an open-source software that served in the past few years as a platform for multiple synthesis applications. It primarily uses labels and variables to allow differentiation of nodes and edges. It features integrated search and parametric optimization algorithms; additional search algorithms can be added as plugins. GraphSynth provides a graphical user interface for the definition of rules and graphs. It has been applied to the synthesis of sheet metal parts (PATEL & CAMPBELL, 2010), disassembly sequences (AGU & CAMPBELL, 2010) and to applications targeted at computationally supporting the conceptual design phase which are presented in the following paragraphs.

Based on a predecessor of GraphSynth, SRIDHARAN & CAMPBELL (2004) analyzed the function structure of 30 products and derived a set of 69 rules. These rules were defined manually and capture the principles of decomposing an overall function into a more detailed function structure. To do so, the overall function is considered as a black box and the rules for adding the necessary subfunctions for translating the input flows into the output flows are derived. This defines a grammar that can be reused later on to automate this task for similar design problems. The approach has been validated in an empirical study showing that students applying this grammar achieved better results than without this kind of tool support.

KURTOGLU & CAMPBELL (2006) have gone one step further and integrated components, i. e. Structure level, as an additional level of abstraction of the design representation. They implemented an automated mapping from a function structure to a component structure (termed configuration flow graph) through the definition of a graph-grammar comprised of a set of 170 rules. The rule set was built based on the analysis of the dissection of 23 electromechanical products stemming from a web-based design knowledge repository. The authors state that the "design knowledge is systematically extracted and integrated into a graph-grammar-based framework". However, details of this extraction process remain unclear. Also in later publications (KURTOGLU ET AL., 2010), the derivation of rules is not further characterized. It is hence assumed that the formulation of the rule set was carried out manually.

The two previous approaches adopt the nomenclature of the Functional Basis (HIRTZ ET AL., 2002b) as a foundation for synthesizing function structures. This ensures consistency and repeatability for their representation. Another common aspect is the large size of the rule set. The grammars are derived using a data-driven approach which, in turn, is based on product data from design repositories and solely includes knowledge about the analyzed class of products. The formalization of established design processes or procedures from design theory, e. g. for a functional decomposition, could provide for a high degree of generality. Under the assumption that the grammars were defined manually, the generation process must be extremely time-consuming. Also, maintenance and keeping the knowledge base up-to-date involves a considerable effort. Another issue is that expanding the scope of application (integration of Structure

level by KURTOGLU & CAMPBELL) directly causes an increase in the size of the rule set. This restricts the scalability of this approach towards a wider range of applications.

Another, yet earlier, graph grammar-based implementation for generating product configurations out of function architectures is the FFREADA[21] algorithm by SCHMIDT & CAGAN (1998) that has been applied to the synthesis of drill powertrains (SCHMIDT ET AL., 2000). This approach encapsulates the formal design knowledge based on string grammars that is specific to this application. Also, the function and component entities are string-based but can be interpreted as graphs.

Design Compiler 43 (ALBER & RUDOLPH, 2003) is a general software aiming at solving design synthesis problems that is based on the software development environment Eclipse[22]. The knowledge and the procedure to solve design problems can be formalized in a graph grammar where rules are matched using subgraph recognition based on regular expressions. The key advantage is the multitude of interfaces provided to transform design graphs into analysis models, allowing for integration with common tools, e. g., computer-aided design and simulation. In a recent application, the design of satellites (SCHÄFER & RUDOLPH, 2005) was automated through the synthesis of component-based satellite models based on a set of application-specific rules.

A grammar-based approach for the synthesis of models of technical processes – according to the Theory of Technical Systems (EDER & HOSNEDL, 2008) – has been developed by STANKOVIC ET AL. (2009). Technical processes are structured as flow-based systems that can be modeled using graphs. The formalism is not based on graph grammars but on the Backus-Naur form (BNF). BNF is a compact and formal language for the definition of grammars that are used to formalize design knowledge to decompose overall technical processes (considered as black boxes) into their subprocesses and operations. The grammars do not contain generic knowledge from the Theory of Technical Systems but are tailored for specific applications. The approach has been validated with the synthesis of a tea-brewing process using ten rules and a stiffened panel assembly using eleven rules (STANKOVIC, 2011).

The reviewed rule-based CDS approaches tackle a variety of application domains and show, hence, their versatility to computationally solve engineering design problems. When comparing the previous rule-based approaches, the following observations can be made: The scope of application is either narrow, the rule set small and the generated designs are composed of many elements of the same type, or of a limited number of different types (SHEA; STARLING & SHEA; LIN ET AL.; BOLOGNINI ET AL.). Or, the scope of application is wider entailing large rule sets and the generated designs are composed of fewer elements of many different types (SRIDHARAN & CAMPBELL; KURTOGLU & CAMPBELL). Besides the scope of application, CAGAN (2001) relates the directedness of the generation mode ("generation vs. search") to the level of knowledge ("simple vs. intensive"). From that connection, the conclusion can be drawn that using directed search techniques, e. g. simulated annealing and simulation (SHEA, 1997), supports the goal of keeping the level of knowledge in the grammar simple, and the number of rules low, as more knowledge is encapsulated in the search method and optimization model, i. e. objectives and constraints. As a counter example, CAGAN (2001) refers to the coffee maker grammar (AGARWAL & CAGAN, 1998), a shape grammar for the generation of coffee makers, that compensates for an undirected generation mode, also termed "naïve" (CAGAN ET AL., 2005), with a knowledge-intensive grammar composed of 100 rules.

None of the approaches generate models on multiple levels of abstraction. Only KURTOGLU & CAMPBELL (2006) automatically map from Function to Structure. Instead of generating a model that spans multiple

---

[21] **F**unction to **F**orm **Re**cursive **A**nnealing **D**esign **A**lgorithm
[22] Website of the Eclipse foundation: http://www.eclipse.org

levels of abstraction, functions are substituted with components, which rather represent a jump from one level of abstraction to another. From this, a conclusion can be drawn that purely rule-based approaches cannot cope with the inherent complexity resulting from the synthesis of such integrated product architectures. This conclusion is aligned with the lessons learned from the first generation of expert systems. This issue is primarily due to fact that the creation and management of the extensive rule sets that would be required to generate such complex models is computationally expensive.

Generally speaking, production rules provide a native means to represent transitions, or problem solving steps, in a state space search (Luger, 2005, p. 310). Thereby, they support a high directedness of generation, e. g. through coupling elaborate search methods (simulated annealing or genetic algorithms) with automated simulation, which, in turn, allows to reduce the size of the rule set. Nevertheless, this factor can only compensate a small part of the issue that with increasing scope the rule set becomes difficult to handle as discussed for expert systems of the first generation in Section 2.2.4. The aptitude of rule-based systems to formalize sophisticated search methods and the fact that experts tend to formulate knowledge in rule-form (Beierle & Kern-Isberner, 2008, p. 73) strengthens the argument that rule-based systems are useful in applications with limited scope from an engineering design perspective as well (Luger, 2005, p. 310).

2.4.2 Model-based CDS approaches

Matrices are a straightforward data structure for encapsulating engineering knowledge in a simple model-based representation. They can easily be created using common office software and be analyzed with specific matrix-based analysis software, as for example the commercial software LOOMEO[23]. The simplicity of matrix-based modeling and the high accessibility to computational support is an advantage. However, due to their limited expressiveness, only simple relationships among modeling elements can be represented. Typically, *design structure matrices* (DSM) and *domain mapping matrices* (DMM) (Lindemann et al., 2008) represent compatibility relations, e. g. between components-components or functions-components, and can be understood as topological constraints. Representing more complex concepts, such as multiple inheritance or parametric constraints, using solely matrix editors, e. g. spreadsheet applications, quickly becomes convoluted for the human modeler. This has little significance for practical use unless the modeling is supported with suitable graphical user interfaces.

Bryant et al. (2006) developed a software named Concept Generator deriving component-based design solutions from a predefined function structure that is based on the building blocks of the functional basis (Hirtz et al., 2002b). A function-to-component matrix, i. e. a DMM, represents knowledge about the ability of components to embody required functionality. The compatibility between components is represented in another matrix that is considered a DSM. Based on matrix-algebra, the initial functional model, represented as a DSM, is converted into a component structure that is subsequently checked for compatibility between the components. Aside from the different knowledge representation, the scope of application of this approach is akin to the work in Kurtoglu & Campbell. However, the drawback of expensive knowledge formalization is mitigated as the knowledge about the function-component-mapping is extracted from a web-based design repository. In the DSM research community various similar compatibility-oriented approaches can be found, such as (Hellenbrand & Lindemann, 2008) or (Gorbea et al., 2008). In contrast to (Bryant et al., 2006), these do not acquire knowledge from a design repository but require that knowledge is formalized in matrices, which is a tedious task. These research

---

[23]Company website: http://www.teseon.com/loomeo

efforts are not targeted at automatically solving design tasks but at investigating the predefined configuration space of products. For that reason, these approaches are not considered model-based contributions to CDS and are not further discussed.

Žavbi & Rihtaršič (2009) use topological constraints to represent the interconnectivity between physical effects (termed wirk elements) that are stored in a manually created library. Their approach aims to support the generation of working structures using the predefined physical effects as building blocks. It is implemented in a software called SOPHY[24]. The constraints are formulated based on the equations of the physical effects allowing for the automated creation of chains of physical effects. The inputs and outputs of the physical law equations are concatenated until the translation of a given input variable to a required output variable is realized. Based on this, potential allocations of physical effects to functions can be investigated. The focus on an equation-based constraint representation makes this approach principally independent of a specific engineering domain. However, this also implies the necessity to manually formalize physical effects from multiple engineering domains in the library. The approach has been validated with the synthesis of working structures for optical laser devices, household appliances and a device for removing toll-road stickers (Rihtaršič et al., 2010).

Using the Cambridge Advanced Modeller (CAM) (Wynn et al., 2010) as a platform, the synthesis approach of Wyatt et al. (2012) aims to support product architecture design. From a general graph-based description of the variant space, the set of all possible product variants can be generated. A graphical modeling language is at the designer's disposal to define a product architecture schema. It can be constructed from a set of different components, relations and constraints. This schema can be considered a metamodel that defines the modeling domain for all derived product models. It implements the key characteristics of frame-based representations: inheritance, aggregation and multiplicity-relationships. Further, topological constraints are used to represent the interconnectivity between components. The approach shows a high degree of generality as the combination of these knowledge representation features provides an expressive foundation for encapsulating engineering knowledge for a wide range of applications. It has been validated with the synthesis of architectures of gear trains, aircraft configurations, vacuum cleaners and engine block mounting. However, the search process, depth-first search, is fairly simple using elementary transformations of the metamodel.

The Knowledge Intensive Engineering Framework (KIEF) (Yoshioka et al., 2004) aims to bridge the gap between a subjectively formulated function structure of the desired product and the creation of a product concept. The identification of suitable physical concepts realizing specific functional requirements is accomplished by the Qualitative Process based Abduction System (QPAS) (Ishii et al., 1993), which is a reasoning system that proposes design solutions based on qualitative state transitions. The knowledge representation is ontology-based; it builds on a cascade of metamodels and organizes abstract-concrete knowledge in a hierarchical manner. This representation allows integrating general engineering knowledge and specific domain knowledge. Metamodeling as a key factor for enhancing the scope of intelligent design support has been emphasized by Tomiyama et al. as early as 1989. The underlying design representation Function-Behavior-State (FBS) (Umeda & Tomiyama, 1995) constitutes the lowest level of a cascade of metamodels; it defines the basic modeling elements and the relations among them. Using the concept of multiple inheritance, higher level metamodels, e. g. a geometric model or a process-interval model, can make use of the low level FBS representation and define more specific element types such as depicted in Figure 2-19. Recent work (Komoto & Tomiyama, 2010) builds on that foundation and

---

[24]**Synthesis Of Phy**sical laws

takes a systems engineering approach to assign components together with their spatial parameters to the working principles that are to be realized. Similar as in the work of (WYATT ET AL., 2012), an expressive model-based knowledge representation foundation provides the basis for a wide scope of applications.
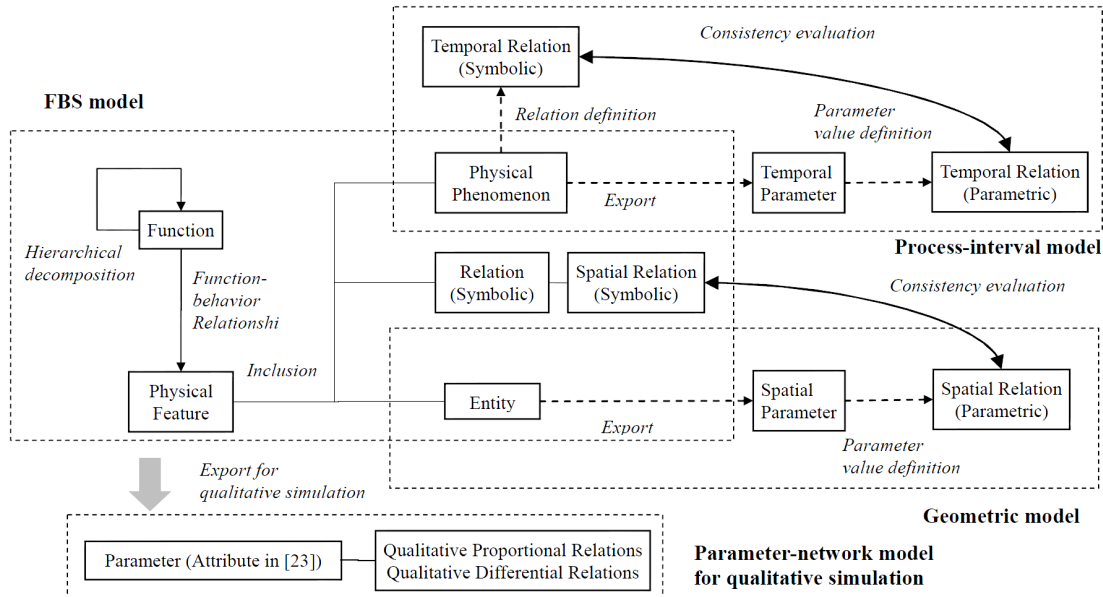
Figure 2-19: Cascade of metamodels building on the FBS model (KOMOTO & TOMIYAMA, 2010)

All presented model-based approaches claim to be generic regarding the scope of application. Real generality can only be observed in the work of (YOSHIOKA ET AL., 2004) and (KOMOTO & TOMIYAMA, 2010). BRYANT ET AL. and ŽAVBI & RIHTARŠIČ use a simple knowledge representation that only encapsulates specific knowledge about the compatibility of specific modeling elements. Instead of having a deep model of the problem domain, a data-driven approach using extensive libraries is realized. These are either filled in by hand or derived from a design repository containing dissected products. As no fundamental design knowledge is contained, widening the scope of application necessarily results in expanding the knowledge base; this requires a considerable effort. Another drawback of such a limited expressiveness is the limited ability for considering complex relationships. This issue is reflected in the simplicity of the validation examples. WYATT ET AL., on the contrary, provide a richer representational foundation and hence are able to consider more complex design problems like in the given validation examples. However, also here the drawback is that for every design problem an architecture schema, i. e. a metamodel, has to be defined. As the approach provides for inheritance, problem-specific metamodels could inherit from metamodels that encapsulate fundamental knowledge. This is envisaged in future work through integrating metamodels containing abstract, reusable components. How a cascade of metamodels can encapsulate various levels of knowledge that support both generality and problem-specifity becomes clear in the approaches based on KIEF. Further, the rich expressiveness of ontologies is used to represent complex interrelations between physical effects. The KIEF-based approaches have been validated using realistic, industrial applications from the area of printing devices. To date, the KIEF knowledge base contains 500 physical effects, 150 relations and 600 parameter types in an efficient, hierarchical structure (KOMOTO & TOMIYAMA, 2012). A recent application (KOMOTO & TOMIYAMA, 2012) shows the versatility and ability to cope with complexity in mechatronic design.

In contrast to the rule-based approaches, the model-based approaches apply simple search methods leading to an exhaustive enumeration of the possible solutions. These kinds of search methods are straight-

forward to implement and are in theory able to find a global optimum. Hence, they represent a practical method to deal with small search spaces. With increasing search space, the computational effort of these brute-force approaches increases tremendously calling for more sophisticated, i. e. more directed, search methods such as applied by the rule-based approaches. In this respect, too, the KIEF-based approaches are an exception as they use logical reasoning. For example, they infer physical effects possibly fulfilling required functionality from the physical concept ontology (YOSHIOKA ET AL., 2004). One disadvantage of purely model-based reasoning is, according to LUGER (2005, p. 312), the lack of clarity of the formalized knowledge describing the problem domain. Although KIEF's reasoning techniques can cope with complex problems, this disadvantage becomes apparent as the underlying reasoning mechanisms remain unclear and are used in a black-box manner.

2.5 Conclusions

An overview of the reviewed CDS approaches is depicted in Table 2-1. Based on this, general observations are given and reference is made to the research issues depicted in the problem description in Section 1.1. Finally, the research goals are refined to justify and detail the further method development in this thesis.

Research Issue 1 addresses the lack of efficiency of knowledge formalization. Regarding rule-based representations, this point accords with the trade-off that can be seen between manageable, small rule sets (SHEA, 1997; SHEA ET AL., 2005; STARLING & SHEA, 2005; LIN ET AL., 2009; BOLOGNINI ET AL., 2007; KERZHNER & PAREDIS, 2009; SCHÄFER & RUDOLPH, 2005) and a wider scope of application (SRIDHARAN & CAMPBELL, 2004; KURTOGLU ET AL., 2010) often leading to large rule sets, which raises the problems of first generation expert systems. Nevertheless, rule-based approaches natively support the definition of transitions[25] in a state space search. Production rules thus provide an efficient foundation for realizing elaborate search methods with a high directedness (CAGAN, 2001) that encapsulate additional knowledge about the solutions' quality in evaluation methods or target functions. How efficient and flexible knowledge representations can be achieved becomes apparent when regarding the hierarchical knowledge structure of approaches (YOSHIOKA ET AL., 2004) and (KOMOTO & TOMIYAMA, 2010); cascading abstract and concrete metamodels based on inheritance seems to be the key factor which will be integrated by the researchers of approach (ŽAVBI & RIHTARŠIČ, 2009) in their future work. Approaches (BRYANT ET AL., 2006) and (ŽAVBI & RIHTARŠIČ, 2009) are based on a flat, database-like knowledge structure that does not allow for reuse of fundamental knowledge in a hierarchical manner. However, the manageability is considered high as the proposed data structure avoids an entanglement of knowledge chunks. None of the approaches provides a systematic formalization support or enables the formalization of evolving knowledge. Instead, the formalized knowledge is assumed to be already encapsulated in the implementation. In this respect, the KIEF-based approaches do not provide further indications concerning how the integrated ontologies can be extended or updated.

The limited scope of application is mentioned in Research Issue 2. This is a critical issue as computational support for the conceptual design phase requires the design representation to span multiple levels of abstraction and granularity. The majority of the reviewed CDS approaches focus only on the composition of components on the Structure level. Although a decomposition of functions (SRIDHARAN & CAMPBELL, 2004) and processes (STANKOVIC, 2011) and a mapping from Function to Structure (SCHMIDT

---

[25]Typically, transitions are called *actions* in AI literature (RUSSELL & NORVIG, 2003). They can be used, for example, to encapsulate heuristics as used for A* search to find global optima.

Table 2-1: Overview of rule-based and model-based CDS approaches

	Authors	Approach/ Implementation	Scope of application	Design representation				Knowledge representation		Search method
				Technical processes	Function	Behavior	Structure	Rule-based	Model-based	
Rule-based approaches										
1	Shea 1997, Shea 2005	elfForm	Truss structures				C: Trusses, joints	9 topology modification rules		Simulated annealing
2	Starling 2005	Parallel grammar	Mechanical gear systems				C: Gear pairs, shafts	7 C-rules, 14 P-rules		Hybrid pattern search and simulated annealing
3	Lin 2009	ROMAX Designer	Gearboxes				C: Gear pairs, shafts	6 topological rules, 3 parametric rules		Simulated annealing
4	Bolognini 2007	CNS-Burst	MEMS				C: Microresonator primitives	5 general graph modification operations	Definition of primitive types	Multi-Objective Burst (heuristic-based search)
5	Kerzhner 2009	DSL-based grammar	Hydraulic circuits				C: Hydraulic components	4 graph transformation rules	SysML-based DSL	Random-based selection of rules
6	Sridharan 2004	Grammar for Function Structures	Household products		Functions –D→ Sub-functions			69 manually defined rules		*not detailed*
7	Kurtoglu 2010	GraphSynth	Electromechanical products		Functions ———	M→	Electromechanical components	170 manually defined rules		Nested breadth-first search
8	Schmidt 1998	FFREADA	Drill powertrains		Functions ———	M→	C: Drill powertrain components	String-based design rules*		Recursive simulated annealing
9	Schäfer 2005	Design Compiler	Satellites				C: Satellite components	Design rules formulated as regular expressions*		*not detailed*
10	Stankovic 2011	Grammatical evolution	Tea brewing process / Stiffened panel assembly	Overall process –D→ Subprocesses				10 BNF rules / 11 BNF rules		Exhaustive search
Model-based approaches										
11	Bryant 2006	Concept Generator	generality claimed, validated with thermal mug, human powered power supply, wall climbing toy		Functions ———	M→	Components		Matrices for function-component mapping and component compatibility	Exhaustive search *(not further detailed)*
12	Žavbi 2009	SOPHY	generality claimed, validated with optical laser devices, household appliances, device for removing toll-road stickers			C: Physical effects			Equation-based topological constraints	Exhaustive search *(not further detailed)*
13	Wyatt 2012	Cambridge Advanced Modeler	generality claimed, validated with gear trains, aircraft configurations, vacuum cleaners and engine block mounting				C: Components		Frame-based and topological constraints	Exhaustive depth-first search
14	Yoshioka 2004	KIEF	generality claimed, validated with laser stereolithography method and printing devices		Functions –D→ Sub-functions –M→ Physical effects –C→ Working principles				Ontology, cascade of metamodels	Ontology-based logical reasoning
15	Komoto 2010	KIEF	generality claimed, validated with photocopier		Functions –D→ Sub-functions –M→ Physical effects –C→ Working principles –M→ Components				Ontology, cascade of metamodels	Ontology-based logical reasoning

Legend: D: Decomposition; C: Composition; M: Mapping; *: Number not specified

& Cagan, 1998; Bryant et al., 2006) was achieved by some researchers, product architectures that can cope with the complexity of design processes of mechatronic products are only generated by the KIEF-based approaches (Yoshioka et al., 2004; Komoto & Tomiyama, 2010). This is reflected in the FBS metamodel in Figure 2-19 as it provides the underlying data structure for various design process steps, e. g. functional decomposition or assignment of physical effects (physical features) to functions.

Only few approaches are based on a computational implementation of established design methods, as stated in Research Issue 3: Key aspects of the Theory of Technical Systems (Eder & Hosnedl, 2008) (Stankovic, 2011), functional decomposition according to Pahl et al. (2007) (Sridharan & Campbell, 2004) and an emulation of parts of the design process of VDI 2221 (1987) (Yoshioka et al., 2004; Komoto & Tomiyama, 2010). Established, paper-based design methods typically claim to be generic and solution-independent. Their computational implementation could foster the development of more generic CDS approaches. Further, reusing existing design knowledge, e. g. from design catalogs or text books, is not widespread. Bryant et al. (2006) import data of dissected products rather than using design knowledge, e. g. from design catalogs. Žavbi & Rihtaršič (2009) propose a method to transfer functional models into chains of physical effects that are manually derived from (Koller & Kastrup, 1998). One good example in this regard is the generation of a "Very Large-Scale Knowledge Base" for KIEF (Ishii et al., 1995) using text books such as (Hix & Alley, 1958) and thus avoiding building the knowledge base from scratch. However, the use of this ontology-based knowledge for computational synthesis is not based on well-known design methods that were intuitively understandable by a designer, but on logical reasoning algorithms. These are hard to grasp for the human designer and hard to formulate by the human expert. This is a major drawback of model-based knowledge representations (Luger, 2005, p. 312). None of the known approaches takes the heterogeneity of the numerous knowledge sources into account and proposes a method to uniformly formalize knowledge from them.

A physical concept ontology (Yoshioka et al., 2004) that aims to map from function models to physical concepts has been implemented in the Knowledge Intensive Engineering Framework (KIEF). It has recently been extended to software for system architecting (Komoto & Tomiyama, 2010). Both approaches are based on representations using the physical value that is subject to be manipulated by a function as a starting point for searching appropriate physical effects. Wu et al. (2008) introduced a method based on bond graph modeling that supports the automated generation of dynamic models from component models. This approach supports the investigation of the dynamic behavior of component networks and, hence, supports the design process step of the embodiment seen in Figure 3-1.

As stated in Research Issue 4, the use of standard modeling languages has not yet achieved any great significance in CDS. The only exception is approach (Kerzhner & Paredis, 2009) using SysML. However, increasing tool support, intensified teaching activities and the ongoing cross-linking with research activities from the area of Model-Based Systems Engineering (MBSE) will stimulate a further dissemination in academia and industry. The advantages for CDS research will be the facilitation of tool integration, e. g. for simulation, using standard interfaces but also the easier integration in development processes in industrial applications. To profit from that trend, the adoption of SysML is not necessarily required. Also self-defined, problem- and domain-specific languages (DSL) that are based on a formal definition, i. e. a formal metamodel, allow access to these advantages. Regarding the definition as stated on page 3, only approaches (Kerzhner & Paredis, 2009) (using SysML), (Wyatt et al., 2012; Yoshioka et al., 2004; Komoto & Tomiyama, 2010) (using DSLs) are considered as being based on formal modeling. As long as a formal metamodel of a modeling domain is available, models of this domain can be transformed into a different domain, such as SysML for example. This technique is termed *model transformation*.

Solely Kerzhner & Paredis (2009) take advantage of the potential of model transformation as a means to map between two modeling languages of two different modeling tools and to keep these views synchronized.

Regarding Research Issue 5 (low maturity of software tools), only a rough estimate of the state of the art can be made that is primarily based on personal impressions and discussions with the respective researchers. The essential reason for this vagueness is that few papers are written about CDS implementations in detail and that the assessment of software quality[26] does not lie – at least so far – in the CDS community's interest. Basically, it can be said that almost no (except 3 and 5) approach reuses existing software, or software libraries, to achieve a higher level of maturity. A reason for this is certainly, that integrating available open-source software often implies that the developed software also has to be issued under an open-source, or compatible, license and the integration of commercial software often involves considerable costs. However, approaches (Kurtoglu et al., 2010; Schäfer & Rudolph, 2005; Wyatt et al., 2012; Yoshioka et al., 2004; Komoto & Tomiyama, 2010) are built on software platforms that are used for multiple research projects and support a wider user and developer base. This is a clear advantage; development efforts can hence be bundled and the increase of software quality benefits from the experience and feedback of more users.
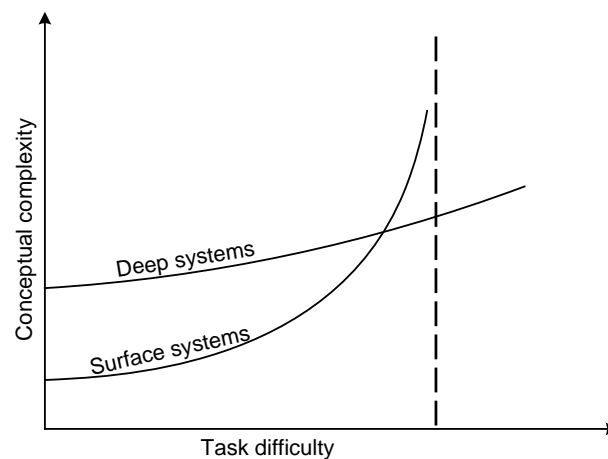
Figure 2-20: Deep systems versus surface systems adapted from (Hart, 1982)

As a general observation it can be stated that rule-based knowledge representations constitute an advantageous basis for realizing efficient search and optimization methods. Model-based approaches, in turn, can cope with higher complexity of the modeling domain. To date, no CDS approach aims to combine these knowledge representation paradigms to benefit from the respective advantages. To some extent the current situation in CDS research is similar to the situation in Artificial Intelligence research in the early 1980s. At that time the first generation of expert systems, like MYCIN, showed the potential for computational problem-solving in a narrow scope. The extension towards more general problem domains was still limited. In 1982, Hart analyzed the status quo of AI research and derived directions for AI research in the 1980s. He differentiates between *surface systems* having no underlying representation of the problem domain and *deep systems* that encapsulate fundamental knowledge. Their different qualitative characteristics showing the relationship between task complexity and conceptual complexity are shown in Figure 2-20. The term conceptual complexity is not clearly defined and not quantitatively mea-

---

[26]According to ISO/IEC 9126 (Jung et al., 2004), software quality can be measured using these characteristics: Functionality, reliability, usability, efficiency, maintainability, portability

surable but comprises the issues related to the generation and maintenance of knowledge bases. Purely rule-based approaches having no directed search method – but rather building on naïve generation – are typically surface systems whereas the deep systems usually incorporate features of model-based knowledge representations. The central statement of this figure is that as the complexity of the problem increases the effort related to the conceptual complexity increases exponentially for surface systems but almost linear for deep systems. Hence, using surface systems is justified for simple problems that can be solved at a cost of lower complexity by a surface system than by a deep system (HART, 1982). The growth of complexity of these systems with an increase of task complexity can be seen in the large rule sets of (SRIDHARAN & CAMPBELL, 2004) and (KURTOGLU ET AL., 2010). To conclude, the advantages of deep systems point the way to expand the scope of application while rule-based systems are essential to formalize heuristics and to realize efficient search methods. As stated by SRIRAM (1997, p. 140), a hybrid knowledge representation could combine the advantages of both systems.

Based on these conclusions, the following chapter is dedicated to the depiction of the method of object-oriented graph grammars, a hybrid knowledge representation that unifies the aforementioned advantages of rule- and model-based approaches.

3 Synthesis of product architectures using object-oriented graph grammars

The aim of this chapter is to introduce the method that leads to the Expected Contribution 1 (high efficiency of object-oriented graph grammars) and Expected Contributiosn 2 (high effectiveness of object-oriented graph grammars). Based on the conclusion of the literature review and analogies drawn from the development of expert systems, Research Issues 1 (inefficient knowledge formalization) and 2 (limited scope of application) are addressed with the development of a hybrid knowledge representation: object-oriented graph grammars.

First, the definition of the design representation, based on Function-Behavior-Structure, and its relation to design methodology is given. Next, object-oriented graph grammars are presented, a general approach for the synthesis of graph-based product architectures. The validation and application of the method to a specific design problem is given through an example of synthesizing automotive hybrid powertrains and the analysis of the computationally generated solution space. This chapter concludes with a discussion of the results.

3.1 Design representation: Function-Behavior-Structure

As discussed in Section 2.2, a top-down transformation of a functional product definition into a component-based product model is the widespread approach in the classic engineering design literature. The Function-Behavior-State (FBS) representation by Umeda & Tomiyama (1995) takes these multiple levels of abstraction into account. The design representation that builds the foundation for synthesizing product architectures in this work also relates to these three levels of abstraction. However, it interprets them from a perspective that is closer to the classic design literature. Rather than considering the specific state of the product behavior, the S-level represents Structure as proposed in a different FBS representation by Gero (2004): "components of the object and their relationship". For brevity and to prevent from confusion, the design representation applied here is termed *FBS** and builds on the levels of abstraction and the design process steps to achieve them as defined in the following:

- On the *Function* level, the design problem is formulated and hence the top-down synthesis process is initialized. The methodology according to Pahl et al. (2007) of characterizing functions by input/output relations described as verb-noun pairs is followed. To perform a functional decomposition, three levels of detail are introduced. An *overall function* is decomposed into *high-level functions*. For this step, a high level of contextual knowledge is required. Hence, it is not carried out computationally but by the human designer. The second decomposition step requires the breakdown of high-level functions into *subfunctions*. Elements of this third category are modeled using the set of functional operators and flows from the Reconciled Functional Basis (Hirtz et al., 2002b) and is carried out computationally.

- On the *Behavior* level, a network of physical effects is synthesized to embody the required subfunctions. They describe the working principles that realize the functions from an idealized, physical point of view. Design catalogs, such as issued by Koller (1994) and Ponn & Lindemann (2011),

provide a large source of knowledge for physical effects. This step provides a valuable starting point for the development of innovative, new components. A method to automate the transfer of paper-based engineering knowledge about physical effects into a computable representation is presented in Chapter 4.

- The lowest degree of abstraction is achieved on the *Structure* level where concrete components are associated to create product architectures that embody the intended physical effects. Further design process steps, such as the adoption of a product modularization strategy, are conceivable but are not in the focus of this research.
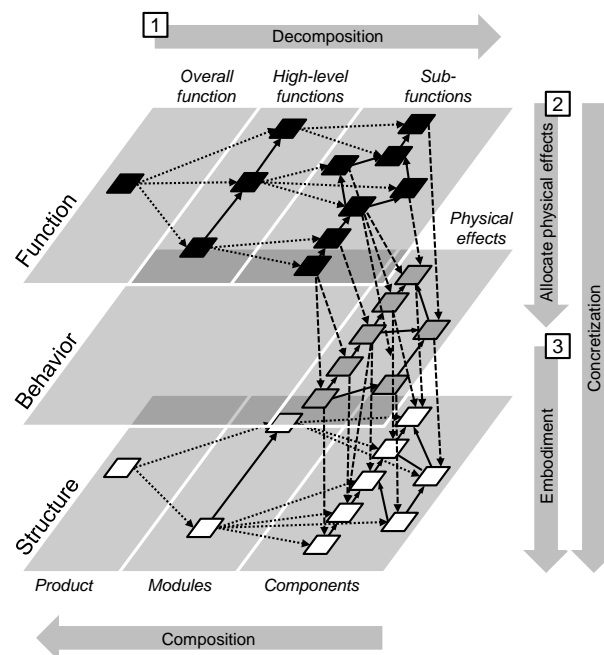
Figure 3-1: Graph-based product architecture based on Function-Behavior-Structure and design process steps

The significance of this two-step-process of generating a component structure from a functional description via a behavioral level has also been highlighted by WELCH & DIXON (1994) as they "believe that behavior, in terms of physical principles and phenomena, provides a natural bridge between functional requirements and physical artifacts." The physical effects are, in their work, modeled based on bond-graphs, which is also the foundation of the method presented in Chapter 4. Instead of modeling the physical characteristics at this fine level, the presented approach in this thesis is based on the description of physical effects in design catalogs. If the design task does not require the development of new components or behavioral reasoning, this level of abstraction can be skipped. In this case, a direct mapping from Function to Structure is possible such as in the work of CAMPBELL ET AL. (2000) and SCHMIDT & CAGAN (1998). GERO (1990) termed this kind of mapping "catalogue lookup".

To summarize, the design representation defines the foundation based on which product architectures are synthesized. It is inspired by the classic design literature involving three levels of abstraction. The design process steps, as depicted in Figure 3-1,

1. decomposition of high-level functions into subfunctions,

2. allocation of physical effects to subfunctions (further elaborated in Chapter 4),

3. embodiment of physical effects with components

are executed computationally. While these basic steps are built on known paper-based design methods, such as functional decomposition (ULRICH & EPPINGER, 2008) or matrix-based selection of physical effects (KOLLER, 1994), here they are formalized, computationally implemented and automated. The advantage of basing the approach on paper-based methods is that it is potentially more intuitive to engineers and can also be applied semi-automated and interactively. Compared to the reviewed state of the art, this approach offers a richer – and thereby more effective – description of the design representation as it interrelates multiple levels of abstraction and leads to the Expected Contribution 2 (high effectiveness of object-oriented graph grammars). This design representation addresses thereby the issue raised by CRAWLEY ET AL. (2004) regarding the importance of developing support for exploring product architectures on multiple levels of abstraction. As further detailed in Section 5.2, this design representation builds on a formal language based on which approaches for model transformation can be developed, cf. Expected Contribution 4 (model transformation).

3.2 Knowledge representation: Object-oriented graph grammars

Conceptually, graph grammars are a formal, generative method consisting of a vocabulary and a set of rules. The vocabulary contains all valid elements represented by nodes and edges, the set of rules defines transformations modeling how these elements can be created, deleted or modified. Similar to natural language that is based on words and grammatical rules, it is also possible to develop a language of designs via formal engineering design grammars. Starting from an initial graph, or starting symbol, the repeated application of different grammar rules generates new designs. The combinatorial expansion of all valid rule sequences is termed *design language*. The advantage of using graphs is the fact that they can be used as the foundation for almost any kind of formal modeling language in conceptual design (e. g. SysML diagrams, function models and geometric models like B-Reps). Further, they provide a strong basis for computational representation and transformation.
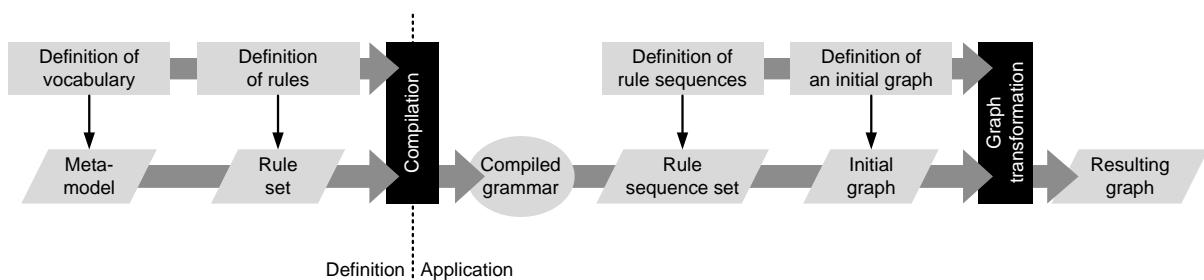
Figure 3-2: Definition and application of object-oriented graph grammars for computational design synthesis

This research strictly follows an object-oriented approach and combines it with graph grammars for design synthesis, Figure 3-2. Therefore, this knowledge representation is called *object-oriented graph grammar*. Through this approach, the definition of a grammar is separated from its application. The definition of the graph grammar includes the metamodel and the rule set. The compilation of the grammar definition results in an executable grammar. Rules within the grammar are not independently executed,

they are embedded in rule sequences in which they can be concatenated to sophisticated control structures. Combining formal design synthesis methods with concepts from object-oriented programming provides significant advantages. Regarding software development, these quality factors have been by MEYER (1997). With respect to Research Issues 1 (inefficiency of knowledge formalization) and 2 (limited scope of application) they can be transferred to CDS research as follows:

- *Extendibility* is the ease of adapting the method and the implementation to a wider range of applications or completely new applications that includes an extension of both vocabulary and rules. Since knowledge is always evolving, this point is of particular interest.

- *Reusability* is the ability to define a grammar for multiple applications and reuse the vocabulary and rules.

- *Compatibility* is the ability to define different sets of vocabulary and rules, e. g. in different domains such as design or manufacturing, that can be interchanged for different applications or product generations.

- *Efficiency* means that the formalization and the encapsulation of knowledge are supported in a way that the demand of resources is as low as possible. This involves computational resources, but more importantly is to efficiently support the time consuming human task of encoding knowledge.

- *Ease of use* means that the structure of the grammar can be understood as intuitively as possible and provides the foundation for the factor of extendibility.

The term "object-oriented graph grammar" has to be differentiated from how it is used in computer science. In both domains, a graph grammar is used to synthesize graph-based models. "Object-oriented" refers in this work to the grammar itself as it embodies the key features of object-orientation, e. g. inheritance. However, in the work of FERREIRA & RIBEIRO (2003), the same term refers to the ability of a graph to represent object-oriented systems, e. g. object-oriented software implementations. In that sense, an object-oriented graph grammar is able to represent the dynamics of object-oriented systems, e. g. the behavior of object-oriented software during its execution.

3.2.1 Grammar definition

The definition of a design grammar, here a graph grammar, determines the design representation, i. e. vocabulary and/or building blocks, and the design synthesis process steps, i. e. rules. While the metamodel contains the building blocks for the synthesis of FBS* product architectures, the rule set encapsulates the design process steps of decomposing functional models, the allocation of physical effects to functions on the Behavior level and finally, the embodiment into components on the Structure level.

The definition of appropriate building blocks depends highly on the design problem. For the synthesis of FBS* product architectures, a general schema of the metamodel is first presented. A more specific metamodel for the synthesis of hybrid powertrains is then provided in Section 3.3.1.

In contrast to the metamodel, the rule set encapsulates general knowledge for the synthesis of FBS* product architectures. Hence, procedural knowledge for the design process is formalized here. The definition of the rule set in Section 3.2.1 is directly applied to the synthesis of hybrid powertrains.

Metamodel: Building Blocks

According to ISO 11179 (International Organization for Standardization, 2004), a *metamodel* is a "data model that specifies one or more other data models". Hence, the metamodel is an upfront definition of the modeling elements for the synthesis problem. In this context, it must not be confused with the term *surrogate model*, which is an approximation model of a simulation model. The application of graph grammars requires a modeling space that comprises all graph elements, i. e. nodes and edges that can be used during the synthesis process. In this approach, the metamodel represents, from a knowledge standpoint, the building blocks of the engineering design grammar. It covers hence the model-based aspects of the hybrid knowledge representation. The metamodel provides the foundation for the formulation of graph transformation rules by specifying node and edge types.
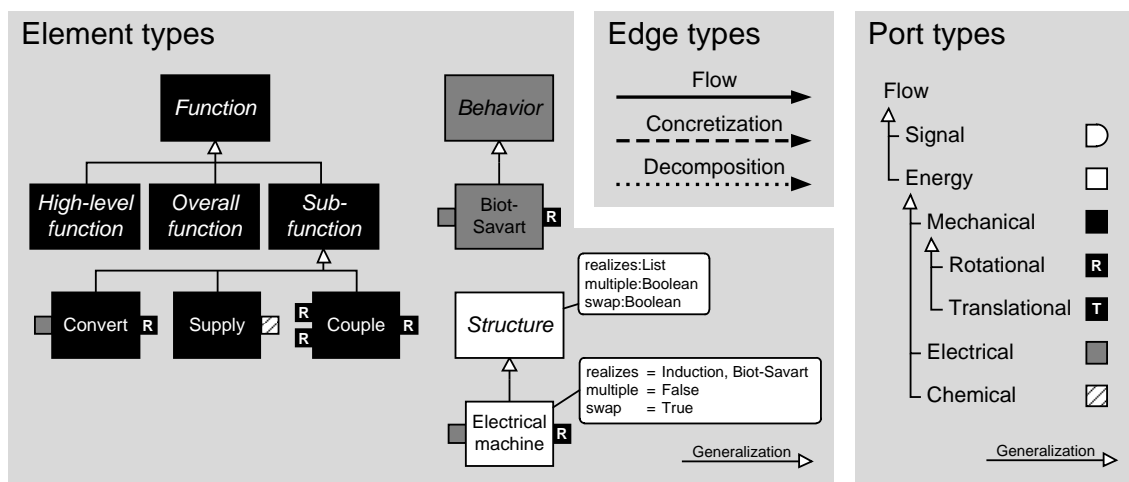

Figure 3-3: General structure of a FBS* metamodel

As this research uses an object-oriented approach, the definition of the metamodel applies analogously to the definition of classes in object-oriented programming. Similar to objects that are instantiated from classes, nodes and edges are instantiated from node types and edge types. The concept of inheritance, which is widely used in programming, provides the foundation for creating a structured definition of the building blocks. This principle is successfully applied by Komoto & Tomiyama (2010) for defining a cascade of metamodels and shows the aptitude to systematically capture engineering knowledge in a "Very Large-Scale Knowledge Base" (Ishii et al., 1995).

In the scope of this research, the metamodel defines a schema of modeling elements, i. e. building blocks, providing the foundation for synthesizing FBS* product architectures. All elements of an equal level of abstraction are interconnected based on the exchange of energy, material and signal. Hence, a set of input/output relations, called *ports*, can be defined in the metamodel. The underlying computational graph representation requires that ports are modeled as nodes as well. This is further detailed and mathematically defined in the formal definition of the underlying port-based modeling language in Section 5.2. To prevent confusion, the term *element* is used to denote modeling elements having *ports* and being defined as element types in the metamodel.

A crucial aspect of this method is the ability to define the valid interconnectivity of building blocks upfront in the metamodel. This definition is independent from any generative process and remains static

during the synthesis process. The concept of ports is introduced in the metamodel, Figure 3-3 through which compatible elements, on any level of abstraction, can be identified. As introduced in Section 2.3.2, ports can be understood as topological constraints defining the valid interconnectivity between element types. The significance and suitability of ports in conceptual design has been emphasized by LIANG & PAREDIS (2004): "The port connections represent interactions consisting of the exchange of energy, matter or signals [...]. Representing design alternatives as configurations of port-based objects is useful at the conceptual design stage when the geometry and spatial layout is still ill-defined." The metamodel definition of the port types adopts the concept of inheritance and reproduces the flow taxonomy from the Functional Basis (HIRTZ ET AL., 2002b). In the use of ports, the approach taken in this research extends previous research in three aspects: First, the notion of ports in this research is not bound to any level of abstraction, e. g. components. They can be seen as a general concept for capturing knowledge about interconnectivity of elements. Second, due to inheritance, the logic behind a taxonomy of ports can be captured, such as depicted in the hierarchy of energy flow ports in Figure 3-3; parallel concepts, e. g. signal flow, can be described in the representation too. Third, ports defined this way provide the foundation for representing design knowledge in graph grammar rules in a more generic way (as presented in the following section), leading to a reduced number of rules that remain valid even when the taxonomy of ports is extended.

In Figure 3-3 the general structure of a FBS* metamodel based on a hierarchy of inheritance is shown: The top-level element types *Function*, *Behavior* and *Structure* split the metamodel into the three levels of abstraction. Their descendants (connected with the *Generalization* arrow) represent more detailed element types on each of the abstraction levels. The association of port types to element types is achieved in the metamodel definition. The existence of associated ports is a prerequisite so that elements can be connected to form a graph. Element types in italic letters are *abstract element types* and cannot be instantiated. They can be used to group element types or can serve as property containers for their descendants, e. g. elements typed *Structure*. Due to inheritance, all properties of an element type are transferred to its descendants. These attributes only have to be defined once and become automatically available to all descendants, e. g. *Electrical machine*.

The focus of this synthesis method is to generate graph-based product architectures for the conceptual design phase. Assessing quantitative characteristics of the model requires additional evaluation methods and is usually considered later in the engineering design process. Hence, representing the general compatibility – independent of the application domain – of components for the embodiment of physical effects is the purpose of the definition in the metamodel. To achieve this, the following attributes are required for the components on the Structure level:

- *realizes*: List of physical effects, which a component is able to embody, e. g. an *Electrical machine* can be used for realizing the physical effect *Biot-Savart*.

- *multiple*: Boolean value defining whether multiple components of a specified type are allowed, e. g. when False the component *Electrical machine* cannot be used repeatedly in a product architecture.

- *swap*: Boolean value defining whether input and output ports can be swapped. If this value is set to *True*, the input flow type can become the output flow type and vice versa. Regarding the *Electrical machine*, this means that it can function as a generator or as a motor.

The representation of subfunctions is, in accordance with the Functional Basis, realized by a functional operator, e. g. *Convert* and *Supply*, linking the input and output ports and representing the respective flows. The input port is located on the left side, the output port is on the right side.

As a matter of course, the definition of a graph's building blocks requires the definition of edge types. Any kind of functional relation between elements that is based on an exchange of energy, material or signal is specified through respective ports. The connection between ports is modeled using a *Flow* edge, Figure 3-3. Further, *Concretization* edges model the mapping between different levels of abstraction. Finally, *Decomposition* edges are used to represent part-of-relationships on either the Function or the Structure level.

The ability to define possible interconnections upfront in the metamodel, and not in the rule set, is an important aspect of this method. Thus, this definition is independent from any generative process and remains static during the synthesis process. This is the key factor for shifting knowledge from the rule-based representation in graph transformation rules to a model-based representation in the metamodel and to achieve, by that, a hybrid representation.

Rule Set: Design Process Steps

The set of rewrite rules includes all valid operations for creating and modifying nodes and edges in a graph, i. e. a FBS* product architecture. Hence, steps for the computational, top-down design process are defined in the rule set and closely reproduce the design process steps, Figure 3-1, carried out by the human designer. Such a generic rule set provides a foundation for modeling general design procedures. Hence, general knowledge that is independent of the design problem (e. g. how to carry out a functional decomposition,) can be formalized and used for many design synthesis problems that are based on a FBS* representation.

All graph transformation rules are based on a separation into a left-hand side graph, L, and a right-hand side graph, R. When a rule is applied to a host graph it searches for the left-hand side and transforms it into the right-hand side. If the left-hand side is not found, no transformation is carried out. The context K ($L \cap R$) contains the set of elements that remain static in the rule application and mark the location of the transformation. Hence, an element on the left-hand side not included in K is deleted whereas an element on the right-hand side not included in K is added. This set-based definition of a graph transformation is called the *Algebraic Approach to Graph Transformation* and the sets K, L and R can also be identified in the rule schema introduced in Table 3-1. For further details, refer to (CORRADINI ET AL., 1997). To provide a higher flexibility and expressiveness, the definition of rules is enhanced with several concepts from programming:

- *Rule parameters* are the variable input when a rule is called.

- *Return values* are the result of the rule termination and can be used for further operations, e. g. as rule parameters for subsequent rules. By default, rules return the result of the rule application. If a rule does not apply because the left-hand side does not match, *False* is returned, otherwise True is returned.

- *Rule constraints* can be used to impose conditions on the matching or the evaluation of attributes. The applied rule language allows implementing them directly in the rule specification in a pro-

Table 3-1: Rule set for the synthesis of FBS* product architectures

Rules	Exemplary rule application
	Rule constraints

① initialize

same_in:Boolean, same_out:Boolean

Rule name
Rule parameters
Negative application condition
Left-hand side R
Rule return values

Rule context K
Rule constraint
Right-hand side R

a, b, z

or

z or z

L   R

K

① : Rule constraints   ✖ : Negative application condition

Boost → **initialize** → Boost T → Couple R R
Rule parameters:
same_in = False
same_out = False

Convert R → **initialize** → Convert R → Biot-Savart R
Rule parameters:
same_in = True
same_out = True

① : If *same_in* is True: Same port type
② : If *same_out* is True: Same port type
③ : If type of *b* is Subfunction, *z* is of type Decomposition
If type of *b* is Behaviot, *z* is of type Concretization

② create chain

a:Node, b:Node, z:Edge, left:Boolean, nb_repetitions:Integer

K

a   a

z or z   z or z   y

b   b   c

L   R

a, c, y, left, nb_repetitions

Boost T → Couple R R
2 x create chain
Rule parameters:
left = True
nb_repetitions = 2
→ Convert R, Boost T, Couple R R, Convert R

create chain
Rule parameters:
left = False
nb_repetitions = 2
Boost T → Convert R → Couple R → Convert R → Convert R

① : Not matching port type
② : Type of *c* has to be in accordance with *nb_repetitions*

③ embody

K

a → a → b

c

realizes = ... ①

L   R

Biot-Savart R → **embody** → Biot-Savart R → Electrical machine R

① : *realizes* contains physical effects to be embodied

④ merge

K

multiple = False

w → a → x
w → a → x
y → b → z   z

L   R

①

Electrical machine R
Electrical machine R
→ **merge** → Electrical machine

① : *a* and *b* must be of the same type

gramming language-like manner. For further details on the rule specification syntax refer to (Geiss et al., 2006).

These aspects provide an added value as they enrich the formalism and provide the basis for formalizing general procedural knowledge, such as the process steps for the synthesis of FBS* product architectures. To take these additional features into account, a new rule representation schema has been developed that is used to represent the rule set in Table 3-1.

The computational design synthesis process starts with a given functional model composed of high-level, user-defined functions. The two subsequent steps of decomposing high-level functions into subfunctions and the allocation of physical effects are treated uniformly. They are considered as a chaining process based on the following rules:

- The rule *initialize* starts the process with respect to the required port configuration of either the high-level function (decomposition) or the subfunction (allocation of physical effects).

- The rule *create chain* produces a chain of elements until a compatible port configuration is found for either the decomposition or the allocation of physical effects.

Further, two additional rules are required to assign components to physical effects (*associate S to B*) and to combine components that should not have multiple occurrences (*merge components*).

The rule *initialize*, Table 3-1, uses the knowledge about valid element combinations already defined in the metamodel through ports. For this purpose, an element a is sought on the left-hand side that is neither decomposed into a subfunction nor embodied into a physical effect. A negative application condition (NAC) represents this fact and is symbolized with a thick cross. With NACs, graph patterns can be defined that forbid the application of a rule if any of them are present in the host graph. The local variable a is only valid within the scope of this rule and identifies a high-level function, e.g. *Displace electrically*, or a subfunction, e.g. *Convert electrical energy to rotational mechanical energy*. While the number of ports is disregarded on the left-hand side, the port configuration in this rule is handled on the right-hand side in the rule constraints ① and ②. They control the search for the new element. The rule parameters *same_in/same_out* define whether the input/output ports of the added element have to be of the same type (or one of its descendants in the port type hierarchy) or not. The effect of setting the rule parameters is illustrated in the example in Table 3-1. Rule constraint ③ takes control of the edge to be added: In the case that a is an instance of the element type *Subfunction*, the edge z is typed *Concretization*, otherwise it is typed *Decomposition*.

After creating the element b, the starting point for the chaining process, i.e. decomposition or allocation of physical effects, is set. The entire graph on the right-hand side (a, b, z) is defined as return value. Consequently, those three elements can be forwarded to the rule *create chain*, Table 3-1. This rule creates a chain of nodes until an identical port configuration is reached on the subfunction or behavior level. For this purpose, the node c with a matching port configuration is selected from the metamodel. As this rule only adds one element, repeated application is necessary. The formulation of rule sequences allows modeling this repetition.

In addition to a, b, z the following rule parameters are required:

- The boolean variable *left* defines in which direction the chaining process is carried out. In the rule depiction in Table 3-1, *left = False* is assumed.

- *nb_repetitions* defines after how many iterations in the chaining process the re-occurrence of a node type is allowed.

According to constraint ①, rule *2: create chain* only applies when a subfunction or physical effect chain is not yet finished. Therefore, the marked ports on the left-hand side *L* do not have to be compatible. If there are no incompatible ports detected, the left-hand side is not matching and the rule does not apply. Besides the matching ports, an additional constraint, ②, for the selection of valid elements from the metamodel is defined through the parameter *nb_repetitions*. The type of the new node *c* might be excluded from certain node types that were already used. These prohibited node types are stored in a list.

The rule *embody* uses the information that is stored in the attribute *realizes* of the components in the metamodel. It represents the physical effects that can be embodied and allows for the identification of component *c*. Hence, *c* can be associated to *a*. The attribute *realizes* contains single effects but also effect groups that can represent an unintended behavior *b*, e. g. friction, that has to be associated to the component.

The execution of the rule *merge* aims to find redundant components and to merge them. Based on the attribute *multiple*, it is defined for each element type whether multiple occurrences are allowed or not. Instances of the element type *Electrical machine*, can be used as an electrical motor (electrical input) but also as an electrical generator (electrical output). Thus, the attribute *swap* has been set to True in the metamodel. Consequently, elements with inverted input/output ports can be merged, as shown in the example in Table 3-1.

3.2.2 Grammar application

While the grammar definition, Figure 3-2, contains the metamodel and the set of rules, these two components of a formal grammar come to life in the grammar application. Making the metamodel and the rule set executable is achieved through a compilation step. While the grammar definition focuses on element and port types, at this point, the grammar operates with real data objects.

Rule Sequence and Initial Graph: Design Strategy

Extending the rule definition language with concepts from programming, as presented in 3.2.1, enhances significantly the expressiveness. Through the ability to set rule parameters and return values, rules can interact with each other throughout their application. Instead of having closed rules, this approach is based on a data flow between the application of rules. By default, rules return either the boolean value *True* when the left-hand side *L* matches or *False* in the case that no rule application is possible. The interplay of rules is realized in the definition of *rule sequences*. The ability to formulate sophisticated rule sequences provides a strong basis to implement problem-specific strategies.

Figure 3-4 depicts a general rule sequence for generating a FBS* product architecture. Based on the result of the rule application (*True* or *False*), the logical structure of a rule sequence can be created and visualized as a flow diagram. Another advantageous aspect of the separation of grammar definition and rule application becomes apparent here. While the metamodel, Figure 3-3, contains domain-specific building blocks, both the rule set and rule sequence are domain-independent and support the synthesis of any FBS*-based product architecture. This means that for a new application domain only the metamodel needs to be defined rather than a potentially large set of rules, as found in many approaches.
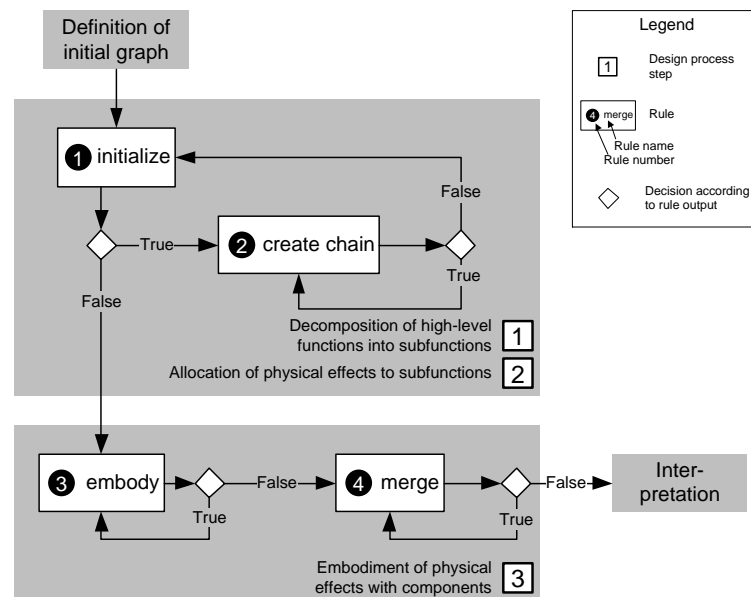
Figure 3-4: General rule sequence for the synthesis of FBS* product architectures

The application of this rule sequence is based on the definition of an initial graph, as the general example in Figure 3-5 illustrates. It provides the foundation on which all subsequent rule applications are applied. It contains the overall function (OF) decomposed into two high-level functions (HLF). In the next step, the interplay of the rules *1: initialize* and *2: create chain* handle both the decomposition process from high-level functions to subfunctions (SF) and the assignment of physical effects (PE). The assignment of components (C) to physical effects is achieved by the rule *3: embody* that is iterated until all behaviors are embodied by components.

Up to this point, independent FBS*-modules are created with each of them realizing one specific high-level function. To create one integrated product architecture, the rule *merge* looks for redundant components and merges them. The advantage of the modular procedure is that promising FBS* modules can be stored and reused. Further, the functional interrelations between the levels of abstraction become apparent. For these reasons, a merge of redundant subfunctions is not implemented as the individual functional modules would no longer be identifiable.

Graph transformation: Synthesizing solutions

The final step of executing a rule sequence is carried out by a graph grammar interpreter (GEISS ET AL., 2006) that interprets the directives in the rule sequence definition and executes the transformation, based on the predefined metamodel and the rule set. The resulting graph provides the basis for evaluating the quality of the design.

The rule sequence entirely defines the synthesis process. So far, the synthesis of new design candidates is mainly based on a randomized selection of suitable elements within the rules *initialize* and *create chain*. In accordance with (CAGAN ET AL., 2005), this can be considered as a mainly naïve generation method. However, due to the definition of compatible port types, a declarative model of feasible solutions is contained in the metamodel. The application of the rule sequence adapts to the elements and ports
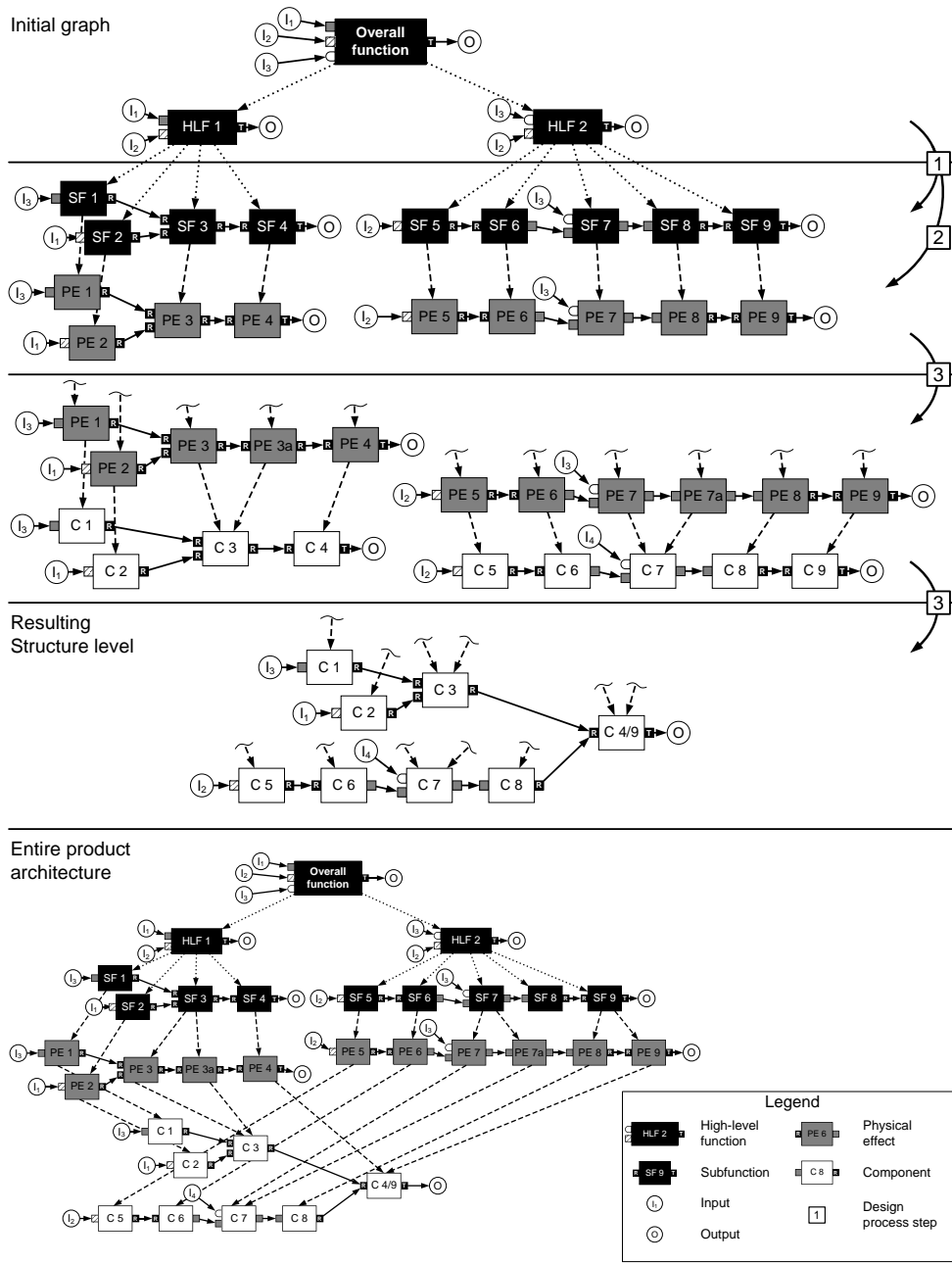
Figure 3-5: Example application of the general rule sequence in Figure 3-4

defined in the metamodel. Therefore, changes in the definition of the modeling elements do not trigger the necessity for changes in the definition of rules and the rule sequence.

Rules *1: initialize* and *2: create chain* randomly select elements for decomposing a high-level function into subfunctions and allocating physical effects to subfunctions. This can be considered as a random walk search. Being a rather slow search method, random walk will eventually find a solution, provided that the search space is finite, (RUSSELL & NORVIG, 2003). As the definition of ports constrain the solution space, the search can be regarded as constrained random walk. From an algorithmic perspective, the definition of ports serves three purposes: First, ports reduce, by restricting element compatibility, the search space significantly. Second, in conjunction with the rule parameter *nb_repetitions* in rule 2, the

length of port-based generated chains is finite. Hence, in accordance with (RUSSELL & NORVIG, 2003), random walk will find solutions for both the decomposition of high-level functions and the allocation of physical effects. Third, as rule 2 only applies if the ports that are checked in constraint ① are not of the same type, a clear termination condition is given for the chaining process. This leads to an additional reduction of the size of the search space. Increasing the efficiency of random walk search methods by reducing the compatibility of components for configuration tasks has extensively been discussed by MITTAL & FRAYMAN (1989). To conclude, the assignment of ports provides a means to control whether the search process is either naïve or knowledge-based (CAGAN ET AL., 2005). The more generic the port is, e. g. using solely energy flow ports, the less constrained the solution space is leading to a higher number of randomly generated solutions. This increases the chance to naïvely generate novel solutions but raises issues in terms of evaluating a high number of solution candidates. Best practice in grammars is to constrain the language to valid solutions and avoid meaningless solutions, e. g. by including knowledge about the differentiation between electrical and mechanical ports.

This research results in the development of the open-source software called booggie integrating and adapting the graph rewrite generator GrGen.NET[27] (GEISS ET AL., 2006) and the graph visualization library Tulip[28]. It is available to the public as open-source software and is presented in detail in Section 5.

3.3 Validation: Synthesis of hybrid powertrains

In the following, the synthesis approach based on object-oriented graph grammars is illustrated and validated through the example of the synthesis of automotive powertrains. The development of automotive drivetrains is characterized by the need to reduce emissions while maintaining the range of performance. This pressure led to an introduction of electrical components, e. g. batteries and electrical motors, allowing for new, innovative product functions, such as start-stop systems and regenerative braking. However, an embodiment of these new functions cannot be achieved as add-ons to existing, conventional powertrains but requires the re-conceptualization of product architectures. Consequently, a wide spectrum of hybrid powertrain architectures is emerging, ranging from low electrification, e. g. mild hybrid, to full hybrid drivetrains using a combustion engine only for charging the battery but not for propulsion (FUHS, 2009). The range of architectural design alternatives has tremendously increased compared to conventional powertrains. An analysis of existing hybrid powertrain architectures by GORBEA ET AL. (2008) showed that 432 architectural alternatives could be differentiated based on state-of-the-art components in 2008. It is assumed that this number continues to increase significantly with emerging technology leading to new components and even more architectural options. To cope with these serious challenges, knowledge-based systems are required to support the design process (STRUSS & PRICE, 2003). Once a product architecture is defined, optimization approaches can be applied to identify design variables such that the fuel consumption is minimized (NEEMA ET AL., 2005). However, to systematically explore the range of demanded embodiments of required functions, a synthesis method is required that actually generates alternative product architectures on multiple abstraction levels. The application of the method presented in Section 3.2 aims to support the conceptual phase of powertrain design by generating design alternatives.

---

[27]Project website: http://www.grgen.net
[28]Project website: http://tulip.labri.fr

Hybrid powertrain architectures are characterized by their high modularity. Due to the limited range of interfaces for the exchange of energy (electrical, chemical, mechanical rotational), the interconnectivity of components is high. This results in a combinatorial solution space and strengthens the arguments for applying computational synthesis methods. Further, the port-based representation of the object-oriented graph grammar approach is a natural fit. As research in this area is highly dynamic, new functions, physical effects and components need to be considered. It is therefore crucial to have a synthesis method that allows for the rapid formalization of evolving knowledge in an efficient way and addresses thereby Research Issue 1 (inefficiency of knowledge formalization). That is to say that a straightforward extension of the metamodel, e.g. by adding new physical effects, has to be supported. A change of the rule set, though, is only required if the overall design strategy for the synthesis of FBS* product architectures is subject to modification.

3.3.1 Grammar definition

Defining the metamodel is the first step in the definition of a formal object-oriented graph grammar. The definition of port types and edge types within the metamodel presented in Section 3.2.1 remains the same and does not have to be adapted to this specific application. Hence, only the set of element types has to be extended (Figure 3-6).

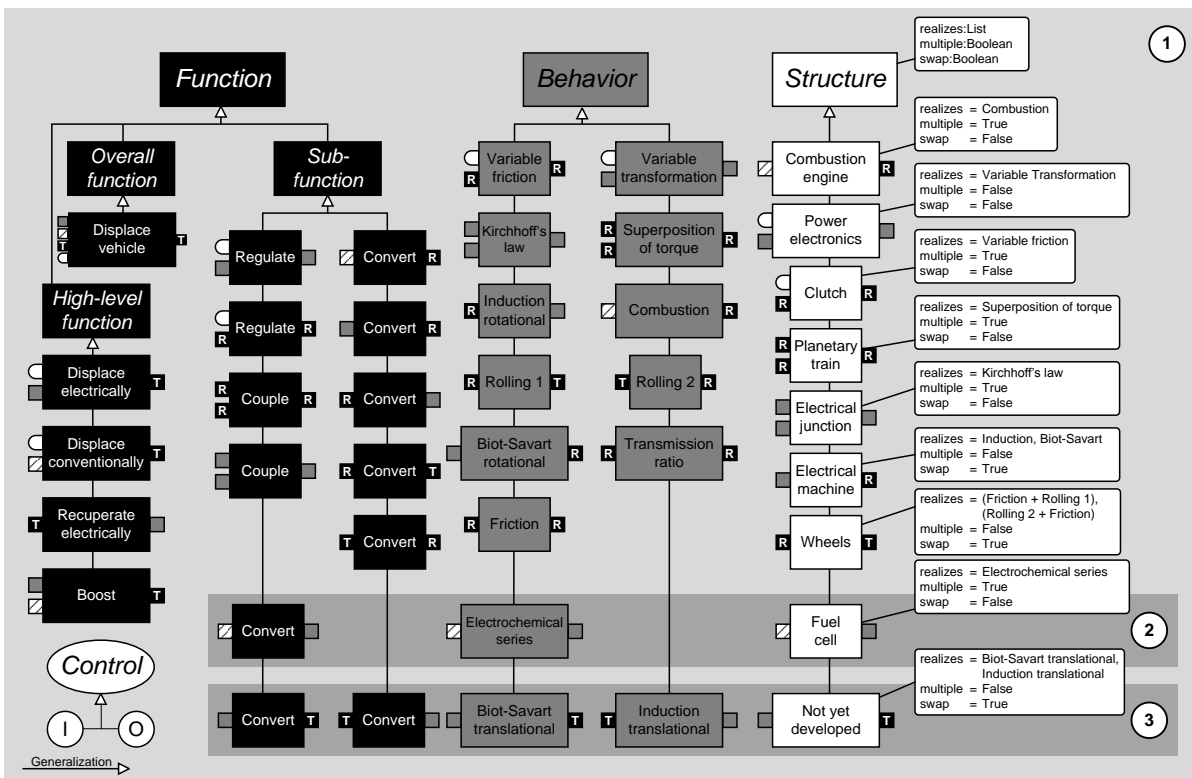Figure 3-6: Problem-specific element types within the evolving metamodel for the synthesis of hybrid powertrains

The principal task of a vehicle powertrain is to convert stored energy into a translational movement relative to the road. The amount of energy converted is controlled by the user of the vehicle by means of signal flow. Secondary tasks, like traction control or a antilock braking system, are not considered here.

Hence, interaction between the elements can be reduced to the exchange of energy and the exchange of a control signal.

The function *Displace vehicle* is considered as the only overall function. It can be decomposed into the high-level functions *Displace electrically*, *Displace conventionally*, *Recuperate energy electrically* and *Boost*. This decomposition step is not carried out computationally but by the human designer. The second, yet computational, decomposition step results in the assignment of subfunctions. Elements of this third category are modeled using the set of functional operators from the Functional Basis (HIRTZ ET AL., 2002b) while the associated port types define the functional flows.

As energy storage is considered to be outside the powertrain's system boundary, all overall and high-level functions include energy input ports. The signal inputs are required for the interaction with the driver. By means of the input and output elements (subtypes of *Control*), the system boundary is marked.

The evolving metamodel is shown in Figure 3-6. The initial metamodel, ①, is extended in two stages: First, the exploration of the physical effect *Electrochemical series* allows for an inclusion of the additional function *Convert chemical energy to electrical energy* and resulted in an additional component, a *Fuel cell*, ②. In the second extension, the two effects *Biot-Savart translational* and *Induction translational* become applicable for translational applications and allow for two additional functions, ③. For example, the German monorail train Transrapid uses these effects for magnetic levitation. However, components for an embodiment of these new effects in the automotive domain are currently not available. At this metamodel stage ③ they are potentially embodied with the placeholder component marked with *Not yet developed*. As this component embodies the two mentioned physical effects, in- and output ports can be swapped. Rule definitions remain as presented in Section 3.2 and can be based on different versions of the metamodel.

3.3.2 Grammar application and discussion of the synthesis results

This section serves two purposes. First, an example synthesis procedure is shown to illustrate the method and its advantages. Second, to discuss the entire solution space and explore the characteristics of generated solutions.

The starting point for synthesizing a FBS* product architecture is a functional model that serves as an initial graph. In the example, Figure 3-7, the overall function *Displace vehicle* transforms the four inputs (electrical energy, chemical energy, mechanical translational energy and a control signal) into mechanical translational energy. To define the required functionality more specifically, this function has been decomposed by the user into the high-level functions *Displace electrically*, *Displace conventionally*, *Recuperate energy electrically* and *Boost*.

For brevity, only the functions *Displace conventionally* and *Recuperate energy electrically* are considered further, Figure 3-8. Starting with the initial graph, the functional decomposition is accomplished based on the rules *initialize* and *create chain* as depicted in Figure 3-4. While energy recuperation is decomposed into rather obvious subfunctions, conventional displacement involves a regulation of the energy flow in the electrical domain and requires therefore auxiliary convert functions. The set of physical effects in the metamodel provides for a direct allocation to subfunctions. For this reason, only one-to-one mappings are required to embody the function structure. Hence, the rule *initialize* directly finds a valid port configuration, consequently, the rule *create chain* finds no valid match on the left-hand side and is not applied.
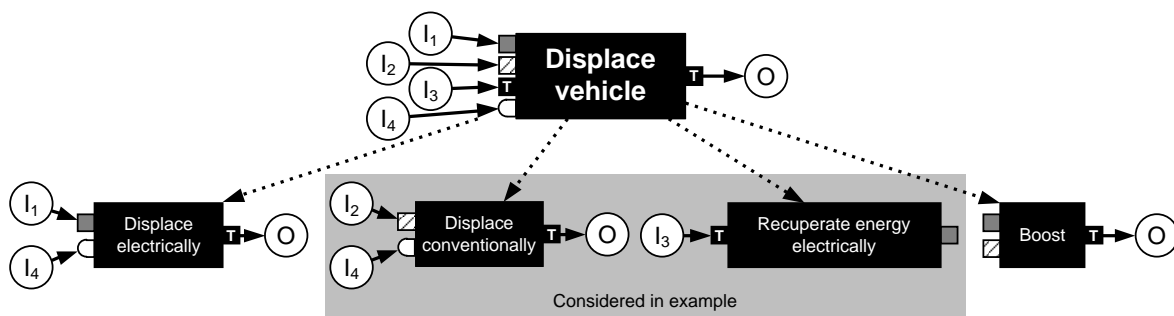
Figure 3-7: Functional model as initial graph

The embodiment of physical effects into components is based on the specification in the metamodel within the components parameter *realizes*, Figure 3-6. Primarily, one component is allocated to one physical effect. However, based on the parameter *realizes* the consideration of unintended effects is also feasible. In the example, an embodiment of the effects *Rolling 1* and *Rolling 2* with *Wheels* necessarily involves the presence of *Friction* in the wheels' bearings reducing the efficiency of transforming rotational mechanical energy into translational mechanical energy. The rule *embody* takes this into consideration when adding the unintended effects based on the *Wheels'* parameter *realizes = (Friction + Rolling 1), (Rolling 2 + Friction)*. This process step can be regarded as a minor contribution to computationally executing the "Analysis"-step of the FBS framework (GERO, 2004) thats aims at deriving the "actual behavior" from the component structure. However, using the rule *embody*, this process step is limited to solely adapting the product architecture's topology; quantitative characteristics of the product remain unconsidered.

The assignment of components might result in the creation of redundant elements. For example, *Wheels* are added twice due to the physical effects *Rolling 1 & 2*. However, one component is sufficient as defined with *multiple = False*. Further, in- and output of *Wheels* can be swapped for the association with components (*swap = True*). Hence through the application of the rule *merge*, redundant *Wheels* and *Electrical machines* are merged and the two individual modules, stemming from the high-level functions *Displace conventionally* and *Recuperate energy electrically*, become one product architecture.

The resulting component architecture corresponds to a classic, series hybrid powertrain and embodies the required functionality of *Displace conventionally* and *Recuperate energy electrically*. Due to the composition of the metamodel, alternative solutions result from variations in the functional model. In this example, Figure 3-8, a variation in the functional decomposition results in two more architectures: a parallel and a through-the-road hybrid. The former represents a standard parallel hybrid architecture, which lacks an electrical output because no output was defined for the high-level function *Recuperate energy electrically*. The latter is a more creative through-the-road solution. Its name stems from the fact that the mechanical energy at the output of the *Combustion engine* is propagated through the *Wheels* back to itself as can be seen from the loop at the mechanical translational port. As a wheel cannot transfer energy to itself, a second *Wheels* element is assumed. Consequently, one could imagine that the front wheels are driven directly by the combustion engine and the rear wheels apply an opposite torque to recuperate the energy from the road and convert it into electrical energy. If electrical energy storage is not considered in the function model, a battery could be included to store electrical energy, e. g. when braking, and to drive the vehicle electrically by the rear wheels. The fact that the *Combustion engine* is constantly applying a torque on the *Wheels* and the speed regulation is achieved through the electrically
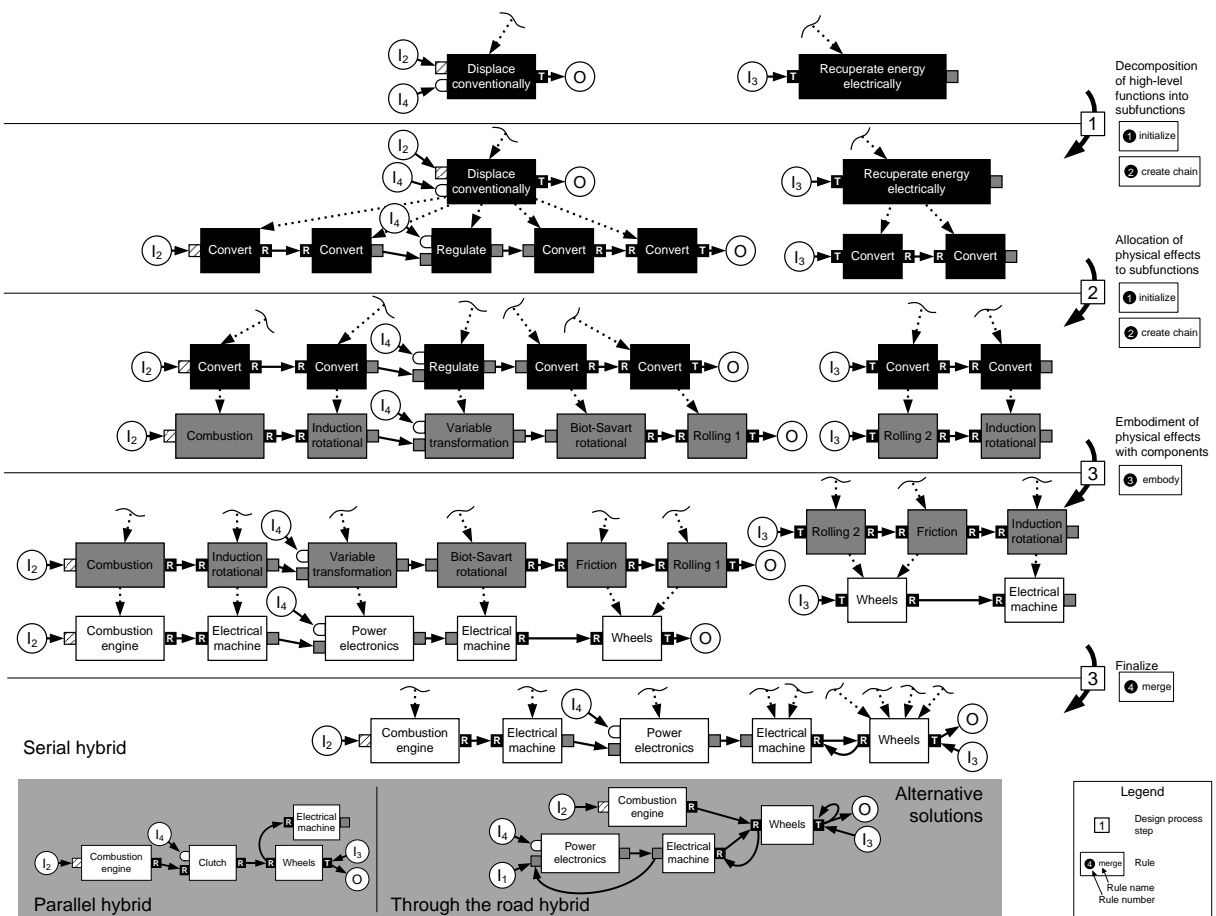
Figure 3-8: Example synthesis procedure

driven *Wheels*, makes this concept novel. However, this is not a solution that can be implemented right away. For example, applying a contrary torque on the wheels braces the powertrain and might result in increased heat production; further, if the electrically generated torque is too high the combustion engine might stall. However, the new solution generated might spark further investigation.

Since the set of generic rules is static, new solutions emerge only from an evolving metamodel as depicted in Figure 3-6. Hence, the formalization of new knowledge within new element types has a direct impact on the solution space. This interrelation is shown in Figure 3-9. Since alternative solutions arise solely from functional alternatives, the development of the solution space can be studied by considering solutions on the function level. The overall function *Displace vehicle* is composed of the four high-level functions. Hence, the number of overall solutions is the multiplication of the number of solutions for the high-level functions. The total number of solutions is finite because the rule *create chain* as presented in Table 3-1 avoids (through the rule parameter *nb_repetitions*) that infinite energy conversion chains are created. Thereby, the number of possible energy conversion chains used for the decomposition of high-level functions can be analytically predicted. The total number of solutions can hence be calculated through combinatorial enumeration. This kind of analysis is only feasible when the total number of solutions is finite and the number of elements and their valid combinations in the metamodel is sufficiently small. As discussed in the following, the incorporation of new elements into the metamodel has a significant impact on the number of possible solutions for a given design problem. This makes the analysis of a solution space increasingly difficult for larger metamodels solely by reason of the sheer number of

solutions to be considered.

The evolvement of the metamodel reflects the engineer's evolving knowledge, e. g. due to new techno-logical developments. Let us assume the following scenario: Research on the physical phenomenon of *Electrochemical series* results in the development of a new component: the *Fuel cell*. Based on this, a new function for a direct conversion of chemical energy into electrical energy is added (metamodel stage ②). Consequently, the high-level functions involving a chemical energy flow (*Displace conventionally* and *Boost*) benefit from this new technology and increase the number of valid solutions for decomposing the overall function *Displace vehicle* from 18 to 50.

Metamodel state	①	②	③

	Number of element types		
Subfunctions	9	10	12
Physical effects	11	12	14
Components	7	8	9
Total	**27**	**30**	**35**

	Total number of solutions*		
Displace conventionally	3	5	14
Displace electrically	2	2	6
Recuperate energy electrically	1	1	2
Boost	3	5	20
Drive vehicle	**3 x 2 x 1 x 3 = 18**	**50**	**3360**

*: The number of solutions in the metamodel stages 2 and 3 includes the solutions of the previous metamodel stages.
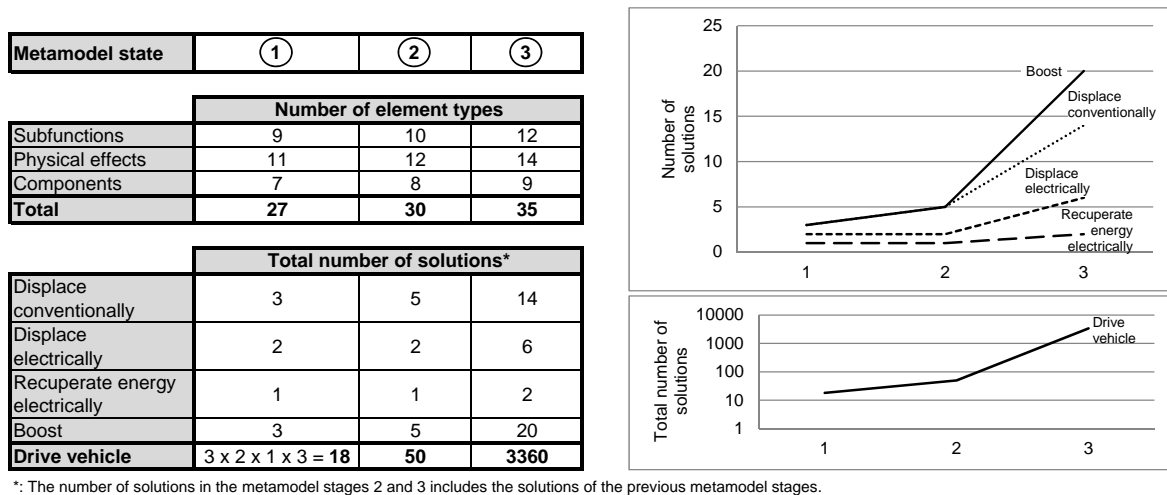
Figure 3-9: Impact of evolving metamodel on the number of possible solutions

Next, the set of functions in the metamodel are further analyzed. The result is that a function allowing for a direct conversion between mechanical translational energy and electrical energy provides for a wide range of new functional alternatives. The two corresponding *Convert* functions are added. This increases the number of possible solutions for all high-level functions. The effect of these new functions is so tremendous because every high-level function has mechanical translation and/or electrical ports and as they all have a mechanical translational output port (in contrast, the first expansion of the metamodel from ① to ② only affects high-level functions with a chemical port). At all these ports, the new functions provide additional alternatives for decomposing the high-level functions leading to a raise of the number of subfunction models realizing the overall function *Displace vehicle* from 50 to 3360. To embody these functions, the physical effects *Biot-Savart translational* and *Induction translational* can be identified from a design catalog (KOLLER, 1994). Components that could be associated to these effects are unfortunately not yet available and present a great technological challenge. Hence, the study of the solution space on the functional level can motivate to trigger further research in that area to be able to provide new technological solutions for hybrid powertrain architectures.

Example solutions that realize all four high-level functions and show noteable characteristics are dis-cussed in the following. Although providing for a general feasibility – assured through matching flow ports – these models require a subsequent post processing or interpretation as depicted in the general rule sequence for the synthesis of FBS* product architectures in Figure 3-4. This interpretation is achieved based on the identification of solution characteristics that are represented as graph structures. The names of the solution characteristics used for the subsequent solution space analysis are highlighted in boldface and are gray shaded in the following figures depicting the hybrid powertrain architectures.

Solution 6_1_1_10[29] in Figure 3-10 only uses modeling elements of the initial metamodel (stage ①). For various reasons (e. g. high component costs, high weight), the existence of two combustion engines is questionable and a merge should be taken into consideration. The direct connection of a clutch with wheels cannot be found in reality; typically, a gearbox and a differential are located between them. As the functional model does not contain a function for adapting the torque level, these components are not added. Besides functional considerations, the necessity to add these two components could also arise from parametric considerations that are omitted for now. For example, the gearbox is required to avoid engine stalling by adapting the torque delivered by the combustion engine. As discussed earlier, the fact that the *Wheels*-element has a connection with itself is noticeable. Splitting up the wheels, such as in the example in Figure 3-8, results in a **through-the-road** solution. Further, solution 6_1_1_10 is **divisible** into two disjoint architectures by removing the *Wheels* element.
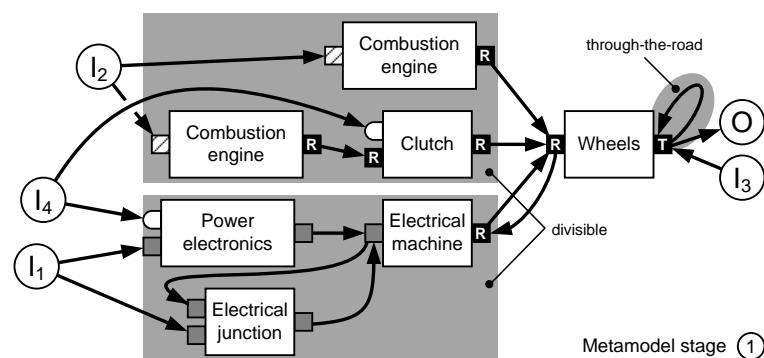

Figure 3-10: Solution 6_1_1_10

Solution 10_1_3_9 in Figure 3-11 builds on metamodel stage ②, recognizable by the presence of the *Fuel cell*. The arrangement of a *Planetary train* linked to an *Electical machine* and a *Combustion engine* can be found in **classic parallel hybrid powertrain** architectures as well. One *Clutch* is redundant to regulate the energy flow between *Wheels* and the *Electrical machine* and can be omitted. Using a *Clutch* here does not violate any physical principles, but controlling the electrical energy supply of the *Electrical machine* would be advantageous. Solutions of the stages ② and ③ can potentially have a *Fuel cell* and a *Combustion engine* in one solution. In these cases it becomes apparent that the flow port hierarchy should be refined as chemical energy can be further subdivided into hydrogen for the *Fuel cell* and gasoline or diesel for the *Combustion engine*.
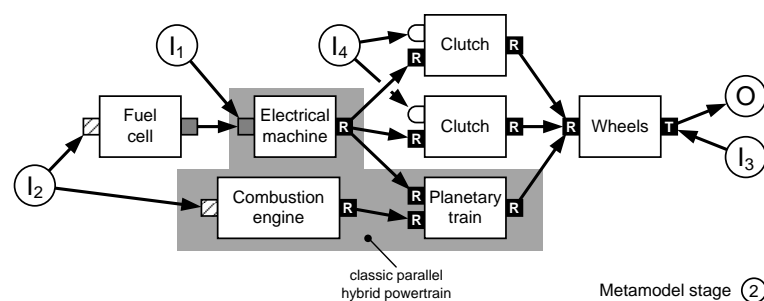

Figure 3-11: Solution 10_1_3_9

---

[29]The four values of the solution index designate the individual solutions for every high-level function and reads as follows: 6th solution for *Boost*, 1st solution for *Recuperate electrically*, 1st solution *Displace electrically* and 10th solution for *Displace conventionally*.

Solution 9_1_1_7 in Figure 3-12 also uses a *Fuel cell*, i.e. metamodel stage ② is applied. A **classic serial hybrid powertrain** topology can be identified when regarding the composition of *Combustion engine*, *Electrical machine* and *Power electronics*. The loop between *Electrical machine* and *Power electronics* should be broken up by using two electrical machines, i.e. a **split up into engine and generator**. The reason why the element *Electrical machine* occurs as engine and as generator lies in the fact that the flag *swap* is set to True in the metamodel, see Figure 3-6, and allows that input- and output-ports can be swapped. The mechanical rotational energy at the output of the *Combustion engine* is then converted into electrical energy by the first *Electrical machine* in generator operation. The *Power electronics* component allows to regulate the electrical energy flow that is converted into mechanical translational energy by the second *Electrical machine* in engine operation. The second *Electrical machine* has an electrical output to feed the recuperated energy into an energy storage device, e.g. a battery.
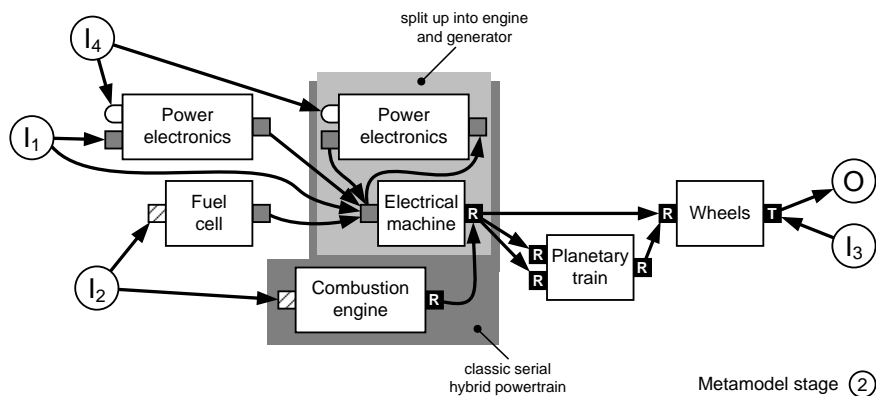

Figure 3-12: Solution 9_1_1_7

Solutions built on metamodel stage ③, such as solution 6_0_0_9 in Figure 3-12, can have the peculiarity of being separated into **disjoint architectures**. This is due to the fact that *Wheels* are not necessarily required to embody every high-level function as the placeholder component *Not yet developed* can be used instead. Hence, not all sub-architectures are inevitably integrated in one single architecture through the merge of *Wheels*. The future, *Not yet developed* component is used here to embody the high-level function of *Recuperate energy electrically* as it has a mechanical translational input and an electrical output. It also embodies, together with the *Power electronics* element, the high-level function *Displace electrically*.
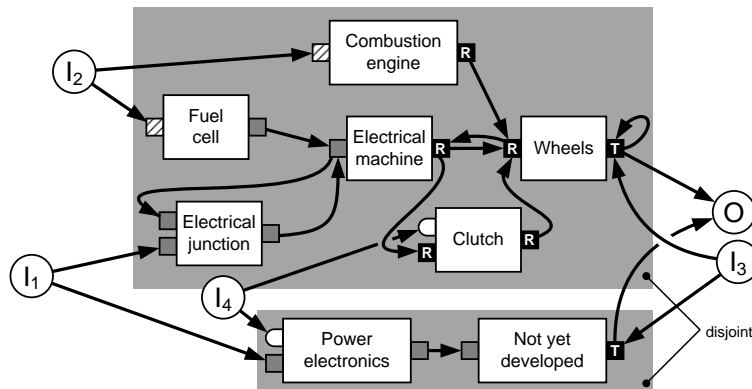

Figure 3-13: Solution 6_0_0_9

The manual analysis of example solutions is considered valuable as it allows to grasp insights into indi-

vidual solutions and the nature of the solution space in its entirety. However, the big number of solutions requires an automated analysis and interpretation of the solution space as not all synthesized powertrain architectures can be analyzed and interpreted by hand. The solution characteristics previously introduced are used for that purpose. They are applied in the left-hand side of rules that are dedicated for the interpretation of every single solution of the solution space. Table 3-2 shows the results of the analysis and interpretation of the solution space.

Table 3-2: Analysis and interpretation of the solution space based on solution characteristics

Metamodel stage		①	②	③
Number of solutions *		18	32	3310
Electrification ratio	before merge	33.6%	40.1%	40.4%
	after merge	35.4%	48.4%	42.3%

	Occurrences of solution characteristics					
Through-the-road	10	55.6%	8	25%	750	22.7%
Divisible	14	77.8%	12	38%	2357	71.2%
Disjoint architectures	0	0.0%	0	0%	172	5.2%
Classic parallel hybrid powertrain	6	33.3%	4	13%	326	9.8%
Classic serial hybrid powertrain	10	55.6%	8	25%	402	12.1%
Split up engine and generator	16	88.9%	16	50%	736	22.2%

*: The number of solutions in the metamodel stages 2 and 3 does not include the solutions of the previous metamodel stages.

Adding new elements to the metamodel has a noticeable effect on the electrification ratio of the generated powertrain architectures. This ratio is calculated for every solution by dividing the total number of components by the number of electrical components. Any component that has at least one mechanical translational port or mechanical rotational port is counted as a mechanical component and the same with electrical ports applies to electrical components. Hence, an *Electrical machine* is counted once for the electrical components and once for the mechanical components. An electrification ratio of 100% means that all components are purely electrical components. The ratios shown in Table 3-2 are the average values of all electrification ratios for every metamodel stage. They are distinguished between the ratio before and after the components are merged. While in stage ① 33.6% of the components are electrical components, this number rises to 40.1% and 40.4% in the subsequent stages. This is due to the addition of the electrical components *Fuel cell* and *Not yet developed*. The individual FBS* modules are mainly merged through the merge of *Wheels*. As they are mechanical components, it is logical that the electrification ratio increases after the merge step that leads to a decreasing number of mechanical components.

A "through-the-road"-configuration appears if two *Wheels* elements are merged having an opposed energy flow. With a decreasing number of mechanical components, the likelihood for *Wheels* to appear decreases. Thereby, a decrease of the occurrence of this characteristic is observed throughout the metamodel stages.

The fact that in metamodel stage ② the direct conversion of chemical to electrical energy is possible, raises the probability of occurrence of *Electrical machines*; they are another component for integrating the independent FBS* modules. Hence, dividing a powertrain into disjoint architectures with splitting a *Wheels*-element into two *Wheels*-elements is possible in fewer cases. The reason why the "divisible" characteristic occurs more often in metamodel stage ③ can be explained with the higher electrification ratio of the powertrain due to the additional electrical components. The remaining mechanical part of the powertrain (mostly consisting of a *Combustion engine* and a *Clutch*) is detached when removing the

Wheels.

All high-level functions have a mechanical translational input- or output-port. In the metamodel stages ① and ②, *Wheels* are the only component with a mechanical translational port. Hence, they occur in every independent FBS* module leading necessarily to one integrated solution while merging. By adding the *Not yet developed* component in stage ③, an alternative component with a mechanical translational port is available. This leads to the occurrence of FBS* modules that cannot be merged with other *Wheels*-based modules and, consequently, to the possibility of disjoint architectures.

The occurrence of the classic parallel and serial configurations shows that this approach is able to generate known, meaningful solutions. With increasing electrification of the solutions, these classic configurations become less frequent. This is logical as their indispensable mechanical constituents are getting replaced with electrical components, e. g. a *Fuel cell* instead of a *Combustion engine*.

In metamodel stage ①, *Electrical machine* is the only component that can realize an energy conversion from or towards the electrical energy domain. Alternative components for that purpose are added in stage ② and ③ leading to a decreased occurrence of *Electrical machines*. Hence, the number of cases that necessitate a split up of electrical engine and generator becomes smaller.

3.4 Discussion

To efficiently support the conceptual phase of the innovation process, methods and tools are required that are based on a general and domain independent design methodology allowing product synthesis on multiple levels of abstraction, but such implementations are rare (ERDEN ET AL., 2008; EIGNER ET AL., 2012). The method presented in this chapter contributes to advances in this area since it introduces an integrated design representation for conceptual design based on the levels of abstraction Function-Behavior-Structure. It's combined with an object-oriented graph grammar approach capable of automated synthesis of product architectures on all levels. Further, extensions to conventional graph grammar methods were developed drawn from object-oriented programming. Thus, the manageable complexity of generated product architectures is higher with regard to the state of the art in CDS.

The definition of the design representation is based on the definition of the metamodel. It contains the set of building blocks and their interconnectivities, termed ports, and their logical structure using the concept of inheritance. Thus, the metamodel enables the formalization of declarative engineering knowledge stemming from the engineer's expertise and including knowledge from design catalogs in a model-based representation. It contains the modeling elements that are required to solve a specific design task. Rules can be defined in a generic way using the metamodel, especially the port types defining a taxonomy of flows. Hence, general and domain-independent design procedures, such as a functional decomposition process, can be translated into a rule-based representation. Inclusion of specific knowledge can also be taken into account; for example, the definition of the embodiment of physical effects through components is not based on a generic mapping but rather on a fixed mapping defined in the metamodel.

The approach is validated through the synthesis of hybrid powertrains. The generated architectures show one of the key benefits of using the presented method for the synthesis of FBS* product architectures. Although the solutions may not be entirely meaningful, e. g. redundant components, the concept of ports ensures that the basic physical compatibility is guaranteed. Hence, the proposed solutions provide a trigger for rethinking conventional solutions. This is a crucial contribution to the ideation of innovative solutions in the conceptual phase. In analyzing a generated, novel product architecture and trying

to understand the product architecture's inner workings, the designer is stimulated to think outside the box. Admittedly, a manual interpretation of 3360 potential solutions cannot be expected from a human user. A special rule set dedicated to the analysis and interpretation of solutions to automate the process is developed. It includes rules for detecting solution characteristics requiring further corrections or identifying solutions being worth further investigation, e. g. retrieving special configurations, such as through-the-road concepts. Especially for powertrains implementing all four high-level subfunctions, a rule set for the subsequent interpretation of the generated product architecture provides added value as these solutions are larger and a manual approach could be time-consuming.

The metamodel, as a repository of declarative knowledge, provides a rich foundation for the rule-based formalization of procedural engineering knowledge. Combining model-based and rule-based knowledge representations into one hybrid representation allows for an increase of the scope of application (SRIRAM, 1997).

As long as a specific metamodel respects the general structure, e. g. the FBS* levels of abstraction, the set of only four generic design rules can be applied to synthesize solutions. This is a core contribution of this approach since it circumvents the issue of exhaustive rule formulation. Through this flexibility, the approach combines advantages of a generic and systematic design method that is applicable to a wide range of design problems.

As engineering knowledge is constantly evolving, e. g. due to the development of new technologies, knowledge formalization is always needed and often a bottleneck in applying engineering grammars. The presented approach supports this issue through the concept of inheritance. It supports extendibility by enabling an extension of the inheritance hierarchy in the metamodel without creating incompatibilities. Further, rules defined on an abstract element type level, or involving the use of ports, remain valid even when elements are added, modified or deleted, as shown in Section 3.3.2.

Given the results obtained with the validation study, it seems fair to conclude that the Expected Contributions 1 and 2 have been achieved. Contribution 1 delivers an increase of efficiency in the knowledge representation, specifically in terms of:

1. A small, generic rule set remains manageable due to the definition of a metamodel to define a wide solution space.

2. A clear and simple process is developed for the definition and application of object-oriented graph grammars (cf. Figure 3-2).

3. The generic rule set and the flexibly expandable metamodel support the formalization of evolving engineering knowledge.

4. The application of a computationally efficient graph transformation library allows to generate and analyze high numbers of solutions in reasonable time. An extension towards more elaborate search and optimization algorithms is feasible, as discussed in (JAKUMEIT ET AL., 2010) and could be subject to future work. For a quantitative assessment of the computing time of the presented synthesis approach, refer to Section 5.4.5.

Object-oriented graph-grammars are a hybrid knowledge formalization that can serve as a formal knowledge foundation for synthesizing graph-based models. They are, by that, an effective representation to support a wide range of applications leading to the achievement of Contribution 2 (high effectiveness of

object-oriented graph grammars). Thereby, this contribution aims to achieve Research Goal 2 (generate known and new solutions), namely the generation of known and new solutions. Known powertrain characteristics, e. g. through-the-road, parallel, serial, such as they are described in known powertrain architectures by Gorbea et al. (2010) occur in various solutions in numerous combinations. New and controversial solutions that merit closer consideration occur as well. This result also corroborates the Expected Contribution 5 regarding the generation of human-competitive solutions and is further discussed in the industrial case study in Section 5.5.

Finally, it only remains to reflect on the developed approach with respect to the quality factors of object-oriented programming that were interpreted from a CDS perspective in the beginning of Section 3.2. The upfront definition of the modeling space by means of a hierarchical metamodel, contributes to the goal of *extendibility* by enabling the subsequent addition and detailing of elements in a structured way. Moreover, this kind of type hierarchy is intuitively understandable and increases the *ease of use*. Due to the concept of inheritance, the definition of overlapping or redundant attributes or elements is abolished. Further, the class taxonomy of the metamodel supports the automated creation of generic rules but also the definition of knowledge intensive rules that apply not only to one type of element but also to all of its subtypes. These features facilitate the encapsulation of engineering knowledge and support hence the goal of *efficiency*. The separation of representation and program execution enhances the *reusability* of metamodels for multiple applications and domains. Moreover, parts of the metamodel can be detached, further detailed and reintegrated. The *compatibility* of different metamodels and rule sets can be assured as long as naming conventions, e. g. for functional modeling, are maintained. This comprises in particular rules that reason on different levels of granularity.

Up to now, quantitative aspects of the design representation and generation are disregarded. The exploration of the solution-space is based on a constrained random walk search that can be considered as a random- and port-based search process seeking for valid input and output configurations. In the presented example, this approach is appropriate because the function set consists of elements that can be entirely defined through their energy- or signal-flow ports; it is in the very nature of powertrains to convert energy flows. However, as there is no explicit definition of which physical effect can be allocated to which subfunction, unsatisfactory results might arise if the nature of the problem domain is less energy-flow-oriented. Further, if the metamodel consisted of a wider spectrum of functions, this approach would create a high number of solutions requiring interpretation, which demands efficient evaluation methods. In theory, this can be achieved taking the presented approach further for analyzing the solution space. This post-processing of the solution space can be taken further to repair or enhance solutions. Nevertheless, as shown with the impact of the evolving metamodel, the number of solutions increases dramatically with every added element in the metamodel. For this reason, a more purposeful selection of physical effects is required that does not only compare the flow inputs and flow outputs of subfunctions and physical effects. An approach to tackle this issue based on the automated formalization of physical effect is presented in the following chapter.

4 Automated allocation of physical effects to functions using abstraction ports

The method presented in the previous chapter is an object-oriented graph grammar that enables the efficient definition of generic design rules based on valid port configurations. These rules are defined in a problem-independent manner and allow the formalization of the process step for the allocation of physical effects to functions within two rules. The application of these generic rules is based on the definition of a flow port hierarchy in the knowledge repository and a set of physical effects that are formalized from design catalogs. The knowledge repository is realized as a metamodel defining the set of building blocks. It defines the modeling elements that are available in the FBS* domain and can be instantiated for synthesizing product architectures (HELMS & SHEA, 2012). For this approach, ports are crucially important and their significance has been discussed in Section 2.3.2.

The synthesis process that is based on flow ports, as depicted in Figure 4-1, compares the input and output ports of a function to the input- and output-ports of physical effects and searches in the knowledge repository to identify possible candidates. The principal advantage is that this process complies with any kind of physical effect in the repository that is based on the notation of flow ports. Hence, seeking compatible input and output port configurations ensures general compatibility, even for new physical effects resulting out of technological advancement. However, this selection process is purely based on the compatibility of the energy, material or signal flow ports. The compatibility of the functional operators, e. g. Convert, Store, Supply (HIRTZ ET AL., 2002b), with the characteristics of physical effects is not considered and might result in an expansion of the solution space that makes the identification of good, or even optimal, solutions difficult. The effect of the *Coriolis force*, for example, has an appropriate input and output characteristic of its flow ports to be allocated to the function *Convert translational mechanical energy to rotational mechanical energy*. Additional features of this effect, though, such as the possibility to modulate the energy conversion through a signal input, are not considered.
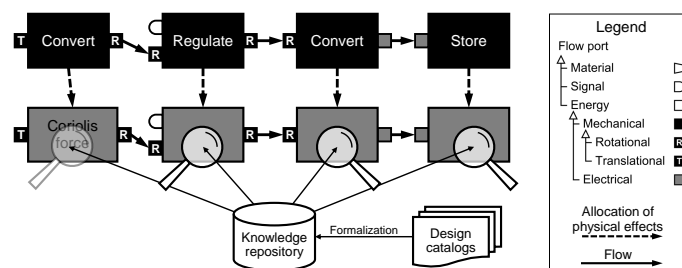

Figure 4-1: Allocation of physical effects to functions based on flow ports

Design catalogs (EHRLENSPIEL, 2009; KOLLER, 1994; PONN & LINDEMANN, 2011; KOLLER & KASTRUP, 1998; ROTH, 2001) provide a large source of knowledge about physical effects. The method presented in the previous chapter solely uses the formalization of physical effects based on flow ports with the above-mentioned drawbacks. Integrating the selection matrices included in the design catalogs would ensure compatibility between functional operators and physical effects. However, this would also result in a

knowledge base that is both inflexible and difficult to maintain and has been seen in previous work (BRYANT ET AL., 2006; ŽAVBI & RIHTARŠIČ, 2009).
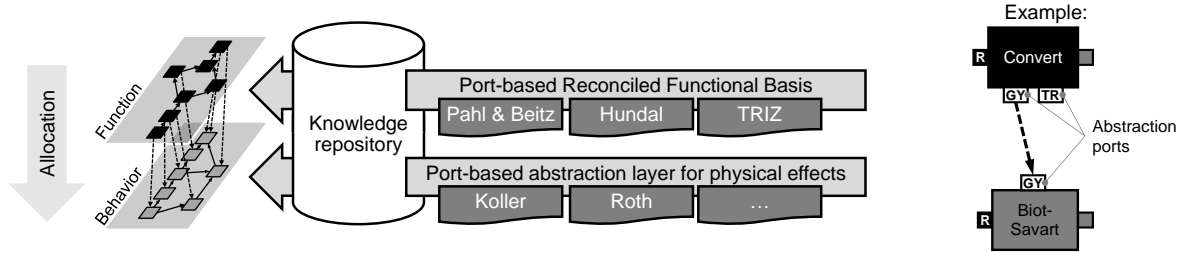

Figure 4-2: Allocation of physical effects to functions based on a port-based knowledge repository

According to Research Issue 3 (lack of reuse of design knowledge), the heterogeneity of the numerous knowledge sources calls for the development of a method that allows to systematically express and classify physical effects at a uniform layer of abstraction. The gap between Function and Behavior could thus be bridged. This allows for standardization, i. e. solving compatibility issues between different design catalogs, supports the uniform formalization of evolving physical effects and provides for a mapping to the established functional modeling. For these purposes, the notion of abstraction ports is introduced. It also provides for a uniform classification regardless of the energy domain, which is of high benefit for the synthesis of product architectures consisting of multiple energy domains. Analogous to flow ports, which represent the conceptual compatibility of flows on the same level of abstraction, *abstraction ports* represent the compatibility between subfunctions and physical effects and provide for increased efficiency when searching computationally for appropriate physical effects. The essential objective of this chapter is to introduce a method that reuses knowledge from design catalogs and allows to formalize it in a uniform way based on the graph-based representation introduced in Chapter 3. Thereby, this chapter leads to the Expected Contribution 3 (formalization of design catalogs).

4.1 Method context

The methodology of bond graphs is a form of object-oriented physical system modeling developed by Paynter (PAYNTER, 1961) to model dynamic systems. The consideration of different energy domains (e. g. mechanics, hydraulics, electronics) is provided based on a uniform modeling concept. Providing a general domain-spanning set of elements, it supports a way to unify all heterogeneous, paper-based knowledge sources by defining a unified abstraction layer for physical effects.

Hence, the presented method aims to assign abstraction ports to physical effects considering solely the nature of the exchange of energy. Based on the analysis of physical effect equations, the future categorization of new effects that have yet to be formulated or discovered is supported.

Figure 4-2 illustrates the general approach within the context of the design representation presented above focusing on the embodiment of subfunctions into physical effects. Both the port-based Reconciled Functional Basis and the port-based abstraction layer for physical effects can be considered as a formalization method that provides uniform access to either functions or physical effects for computational design synthesis.

Table 4-1: Domain-specific state variables

Energy domain	Effort e			Flow f			Momentum p			Displacement q		
	Name	Sign	Unit	Name	Sign	Unit	Name	Sign	Unit	Name	Sign	Unit
Mechanical translational	Force	F	N	Velocity	v	m/s	Linear momentum	p	Ns	Linear displacement	x	m
Mechanical rotational	Torque	T	Nm	Angular velocity	ω	1/s	Angular momentum	L	Nms	Angle	α	rad
Electrical	Voltage	U	V	Current	I	A	Flux linkage	λ	Vs	Charge	Q	As

The core contribution of this method is the assignment of abstraction ports, as depicted in the example seen in Figure 4-2: The assignment of flow ports to the subfunction Convert rotational mechanical energy to electrical energy and to the physical effect Biot-Savart follows the general functional modeling methodology (PAHL ET AL., 2007; EHRLENSPIEL, 2009; PONN & LINDEMANN, 2011). The abstraction ports ensure that the physical effect is conceptually able to fulfill the required functionality. This is exemplified through the ports *GY* and *TF*. Convert requires that a physical effect has to be associated with one of these abstraction ports. This is the case for the physical effect Biot-Savart providing the port *GY*. So far, abstraction ports only consider concepts for the exchange of energy and are derived from bond graph elements. They are introduced in the following section along with their function-allocating ability.

Although abstraction ports apply to all energy domains, the examples in this thesis concentrate on the mechanical translational, mechanical rotational and electrical domains.

4.2 Bond graph elements – the foundation for abstraction port types

Modeling physical systems with bond graphs is based on a flow of energy. The energy flow between two elements has the physical dimension of power, being the product of the two variables effort e and flow f. The variables momentum p and displacement q are the time integrals of effort and flow. Each of these generalized (i. e. state) variables has a domain-specific significance, as depicted in Table 4-1.

The general relationships between the state variables can be illustrated using the tetrahedron of state (PAYNTER, 1961), Figure 4-3. The different mathematical operations describing the relationships between effort and flow variables constitute the different types of bond graph elements. These can either be elements of a single energy domain (one tetrahedron) or elements mapping potentially from one to another
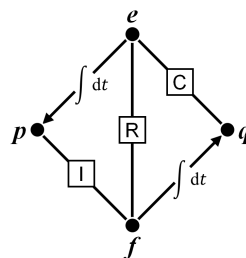

Figure 4-3: Tetrahedron of state, adapted from (PAYNTER, 1961)

Table 4-2: Bond graph element and functional operators

	Tetrahedron(s) of state	Equation schema	Functional operators	Assigned ports
Resistor	(diagram: e, f, p, q, R, C, I)	$e = \text{k} \cdot f$	Decrement, Prevent, Inhibit	R
			Decrease	R M
Capacitor	(diagram: e, f, p, q, R, C, I)	$e = \text{k} \cdot q$ or $e = \text{k} \cdot \int f \, \text{dt}$	Store, Supply	C
Inertia/ Inductance	(diagram: e, f, p, q, R, C, I)	$f = \text{k} \cdot p$ or $f = \text{k} \cdot \int e \, \text{dt}$	Store, Supply	I
Transformer	(diagram: p_1, q_1, f_1, p_2, q_2, f_2, TF, R, C, I)	$e_1 = \text{k} \cdot e_2$ and $f_2 = \text{k} \cdot f_1$	Increment, Decrement, Convert	TF
			Increase, Decrease, Convert	TF M
Gyrator	(diagram: e_1, e_2, p_1, q_2, f_1, f_2, GY, R, C, I)	$e_1 = \text{k} \cdot f_2$ and $f_1 = \text{k} \cdot e_2$	Convert	GY
			Increase, Decrease, Convert	GY M

domain (two tetrahedrons), cf. Table 4-2. Consequently, the equation schemes stand for the respective bond graph element types and allow for their identification. Primarily, the relation between the state variables – either integrative or derivative – is important. Hence, any other physical characteristics linking effort, flow and their respective time-integrals are lumped in the parameter k.

Only the bond graph elements that are suitable for the classification of physical effects from design catalogs and the assignment of abstraction ports are briefly presented below. Their ability to be allocated to functions is depicted as well. For a comprehensive introduction to physical systems modeling with bond graphs, refer to (BROENINK, 1999).

- *Resistor* (R): The resistor represents an irreversible dissipation of power. Normally, power is dissipated to the surroundings in the form of heat. The relationship between effort and flow is direct and linear, i. e. the effort is proportional to the flow. Examples are dampers, frictions and electric resistors.

- *Capacitor* (C): A capacitor represents the storage of energy as a displacement, being the time integral of the flow. The *C*-element is energy conservative. This means that in an ideal *C*-element,

energy can be stored and subsequently released in exactly the same amount. Examples are springs and capacitors.

- *Inertia/Inductance* (*I*): *I*-elements represent inertia in mechanical systems and inductance in electrical systems. This element represents the storage of energy as momentum. Like *C*-elements, *I*-elements are energy conservative too. Examples are masses, inertias and inductors.

- *Transformer* (*TF*): Transformers describe a relationship (based on the modulus contained in *k*) between two efforts or two flows. These can be within the same domain, or of two different domains and are represented with two tetrahedrons of state. The efforts are proportional to each other, as are the flows. The transformation can be within one domain, e. g. lever, or between two domains, e. g. winch.

- *Gyrator* (*GY*): While transformers describe a pair-wise relation of efforts and flows, gyrators relate an effort on the one side (left tetrahedron) to the flow on the other side (right tetrahedron) and vice versa, based on the gyrator ratio contained in *k*. Most of the realizations of gyrator effects represent a domain transformation, such as an electric motor or generator.

4.3 Assignment of abstraction ports to functions

The assignment of flow ports is in line with the functional modeling paradigms (Pahl et al., 2007; Ehrlenspiel, 2009; Ponn & Lindemann, 2011) and represent a flow of energy, material or signal as detailed earlier (Helms et al., 2009). The assignment of abstraction ports, in contrast, is based on functional operators. The procedure for the assignment of abstraction ports is to first determine all unsuitable functional operator classes and all of their subtypes for one of the following reasons:

- The primary classes *Signal* and *Support* do not represent a flow of energy.

- Although *Branch-*, *Channel-* and *Channel*-based functions can be applied on energetic flows, the exchange of energy is a subordinate characteristic. These functions are primarily described with their flow port characteristics.

- The secondary level operator *Actuate* can be used to model the commencement of a flow. It does not represent an exchange of energy but rather the act of switching a flow on or off.

- The tertiary operators *Shape* and *Condition* represent indeed an exchange of energy. However, due to their potential complexity, the embodiment requires a network of bond graph elements and is therefore out of scope and should be considered in future work.

For the remaining operators of the Functional Basis (Hirtz et al., 2002b), it was possible to manually allocate the appropriate bond graph element types, as depicted in the last column of Table 4-2. For example, the function *Decrement* is described as: *"To reduce a flow in a predetermined and fixed manner"*. The nature of bond graph *R*- and *TF*-element types, allows them to reduce the magnitude of an energy flow. By choosing the magnitude of the R-element's constant, *k*, or the modulus of the TF-element, the reduction of the flow can be set.

Some functions require the presence of a modulating signal port, e. g. *Increase: "To enlarge a flow in response to a control signal"*. This aspect presents an additional requirement on the physical effect for the embodiment and results in the additional assignment of an M port. A special case in that context is

the operator *Convert: "To change from one form of a flow [. . .] to another"*. There is no specification whether the energy conversion has to be embodied in a fixed manner or whether the ratio effort/flow can be modulated with a signal input. Hence, in terms of the abstraction portsassignment, two versions are introduced: One with and one without an M port, see Table 4-2. Finally, the operator *Convert* can be embodied based either on *GY*- or on *TF*-elements. Hence, both abstraction ports are assigned.
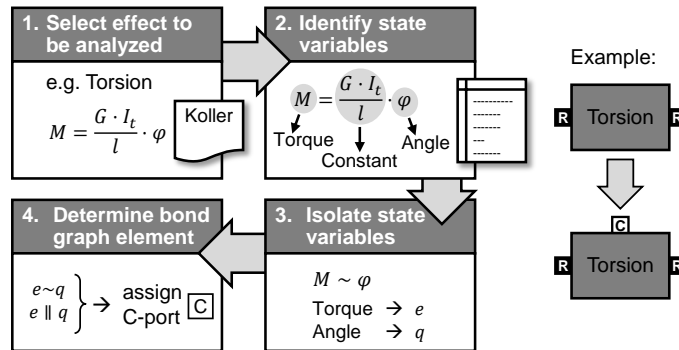

Figure 4-4: Example assignment of a *C*-PORT

4.4 Assignment of abstraction ports to physical effects

4.4.1 General approach

The assignment of abstraction ports to physical effects using bond graph elements is based on the manner in which the occurring state variables in the given formula are combined. This means that in each physical effect formula, only the specific state variables are relevant for the allocation of an abstraction port. Using the physical effect Torsion as an example, 4-4 shows the proposed approach for assigning the formulas of elementary physical effects to a corresponding bond graph element. The approach includes the following steps:

1. Select the physical effect to which abstraction ports are to be added from a design catalog, here: (KOLLER, 1994).

2. Identify state variables in the equation. In the case of Torsion, the variables torque and angle can be identified. The different uses of physical variables are matched by a conversion table.

3. Isolate state variables; other parts of the formula irrelevant for the allocation process are ignored.

4. Match bond graph element based on the nature of the equation format of the identified state variables and assign abstraction port.

By considering which of the state variables are present in a physical effect formula and by which equation scheme they are related, it is possible to allocate the physical effect to the corresponding bond graph element. This last step of identifying the bond graph element, i. e. the assignment of a abstraction port, is schematically depicted in Figure 4-5. The special case of three state variables is considered in the following subsection. In the next step, the two state variables are compared to all the possible combinations defining a bond graph element type. In the case that these variables have a parallel orientation – which is only the case if they originate from the same energy domains – a direct allocation of the ports R, C and I
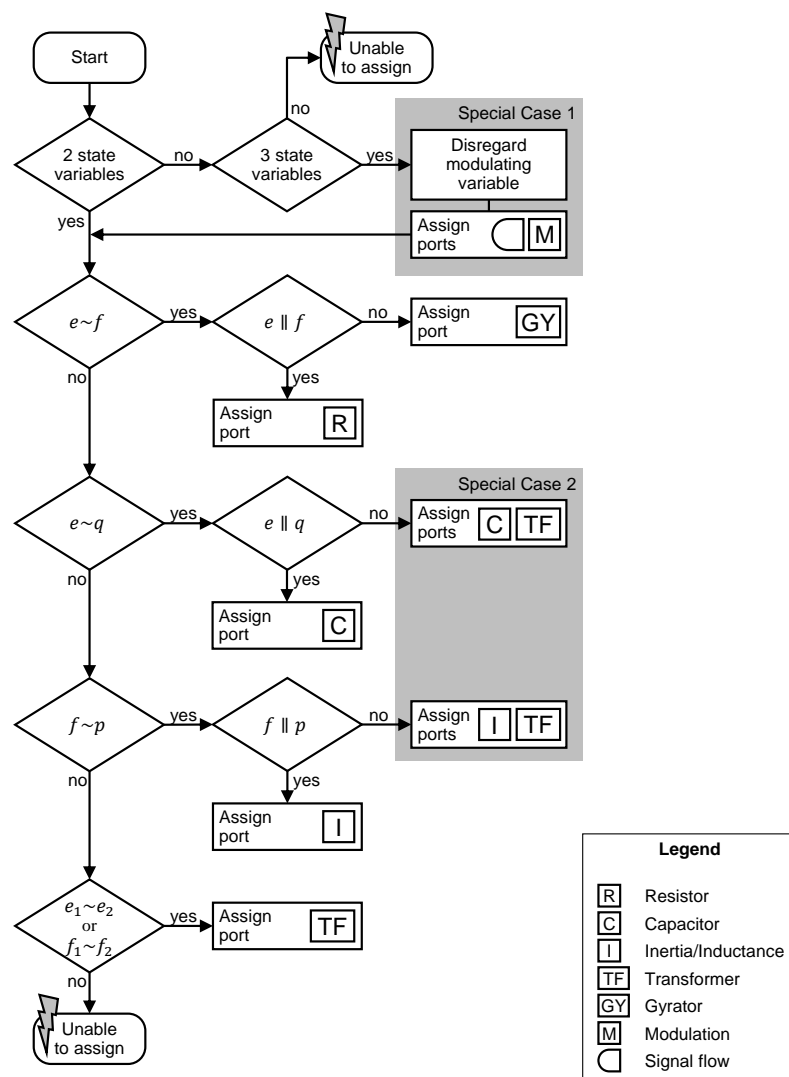
Figure 4-5: Process for assigning abstraction ports

is possible. Parallel in this context means that effort and flow act in parallel directions, such as force and speed, as time derivative of the displacement, in a spring. Non-parallel effort-flow relations result in the assignment of the *GY* port. Non-parallel relations of either effort-displacement or flow-momentum are combinations of two bond-graph characteristics and are considered as the second special case, cf. Section 4.4.3. A TF port can be directly assigned in the case that an effort relates to another effort variable or, vice-versa with two flow variables.

This assignment procedure is exemplified with the physical effect of Biot-Savart. This physical effect is represented in the design catalog with the equation $F = I \cdot l \cdot B$. From this equation, the variables F (mechanical translational force) and I (electrical current) are relevant for allocating the physical effect to a bond graph element while l and B are disregarded. The state variables F and I represent a proportional relationship between an effort and a flow from two different domains ($d1$: mechanical translational and $d2$: electrical): $e_d1 \sim f_d2$. Considering the law of Biot-Savart in greater detail, it turns out that F and I act in directions that are not parallel ($\nparallel$). The parallelism of effort and flow are a prerequisite for the assignment of *R*, I, and *C* ports. Consequently, a *GY* and not an *R* port is assigned.

Some of the functional operators in Table 4-2, e. g. *Increase*, *Decrease*, require that an input signal allows to modulate the intensity of the effect. The identification of suitable effects is considered in the following section.

4.4.2 Special case 1: Modulated elements

According to Rosenberg & Karnopp (1983), modulated elements include a coupling for a further input signal that varies the modulus of the *TF* element, the gyrator ratio of the *GY* element or the resistance of an *R* element. In bond graph methodology, these elements are denoted as *MTF*, *MGY* and *MR*. According to the notion of abstraction ports, the additional *M* port is introduced. Further, these elements are equipped with an additional signal port modeling the input signal for the modulation. To further comply with the assignment procedure seen in Fig 4-5, a decision point is introduced where the modulated state variable is eliminated. This is exemplified with the physical effect Coriolis force in Figure 4-6.
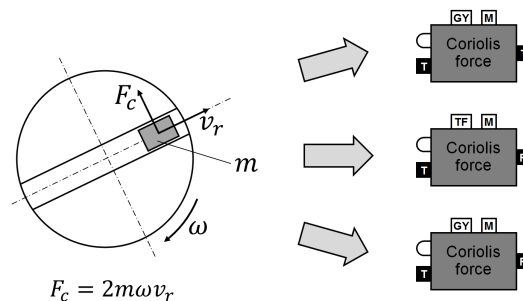

$$F_c = 2m\omega v_r$$

Figure 4-6: Schematic drawing of the Coriolis force, adapted from (Koller, 1994), and alternatives for assignment of abstraction ports

In the case of the physical effect of Coriolis force, a *GY*-element in the mechanical translational domain ($F_c \sim v_r$) can be identified. ω can be regarded as the input signal that varies the modulus of the element, cf. Table 4-2. Alternatively, the Coriolis force could be interpreted as a *TF*-element between the mechanical translational (v_r) and the mechanical rotational (ω) domains with the force F_c considered as being the modulating element. Lastly, the Coriolis force could be looked at as a being a *GY*-element between the mechanical translational (v_r) and the mechanical rotational (ω) domains with F_c as the modulating element. The physical effect Coriolis force can therefore be mapped to three specific bond graph elements. This means that three separate physical effects could be noted for the Coriolis force, each one with a different state variable as the modulating variable. This may result in each effect being assigned a different abstraction port depending on the physical domain and the nature of the two state variables that portray the constitutive law for the particular variation of the physical effect. However, noting three separate effects may, in the case of some modulated physical effects, not always result in three technologically useful effects. Nonetheless, their suggestion to the designer during the engineering process may expand the solution space and possibly lead to an innovative solution. This also demonstrates the need for the additional abstraction ports.

4.4.3 Special case 2: Combined elements

A further special case that needs to be taken into account is the fact that the equations of certain physical effects are a combination of several bond graph elements. So far, only the combination of two bond graph elements within one equation of a physical system is considered. Consequently, these effects do

not correspond to any specific equation scheme. An example of such a case can be found in the formula for the piezo effect, Figure 4-7:

$$x = d \cdot U \tag{4.1}$$

Equation 4.1 shows a direct relationship between a displacement x of the mechanical translational domain and an effort U of the electrical domain through the piezoelectric coefficient d and can be considered as $e_{d2} \sim q_{d1}$. A constitutive law linking displacement q and effort e describes a C element, however, only if both e and q belong to the same domain. This relationship can therefore not be directly mapped to one single bond graph element. Another way to describe the piezo effect in terms of bond graph elements is needed. With the help of Hooke's law

$$F = c \cdot x \tag{4.2}$$

the effort (F) can be related to the displacement (x) in the mechanical translational domain using the spring constant c. The piezo effect can hence be described as follows:

$$F = c \cdot d \cdot U \tag{4.3}$$

While Equation 4.2 describes a C element in the mechanical translational domain, Equation 4.3 describes a TF-element that links the effort of domain 1 (F) to the effort of domain 2 (U).

The piezo effect can therefore be described by a combination of a C-element that relates the displacement to effort within the same domain and by a TF-element that relates the effort of one domain to the effort of another and can be thought of as a trans-domain storage element. Hence, this is an example for the displacement in one domain being stored as energy in another domain. Consequently, two abstraction ports can be assigned: C and TF. Analogously, there are physical effects, e. g. Coulomb's force that can have both I- and TF-element characteristics and are also equipped with two abstraction ports: I and TF.
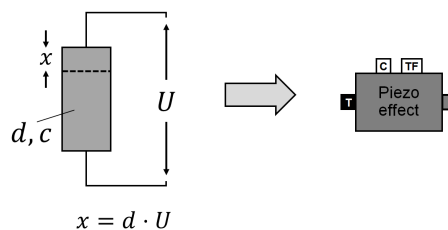

Figure 4-7: Schematic drawing of the piezo effect, adapted from (KOLLER, 1994), and assignment of abstraction ports

4.5 Validation: Formalization of design catalogs

This approach is validated by assigning abstraction ports to the physical effects of the design catalogs by KOLLER & KASTRUP (1998) and PONN & LINDEMANN (2011), Table 4-3. These catalogs were selected because of their clarity, systematic structure and the availability of equation-based descriptions of physical effects. They differ in that they contain different physical effects and use a different notation for the physical variables, which is taken into account using a conversion table as illustrated in Figure 4-4.

The results of the assignment of abstraction ports are depicted in Table 4-3. For both catalogs, all effects representing an exchange of energy that is described by an equation (60.0% and 80.5%) are suitable for the allocation of abstraction ports as detailed in Section 4.4.1. Hence, unsuitable effects (40.0% and 19.5%) can immediately be identified and do not have to be analyzed according to the assignment process in Figure 4-5. However, only effects from the energy domains mechanical translational, mechanical rotational and electrical are considered. Effects from other domains (26.2% and 29.9%), e. g. hydraulic, thermodynamic, magnetic, can also be allocated to corresponding abstraction ports but are not explicitly considered in this research. Although a high number of effects could be allocated immediately (18.2% and 28.7%), the extension towards the consideration of the special cases (15.6% and 21.8%) provides a valuable supplement.

Table 4-3: Result of assignment of abstraction ports to physical effects from design catalogs

		Number of physical effects			
		Koller 1994		Ponn & Lindemann 2011	
Suitable for allocation	Immediate allocation	50 (18.2%)		25 (28.7%)	
	Special cases	43 (15.6%)	60.0%	19 (21.8%)	80.5%
Different physical domain but suitable for allocation		72 (26.2%)		26 (29.9%)	
Unsuitable for allocation	No exchange of energy	57 (20.7%)	40.0%	13 (14.9%)	19.5%
	No equation	53 (19.3%)		4 (4.6%)	
		Total: 275		Total: 87	

An example allocation of physical effects to functions is shown in Table 4-5. Based on the assignment of abstraction ports to functions, Table 4-2, and to physical effects (using the process shown in Figure 4-5), a mapping between them is represented. Based on this, suitable physical effects for the realization of functions can be identified. All of the three functions can be carried out with two different abstraction ports, e. g. in the case of the function Convert mechanical translational energy to electrical energy, a physical effect with either a *GY* or a *TF* port is required. While the first four effects (Electrostatic force – Induction) solely contain these required ports, the last three bring in the additional characteristics of capacitive bond graph elements (*C* port). Hence, this additional port gives an indication what unintended behavior should be considered when assigning the piezo effect to this function, for example. Alternatively, the additional (*C*) port characteristic could be used to realize a functional operator that requires a capacitive behavior, e. g. Supply.

The process presented for assigning abstraction ports is well suited for a computational implementation. The architecture of a software prototype that validates the automated assignment of physical effects to functions based on abstraction ports is described below.

The program is divided into two main parts and is schematically illustrated in Figure 4-8. The first part

handles the assignment of physical effects to the corresponding bond graph elements, i. e. abstraction ports. The second part assists the designer during the concretization process by providing a search for suitable physical effects to fulfill the required functions. Details of these parts are presented in the following subsections. The source code of all program parts is written in the Python programming language.
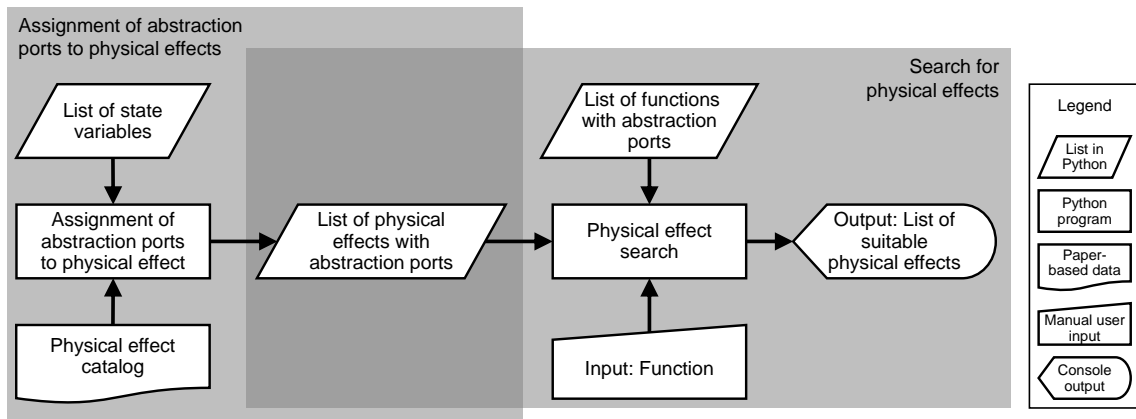

Figure 4-8: Schematic outline of the individual program parts implementing the process for assigning abstraction ports

4.5.1 Assignment of physical effects to bond graph elements

The first part of the program handles the assignment of physical effects from different, heterogeneous physical effect collections, e. g. (KOLLER & KASTRUP, 1998; ROTH, 2001; PONN & LINDEMANN, 2011), to corresponding bond graph elements, as previously depicted in Figure 4-4.

Four individual software components are required. First, the *Assignment of abstraction ports to physical effects* that implements the core method presented in Section 4.4. To assign an abstraction port to a physical effect, this component requires two pieces of information:

- The *List of state variables* that contains all state variables and relates them to their respective energy domains and formula symbols, cf. Table 4-1.

- The *Physical effect catalog* is the source of physical effects. Physical effects are characterized by means of their name and equations, whereas occurring state variables are defined according to the formula symbols defined in the list of state variables. Ultimately, these effects are derived from effect catalogs.

If a new physical effect is to be registered, the user has to enter each symbol in the physical effect formula one at a time. Next, the equation symbols are compared to the domain specific state variables stored in the *List of state variables*. The occurring state variables in the equation can thus be identified and isolated. Next, the relationship – or constitutive law – between the identified state variables is determined based on the allocation algorithm in the source code of the program part *Assignment of abstraction ports to physical effects* and the domain-specific information of the identified state variables.

Following a successful assignment, the effect, including its domain specific information, is stored in the *List of physical effects with abstraction ports*.

For example, the physical effect Biot-Savart from the design catalog by Ponn & Lindemann (2011) would be assigned to a bond graph *GY*-element that acts between the mechanical translational and the electrical energy domains. The entry for this effect in the *List of physical effects with abstraction ports* looks like this:

```
['63', 'Biot-Savart', 'GY', 'mechanical-translational', 'electrical']
```

63 is the ID of the physical effect *Biot-Savart* having a *GY* abstraction port and the two flow ports *mechanical-translational* and *electrical*.

4.5.2 Search for suitable physical effects for allocation to functions

The second part of the program (see Figure 4-8) uses the *List of physical effects with abstraction ports* and makes them available to be assigned to functions. The component *Physical effect search* is the core of this part as it allows the user to select the function for which suitable physical effects are required. Further, the user has the ability to constrain the search through required or undesired abstraction ports but also through required or undesired energy domains.

In order for the component *Physical effect search* to perform such a task, it requires the *List of physical effects with abstraction ports* as well as the *List of functions with abstraction ports* as input. The former list contains the assignment described in the previous section. The latter is incorporated into the source code of the *Physical effect search* and contains the content presented in Table 4-2. Table 4-4 represents the search results for given functional operators. The number of suitable effects for every operator is indicated in the second row. This number is split up according to the distribution of abstraction ports and further split up into the involved energy domain that are represented by the flow ports.

4.6 Discussion

A large amount of engineering design knowledge is contained in the physical effect collections of paper-based design catalogs. This research contributes to advances in the area of computational design synthesis by introducing an approach that allows this knowledge to be made available to computational synthesis methods. The heterogeneity of the knowledge sources requires a uniform formalization layer to enable physical effects to be formulated in a single standard manner and to be able to integrate with a design representation, such as FBS*.
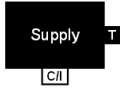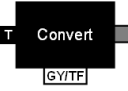
The approach presented was inspired by the bond graph theory providing a set of abstract, physical modeling elements and led to the introduction of the notion of abstraction ports as a classification scheme for physical effects. Hence, abstraction ports provide a stepping stone in the concretization process between the levels of abstraction Function and Behavior. The approach concentrates on physical effects that are based on a flow of energy. Only these can be modeled using bond graph elements. They were analyzed according to their ability to be allocated to functional operators of the Reconciled Functional Basis. Further, the equation schemes of the bond graph elements were defined, as they serve as a blueprint on which abstraction ports can be assigned to the equations of physical effects. The process for assigning ports is elaborated in an algorithmic manner and has been successfully validated using the formalization of two design catalogs.

Table 4-4: Physical effects found for given functional operators, sorted according to their abstraction port and flow port

Functional operator	Effects found	Abstraction port		Flow ports	
				Search results	
			1	electrical	electrical
			1	electrical	mechanical-rotational
		12 GY	2	electrical	mechanical-translational
			1	mechanical-translational	electrical
			3	mechanical-translational	mechanical-rotational
Convert	38		4	mechanical-translational	mechanical-translational
			5	electrical	electrical
			1	electrical	mechanical-translational
		26 TF	1	mechanical-rotational	mechanical-translational
			3	mechanical-translational	electrical
			1	mechanical-translational	mechanical-rotational
			15	mechanical-translational	mechanical-translational
			1	electrical	electrical
			1	electrical	mechanical-rotational
		12 GY	2	electrical	mechanical-translational
			1	mechanical-translational	electrical
			3	mechanical-translational	mechanical-rotational
			4	mechanical-translational	mechanical-translational
Decrease	49	9 R	5	electrical	electrical
			4	mechanical-translational	mechanical-translational
			5	electrical	electrical
			1	electrical	mechanical-translational
		28 TF	1	mechanical-rotational	mechanical-translational
			3	mechanical-translational	electrical
			1	mechanical-translational	mechanical-rotational
			17	mechanical-translational	mechanical-translational
			8	electrical	electrical
		14 R	1	mechanical-rotational	mechanical-rotational
			5	mechanical-translational	mechanical-translational
			5	electrical	electrical
Decrement	39		1	electrical	mechanical-translational
		25 TF	1	mechanical-rotational	mechanical-translational
			3	mechanical-translational	electrical
			1	mechanical-translational	mechanical-rotational
			14	mechanical-translational	mechanical-translational
			1	electrical	electrical
			1	electrical	mechanical-rotational
		12 GY	2	electrical	mechanical-translational
			1	mechanical-translational	electrical
			3	mechanical-translational	mechanical-rotational
			4	mechanical-translational	mechanical-translational
Increase	46		3	electrical	electrical
		7 R	1	mechanical-rotational	mechanical-rotational
			3	mechanical-translational	mechanical-translational
			6	electrical	electrical
			1	mechanical-rotational	mechanical-translational
		27 TF	3	mechanical-translational	electrical
			1	mechanical-translational	mechanical-rotational
			16	mechanical-translational	mechanical-translational
			5	electrical	electrical
			1	electrical	mechanical-translational
Increment	25	25 TF	1	mechanical-rotational	mechanical-translational
			3	mechanical-translational	electrical
			1	mechanical-translational	mechanical-rotational
			14	mechanical-translational	mechanical-translational
			8	electrical	electrical
Inhibit	14	14 R	1	mechanical-rotational	mechanical-rotational
			5	mechanical-translational	mechanical-translational
			8	electrical	electrical
Prevent	14	14 R	1	mechanical-rotational	mechanical-rotational
			5	mechanical-translational	mechanical-translational
			4	electrical	electrical
		16 C	1	mechanical-rotational	mechanical-rotational
Store	24		11	mechanical-translational	mechanical-translational
		8 I	3	mechanical-rotational	mechanical-rotational
			5	mechanical-translational	mechanical-translational
			4	electrical	electrical
		16 C	1	mechanical-rotational	mechanical-rotational
Supply	23		11	mechanical-translational	mechanical-translational
		7 I	2	mechanical-rotational	mechanical-rotational
			5	mechanical-translational	mechanical-translational

Table 4-5: Example mapping of functions to physical effects

Function	Physical effect	Assigned ports
Supply (C/I)	Newtons second law	I
	Gravitation	I
	Elastic strain	C
	Elastic inflection	C
	Shear	C
	Lateral contraction	C
	Buoyancy	C
	Thermal expansion	C
Convert (GY/TF)	Electrostatic force	TF
	Electric charge	GY
	Biot-Savart	GY
	Induction	GY
	Piezo effect	TF C
	Coulomb's force	TF C
	Charge in electric field	TF C
Decrement (R/TF)	Ohm's law	R
	Hall effect	R
	Vacuum discharge	R
	Amplifier	R
	Induction	TF
	Collision ionisation	TF
	Transductor	TF
	Secondary electron amplifier	TF

The advantage of the described method lies in the fact that it is not necessary to consider the structure of an equation. This means that simply the occurrence of state variables in the formula that describe a physical effect is sufficient for correctly allocating abstraction ports; their position within the equation is not important. This approach bypasses the problem arising through formulas for certain physical effects being structured differently in different design catalogs. The problem that different design catalogs might use different equation symbols to represent a certain variable can be overcome by using a conversion table to translate the given equation symbols to equation symbols in a uniform set. Further, adding new effects to the knowledge repository, which might arise due to technological advances, is not an issue as long as the effect is based on an exchange of energy and its equation is composed of known variables. Hence, with the achievement of the Expected Contribution 3 (formalization of design catalogs), advances in terms of the Research Goals 1 (increase effectiveness and efficiency of knowledge formalization) and 3 (formalization of existing design knowledge resources) are delivered. Beyond that, the approach is not bound to a physical domain as the underlying bond graph methodology is a domain-independent approach for modeling physical systems. Hence, it is fair to assume that a successful mapping for the three domains being considered is transferable to the remaining ones.

The approach classifies physical effects according to their ability to be allocated to functional operators by introducing abstraction ports valuable especially for computational design synthesis approaches. The synthesis method presented in Chapter 3 concentrated on the allocation of functions to physical effects based on input and output flow ports. Solutions were generated as long as the set of available functions and physical effects was small enough, meaningful, yet surprising. However, scalability in terms of integrating new elements was not sufficiently supported due to the combinatorial expansion of the solution space. The introduction of abstraction ports, in addition to flow ports, increases efficiency when searching for suitable physical effects for the concretization of functions, cf. Table 4-5. It provides, furthermore, for the identification of unintended physical effects and gives indications about how physical

effects can be associated to multiple functions in the case that they have multiple abstraction ports. An example for this is the piezo effect as depicted in Figure 4-9. Previously, the mapping to functions was solely based on matching input and output flow ports and only captured the effect's capability to convert mechanical-translational energy to electrical energy. Now, the additional capacitive nature of the effect can be represented as well and leads to assignment of a *C*-port. Thereby, this effect is qualified to be allocated to the additional function *Store mechanical mechanical-translational energy*, although it is also valid to map only the *Convert* function. Graph grammar rules, similar to those depicted in Table 3-1, could perform this task automatically. As such rules could be based on port matching, the definition of generic rules is feasible that does not depend on the actual elements in the metamodel but on the set of abstraction ports. Unintended effects, such as friction being necessarily associated with wheels, were previously hardcoded in the metamodel in the *realizes*-variable of components as depicted in Figure 3-6. Using abstraction ports, those unintended aspects of a physical effect become apparent simply through detecting unconnected abstraction ports.
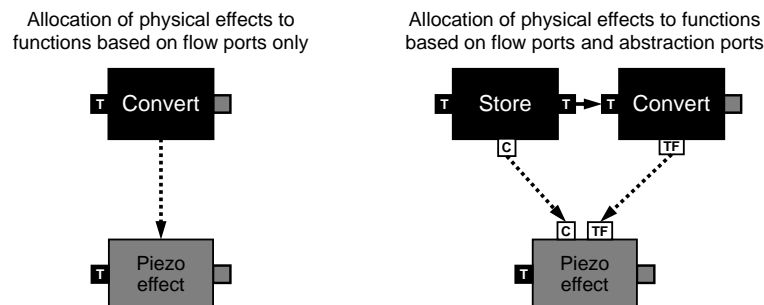

Figure 4-9: Example allocation of piezo effect to functions

The algorithmic nature of the assignment process is suited for computational implementation. This is shown with a software prototype consisting of two main components. In the first part, the automated assignment of abstraction ports to physical effects is carried out. This builds the basis for the second component that searches for suitable physical effects for functions. So far, this prototype proposes suitable physical effects for a given function to the human designer.

Integration with the object-oriented graph grammar implementation for computational design synthesis (called booggie) will support automated search for alternatives for allocating physical effects to functions.

Further, the concepts of metamodeling support the definition of abstraction and flow ports and provides for the formalization of design rules. The identification of abstraction ports out of the equation of physical effects should be implemented as a plugin to booggie.

5 Development of the software prototype booggie

The computational design synthesis approach that is based on object-oriented graph grammars has been implemented in the open-source software prototype *booggie*[30]. This chapter starts with a presentation of the software requirements that need to be fulfilled in the implementation of object-oriented graph grammars. In an abstract sense, booggie is a tool to define domain-specific modeling languages based on which grammar rules automate the generation of graph-based models. For that reason, the formal foundations for defining and using modeling languages in booggie are presented in Section 5.2. Thereafter, the implementation of the requirements in a software prototype is, in the first step, portrayed from a global perspective detailing the overall software architecture and the applied implementation paradigms. Afterwards, the implementation of selected components is described. The usability of the implementation in an industrial context is illustrated and validated in a study that targets the automated configuration of aircraft cabin layouts. This chapter closes with a discussion and opportunities for future work.

5.1 Software requirements

Software requirements can be differentiated between functional and non-functional requirements (Bruegge & Dutoit, 2010, p. 159ff). While the first category describes the interactions between a piece of software and its environment (here: the user), the latter captures aspects regardless of the required functionality and primarily addresses issues related to the software quality in general.

In the following, a condensed overview of the most important requirements is given that have to be fulfilled to achieve the Expected Contributions 5 and 6 and to illustrate, or validate, the achievement of the other expected contributions. In the upcoming sections 5.3 and 5.4 reference is made to these software requirements (abbreviated as SWReq).

5.1.1 Functional requirements

First, the software prototype has to support the entire knowledge formalization process as presented in Section 3.2. This leads to the definition of five fundamental software components as depicted in Figure 5-1. The nature of object-orientation requires the separation into a definition and an application part that are respectively concluded either through grammar compilation or graph transformation, represented as black boxes in Figure 5-1. Regarding the overall knowledge formalization process, the following requirements are essential:

Software Requirement 1. Provide a solution-independent implementation supporting the knowledge formalization process as depicted in Figure 3-2.

Software Requirement 2. Develop user interfaces, termed *perspectives*, to guide the user through the knowledge formalization process. For these, intuitive interpreters – or at least the foundation for future development of intuitive interpreters – have to be implemented. Emphasis is to be put on the ease of

---

[30]**b**rings **o**bject-**o**riented **g**raph **g**rammars in to **e**ngineering

use of the Metamodel perspective and a visual rule interpreter; these are the main input channels for formalizing model-based and rule-based engineering knowledge.
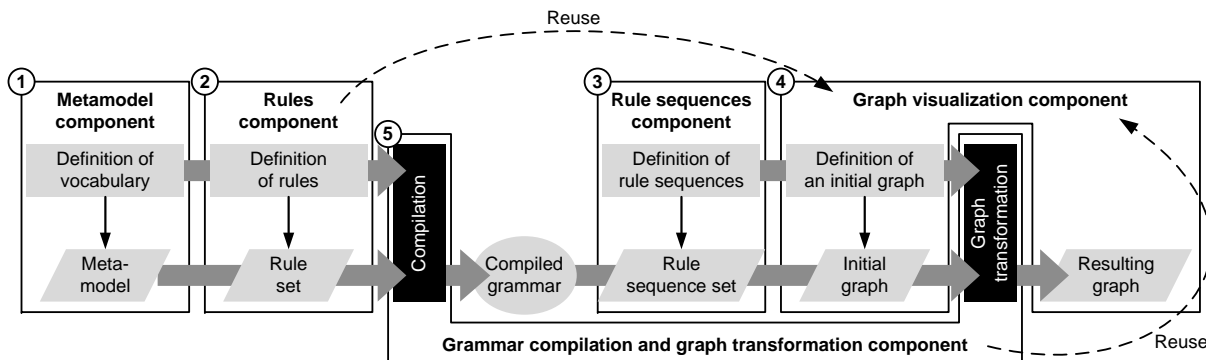

Figure 5-1: Required software components according to the knowledge formalization process, cf. Figure 3-2

The core idea of this research is the combination of model-based and rule-based knowledge formalization in an integrated knowledge representation that is reflected by the knowledge formalization process and the respective software components (Figure 5-1). The requirements regarding the expressiveness of the model-based knowledge representation all refer to the definition of metamodels and can be subsumed as follows:

Software Requirement 3. The metamodel has to support the definition of modeling elements and their respective attributes. To support an efficient definition of the modeling domain in a hierarchical manner, inheritance and the definition of abstract element types are required.

Software Requirement 4. Enable the representation of topological constraints between element types, i. e. ports. Similar to the element type hierarchy, a port type hierarchy should support inheritance and abstract port types.

Software Requirement 5. Provide for the definition of the visual appearance of element types in the metamodel such that it can be accessed during the graph transformation, e. g. for layouting or arranging elements.

As counterpart to the model-based knowledge representation, the implementation of the rule-based knowledge representation has to consider these points:

Software Requirement 6. Two classes of procedural statements are required:

- *Graph transformation rules*, i. e. production rules or simply rules in short, encapsulate operations that represent modifications of a graph-based model.

- In *scripts*, a programming language can be used to define operations in a programming-like manner, e. g. for invoking the analysis of the current models with external simulation tools.

Software Requirement 7. The expressiveness for the definition of graph transformation rules should cover rule parameters, return values and rule constraints, as detailed in Section 3.2.1.

Software Requirement 8. The definition of generic graph transformation rules requires that abstract element and port types can be used in the rule definition.

Software Requirement 9. The definition of rule sequences should allow the concatenation of graph transformation rules and scripts to complex, logical structures. While performing the transformation, the intertwined execution of these two rule types has to be provided.

Targeted at applications in engineering design, the visual representation of generated models is crucial for the user to get an impression of the generated solution and to gain confidence in the computational results. Particularly, these two requirements are to be considered:

Software Requirement 10. A 3D-visualization of the graph-based models is required that enables, for example, the representation of the preliminary geometry of products or the visual partitioning into different levels of abstraction.

Software Requirement 11. Generated graphs should not be just static images of the model but rather allow an interaction with the user, e. g. for intuitively adding or deleting modeling elements.

5.1.2 Non-functional requirements

The following non-functional requirements refer to the implementation at large:

Software Requirement 12. The implementation should be platform independent such that potential applications and future research is not restricted by the selected operating system.

Software Requirement 13. A modular and expandable software architecture is required such that the implementation can be easily extended.

Software Requirement 14. Reusing mature open-source software should allow for a high overall software maturity.

In general, non-functional requirements address issues such as usability, reliability, performance and supportability (BRUEGGE & DUTOIT, 2010, p. 160). Although these aspects were taken into account for the implementation, their scientific significance is considered low and are hence not further discussed.

5.2 Formal definition of the booggie modeling language (bgML)

To define a modeling language, another language is needed. This language, called the metamodeling language, requires either another layer describing its elements or it is reflective, i. e. the modeling elements of the layer are used to define the layer itself. The OBJECT MANAGEMENT GROUP (2007) introduced a four layered metamodel stack to define the Unified Modeling Language (UML) using the Meta-Object Facility (MOF). In this definition, M0 contains the runtime instantiations, i. e. real data, of its upper layer, M1, which contains the type definitions of the runtime instances. M2 contains the modeling elements for modeling M1 and so on. An illustrative example is depicted in Figure 5-2.

In this research, these four levels are required as well. Just like with UML, M0 contains real objects, which belong here to the engineering domain. These objects are not runtime instantiations but real existing things, e. g. constituents of the (future) product. A model of these objects is represented as a booggie graph in M1. Its modeling elements, i. e. ports and elements, are instantiated from the definition in M2. In Section 3, M2 is termed *metamodel*; one specific metamodel that defines the modeling elements for FBS*-based product models of hybrid powertrains is depicted in Figure 3-6. Within M3, it is specified
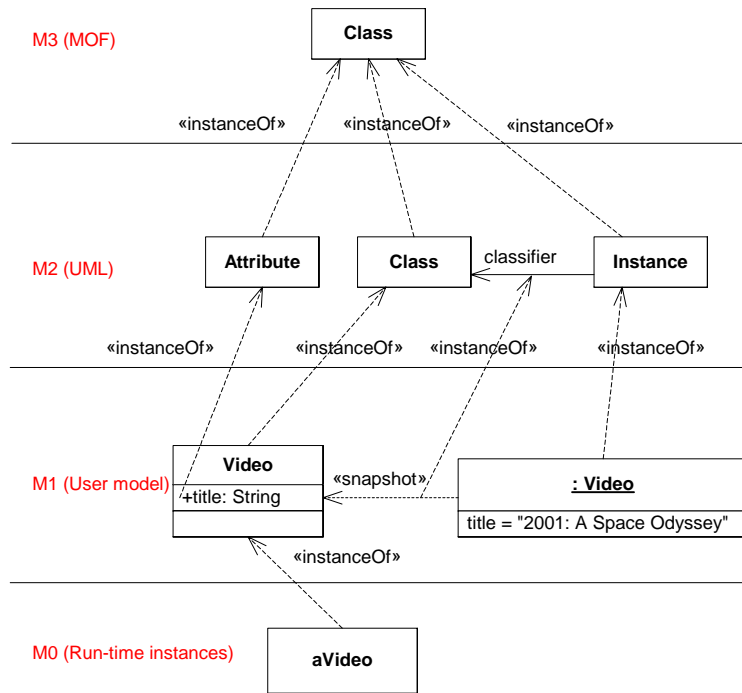
Figure 5-2: Example of the four-layer modeling language stack (OBJECT MANAGEMENT GROUP, 2007)

that element types and port types are available for representing M2 models. The modeling syntax for M3 is the graph model language in GrGen.NET (JAKUMEIT ET AL., 2010) that allows specifying typed and attributed multigraphs with multiple inheritance. In M3 the link is made between the metamodel of the hybrid knowledge representation (represented in Section 3) and the graph-based representation that is used for graph transformations in GrGen.NET. Hence, from a formal point of view, GrGen.NET's graph model language can be considered as M4. Figure 5-3 illustrates the interrelation between M0, M1, M2 and M3.

In this section, a formal definition is given of the booggie modeling language (bgML) spanning from M1 to M3. Since bgML is a formal modeling language, all domain-specific DSLs that are developed in booggie can be considered formal as well. The advantages of using formal modeling languages in terms of enabling model transformation address Research Issue 4 (lack of modeling standards and tool integration) leading to the Expected Contribution 4 (foundation for model transformation). In this section, a formal perspective is adopted to describe the language details of M1, M2 and M3 within booggie.

5.2.1 booggie graph (M1)

Any booggie graph is a specialization of a typed and attributed multigraph[31] with inheritance on node and edge types that is defined as the tuple $G = (N, E, I_N, I_E, A_N, A_E)$. N is the set of nodes and E is the set of edges. I_N and I_E are graphs describing the inheritance hierarchy of nodes and edges. They are defined in a metamodel. A_N and A_E are sets of attributes of the respective nodes and edges. How a booggie graph is derived from this definition is discussed in the following.

Ports are one of the main concepts of graphs in booggie. They enable the interconnection of elements, rather than connecting the elements themselves with edges. This allows for having defined interfaces

---

[31]A multigraph can have multiple edges between the same nodes. These edges are also called parallel edges.
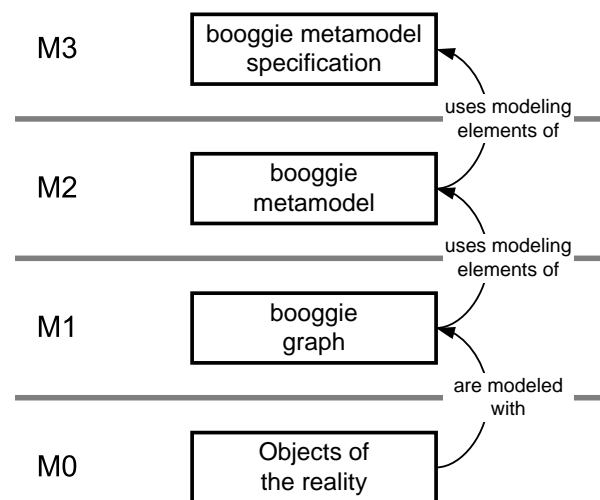
Figure 5-3: Four-layer modeling language stack of booggie

and is helpful when rules are formulated that match for ports, regardless of the element they are assigned to. 5-4 shows a booggie graph with two elements, connected with an edge that is attached to a port on each of its ends. It illustrates the fact that ports are not isolated nodes of the graph, but always have to be attached to an element. To introduce ports into the previous graph definition, the general set of nodes N is split up into two sets N_{El} and N_P, which contain the elements and ports. These two sets are disjoint from each other and together provide all the nodes of the graph such that $N = N_{El} \cup N_P$ and $N_{El} \cap N_P = \emptyset$. Accordingly, the set of node attributes A_N is split up into A_{El} and A_P. Attributes can only be applied to elements and ports. Every attribute belongs to exactly one item and is assigned to them by the function $attr : A \rightarrow N_{El} \cup N_P$.
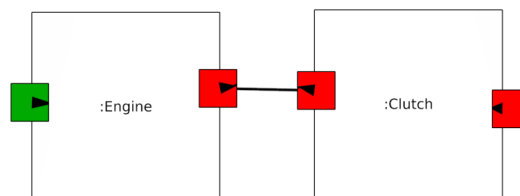

Figure 5-4: Example booggie graph: Ports are depicted by small squares attached to the two elements

An element in a booggie graph cannot be connected with any other element by an edge but provides a set of ports for this reason. This set can also be empty, which means that the element can never be connected with another element. It stays isolated as long as a rule adds a port to it that can be connected to another port. Each port belongs to exactly one element, i.e. $\forall x \in N_P \, \exists \, element(x) \in N_{El}$. This is expressed by the function $element : N_P \rightarrow N_{El}$, which maps a given port to the element it belongs to. One of the following three direction values $D = \{in, out, undirected\}$ is assigned to every port in N_P and defined by the following direction function: $dir : N_P \rightarrow D$. Thereby, it can be represented whether a port is considered as input, output or as unspecified in that respect.

booggie edges in E do not have attributes and are not defined in a hierarchy of inheritance. They are used to represent the links between ports. Through ports, the direction of the connection between two elements, e. g. the direction of an energy flow, is defined. However, the internal representation of edges requires the identification of their adjacent ports that are termed source and target but do not interfere

with the direction determined by the ports. Edges cannot be connected to any node but only to nodes in N_P, i.e. ports. For this reason, the functions s (returning the port at the source of an edge) and t (returning the port at the target of an edge) only map to ports, i.e. $s, t : E \rightarrow N_P$.

To form a valid booggie graph, all edges have to satisfy the following constraints:

- No dangling edges are allowed, i.e. $\forall e \in E \; \exists x, y \in N_P : \; s(e) = x \wedge t(e) = y$

- Any port may have at most one adjacent edge: $\forall n \in N_P : |\{e \in E : s(e) = n \vee t(e) = n\}| \leq 1$. This also implies that edges may not connect a flow port with itself. This way, a port can only be either in the state of being unconnected or of being connected to another port.[32]

- Two connected ports must have matching flow directions: ports with *out* direction may only connect to *in*-directed ports, whereas *undirected* ports may be connected to ports of any direction. Hence, $\forall e \in E : \; (dir(s(e)) = out \Rightarrow dir(t(e)) \in \{in, undirected\}) \wedge (dir(t(e)) = in \Rightarrow dir(t(e)) \in \{out, undirected\})$ must hold, i.e. if one of the ports is *out*- or *in*-directed, the direction of the edge is already determined. This constraint also implies that no two *out*- or two *in*-ports may be connected.

- The types of two connected ports have to match in such a way that they are either of the same type, or one type is the ancestor of the other one, i.e. $\forall e \in E : type(s(e)) \in clan(type(t(e))) \vee type(t(e)) \in clan(type(s(e)))$ (the function *clan* returns all ancestors of a type and is part of the metamodel definition).

Consequently, a booggie graph is defined as the tuple

$$G_{bg} = (N_{El}, N_P, E, I_{El}, I_P, A_{El}, A_P, s, t, dir, element).$$

For an overview of the nomenclature used in this section, refer to Appendix 9.5.

5.2.2 booggie metamodel (M2)

The booggie metamodel is used to define the vocabulary, i.e. element types and port types, together with its logical inheritance structure that can be instantiated in a booggie graph. As every typed graph, every booggie graph relates to a metamodel by means of its two inheritance graphs I_{El} and I_P that are defined as $I_{El} = (N_{ElT}, E_{ElT}, A_{ElT}, p)$ and $I_P = (N_{PT}, E_{PT}, A_{PT})$. The sets N_{ElT} and N_{PT} contain the element types and port types as nodes; the edges representing the inheritance dependency between them are represented in the sets E_{ElT} and E_{PT} as directed edges. The graph structure of I_{El} and I_P becomes apparent in the metamodel definition in Figure 3-3.

In contrast to the port type inheritance graph, I_{El} is extended by the function p that acts as a descriptor of which port types are associated to which element types: $p : V_E \rightarrow V_P \times D \times R$, where D is the set of the three possible directions of a port (in, out, undefined), and R is a range interval from zero or a positive integer to any greater or equal integer, i.e. $R \subseteq \mathbb{N}_0$. With this function p, every element type contains the information about how many and which ports an instantiated element in the graph may have. The range defines a lower bound of ports (of the associated port type with the specified direction) that an element

---

[32]For the sake of better readability, this constraint is violated in the depiction of the hybrid powertrain examples in Section 3.3.2. Hence, ports can be found that have multiple edges. Within booggie, this could not occur since missing ports are added automatically to achieve a valid graph.

must have and an upper bound of maximum ports of this sort that it may have. Every type, represented by a node in either of the two inheritance graphs, is described by a globally unique type name. This means that a bijective mapping, $typename : V_E \cup V_P \to N$, with N being a set of character strings containing all type names, exists.

In booggie, a mechanism is implemented that validates the edge constraints from the previous section and the multiplicity constraints in R. Elements and edges that violate against them are highlighted in red.

5.2.3 booggie metamodel specification (M3)

Within M3, the foundation is laid for defining I_{El} and I_P. This is accomplished by defining the two root nodes of the element type and port type hierarchy using the syntax of GrGen.NET's graph model language. As depicted in Figure 5-5, these root nodes are defined as the node types[33] *PortType* and *ElementType*. All modeling elements that are defined in metamodels, e. g. in Figure 3-6, inherit from these two types. Hence, all their descendants have the layout properties that are defined here. By default, elements are visualized as big white squares and ports as small white squares. These properties can be overwritten by their ancestors but can also be manipulated in rules as demanded in SWReq 5.
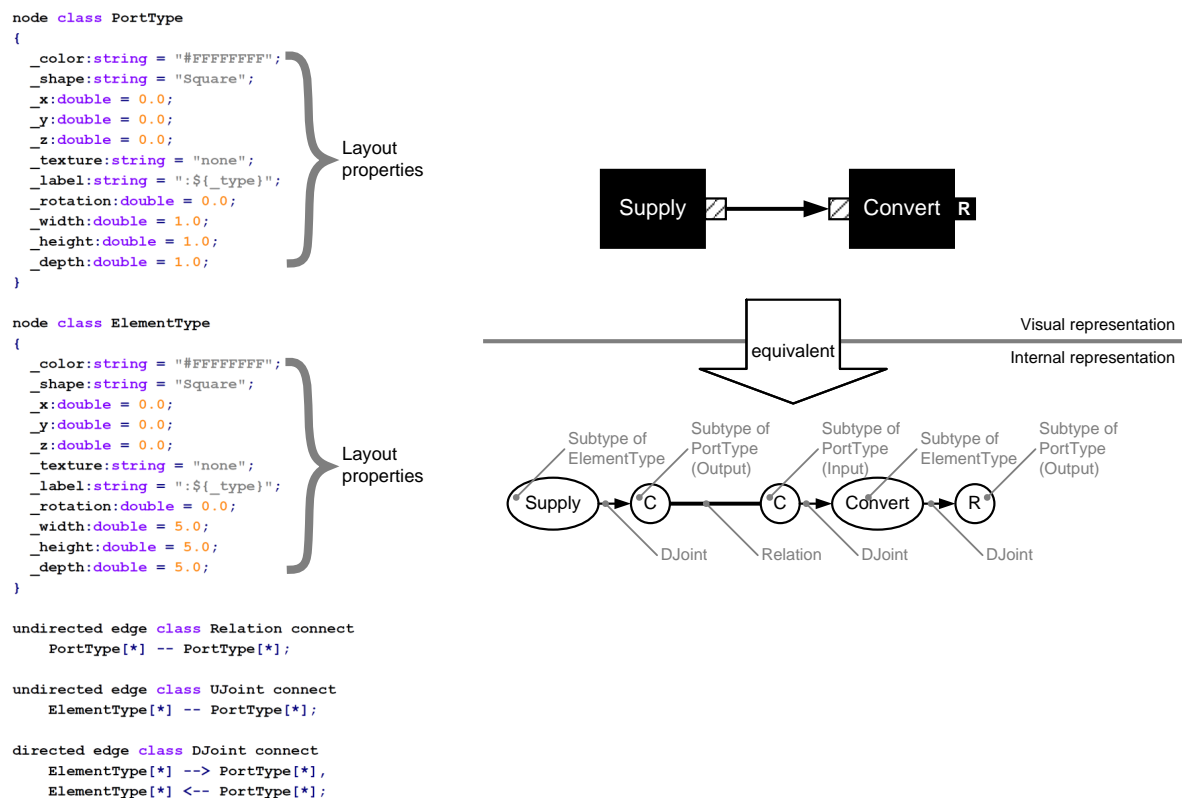


```
node class PortType
{
    _color:string = "#FFFFFFFF";
    _shape:string = "Square";
    _x:double = 0.0;
    _y:double = 0.0;
    _z:double = 0.0;
    _texture:string = "none";
    _label:string = ":${_type}";
    _rotation:double = 0.0;
    _width:double = 1.0;
    _height:double = 1.0;
    _depth:double = 1.0;
}

node class ElementType
{
    _color:string = "#FFFFFFFF";
    _shape:string = "Square";
    _x:double = 0.0;
    _y:double = 0.0;
    _z:double = 0.0;
    _texture:string = "none";
    _label:string = ":${_type}";
    _rotation:double = 0.0;
    _width:double = 5.0;
    _height:double = 5.0;
    _depth:double = 5.0;
}

undirected edge class Relation connect
    PortType[*] -- PortType[*];

undirected edge class UJoint connect
    ElementType[*] -- PortType[*];

directed edge class DJoint connect
    ElementType[*] --> PortType[*],
    ElementType[*] <-- PortType[*];
```

Figure 5-5: Metamodel specification (M3) using GrGen.NET's graph model language and example of the visual representation of a booggie graph and it's internal representation according to M3

Besides these basic node types for elements and ports, some edge types are defined here that are not further detailed in the preceding formal description of bgML. These have only little importance for

---

[33]Please note that GrGen.NET's graph model language adopts the term *class*. However, as defined on page 31, the term *type* is used here as it designates blueprints of instances in the context of modeling.

using bgML from a user perspective and are not further detailed in M2, i. e. these edge types have no descendants. Instances of the edge type *Relation* are used to create connections between ports. Edges of type *UJoint* link undirected ports to their elements and edges of type *DJoint* link directed ports, i. e. input and output ports, to their elements. *UJoint* and *DJoint* element types are only required for booggie's underlying representation such that the dependency between an element and its ports can be represented using GrGen.NET's graph representation. In the visual representation of a booggie graph, these edges have no significance for the user as they are not visualized.

The example in Figure 5-5 illustrates the use of these element and port types for representing a booggie graph. The element types *Supply* and *Convert* are defined in M2 in I_{El} (cf. metamodel in Figure 3-6), the two port types *chemical* and *mechanical rotational* are defined in M2 in I_P; (cf. metamodel in Figure 3-3); these are hence either subtypes of *ElementType* or of *PortType*. As all relations between elements are characterized by ports, there is no need to further specify edge types between them and instances of *Relation* are directly used in the booggie graph. *DJoint*-instances are used to connect directed ports to the elements, undirected ports (represented as *DJoint*-instances) are not used here.
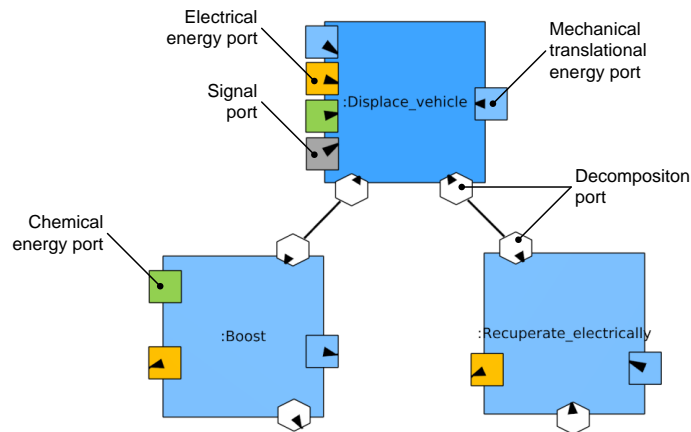
Figure 5-6: Overall function *Displace vehicle* decomposed into the high-level functions *Boost* and *Recuperate (energy) electrically* by means of decomposition ports

Contrary to this example, the metamodel represented in Figure 3-3 contains three edge types (*Flow*, *Concretization*, *Decomposition*) instead of the corresponding port types. The reason for this inconsistency is that the synthesis of hybrid powertrains was developed based on a predecessor of booggie not requiring that strict application of ports. In booggie, all relations have to be port-based. As depicted in Figure 5-6 instead of a decomposition edge, decomposition ports are used.

5.3 Software architecture

The *Model-View-Controller* (MVC) architectural pattern is a widespread approach in software development that was first formulated by REENSKAUG (1979). As depicted in Figure 5-7, MVC classifies an application in three different parts:

- *Model* should be a one-to-one mapping to the user's mental model representing a segment of the reality in the user's perception. booggie refers to a mental model that interprets product architectures as graphs built up of elements and ports. The computerized model does not necessarily have

to be part of the software tool; it could, e. g., be implemented in an external database. However, in booggie it is part of the implementation.

- *View* is a visual representation of the model and adjusts to the user's input. A single model could have multiple views for different purposes.

- *Controller* manages the presentation of views and handles user input. According to the user's input, the controller modifies views and updates the model.
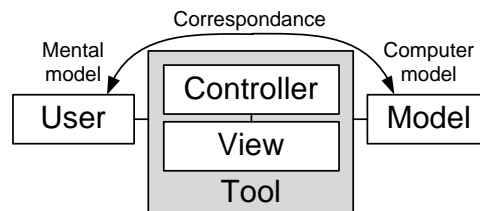

Figure 5-7: Model-View-Controller schema according to Reenskaug (1979)

Subdividing an application into these three parts clearly structures the software architecture and enables several advantages. The most significant benefit of MVC is the decoupling of components; this increases the reusability of single program parts. Hereby, complex software structures are subdivided into smaller, manageable components, which cannot only be reused for different purposes but also can be replaced by components with a similar or extended functionality. Due to this separation, an application is easier to maintain and test because each component has to be tested only once and occurred errors can be located easily. This principle also results in a less complex program structure because the code for solving one concern is not cluttered with the code of another one, which again results in a higher flexibility and reusability (Hürsch & Lopes, 1995). For example, the graph visualization component of booggie is reused several times, like visualizing graph transformation rules, initial graphs and graph transformation results. It only takes care of presenting a graph and has no bindings to other components that may interact with this graph. Hence, it could be reused for any visualization purpose independent from the context.

For a more detailed description of the implementation of the MVC pattern in booggie and the realization of the Software Requirements 1, 2 and 13, refer to Appendix 9.4.

5.4 Implementation of selected (sub-)components

booggie is implemented in the programming language Python[34] that is clear to read and write and easy to learn. Python is a scripting language, hence, no compilation is required since Python is interpreted at run time. It supports many programming paradigms, such as object-oriented programming. A Python-based adaption (PyQt) of the powerful GUI development library Qt has been used. The combination of Python and PyQt makes it possible to develop platform independent applications and run them unchanged on all the supported platforms. booggie has successfully been executed on various Windows versions and on the Linux distributions Ubuntu, Debian and Mandriva, cf. SWReq 12. A wide variety of scientific libraries are available making Python, concerning the functionality, similar to Matlab. These advantages and the fact that Python can be combined with compiled languages like C++ makes Python an excellent choice for developing scientific software prototypes in the field of scientific computing (Langtangen, 2011).

---

[34]Official Python website: http://www.python.org

Over the years, booggie evolved into a large software project. It currently consists of more than 170,000 lines of code[35]. Hence, a description of the entire implementation would exceed the scope of this thesis. Instead, some of the key components, or their subcomponents, are described that implement the main requirements. The management and execution of this software development project is carried out using the facilities offered by Sourceforge[36]. The source code and additional project information are accessible on the project website at http://booggie.org.

5.4.1 Metamodel perspective

Within the metamodel perspective (Figure 5-8) the modeling domain is defined. For illustration, a simple example is used consisting of the geometric primitives circles and square in the colors blue, green and red. These elements are hierarchically defined in the Element Types docklet, cf. SWReq 3. The use of inheritance is exemplified with two float number attributes that are defined for the element type Primitive and inherited by all subtypes. The length attribute contains the default value 5 and the attribute for representing the surface size is set to 0 and has to be calculated in a later step. A special window for the definition of the elements visual appearance (size, position, shape, color) is available. The visual attributes of the elements and ports can be accessed during graph transformation, cf. SWReq 5. The compatibility, i. e. topological constraints, of the elements is defined based on port types. The port types are defined such that only elements of same geometry or of same color can be combined. The port types themselves are defined in the Port Types docklet, cf. SWReq 4. In the center view, the element type RedCircle is presented in detail containing the inherited attributes and the assigned ports that are differentiated according their direction (every element has one incoming and one outgoing port to be connected with compatible elements).

While booggie in general is kept solution-independent (SWReq 1), the Metamodel perspective enables booggie projects to be tailored to specific problem domains. Through the definition of element types and port types, domain-specific languages (DSL) can be defined. These specify modeling domains and provide the foundation for formulating graph transformation rules allow automating model manipulations. booggie itself provides in the Metamodel perspective a formal metamodeling language for the definition of DSLs that is called *booggie modeling language* (bgML). A formal definition of bgML is given in Section 5.2.

Using bgML for defining hierarchical tree structures of element types and port types is not based on a visual, graph-based modeling environment. Instead, a folder structure is utilized that is commonly known from Windows Explorer and allows an intuitive approach (SWReq 2) towards hierarchies.

5.4.2 Rules perspective

Graph transformation rules and scripts are defined in the Rules perspective (SWReq 6). For the definition of graph transformation rules, a visual rule interpreter is developed, see Figure 5-9. Through drag-and-drop, element types from the metamodel are instantiated and can be placed on the left-hand side (defining the source graph to be sought) or on the right-hand side (defining the target graph as a result of the rule application) of the rule. Thereby, the definition of graph transformation rules is simple (SWReq 2). The textual rule description, i. e. the rule code, according to the GrGen.NET syntax, i. e. the rule (source) code, is generated automatically. GrGen.NET is the library that is integrated in booggie to perform the

---

[35]see statistics at http://www.ohloh.net/p/booggie
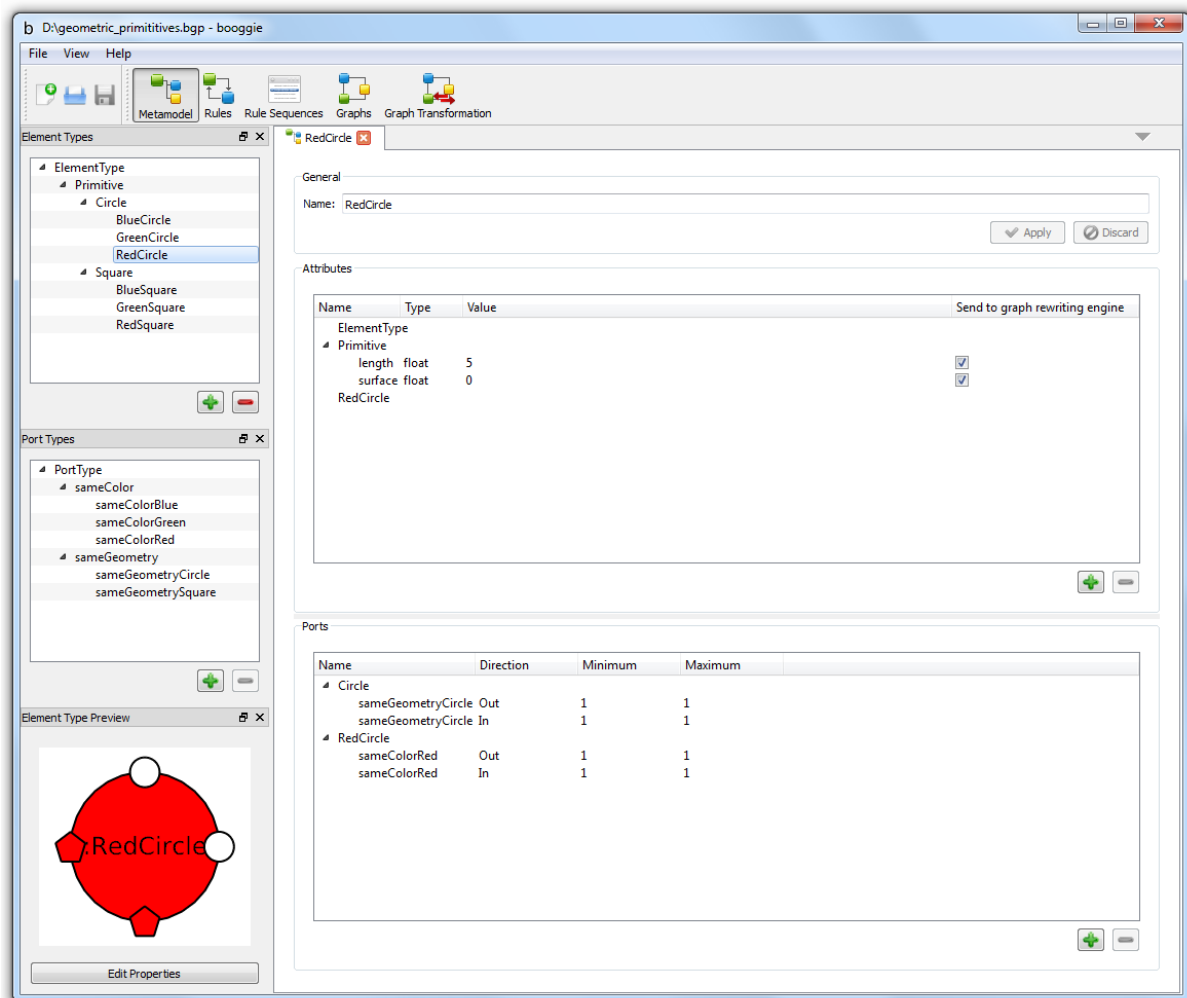[36]Website: http://sourceforge.net

Figure 5-8: Metamodel perspective

compilation of the graph grammar and the execution of the graph transformations (Jakumeit et al., 2010) (the integration of GrGen.NET is discussed in more detail in Section 5.4.5). The formulation of rules is exemplified here with a rule adding a red square to a red circle, see Figure 9-4. The user can directly access the rule code entailing the visual graph widgets to disappear. Thereby, the entire GrGen.NET syntax can be used allowing for the definition of sophisticated rules such as required for the functional decomposition as described in Section 3.2.1.

A simple function for calculating the surface of circles and squares illustrates the definition of scripts in Figure 5-10. The local variable *primitive* contains the element of which the surface is to be calculated and adapts, according to the respective shape, the method for calculating the surface. Hence, this rule is generally applicable to all current ElementTypes and can be extended to additional geometries. Another particularity to be mentioned is the import of the external math module that is required to access the value of pi. In addition to this standard module, any other self-programmed module can also be imported. Thus, the execution of external programming code is possible and interaction with external tools and data sources can be realized.

Visual rule definition:

Left-hand side

Right-hand side

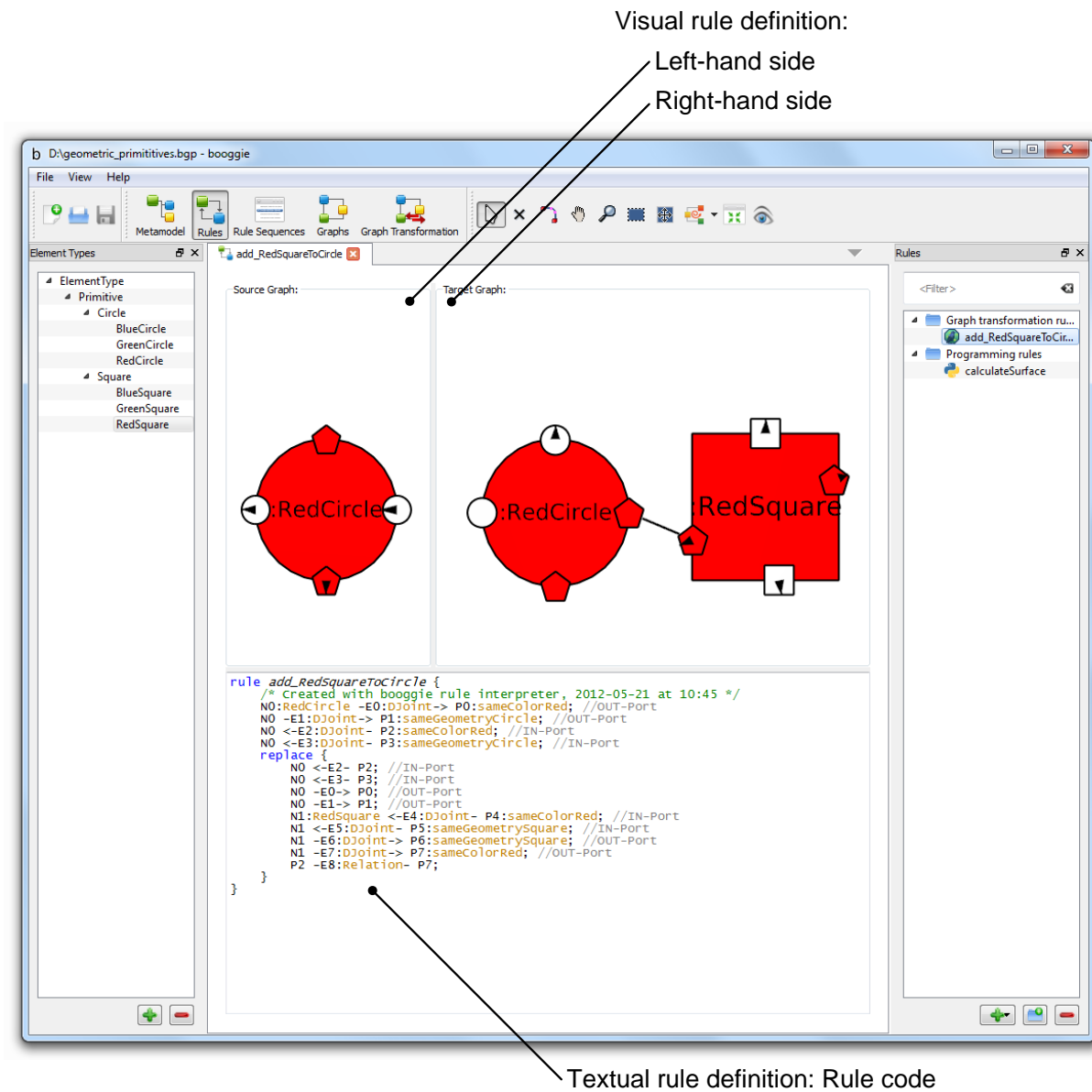Textual rule definition: Rule code

Figure 5-9: Rule interpreter for the visual and textual definition of graph transformation rules

5.4.3 Rule Sequences perspective

The concatenation of graph transformation rules and scripts can be done in the Rule Sequences perspective, see Figure 5-11. Rule sequences, as for example defined for the synthesis of hybrid powertrain architectures in Figure 3-4 on p. 63, allow to formalize a synthesis strategy. The powertrain example focused on the graph transformation aspects of this synthesis approach – essentially adding, modifying and deleting nodes and edges of a graph. However, in many applications it is required to perform actions during the synthesis procedure that can better be described in a programming language or require the richer functionality of external programming libraries or software tools. Such use cases can range from the simple calculation of a random number to the import of data from external data sources or the execution of an external tool, e. g. a simulation software. Within this software prototype, a mechanism is realized that can handle rules and scripts in a uniform way. It is illustrated with the example in Figure 5-11 that shows a rule sequence composed of rules and scripts and consists of four steps:
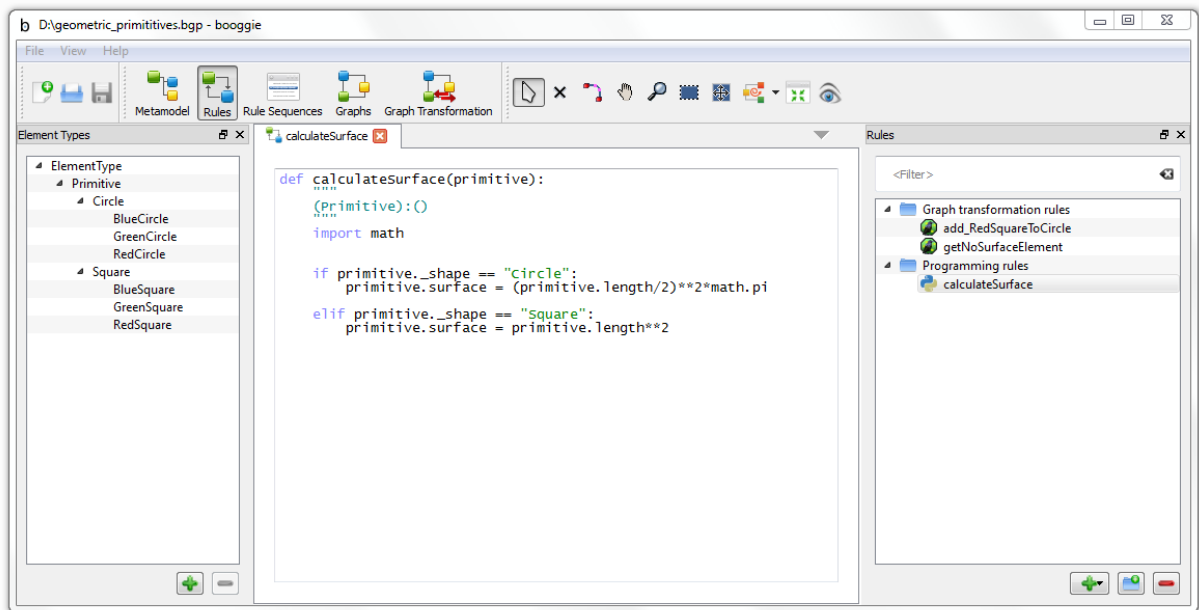
Figure 5-10: Text editor for the definition of scripts in the Rules perspective

1. Ten elements are randomly created in ten iterations as defined with *[10]*. The decision about which element type to add is randomly made with the generation of a random integer number in the script *getRandomInt*. This random value is stored in the variable *randInt* and further used as rule parameter for the graph transformation rules.

2. Two random blue elements are connected using the rule *connectTwoBluePrimitives*.

3. The rule *add_RedSquareToCircle* adds a red square to a red circle. The embracement with square brackets defines that this rule applies to every occurrence of a red circle.

4. The rule *getNoSurfaceElement* detects an element in its left-hand side having no surface value and returns it to the variable *element*. The script *calculateSurface* receives this element as a parameter and calculates the value of the element's surface. The asterisk operator * defines that this operation is repeated until no element without surface value can be found.

The example illustrates the information flow from rules to scripts and vice versa. This possibility to concatenate rules and scripts in one control structure is a distinguishing feature to existing graph grammar implementation. Further, this feature is the underlying basic principle for enabling interfaces between booggie and external tools and to make these interfaces available in the synthesis procedure. Thereby, the computational implementation of Expected Ccntribution 4 becomes possible.

The realization of this uniform sequence handling, and hence the realization of SWReq 9, is based on a wrapper of GrGen.NET that controls the information flow between rules and scripts; it is further described in Section 5.4.5
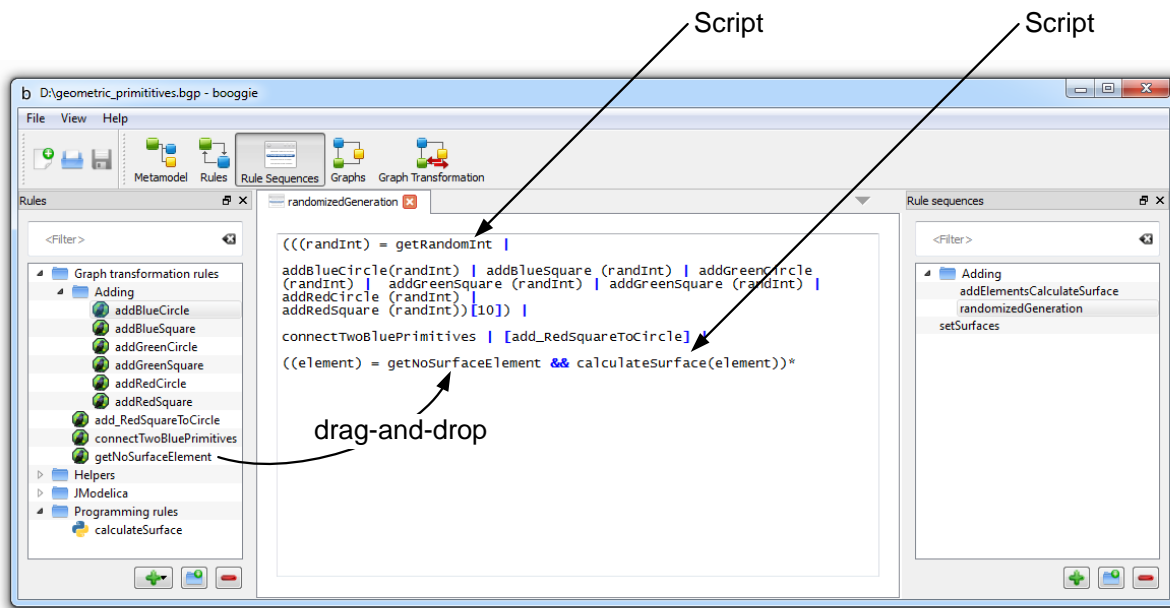
Figure 5-11: Concatenation of rules and scripts in the Rule Sequences perspective

5.4.4 Graph visualization

The graph widget[37] is a central part of the booggie application. It encapsulates the functionality for displaying and manipulating graphs and is reused in several other components such as the Rules perspective. This widget can be imagined as a black box that accepts a booggie graph instance and renders a graph as OpenGL scene graph. The user can then manipulate the graph and the changes affect the underlying booggie graph object.

Tulip[38] is a 3D information visualization framework that is used to implement the graph visualization, cf. SWReq 10. One of Tulip's key benefits is the use of OpenGL for graph rendering. OpenGL allows performing the graph drawing directly from the graphical processor unit (GPU). This increases the rendering speed compared with implementations that use the central processing unit (CPU) exclusively for drawing. Tulip's data structures are optimized for handling large graphs and the framework is largely plugin-based. Common functionality such as interacting with the graph or layouting the graph is achieved by using runtime replaceable components rather than static hard coded parts, cf. SWReq 11. This means that those components can be dynamically loaded during runtime, enabling the development of custom solutions that can be easily added to the existing application without any further work. Based on this principle, booggie reuses various graph layouting algorithms of Tulip.

Tulip is written in the programming language C++ meaning that the integration with Python had to be achieved manually. For this task, the SIP bindings framework[39] is used to wrap the Tulip classes as Python objects. Wrapping allows the access of the class instances from within Python while transparently performing type conversions between the Python and C++ domain.

---

[37]The term *widget* is used within Qt and PyQt for the graphical elements that make up a user interface.
[38]Project website: http://tulip.labri.fr
[39]Project website: http://www.riverbankcomputing.co.uk/software/sip

The original Tulip view component is reused in a widget coupling a booggie graph with Tulip's internal graph presentation. Both graph data structures are kept consistent between the two domains. Both components provide their own means of representing a graph. This decision is made to stay independent of the actual Tulip library used to allow for later integration of alternative graph visualization components.

5.4.5 Graph Transformation perspective

GrGen.NET is a graph rewrite generator that offers a domain neutral language for graph modeling (JAKUMEIT ET AL., 2010). To achieve domain neutrality, GrGen.NET provides the possibility to create a metamodel for defining the modeling domain. The metamodel definition supports type inheritance allowing easy integration with the object oriented approach of booggie. GrGen.NET offers exceptional graph transformation speed, even for large graphs, by applying various performance optimization techniques to its search process and has proven to be significantly faster than its competitors. For example, when searching a matching subgraph in a host graph while checking the applicability of a rule's left-hand side, GrGen.NET considers the best starting point using heuristics and taking the knowledge about the last successful match into consideration. This is done in a conservative way such that there is no risk that matches are not found (GEISS & KROLL, 2007).

Further, GrGen.NET covers the required expressiveness and can conveniently be integrated in booggie using its application programming interface. Moreover, a shell application is available, GrShell, that includes a debugging interface that is useful for the development of rules and rule sequences.

Within GrGen.NET, three formal languages are defined (JAKUMEIT ET AL., 2010) based on the language specification language ANTLR[40] (PARR, 2007):

- The *graph model language* allows the specification of the metamodel through typed and attributed multigraphs with multiple inheritance on node and edge types and, further, the consideration of abstract types. Within booggie, this language plays a minor role because the developed approach is not based on typed and attributed graphs but on elements and ports. These are required for the definition of object-oriented graph grammars such as described in Chapter 3 and are explicitly defined in bgML as presented in Section 5.2. Hence, bgML is an intermediate language layer that builds on the graph model language and provides the foundation for defining element- and port-based metamodels within booggie. This stack of modeling languages is further detailed in Section 5.2.

- The *rule set language* can be used to define rewrite rules. It covers the pattern specification (left-hand side graph) and the rewrite specification (right-hand side) and allows recalculating attributes of graph elements during rule application. Further, rule parameters, return values and rule constraints can be defined, cf. SWReq 7. Rules can be applied on any level of abstraction of the node and edge types of the metamodel, cf. SWReq 8

- The *rule application control language* is a means of composing complex graph transformations using single graph transformation rules. Graph transformation sequences – and all of their subsequences and down to individual rules – have a boolean return value. For a single rule, true means the rule was successfully applied to the host graph, false means the left-hand side could not be matched to the host graph.

---

[40]**AN**other **T**ool for **L**anguage **R**ecognition

For a comprehensive introduction to GrGen.NET, refer to the official manual[41].

GrGen.NET is written in C# and generates .NET assemblies containing the compiled graph transformation engine. To interface with these generated assemblies, language bindings between Python and .NET are needed. This functionality is provided by the Python-for-.NET[42] package that integrates .NET's Common Language Infrastructure (CLI) into the Python interpreter. Python for .NET supports the Microsoft CLI as well as Mono [43], which is an alternative implementation of the CLI supporting multiple platforms such as Linux or Mac OS. Thereby, GrGen.NET is platform independent.
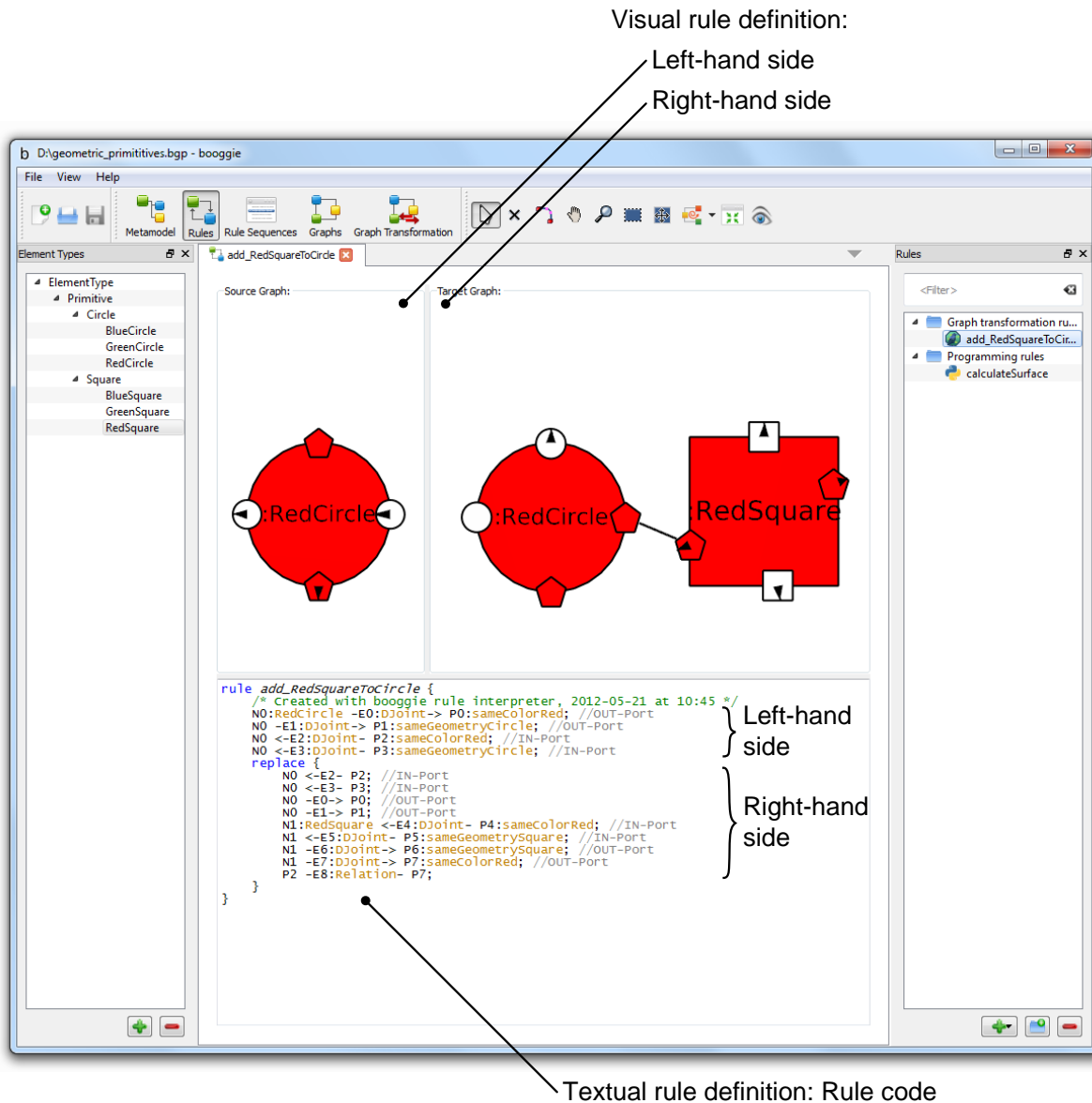
Figure 5-12: Graph transformation perspective

The interface to GrGen is implemented using an additional C# wrapper that is accessible from the Python domain. This wrapper is needed to work around some shortcomings of the Python for .NET component

and to move the transformation engine into its own thread to avoid an unresponsive user interface during graph transformation. This wrapper also handles the execution of scripts. Both, graph transformation rules and scripts are uniformly treated and can be concatenated to rule sequences. Rules and scripts can exchange information based on rule parameters and return values. While graph transformation rules use GrGen.NET's rule language, scripts can be programmed in Python. If the wrapper, during graph transformation, detects a programming rule, the further execution of the graph transformation is paused and the script is fed to the Python interpreter together with the rule parameter as defined in the rule sequence. When the programming code is executed, the wrapper receives the return value and injects it in the rule sequence where it can be further used. With this mechanism, SWReq 9 is met.

The Graph Transformation perspective allows the user to control the graph transformation and to inspect the generated graphs. Figure 5-12 shows a screenshot of this perspective. A graph transformation is always executed on a host graph that might be an empty graph as well. Hence, the first step is to open a graph from the Graphs docklet. In the top left docklet, rule sequences are available that can be applied on the graph in the center view. To select a rule sequence for transformation, it has to be drag-and-dropped into the text field under the graph widget. When clicking on Rewrite, the grammar is compiled and the graph transformation is performed. Every time the metamodel or the rule set is modified, the grammar is automatically recompiled. The graph resulting from the transformation is subsequently represented in the center view.

In the application example, the rule sequence depicted in Figure 5-11 is applied on an empty graph and the generated solution is saved as Solution_1. In the Selection Info docklet, the attributes of the selected element are listed including attributes regarding the visual appearance but also attributes specifically defined in the metamodel, such as the surface that is now 19.63. The Transformation Logs docklet is a console that indicates by default information about the executed transformation but can also be used to print additional information or data, e. g. for debugging, during the graph transformation.

The components for graph visualization and graph transformation, but also the GUI library Qt, are based on external libraries that are implemented in other programming languages than Python. Hence, wrappers are implemented or reused to integrate them into booggie. This allows for a smooth flow of objects between these libraries and the core application. The possibility to wrap external, precompiled libraries using wrappers, such as SIP or Python for .NET, is one of the big advantages of Python. Figure 5-13 illustrates the dependencies between booggie, external libraries and wrappers. The implementation of wrappers represents a considerable effort. However, the advantage of reusing mature software components (cf. SWReq 14) with a rich functionality outweighs this effort by far.
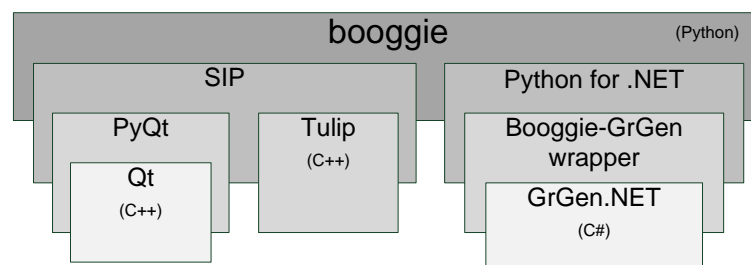

Figure 5-13: Integration of external libraries using wrappers

The high graph transformation performance of GrGen.NET has been mentioned earlier and booggie profits from that. To give an impression how fast the generation process in booggie is, the synthesis pro-

cess for automotive hybrid powertrains, as presented in the validation study in Section 3.3, was executed several times on two different computers, a laptop (Lenovo X121e) and a desktop pc (DELL Precision T5400). The results are depicted in Figure 5-14.

Notebook: Lenovo X121e — Operating system: Windows 7 Professional 64 Bit; Working memory: 8 GB; Processor: AMD E-459, Dual-Core, 1.65 GHz

Run	1					2					3					Average values	
Compilation time (in s)	11.2					10.2					12.6					11.33	
Solution	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2 - 5
Rule applications	157	261	228	156	216	218	176	289	309	296	278	214	170	335	256	237	
Generation time (in ms)	1279	733	858	343	608	1450	515	858	874	811	1747	577	390	873	577	1492	668
Time per rule application (in ms)	8.15	2.81	3.76	2.20	2.81	6.65	2.93	2.97	2.83	2.74	6.28	2.70	2.29	2.61	2.25	7.03	2.74
Average time per rule application (in ms)	2.90					2.87					2.46					2.74	
Average reduction of time per rule application	64%					57%					61%					61%	

Desktop PC: DELL Precision T5400 — Operating system: Windows 7 Professional 64 Bit; Working memory: 4 GB; Processor: Intel XEON, Quad-Core, 2.83 GHz

Run	1					2					3					Average values	
Compilation time (in s)	4.0					4.2					4.1					4.10	
Solution	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2 - 5
Rule applications	239	299	190	215	264	210	184	195	190	215	218	220	220	262	246	224	
Generation time (in ms)	609	156	140	156	219	406	156	140	140	172	452	202	156	203	203	489	170
Time per rule application (in ms)	2.55	0.52	0.74	0.73	0.83	1.93	0.85	0.72	0.74	0.80	2.07	0.92	0.71	0.77	0.83	2.18	0.76
Average time per rule application (in ms)	0.70					0.78					0.81					0.76	
Average reduction of time per rule application	72%					60%					61%					64%	

Figure 5-14: Analysis of the computing time for the generation of FBS* product architectures

On every computer, three runs were executed, each consisting of the grammar compilation and the generation of five solutions while the first solution is always the one that was generated directly after the grammar compilation. The generation of one FBS* product architecture requires an average of approximately 230 rule applications. As the synthesis process uses constrained random walk, a fixed number of rule applications can not be given. In order to make the computing time comparable, the generation time is divided by the number of rule application. It becomes apparent that one rule application takes significantly longer for the first solution whereas the generation time for the following four solutions are all in the same order of magnitude. The time required for one rule application is for the solutions 2 –4 between 57% to 72% lower than for solution 1. The reason for this reduction in computing time is GrGen.NET's internal optimization of the search process.

The generation of five solutions does not require a lot of working memory. Hence, the difference of 4 GB between the two configurations is not relevant. However, the processor performance has an essential impact on the computing time. On the desktop pc, the generation time is lower by a factor of 3 to 4. The fact that the desktop has a quad core processor has no influence on the computing time as booggie is not using any parallelization. The reason for the lower computing time is the higher clock signal of the processor. Despite that significant difference, the generation of one solution in lower than three seconds on the slower computer is still fast enough to generate a high number of solutions in reasonable time.

5.5 Validation: Synthesis of aircraft cabin layouts

This section summarizes the approach and results of a case study conducted together with European Aeronautic Defense and Space Company (EADS) targeted at the synthesis of aircraft cabin layouts. After a short introduction to the motivation and goals of the project, the CDS approach is presented followed by the synthesis results and observations on the software use and method application.

The configuration of aircraft cabin layouts is a knowledge-intensive task that is manually carried out by experts. The goal of this task is to define the composition of the cabin consisting of cabin items, i. e.

components. The problem description is composed of the customer requirements typically stemming from the airline, such as the number of seats per class for given service ratios. Service ratios relate service characteristics to the number of seats, e. g. number of seats per cabin attendant. Constraints from the aviation authorities and internal requirements, e. g. from the production, have to be considered as well.

In reference to the FBS* product model presented in Section 3.1, the Structure level is of primary importance here. The Behavior level is omitted as the cabin items are predefined and do not have to be developed based on physical effects. The analysis of the Function layer could provide added value in terms of generating functionally innovative cabin layout designs and exploring the interplay of functions and their embodiment as components. However, to frame the scope of this study, the Function layer is disregarded as the functionality of the aircrafts of the Airbus A320 aircraft family[44] is not considered in this design task.

SABIN & WEIGEL (1998) define configuration as a special case of design having two key features: the assembly of an artifact from " a fixed set of well-defined component types" and "components interact with each other in predefined ways". In (FELFERNIG ET AL., 2011) this definition still reflects the common understanding of configuration research. FELFERNIG & SCHUBERT (2011) add to this the aspect of "configuration knowledge" that is utilized by configuration systems. It is composed of a "product structure" and constraints regarding the compatibility of components, requirements and resources. Hence, configuration is characterized by the predefinition of the building blocks of a solution and a constraint-based definition of the configuration template. In contrast, engineering design is characterized by a high degree of uncertainty and lack of knowledge about future solutions as discussed in Chapter 1. From this perspective, this case study can be seen as a configuration problem as building blocks and configuration knowledge are predefined by EADS and issues related to the fuzzy front end are disregarded.

Despite considering this case study as a configuration study, the task of configuring aircraft cabins in general is faced with challenges that are special cases of those generally described for the conceptual design phase in Section 1.1. The designers do not consider and investigate a wide range of solutions as the configuration space is too vast and characterized by numerous constraints. A manual exploration of a high number of solutions is not feasible, hence, configurations are generated that are, with respect to the customer requirements, good enough although there could be better solutions. This procedure is prone to errors and often results in extensive rework leading to delay and increasing development costs. Further, a quick reaction to changing customer requirements is only possible with massive manual rework. Hence, with a quick generation and analysis of a high number of different layouts and the identification of the best solution these problems are addressed. This can be seen as a considerable step towards the goal of generating optimal aircraft cabin layouts.

The aim of this case study is to show that the approach of object-oriented graph grammars, as implemented in booggie, can deliver a contribution to tackle these challenges in an industrial context and provides a solid foundation for future work. Thus, the Expected Contributions 5 (human-competitive solutions) and 6 (mature software prototype) are validated. The achievement of Expected Contribution 6 builds on the fulfillment of the software requirements and is discussed in Section 5.6.

---

[44]This aircraft family includes the A318, A319, A320 and A321. It can carry between 132 (A318) and 220 (A321) passengers and has a range of 3100 to 12000 km.

5.5.1 Structure of aircraft cabins and definition of the metamodel

In the following, a brief introduction into the overall structure and nomenclature of aircraft cabins of the Airbus A320 aircraft family is given. First, *logical items* are introduced that are not physically embodied but allow subdividing the entire cabin system into modules, as illustrated in Figure 5-15. Typically, the cabin is divided into three zones: the *door compartment zones* A and C and the *passenger compartment zone* B. Zones are further partitioned into subzones for the left-hand side (LH) and right-hand side (RH). Additionally, the door compartment zones are subdivided according to the exit doors' center line (resulting in four subzones for A and C) and the passenger compartment zone is subdivided into the travel classes. As this case study only considers layouts with one travel class, i.e. economy class, zone B is only subdivided into B1_LH and B1_RH. Subzones in zone B are filled with rows that are primarily characterized by their depth, termed pitch.
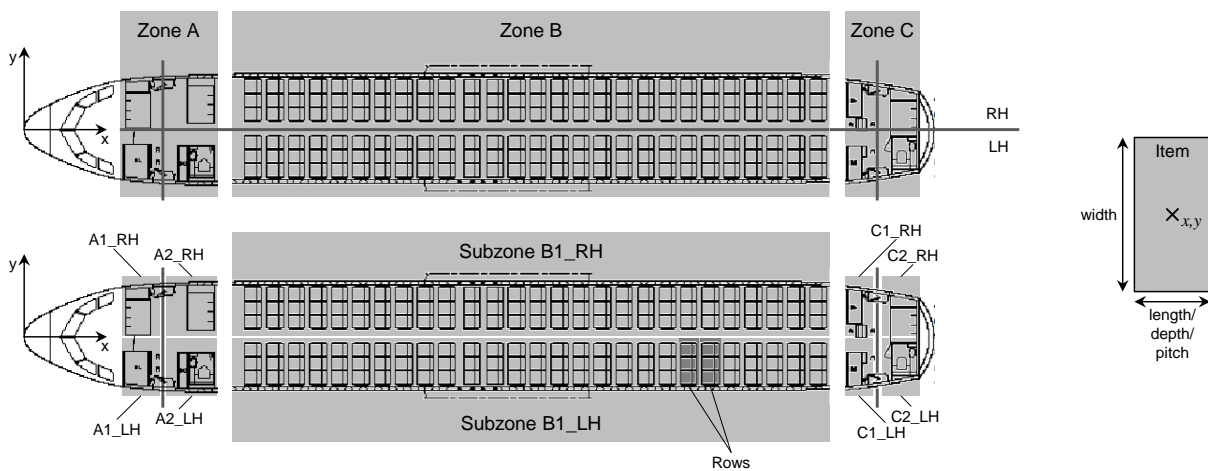
Figure 5-15: Logical items composing an aircraft cabin and spatial definition of items

Physical items, in turn, are reflected in real existing components of the aircraft cabin, i.e. cabin items. Both, logical items and physical items are located in the coordinate system, as depicted in Figure 5-15, at the *x*, *y*-position in the center point and are spatially defined by their length, depth or pitch and width. The following types of physical items are taken into consideration:

- Seats can be differentiated into *passenger seats* (abbreviated as *PAX seat*[45]) and *cabin attendant seats* (*CAS*). PAX seats for the economy class are installed in groups of three, termed triple seat block (*TSB*). CAS are mainly located in the door compartment zones and are mostly wall-mounted, folding seats. Strictly speaking and as an aside, seats should be considered as logical items as they are physically embodied in the TSB. However, for simplification reasons, the partners at EADS chose to consider them as physical items.

- In the galley food is cooked and prepared. *Wet galleys* (*WG*) have, in contrast to *dry galleys* (*DG*), access to the aircraft's water system and are only located in the door compartment zones. Galleys provide space for *trolleys* being mobile containers for the catering goods. Usually, a galley contains three or four trolleys (*WG3, WG4, DG3, DG4*).

- *Lavatories* (*LAV*) are only located in the door compartment zones.

---

[45]PAX is the aviation-specific nomination for "persons approximately".

- There are various free areas that are explicitly modeled: *Unused space* is left over space that stems from the procedure of adding rows and is not sufficiently large to add an additional row. In front of the first row, additional *legroom* has to be considered and, behind the last row, space for the seat inclination, termed *recline*, has to be added. The space for the passengers to reach the emergency exits is relevant to security and must not be obstructed with cabin items. This space is partially distributed over the four door compartment subzones and termed *partial passageway* (*PPW*). In the case of an emergency, the cabin attendants have to assist the passengers in leaving the aircraft through the emergency exits. Dedicated areas are defined for that purpose and are termed *assist space* (*ASP*).

- *Partitions* and *curtains* are used as blinds, e. g. between door compartment zone and passenger compartment zone, or to separate travel classes.

According to the object-oriented graph grammar approach, these elements are defined in the metamodel, Figure 5-16. Again, it becomes apparent how using a hierarchy of inheritance eases the definition of the modeling elements. Attributes that apply to all elements only have to be defined once in the topmost element type, e. g. the x-/y-/z-coordinates. Ports contain the information about the logical composition of the cabin layout, e. g. a *Row* contains a *SeatBlock*.
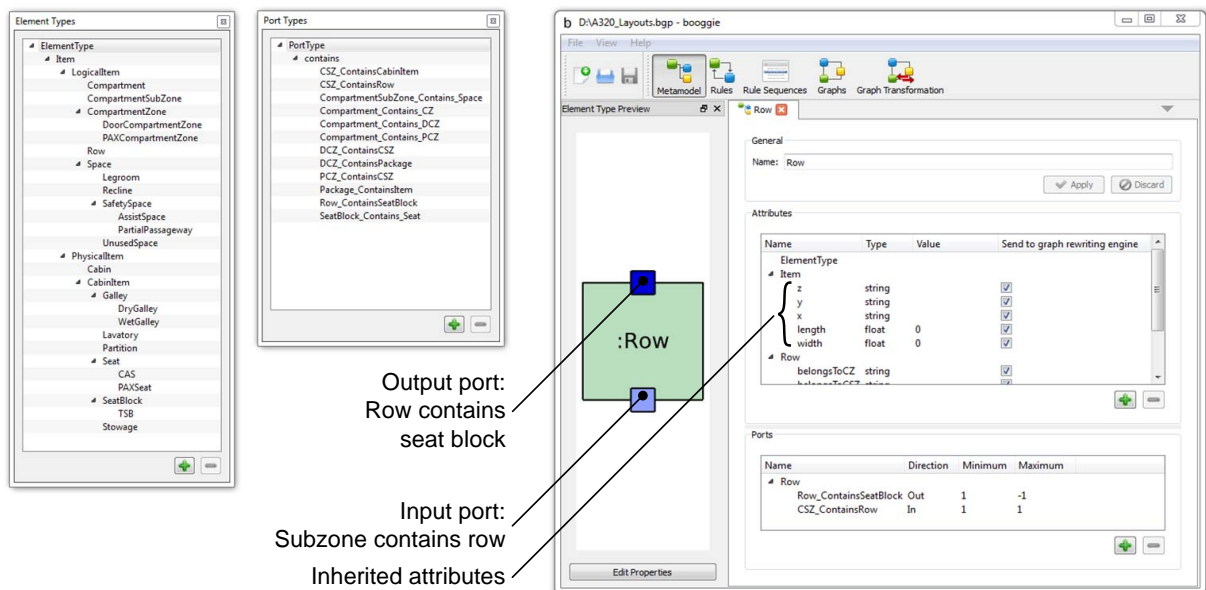
Figure 5-16: Metamodel for aircraft cabin layouts: Element type hierarchy, port type hierarchy and definition of the element type *Row*

The entire aircraft cabin can be characterized by certain parameters that can be either evaluation criteria of generated layouts or requirements provided by the customer. The following layout parameters are crucial for an early cabin layout and are considered in this case study:

- A high *number of seats* is primarily in the airline's interest as it stands for the achievable turnover.

- The *pitch* value defines the distance between rows and is the key indicator for seating comfort.

- *Service ratios* indicate the quality of service on board. The key service ratios are the *lavatory ratio* (PAX/lavatories), the *trolley ratio* (PAX/trolley) and the *cabin attendant ratio* (PAX/attendant).

As the number of passengers is in the numerator, a lower service ratio stands for higher service quality.

- *Unused space* is not profitable and should be as low as possible. If unavoidable, this space is used as stowage.

Typical values for these parameters are depicted in Table 5-1. These statistical values were raised by EADS in an analysis of 54 aircraft cabin layouts of the type A320. All layout parameters, except the cabin attendant ratio, are divided into high density and high comfort ranges.

Table 5-1: Typical layout parameters divided into high comfort and high density layouts

	High density			High comfort		
	avg	min	max	avg	min	max
Number of PAX seats	172.3	165.0	180.0	152.6	136.0	159.0
Pitch (in inch)	29.5	28.0	31.0	33.5	32.0	36.0
PAX per lavatory	57.4	55.0	60.0	50.9	45.3	53.0
PAX per trolley	16.4	13.2	20.8	11.6	10.4	14.2
PAX per cabin attendant	32.0	30.0	33.6	32.0	30.0	33.6

5.5.2 Definition of rules, scipts and the rule sequence

The overall CDS approach taken in this case study has some similarity with the procedure implemented in the synthesis of hybrid powertrains. First, modules are generated in isolation and are then considered in the context of the entire architecture. The synthesis approach that is implemented in a rule sequence is depicted in Figure 5-17.
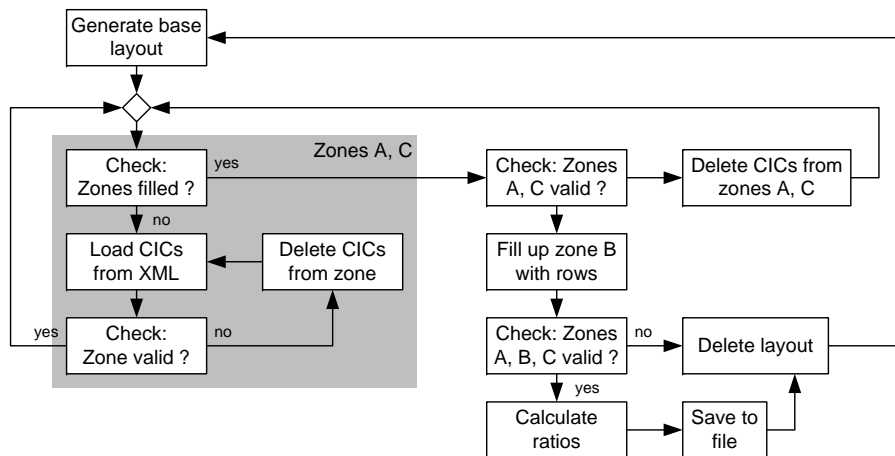
Figure 5-17: Rule sequence for the synthesis of aircraft cabin layouts

None of the rules are defined using solely the visual rule editor. This is due to the fact that the supported expressiveness in the visual editor is too low for the definition of the rules for this application. Starting with the visual rule editor to define the basics of a rule (e. g. identification, addition, deletion of elements and ports) and extending afterwards the rule definition using the textual editor turned out as a good practice. Rules and scripts are structured in a hierarchical folder structure that helps to keep the overview also when the rule set is extended later on. In the following, every rule of the sequence in Figure 5-17 is briefly described and the result of its application in booggie is presented. These rules are considered as high-level rules that are composed of the rules and scripts defined in the rule set. Using GrGen.NET's

command *exec*, rules can execute other rules and scripts and support a hierarchical rule application. For example, the high-level rule *Generate base layout* is composed of the three rules *init*, *initializeParameters* and *createCabinItem* of the rule set as shown in Figure 5-18.
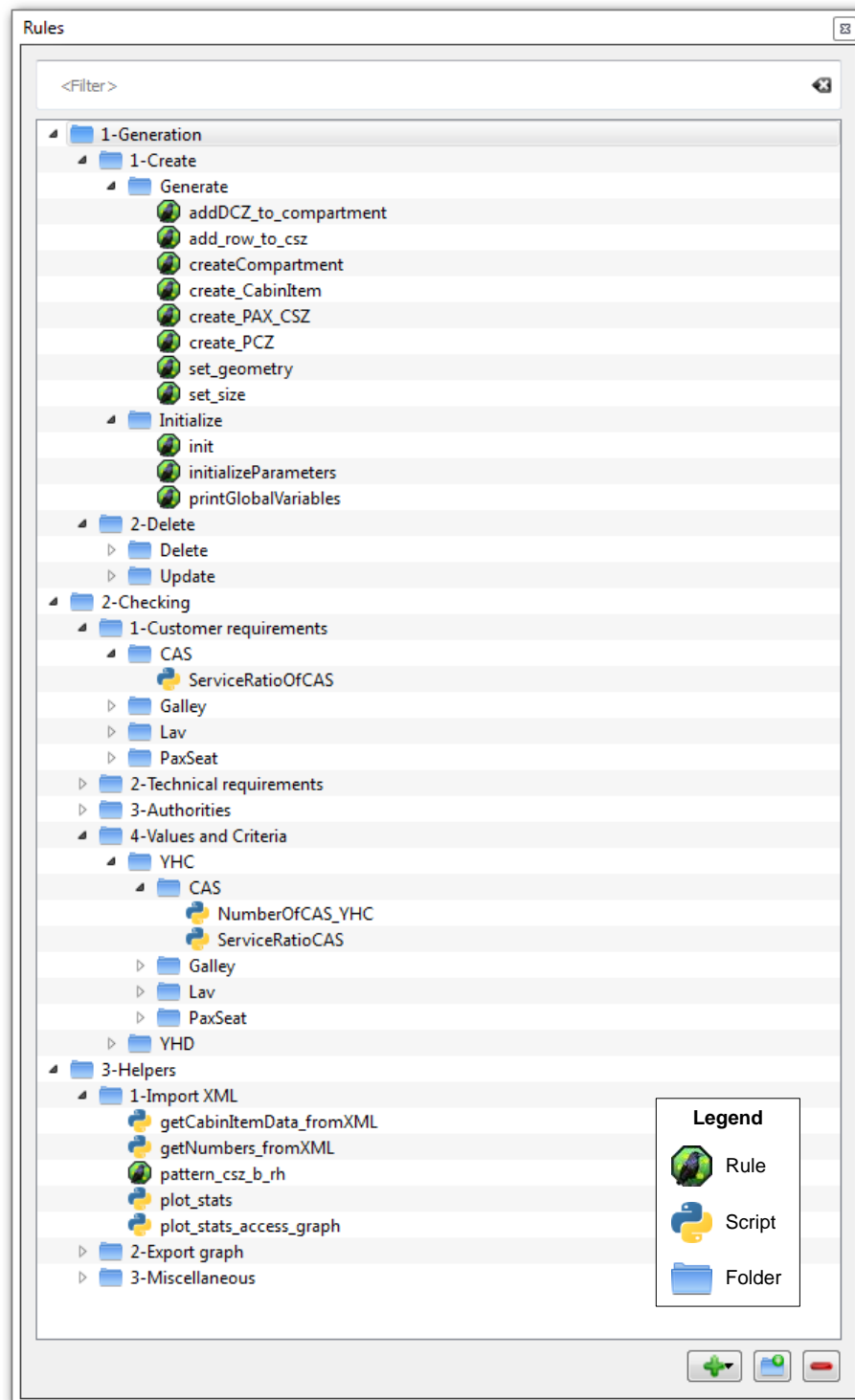

Figure 5-18: Hierarchically structured rule set

Generate base layout sets the overall dimensions of the cabin layout by creating three zone elements:

door compartment zones A and C and the passenger compartment zone B as depicted in Figure 5-19. The dimensions are set according to one specific aircraft type of the A320 aircraft family. In the following, an aircraft of the type A320 is considered.
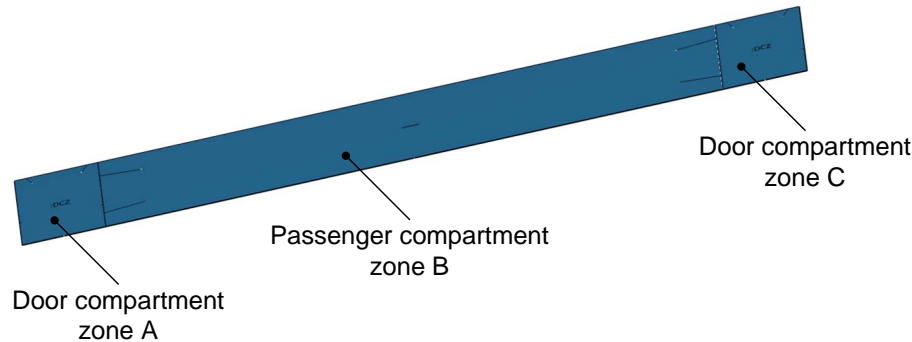
Figure 5-19: Base layout of an aircraft cabin defined through the zones A, B, C

The operations for the synthesis of door compartment zone are identical for zone A and for zone C. Hence, they are defined only once and are applied twice. To ensure, that a door compartment still has to be filled with cabin items, the rule *Check: Zone filled ?* is applied.

If a door compartment zone is not yet filled, the rule *Load CICs from XML* accesses an XML file containing cabin item combinations (CIC). The CICs are predefined and validated by Airbus engineers and reflect best practices for laying out door compartment subzones. A CIC contains, for example, a lavatory and two cabin attendant seats mounted on the lavatory's wall. Further, a partial passageway and assist space is usually included, cf. Figure 5-20. A script is defined for acquiring CICs; for each of the four subzones it queries an XML file for available CICs and randomly picks one of them. A subordinate graph transformation rule instantiates the cabin items according to the CIC-information from the metamodel and places them on the graph. Hence, after this operation the door compartment zone is entirely equipped with components, which is illustrated in Figure 5-21. This operation is repeated for every door compartment zone and results in the placement of cabin items in the front and the back part of the aircraft cabin as shown in Figure 5-22.

While the individual CICs are verified by engineers, the combination of the four CICs of a door compartment zone can still exhibit invalid combinations of components. Most of these constraints state that a number of cabin items has to be within a certain range. These values are not statically defined in rules, but have to be defined in the first step when generating the base layout. Hence, the rule set for checking these requirements is generic as these checking rules can adapt to every layout. Alternatively, these values could be defined in the metamodel. However, this leads to minor disadvantage that value changes, e. g. when generating layouts for different airlines, require that the grammars have to be recompiled, as detailed in Section 5.4.5. The rule *Check: Zone valid ?* checks the following constraints that a valid door compartment zone has to fulfill. These constraints are either customer requirements (C), technical requirements derived from the aircraft engineering (T) or requirements defined by the aviation authorities (A). The customer requirements are typical requirements that are demanded by most airlines.

- Lavatory doors must not open towards passenger seats (C).

- A galley must not be opposite to a seat row (C).

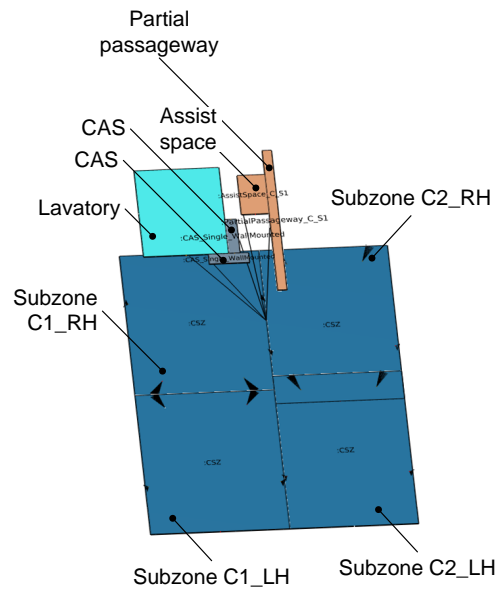- The maximum number of lavatories per door area is limited (zone A: 2, zone C: 4)(T).
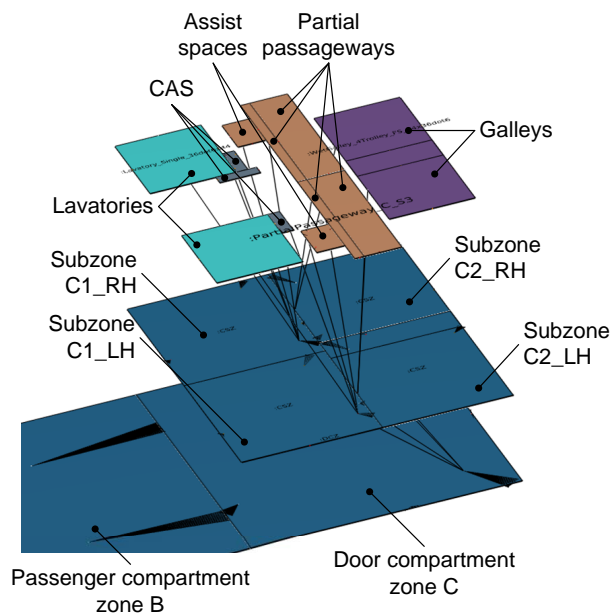
Figure 5-20: CIC for subzone C1_RH

Figure 5-21: Door compartment zone C equipped with the components of four imported CICs

- A cabin attendant seat must be located near each regular exit that also serves emergency exit. Other flight attendant seats must be evenly distributed among the required emergency exits (A).

If one of these requirements is violated, the zone is invalid and the rule *Clear zone* is fired and the procedure for adding CICs is repeated. After both door compartment zones are filled, there are constraints that both zones jointly have to fulfill:

- A minimum of one galley per class is required that has to be located in the class itself to avoid crossing of other classes and to prevent from disturbing catering (C).
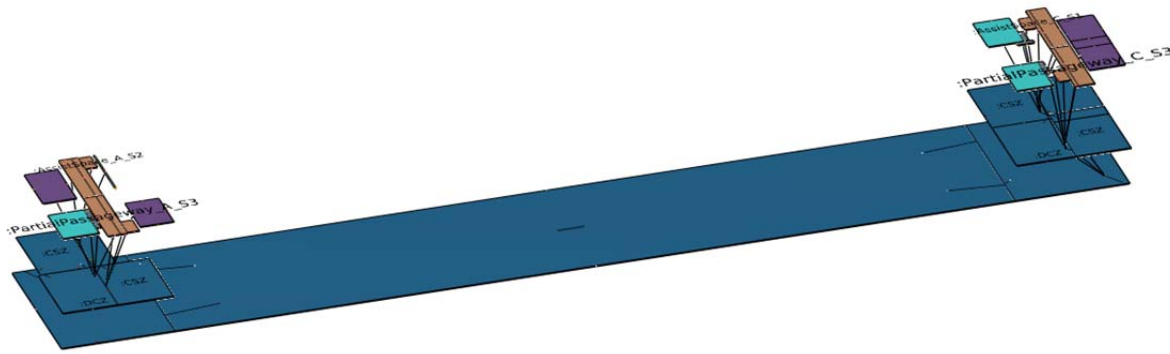
Figure 5-22: Preliminary cabin layout with equipped door compartment zones A and C

- A minimum of one lavatory per class is required that has to be located in the class itself to avoid crossing of other classes and to prevent from disturbing catering (C).

- The maximum number of wet galleys per aircraft is limited to four due to the available systems capacities and load limits for the different galley locations (T).

- The maximum number of lavatories per aircraft is limited to four due to the available vacuum system controller capacity (T).

If one of these requirements is violated, the zones A and C are cleared applying rule *Clear zones A, C* and the synthesis process starts all over again with adding CICs to the zones A and C. If the test is successfully passed, zone B is filled with rows, triple seat blocks and seats using rule *Fill up zone with rows* based on the pitch value. Usually, this value is defined by the airline. A typical pitch value for an economy class seat row is 32". Based on the pitch value, zone B is filled up with rows until there is no space left. In this study, no fixed pitch value is used. Instead, the pitch range between 28" and 36" is explored. Standard values are assigned to legroom (12") and recline (5").

The entire layout composed of the zones A, B and C, as depicted in Figure 5-23, with their respective cabin items has to fulfill these constraints that are checked by the rule *Check: Layout valid ?*:

- The total number of seats is limited to the total egress rate[46] of all exits on one fuselage side. On-wing exits have an egress rate of 35, regular exits have an egress rate of 55. As an A320 airplane has two regular exits and two on-wing exits on every fuselage side, the maximum number of seats is limited to 180 (A).

- The minimum number of cabin attendant seats is limited to one CAS per 50 PAX. Hence, four cabin attendant seats are required (A).

Again, if one of these requirements is violated, all zones are cleared with the rule *Clear zones A, B, C* and the synthesis process starts all over again with adding CICs to the zones A and C. If the check returns a valid layout, the rule *Calculate layout parameters* determines the parameters that are required to assess a layout's quality and to compare different layouts. Finally, the layout is stored in a file using the rule *Save to file* from where it can be accessed for subsequent analysis. Finally, all zones are cleared (rule: *Clear zones A, B, C*) and another layout is generated. After every pass, the pitch value that is used to dimension the rows is increased by 1". This rule sequence represents an infinite loop having

---

[46]The egress rate defines the number of persons that can leave the airplane in an emergency case through one door per minute.

no termination criterion. Nevertheless, a maximum number of iterations can be defined, or the process stops automatically when the working memory exceeds, which is the case after approximately 70 - 90 layouts.
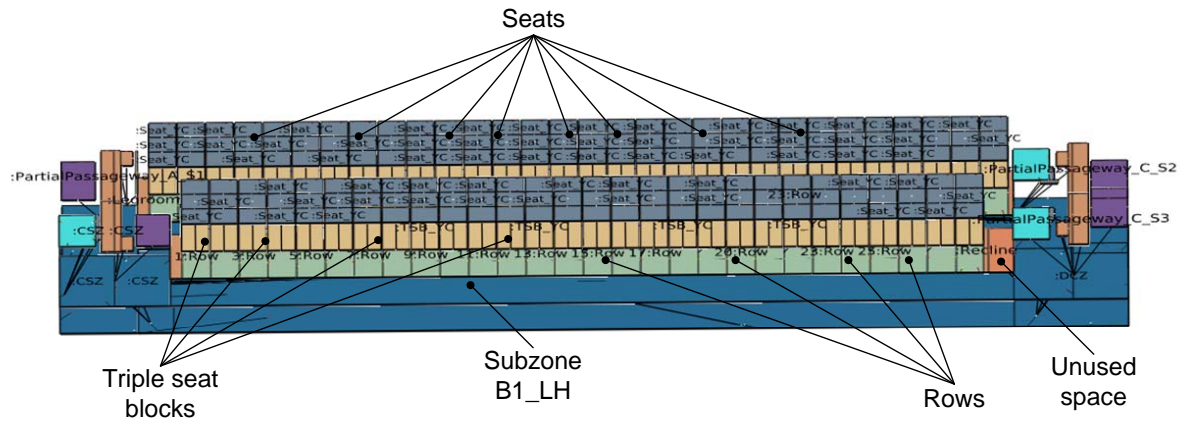

Figure 5-23: Synthesized layout of an aircraft cabin

5.5.3 Grammar application and discussion of the synthesis results

In contrast to the synthesis of hybrid powertrains, the application of an aircraft cabin configurator is not aimed at sparking innovation by generating potentially creative solutions. Rather, the cabin designer is supported in considering a wide range of solutions and gathers an understanding about the solution space. Therefore, alternative solutions can quickly be found and the range of possible solutions can be negotiated with the customer.

For the specification of an A320, 81 valid, distinct layouts are generated covering a pitch range from 28" to 36". This relatively low number is due to the fact that only three different valid CIC-combinations for each of the door compartment zones could be generated based on the available data in the given XML file. However, an increase of the number of solutions is simply possible by adding more CICs to the XML-file. For the identification of distinct solutions a MySQL database is used that can cope with much larger number of solutions. To illustrate this, 2109 cabin layouts are generated containing multiple occurrences of identical layouts. The solution post-processing approach is depicted in Figure 5-24.
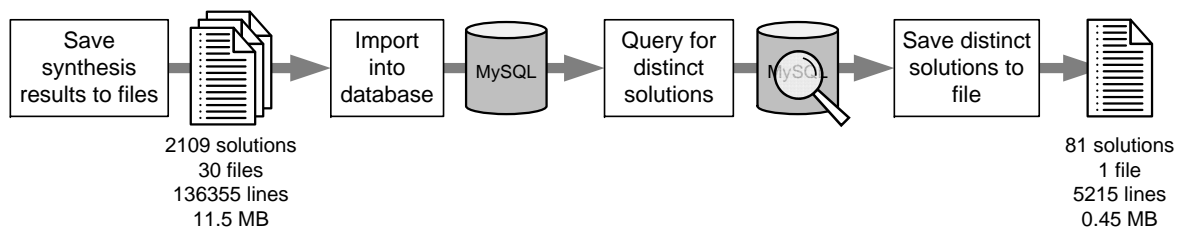

Figure 5-24: Solution post-processing approach using a MySQL database

Applying the synthesis procedure described in the rule sequence in Figure 5-17 results in the automated generation of result files. Memory overflows require executing the synthesis procedure for 2109 cabin layouts in 30 runs resulting in 30 separate result files each containing 70 – 90 layouts. For every individual layout, the result files contain all cabin items of the door compartment zones and all seat blocks of the passenger compartment zone. Each element is described with its x-/y-/z-coordinates, width, length,

type name and an Airbus-specific object ID. Thereby, the description of one layout spans on average 65 lines in a text file. The 2109 solutions generated, split up into 30 files, require 136355 lines and 11.5 MB. Analyzing such an amount of data, e. g. to detect distinct and/or identical solutions, requires a more advanced approach than using a simple spreadsheet software that is not designed for handling that amount of data and performing complex analyses. Handling big amounts of data is exactly the scope of application of databases.

The open-source, relational database system MySQL is used for this purpose. The database management and query tool MySQL Workbench[47] is freely available and allows to generate database tables from text file input. Once the data is in the database, SQL queries can be formulated to analyze the set of solutions. The query depicted in Figure 5-25 identifies distinct solutions through comparison of the cabin layout parameters of all solutions and generates an aggregated output stating the layout parameters for all distinct 81 solutions. To allow for further use of the layout data, one solution file is generated containing the 81 solutions with their individual cabin items. This file comprises essentially the same information as the initial 30 files but without redundant layouts. As it has only a size of 0.45 MB and contains 5216 lines, it can be analyzed in spreadsheet software without any problems.
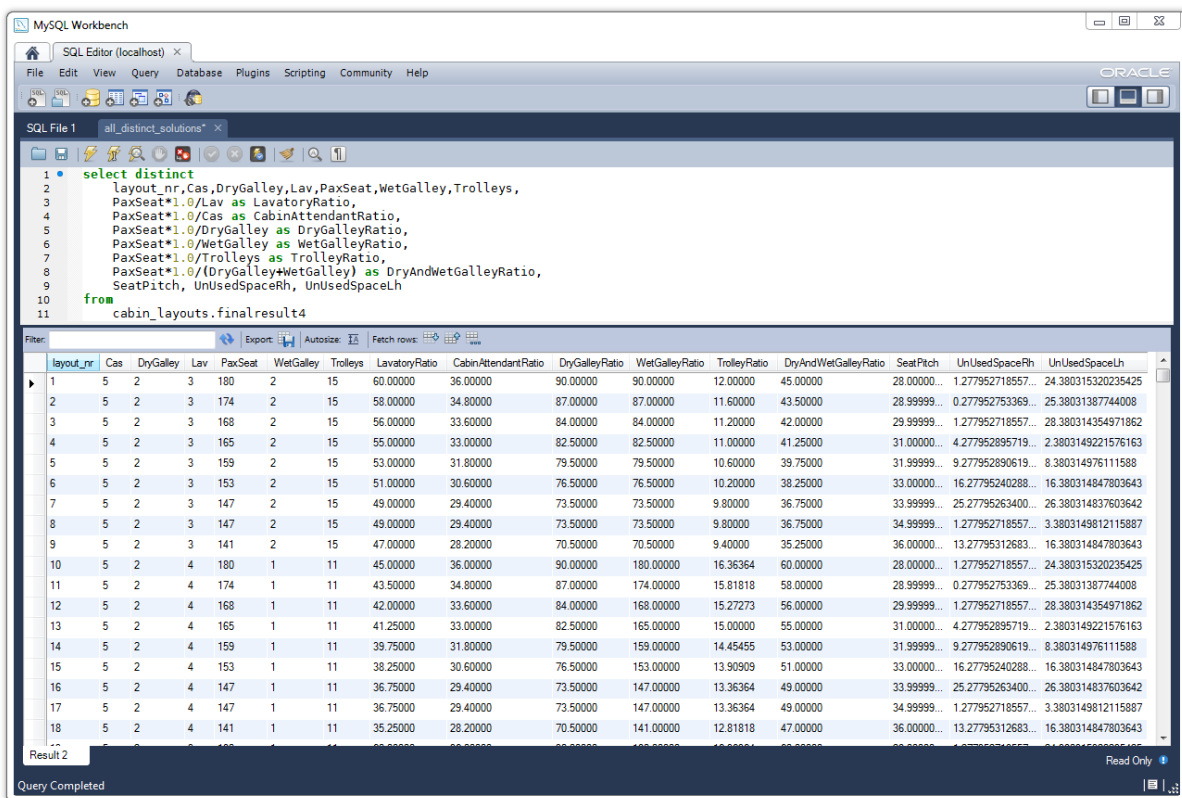

Figure 5-25: MySQL query generating an aggregated output of all distinct cabin layouts

The identification of identical solutions is based on the comparison of the cabin layout parameters. Figure 5-26 shows how often identical layouts occurred in the generated set of 2109 solutions. The first bar means that four different layouts occurred only once; the highest bar means that 15 layouts occurred that have 30 repetitions each. Although such a high number of layouts have been synthesized, it is noticeable that still four solutions occur only once and eight solutions occur twice. Hence, it is fair to conclude that

---

[47]Freely available at: www.mysql.de/products/workbench/

there may be more solutions that are not yet generated. Most layouts appear between 21 and 30 times and the average number of occurrence of identical layouts is 26. This outcome clearly shows the shortcoming of random walk search for design generation. To potentially find one more solution, a high number of redundant solutions has to be generated and one can never be sure that all solutions in the solution space are found. The question as to why the number of repetitive solutions is not equally distributed cannot be conclusively clarified. One explanation is that the 30 repetitions are related to the 30 synthesis runs that result in the generation of 30 files. The rules for checking the constraints may induce patterns in the generation that persist despite the randomization.
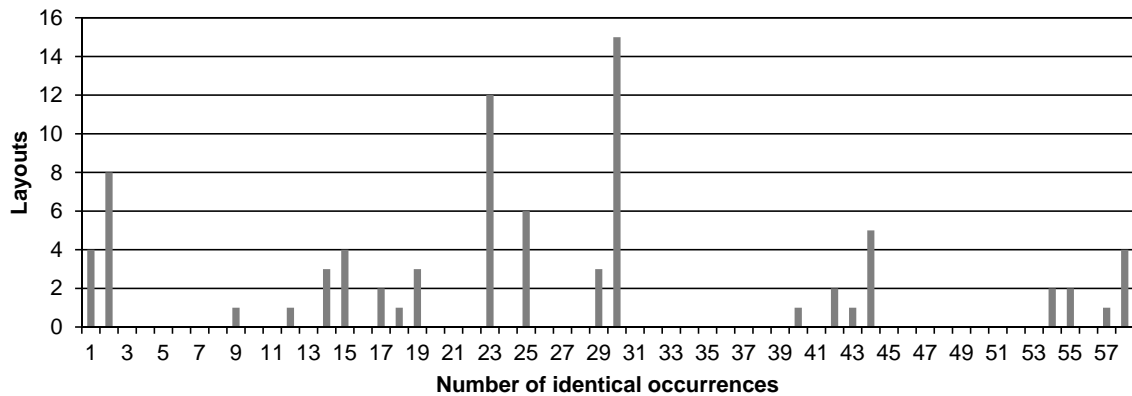
Figure 5-26: Repetitiveness of generated cabin layouts

The generated solutions cover a range of number of PAX seats from 141 to 180. Figure 5-27 shows the relation between the number of PAX seats and the mean values of service ratios and pitch. The average pitch value of the generated layouts decreases with an increasing number of PAX seats. This is simply because more seat rows have to be placed in the cabin leaving less space, i. e. pitch, for every row. Further, it can be said that the service ratios increase with increasing number of PAX seats. This observation is in accordance with the definition of *service ratio* on page 113; a higher number of passengers have to share the service facilities, which is expressed with a higher service ratio value. Further, with increasing number of seats also the available space for adding service facilities decreases.
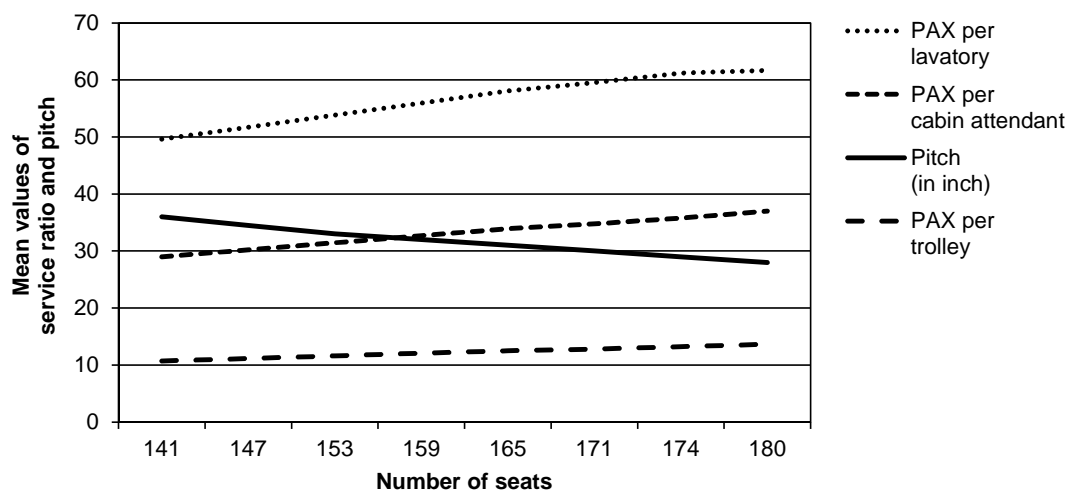

Figure 5-27: Relationship between number of PAX seats and mean values of service ratio and pitch

The generated cabin layout solutions can be classified according to the typical values of the layout pa-

rameters presented in Table 5-1 as shown in Table 5-2. The table reads as follows: *36 solutions are high density solutions with respect to their number of PAX seats, i. e. between 165 and 180.* Most of the layout parameters are within the bounds of high-density- or high-comfort-solutions. Regarding the service ratios, some solutions exceed these boundaries (*very high density* and *very high comfort*), e. g.: *18 solutions are very high density solutions with respect to the PAX per lavatory ratio, i. e. that value is higher than 60.* Due to validity checks, these solutions are not invalid but rather unconventional and might be worth further inspection by the cabin layout designer. Due to these checks, no solutions occur with more than 180 PAX seats as they would violate security regulations. Regarding the *PAX per cabin attendant ratio* it is striking that more than 80% of the solutions are out of the bounds of typical values. This number can lead to the conclusion that CAS are either over- or under-represented in the CIC definitions used. The fact that 18 solutions have a *PAX per trolley* ratio that is above the typical high comfort ratio leads to the conclusion of an over-representation of galleys in the CIC definition (trolleys are contained in galleys).

Table 5-2: Classification of the synthesized solutions according to the typical layout parameters presented in Table 5-1

	Number of solutions, classified according to layout parameters				
	Very high density	High density	both	High comfort	Very high comfort
Number of PAX seats	> 180 –	165 - 180 36	160 - 164 –	159 - 136 45	< 136 –
Pitch (in inch)	< 28 –	28 - 31 27		32 - 36 54	> 36 –
PAX per lavatory	> 60.0 18	55.0 - 60.0 20	55.1 - 52.9 –	45.3 - 53.0 25	< 45.3 18
PAX per trolley	> 20.8 –	14.3 - 20.8 20	13.2 - 14.2 12	10.4 - 13.1 24	< 10.4 25
PAX per cabin attendant	> 33.6 33		30.0 - 33.6 15		< 30.0 33

In general, it can be said that the random-based selection of CICs leads to unconventional solutions, as shown in Table 5-3. This table reads as follows: *Two solutions have five (x = 5) layout parameter values that are typical for conventional high density solutions.* Out of the 81 layouts, only four layouts are within the limits of all five typical layout parameters (left column, x = 5); 21 solutions have four typical layout parameters. Hence, although conventional solutions occur, the majority of the synthesized cabin layouts can be characterized by a mixture of high density and high comfort layout parameters.

Table 5-3: Distribution of solutions according to the occurrence of typical layout parameters presented in Table 5-1

	Number of solutions with x typical layout parameters				
	x = 5	x = 4	x = 3	x = 2	x = 1
High density	2	9	14	13	16
High comfort	2	12	22	17	17
	conventional				unconventional

In the following, three cabin layouts are discussed with their respective layout parameters and notable layout properties. For clarity, only the crucial physical items are displayed that are taken into account for the calculation of the layout characteristics enabling the comparison of cabin layouts.

Solution 46 contains 141 PAX seats with a pitch value of 36" and is a rather conventional high comfort

cabin layout; three layout parameters are within the bounds of typical high density layouts and two parameters (PAX per lavatory and PAX per cabin attendant) are even in the range of very high comfort cabin layouts. The asymmetry in the front area (Zone A) is noteworthy. It is due to an almost empty subzone A2_RH that only contains a partial passageway and an assist space but no other physical items. Because of that, the first row of subzone B1_RH is moved forward allowing to place one additional triple seat block on the right side.

Layout parameters:

- Number of PAX seats: 141   high comfort
- Pitch: 36"   high comfort
- PAX per lavatory: 35.25   very high comfort
- PAX per trolley: 12.81   high comfort
- PAX per cabin attendant: 28.2   very high comfort

Legend

- ASP: Assist space
- CAS: Cabin attendant seat
- DG3:   Dry galley with three trolleys
- DG4:   Dry galley with four trolleys
- LAV: Lavatory
- PPW: Partial passageway
- TSB: Triple seat block
- WG4:   Wet galley with four trolleys

Figure 5-28: Solution 46 – Rather conventional high comfort cabin layout

Solution 44 contains 180 PAX seats with a pitch value of 28" and is a conventional high density cabin layout; four layout parameters are within the bounds of typical high comfort layouts and one parameter (PAX per cabin attendant) is even in the range of very high density cabin layouts. However, adding one cabin attendant seat in subzone C1_LH would not require any other modifications in the layout and would shift that property into the high density bounds. It is remarkable in this solution that space remains unused with a depth of 34.08" at the end of Zone B. Although the unused space would suffice to add an additional row, i. e. two triple seat blocks, that would cause a violation of the maximum limit of 180 PAX seats and is therefore not done. However, that unused space could be used to add physical items, e. g. lavatories and galleys, to Zone A and to increase the service quality for the passenger.

Solution 72 can be seen as a solution that is similar to solution 44 but uses the unused space in Zone B for additional physical items in Zone A. Thereby, this layout shows some high density characteristics as it contains 180 PAX seats with a pitch value of 28". In contrast, the other service ratios are within the (very) high comfort bounds, whereas the PAX per cabin attendant value is the overlapping range between high density and high comfort. A similar layout could not be found in the cabin layout statistics of EADS and is, therefore, considered as an unconventional solution that might be worth further investigation.

5.6 Discussion

From the industrial perspective, the application of object-oriented graph grammars implemented in booggie was a success as outlined in the final project assessment in the Appendix 9.6. This is especially noteworthy considering the fact that this study was primarily carried out on the part of EADS with only
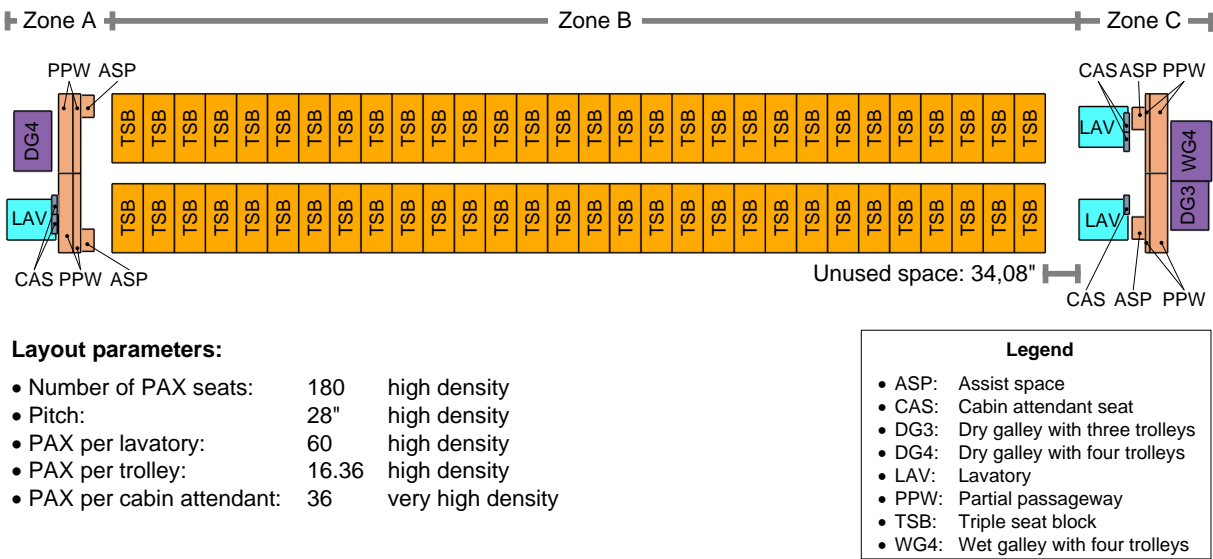
Zone A   Zone B   Zone C

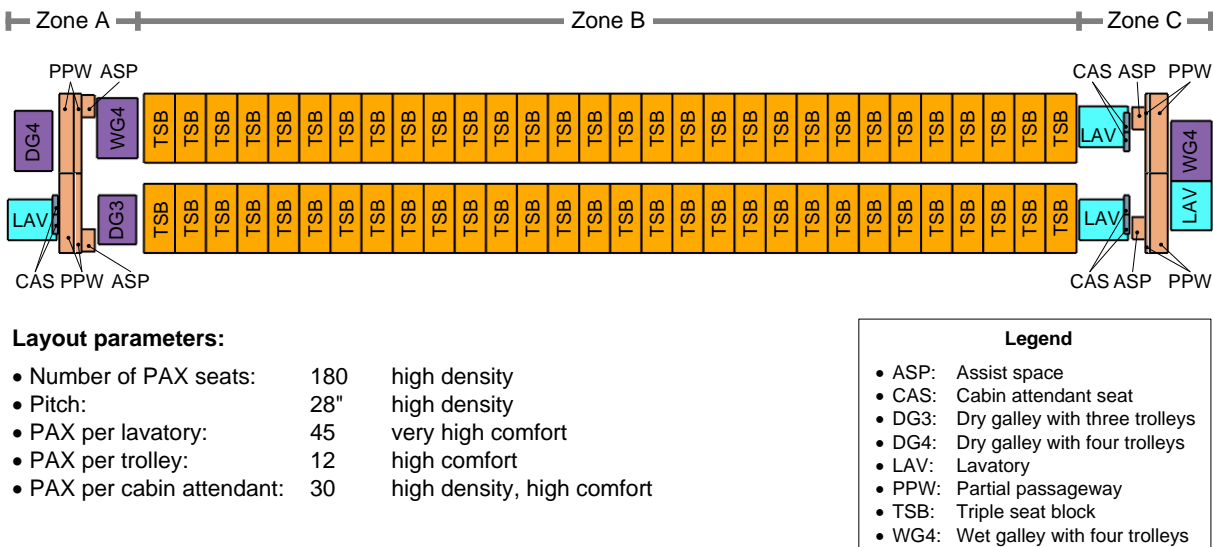Figure 5-29: Solution 44 – Conventional high density cabin layout

Layout parameters:

- Number of PAX seats: 180   high density
- Pitch: 28"   high density
- PAX per lavatory: 60   high density
- PAX per trolley: 16.36   high density
- PAX per cabin attendant: 36   very high density

Legend

- ASP: Assist space
- CAS: Cabin attendant seat
- DG3:   Dry galley with three trolleys
- DG4:   Dry galley with four trolleys
- LAV: Lavatory
- PPW: Partial passageway
- TSB: Triple seat block
- WG4:   Wet galley with four trolleys

Zone A   Zone B   Zone C

Figure 5-30: Solution 72 – Unconventional cabin layout with mixed layout characteristics

Layout parameters:

- Number of PAX seats: 180   high density
- Pitch: 28"   high density
- PAX per lavatory: 45   very high comfort
- PAX per trolley: 12   high comfort
- PAX per cabin attendant: 30   high density, high comfort

Legend

- ASP: Assist space
- CAS: Cabin attendant seat
- DG3:   Dry galley with three trolleys
- DG4:   Dry galley with four trolleys
- LAV: Lavatory
- PPW: Partial passageway
- TSB: Triple seat block
- WG4:   Wet galley with four trolleys

little support from the author. As shown in Table 5-3, the generation of human-competitive solutions is possible: Known solutions are generated that are within the bounds of conventional layout parameters as well as new, or unconventional, solutions that are worth further inspection. Thereby, this case study corroborates the achievement of contribution 5. However, there are still numerous aspects that can be included in the configuration of aircraft cabins, as for example:

- the placement of the on-wing emergency exits and the adjustment of the corresponding PAX seats.

- a differentiation between business class and economy class seats and an optimized distribution of the respective seat numbers.

- the consideration of more CICs.

In general, it can be said that this study can be further pursued by increasing the level of detail. Expanding the hierarchy of inheritance in the metamodel is the straightforward way to achieve a more fine-grained representation. For example, expanding the element type *PAXSeat* with the descendants *First*, *Business* and *Economy* would not require a reformulation of the existing rules. Another point mentioned in EADS' final project assessment is a further increase of expressiveness of the metamodel. The ultimate goal would be to cover the same features, as for example, in an OWL ontology and to allow the application of reasoners based on that. Logical constraints or geometrical dependencies, e. g. to forbid a galley being placed opposite to a seat row, can efficiently be represented using an ontology's descroption logic. Thereby, the constraints that are checked in the *Check... valid ?*-rules could be represented using the model-based representation itself. As a reply to that request, it can be said that ports could be used to incorporate some of these requirements. As discussed earlier, ports can be used to formulate topological constraints and can impose multiplicity constraints. Solving these constraints would require the integration of constraint solving algorithms, which is left for future work.

The software development of booggie strongly benefited from this case study as the software could be tested in a real industrial environment. Potential improvements and shortcomings were identified that had to be addressed in the development. Within the case study, all components and features of booggie were used and, thereby, tested in an industrial application context. All software requirements presented in Section 5.1 are implemented. Thus, it seems justifiable to state that the current state of booggie can be regarded as feature complete. A formal evaluation of the software, such as proposed by BRACEWELL & SHEA (2001), should be targeted in future work.

The fact that known and new meaningful solutions can be generated and that the software prototype is mature enough such that EADS engineers can work with it with little instruction, leads to the conclusion that the Expected Contributions 5 (human-competitive solutions) and 6 (mature software prototype) are achieved. Nevertheless, some shortcomings still persist in booggie and should be addressed in future work. To promote a wider application in industry, non-functional requirements regarding the stability and performance are essential. The most important issues are briefly discussed in the following.

While generating solutions with large graphs, e. g. aircraft cabins, booggie runs out of memory. This issue can be seen from two perspectives and both should be taken into consideration for future work. First, booggie cannot be executed as 64-bit software, which limits the potential available working memory to 4 GB. Regarding the overhead for the operating system and required auxiliary software, approximately 2.5 GB should be available. However, booggie crashes typically when $1.4 - 1.6$ GB of the working memory is used. Second, booggie's memory usage should be decreased. Especially the internal graph representation can be reduced.

Representing the connection of two elements using ports requires four graph nodes (two elements and two ports) and three edges. Instead, the use of typed edges, which would contain the same information as represented in the ports, would only require two nodes for the elements themselves and one edge. The amount of overhead that is induced by this representation can be demonstrated with the cabin layout in 5-23: This layout consists of 377 elements and 377 relations representing the essential information of this design. In order to adopt the port-based representation, 1143 nodes for representing port and 1143 *DJoint-* and *UJoint*-edges are required. Hence, to capture the essential 754 elements, 3040 elements are required. The following steps are advised to tackle this issue: As a short-term solution, the compilation of booggie as 64-bit software and the investigation of memory leaks should be addressed. A more substantial improvement could be achieved by redeveloping the applied port-based graph data structure.

It should be possible to exhibit the same expressiveness as formally described for the booggie modeling language in Section 5.2, using typed edges instead of pushing up the number of graph elements by using a node for every port.

Support for the visual definition of graph transformation rules, as demanded in SWReq 2 and SWReq 6 is implemented in the Rules perspective. This feature allows to apply booggie in the wider engineering domain where designers are lacking the knowledge and motivation to use a programing language for defining rules. So far, the Rules perspective only supports the definition of basic rule features. A high priority should be placed on the further development of this functionality. It is the main point where intuitive software support can deliver a considerable contribution to bring this approach to a larger user group. It is suggested to investigate the approach of story diagrams as presented by FISCHER ET AL. (2000) as an alternative visualization approach for representing rules and rule sequences as flow charts. Nevertheless, a tremendous effort is required to consider all features of the applied rule language in a visual rule interpreter.

6 Discussion and future work

The research presented in this thesis contributes to the goal of the CDS research community to bring CDS approaches into "use in every day design practice" (CHAKRABARTI ET AL., 2011). As discussed in Chapter 1, the conceptual design phase could benefit from the support of CDS methods since it is characterized by high uncertainty and requires a continuous adaption and modification of design artifacts, such as product models. The promise of current CDS research complies to a large extent with the needs for computational support in the conceptual design phase. Nevertheless, it can be stated that the acceptance of this research field, especially in an industrial context, is still low.

The reasons for the weak dissemination of CDS research for supporting the conceptual design phase are summarized in five research issues in Section 1.1. These are the motivating factors to conduct this research. The principal reasons are the inefficiency of knowledge formalization (Research Issue 1) requiring a high effort to transfer expert knowledge into the computer. Research Issue 2 states that the limited scope of application impedes tackling full scale industrial problems. Research Issue 3 indicates that existing design knowledge and methods are not reused so that using CDS approaches starts with creating a new process and with the formalization of first principles instead of building on previous efforts. Fourth, the lack of using modeling standards and tool integration corroborates the weak dissemination as it hinders integration of CDS prototype software in tool chains and development processes (Research Issue 4). Fifth, as CDS implementations are primarily proof-of-concept prototypes, their maturity is low and not suited for industrial applications (Research Issue 5).

In the following, the contributions of this research are summarized that aim to address these research issues and to achieve the research goals as discussed in Section 1.2. Further, limitations and opportunities for improvement and future work are also discussed.

6.1 Research contributions

It can be seen in the evolution of expert systems that formalizing knowledge using a hybrid formalization, that builds on model-based and rule-based paradigms, provides advantages in terms of the efficiency and effectiveness of the knowledge representation. Such a representation has been developed in the approach of object-oriented graph grammars as presented in Chapter 3. Contribution 1 is achieved as object-oriented graph grammars are more efficient than previous knowledge representations in CDS since they:

- allow the formulation of generic design rules and therefore keep the rule set small and manageable.

- are based on a clear and simple process for the formalization of engineering knowledge and provide guidance for the human designer to use this approach.

- incorporate a flexibly expandable metamodel for the formalization of evolving engineering knowledge.

- build on graph transformations for the generation and modification of models. This method is computationally efficient and serves as a foundation for developing elaborate search and optimization algorithms.

Graph-based models are widespread in the engineering domain. Hence, building on typed graphs as an underlying model representation, the developed approach of object-oriented graph grammars is an effective means to represent knowledge for a wide range of applications and for multiple levels of abstraction. The effectiveness of the representation is shown in the validation study in the Section 3.3 and validates Contribution 2 (high effectiveness of object-oriented graph grammars). As discussed in the review of previous work in Chapter 2.4, this is a key contribution compared to previous work.

This hybrid representation is inspired by paradigms from object-oriented programming. As discussed in Section 3.2, some of the advantages of object-oriented programming, compared to imperative programming, can analogously be seen here in terms of extendibility, ease of use, efficiency, reusability and compatibility.

To avoid the necessity of formalizing design knowledge from first principles, a method is developed (presented in Chapter 4) that supports automatically formalizing knowledge from design catalogs, which contain the description of physical effects using equation-based representations and constitutes Contribution 3 (formalization of design catalogs). The method of automatically assigning abstraction ports to physical effects based on their equation structure also contributes to the goal of making knowledge formalization more efficient (Research Goal 1). This approach is validated with the formalization of two design catalogs and a simple software prototype.

The software prototype booggie implements the method of object-oriented graph grammars based on a formal modeling language that is mathematically defined and described in Section 5.2. Formal modeling languages that include a metamodel for the definition of the modeling elements have the advantage that they can be transferred into other representations. This approach, called model transformation, serves as foundation for realizing a mapping to other modeling languages, e. g. the increasingly important language SysML, and for interfacing with other tools, e. g. simulation tools. Thereby, Contribution 4 is accomplished.

Using previous work, e. g. design catalogs and software libraries, not only increases the efficiency for formalizing knowledge but also helps to develop mature software. Previous CDS implementations are typically built from the ground up and often do not leave the prototype stage. booggie, in contrast, is built on two open-source libraries for graph transformation and graph visualization and constitues Contribution 6. The development of a tool that exhibits such high performance in graph transformation and in 3D-graph rendering would not have been possible within this limited time frame and shows the potential of open-source software.

This research has been validated with two case studies: the synthesis of product architectures for hybrid automotive powertrains and the synthesis of aircraft cabin layouts. The developed method, implemented in the software prototype booggie, allows systematic exploration of the solution space. This serves primarily two purposes: first, to gain an overview of known and potentially new solutions; second, to stimulate the designer to think outside of the box and to promote thinking about new possibilities. Therefore, it can be said that the developed approach supports generation of human-competitive solutions, cf. Contribution 5, and has successfully been applied to industrial design problems.

6.2 Future work

Limitations of the developed approaches are discussed at the end of Chapters 3, 4 and 5. Nevertheless, overarching issues are briefly discussed in this section.

This research focused on the modeling, synthesis and evaluation of graph-based topologies. Parametric characteristics of these topologies and of their individual components were out of the scope of this thesis. For example in the hybrid powertrain case study in Section 3.3, the conversion of rotational speed and torque, which is, e. g., inevitable when using a combustion engine, is disregarded on all abstraction levels and leads to the absence of a gearbox. The lack of considering quantitative aspects represents a key limitation of this work and leaves room for future work. Extending the current work in this direction can be divided into three categories: modeling, synthesis and evaluation.

Integrating parameters into the metamodel to support parametric modeling, as depicted in Figure 3-6, and making them available for manually modeling product architectures does not pose any problems, as the modeling of various kinds of value types is fully supported. Thus, components could be quantitatively characterized assigning parameters and concrete values to them. For example, all components could inherit by default from a generic *Volume* element type that comprises parameters such as volume, mass, density and center of gravity. Physical effects could also be characterized by their typical operation ranges, e. g. a power limit.

Also with respect to the method presented in Chapter 4, quantitative characteristics of the physical effects are not considered. The geometric scalability of a physical effect, for example, is disregarded. Further, the detection of an effect's dominance in the case that multiple abstraction ports are assigned, e. g. for the piezo effect, is not supported. However, in the design process step of assigning physical effects to functions, the aim is to investigate the solution space for possible alternatives (PONN & LINDEMANN, 2011) and to consider quantitative characteristics as subordinated. The approach shown provides for this goal a foundation for a computational approach.

Regarding the synthesis of models with quantitative characteristics, object-oriented graph grammars can serve as a foundation for modeling and solving constraint satisfaction problems (CSP). As discussed in Section 2.3.2, the algorithmic stream of CSP research shows potential for combining a graph-based representation with constraint modeling. Research efforts in that direction are under way to combine object-oriented graph grammars and CSP to automatically generate parameterized designs (MÜNZER ET AL., 2012). This shows the potential of this approach; it also uses hybrid automotive powertrains as the application scenario. It is planned to further develop this research towards a CSP library for booggie.

Once parametric models are generated, the automated simulation is an obvious next step for the quantitative assessment of product architectures. The approach of model transformation can be integrated to transform a product architecture into a simulation model. Figure 6-1 shows the transformation of a parallel hybrid powertrain product architecture model into the respective simulation model. A graph-based, or block-oriented, modeling approach can be used such as applied in the simulation language MODEL-ICA[48] or in the simulation tool AMESim[49]. The advantage in doing so is due to the fact that graph-based simulation models can be synthesized computationally, based on simulation knowledge formalized in

---

[48]Website of the MODELICA association: http://www.modelica.org
[49]Company website: http://www.lmsintl.com/imagine-amesim-suite

graph grammars. The advantages for the conceptual design phase could be immense, e. g. synthesizing fully parameterized solutions in response to changing customer requirements or investigating new product architectures in an early stage.
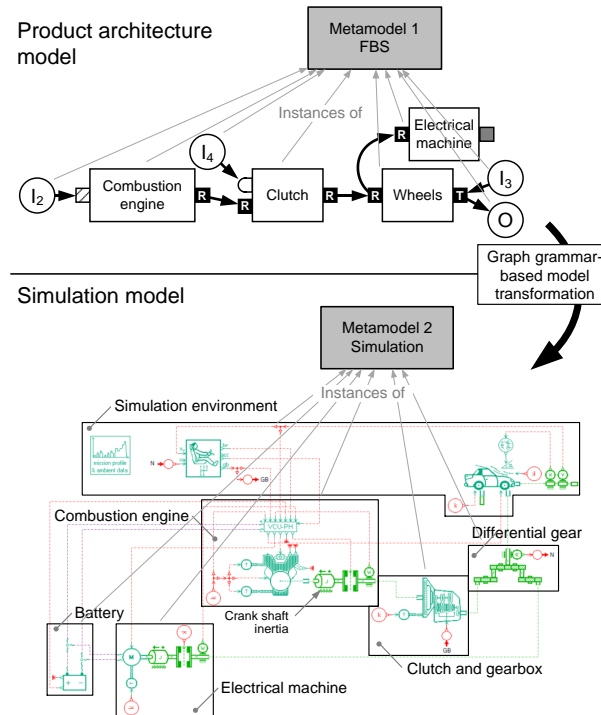

Figure 6-1: Model transformation of a product architecture into a simulation model, e. g. in AMESim

The following steps have to be performed to establish an integration of simulation into the synthesis of product architectures:

1. *Definition of a simulation metamodel:* Just as all modeling elements for synthesizing a FBS* product architecture are defined in the metamodel, cf. Figure 3-6, the modeling elements for creating simulation models have to be defined in a simulation metamodel. Existing libraries of simulation tools and languages are structured in a modular way. Therefore, the individual element types and their potential connections, i. e. port types, can be identified and a metamodel can be created in the applied representation.

2. *Definition of model transformation rules:* Once metamodels for both modeling domains (FBS* and simulation) are defined, rules can be formulated that capture the knowledge for mapping components of the product architecture model to the simulation model. This is primarily the part where expertise about the generation of simulation models is required. For example, the simulation of a combustion engine does not only require one single simulation block but various elements for taking the physical properties into account, e. g. the inertia of the crank shaft, see Figure 6-1. Further, the environment for running a simulation study can be generated based on the definition in a graph grammar rule, such as a driving cycle or vehicle properties. These rules provide the model transformation rule set.

3. *Tool integration:* Based on the application of model transformation rules, all the content required for running simulation studies is contained in the model. However, no tool would be able to run a simulation because the model is not represented in a compatible data format. Fortunately, current

simulation tools are equipped with application programming interfaces (API) and can hence be accessed from the outside. This can be used for an integration with booggie.

Previous work on the automated composition of mechatronic simulation models shows the feasibility of the outlined approach. Eigner & Zafirov (2009) describe a method to automatically derive simulation models out of function-based descriptions of production facilities. Paredis et al. (2001) adopt a port-based approach to compose simulation models taking geometric information from CAD models into consideration.

To extend the scope of industrial applications but also to integrate additional domains, e. g. service, aspects from model-based systems engineering should be considered. The link to the formal modeling language SysML (Wölkl & Shea, 2009; Kruse et al., 2012) could provide straightforward integration with advanced modeling environments. Interfacing with other disciplines, e. g. requirements engineering. In this case, the modeling foundation of SysML is an underlying graph-based representation.

The integration of CSP, simulation and SysML presents primarily a challenge in the conversion of models, i. e. the transformation of model representations. The ability to convert between modeling languages is a basic prerequisite for any implementation issue, such as tool integration, to be tackled. The implementation of model transformation approaches calls for a formal language definition. Concerning this matter, the formal definition of the booggie modeling language is presented in Section 5.2.

The approach of object-oriented graph grammars directly led to the implementation of booggie. The validation study, the synthesis of hybrid powertrains, has also been realized based on booggie. In contrast, the method for automatically assigning abstraction ports to functions and physical effects is not yet incorporated in booggie. A comprehensive validation study bringing together the results described in the Chapters 3, 4 and 5 could not have been accomplished within the scope of this thesis. However, the method is compatible and the implementation is straightforward as abstraction ports are just another type of port. The automated assignment of abstraction ports to functions and physical effects could be understood as a pre-processing of the metamodel. Thus, the definition of physical effects as modeling elements in the metamodel could be automated and could guarantee that the metamodel is kept in sync with an equation-based description of available physical effects. Implementing this as an add-on to booggie would increase the versatility of booggie as a product concept generation tool.

This research combines a rule-based representation with a model-based presentation. Further extensions to the model-based representation can also be considered. One direction is an increase of expressiveness of the metamodel and the further development towards the integration of description logics. Logical reasoners could be applied to answer questions like *Are there any solutions for a given port-matching problem?* and could answer them with certainty. Ports are a small, first step in that direction but using formal ontology languages could allow capturing logical relations on a more expressive foundation that is compatible with logical reasoners. The challenge here is to establish consistency and interoperability between the grammar-oriented metamodel representation and an ontology-based representation. If this is accomplished, logical reasoners could be an alternative, or complementary, approach to graph transformation. Current research efforts are targeted at solving port-matching problems not based on graph transformation rules but interpreting them as satisfiability problems. Hence, this can be solved using so-called SAT solvers. First results (Münzer, 2010b) show that this is a promising step to strengthen the model-based side of the hybrid knowledge representation.

As a final conclusion, it is expected that graph-based representations will continue to play a major role in engineering design and especially in the conceptual design phase. The trend towards products covering multiple domains continues and requires domain-neutral modeling approaches. Further, the research field of model-based systems engineering is constantly gaining significance as a response to increasing complexity of products and development processes. It emphasizes the use of standard modeling languages, such as SysML, or domain-specific languages, which all build on graph-based representations. On the other hand, computational support for handling graphs will also increase as there is a significant need to efficiently analyze massive graph structures, e. g. derived from social networks. In the future, a large fraction of CDS research should build on graph-based representations. Rapid progress can only be expected if the efforts from the field of computer science, i. e. libraries, tools, algorithms, are leveraged.

As discussed in Section 2.5, the current state of the art in CDS research is reminiscent of the situation in artificial intelligence research during the transition of the first to the second generation of expert systems in the early 1980s. After the deficiencies of the first expert system generation were remedied, continued success of the second generation could be observed. The vision for CDS researchers is to draw on the lessons learned from expert systems, to overcome the shortcoming of the first generation of CDS approaches and to build a solid foundation for the development of the second generation of CDS. The consideration of this historical analogy and the fact that constantly evolving computational technologies are freely available to develop powerful CDS software, should make CDS researchers confident that the ambition of bringing CDS approaches into "use in everyday design practice" (CHAKRABARTI ET AL., 2011) is achievable.

7 Conclusion

Computational Design Synthesis (CDS) aims to support product development through formalization and automation of knowledge-intensive design tasks. Especially the conceptual design phase could profit from CDS approaches as this phase is characterized by high uncertainty making it difficult to evaluate design quality and to systematically explore the set of solutions. However, CDS still has little acceptance in industry primarily due to the high effort required for knowledge and task formalization, the limited scope of application of CDS approaches, the lack of reuse of existing paper-based design knowledge, the lack of modeling standards and tool integration, and a low maturity of software tools. The promising opportunities of CDS to increase development efficiency and innovative power provide the motivation to address these problems in this thesis and to contribute to the goal of increasing the applicability of CDS in every day design practice.

A historical analogy to the development of expert systems leads to the hypothesis that a combination of rule- and model-based knowledge formalization is beneficial to address these research issues. This assumption is corroborated with a review of the related state of the art coming to the conclusion that such a hybrid knowledge can combine the advantages of rule- and model-based representations.

Chapter 3 introduces the approach of object-oriented graph grammars for the computational synthesis of graph-based product architectures using a Function-Behavior-Structure representation. It allows to capture declarative engineering knowledge upfront in a port-based metamodel and to formulate generic, procedural design rules in a graph grammar. The contributions of increasing efficiency and effectiveness of knowledge formalization are validated through the synthesis of product architectures of automotive hybrid powertrains. A solution space with 3360 powertrain architectures is generated and noteable solutions are identified through the search for specific solution characteristics, e. g. serial or parallel configurations.

Chapter 4 presents an approach to characterize and integrate physical effects contained in design catalogs. Through an automated analysis of the equation structure, abstraction ports are assigned to physical effects and represent valid mappings between functional operators and physical effects. The contribution regarding the formalization of paper-based engineering knowledge is validated with the formalization of 137 physical effects of two design catalogs and the development of a software prototype that searches suitable physical effects for a given function.

The implementation of the object-oriented graph grammar approach in the software booggie is presented in Chapter 5. This software is built on a formal language definition (the booggie modeling language) that represents the foundation for achieving the contributions of model transformation and tool integration. booggie is a modular, open-source and platform-independent software that incorporates an efficient graph transformation library and a 3D graph visualization library. The automated generation of aircraft cabin layouts validates the practical usability of the developed software platform in an industrial context and illustrates the contributions discussed in the previous chapters.

The expected contributions are achieved leaving, nevertheless, potential for improvements and future work. Besides an increase of the software maturity, future work should include an integration of object-oriented graph grammars and the automated assignment of abstraction ports in one implementation,

the extension of the synthesis of product architectures towards parametric synthesis and evaluation using simulation, model transformation for the transformation towards other modeling languages, e. g. SysML, and the integration of SAT solvers to enable the solution of logical port-matching problems.

8 References

AAMODT & PLAZA 1994
  Aamodt, A.; Plaza, E.: Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches. Ai Communications 7 (1994) 1, pp. 39–59. ISSN: 09217126.

AGARWAL & CAGAN 1998
  Agarwal, M.; Cagan, J.: A blend of different tastes: the language of coffeemakers. Environment and Planning B: Planning Design 25 (1998) 2, pp. 205–226. ISSN: 02658135.

AGARWAL ET AL. 2000
  Agarwal, M.; Cagan, J.; Stiny, G.: A micro language: generating MEMS resonators by using a coupled form - function shape grammar. Environment and Planning B: Planning and Design 27 (2000) 4, pp. 615–626. ISSN: 0265-8135.

AGU & CAMPBELL 2010
  Agu, D. I.; Campbell, M. I.: Automated Analysis of Product Disassembly to Determine Environmental Impact. International Journal of Sustainable Design in press (2010) 3, pp. 241–256. ISSN: 17438284.

AHMED ET AL. 2005
  Ahmed, S.; Kim, S.; Wallace, K. M.: A Methodology for Creating Ontologies for Engineering Design. In: Volume 3: 25th Computers and Information in Engineering Conference, Parts A and B, vol. 2005, pp. 739–750. ASME 2005. ISBN: 0-7918-4740-3. ISSN: 15309827.

ALBER & RUDOLPH 2003
  Alber, R.; Rudolph, S.: 43"-A generic approach for engineering design grammars. In: AAAI 2003 Spring Symposium. Palo Alto, USA 2003.

ALTHOFF ET AL. 1989
  Althoff, K.; Kockskämper, S.; Traphöner, R.; Wernicke, W.; Faupel, B.: Knowledge acquisition in the domain of CNC machine centers; the MOLTKE approach. In: Boose, J.; Gaines, B.; Ganascia, J.-G. (Eds.): EKAW-89; Third European Workshop on Knowledge-Based Systems, pp. 180–195. Paris, France 1989.

ALTSHULLER 1984
  Altshuller, G. S.: Creativity as an Exact Science. Amsterdam, The Netherlands: Gordon and Breach 1984. ISBN: 0-677-21230-5.

ANDREASEN 1992
  Andreasen, M. M.: Designing on a 'Designers Workbench' (DWB). Proceedings of the 9th WDK Workshop (1992).

ANDREASEN & HEIN 1987
  Andreasen, M. M.; Hein, L.: Integrated product development, Sadhana (India), vol. 22. Berlin, Germany: Springer 1987. ISBN: 3-540-16679-3.

ANTONSSON & CAGAN 2001a

Antonsson, E. K.; Cagan, J. (Eds.): Formal Engineering Design Synthesis. Cambridge, UK: Cambridge University Press 2001. ISBN: 0-521-79247-9.

ANTONSSON & CAGAN 2001b

Antonsson, E. K.; Cagan, J.: Introduction. In: Antonsson, E. K.; Cagan, J. (Eds.): Formal Engineering Design Synthesis. Cambridge, UK: Cambridge University Press 2001. ISBN: 0-521-79247-9.

BAADER 2010

Baader, F.: Basic Description Logics. In: Baader, F.; Calvanese, D.; McGuinness, D. L.; Nardi, D.; Patel-Schneider, P. F. (Eds.): The Description Logic Handbook, chap. 2, pp. 47–104. Cambridge, UK: Cambridge University Press 2010, 2nd edition. ISBN: 978-0-521-15011-8.

BAADER ET AL. 2010

Baader, F.; Küsters, R.; Wolter, F.: Extensions to Description Logics. In: Baader, F.; Calvanese, D.; McGuinness, D. L.; Nardi, D.; Patel-Schneider, P. F. (Eds.): The Description Logic Handbook, chap. 6, pp. 237–282. Cambridge, UK: Cambridge University Press 2010.

BARLETTA & HENNESSY 1989

Barletta, R.; Hennessy, D.: Case adaptation in autoclave layout design. In: Hammond, K. (Ed.): Proceedings of the DARPA CaseBased Reasoning Workshop, pp. 203–207. DARPA, Morgan Kaufmann 1989.

BARR & FEIGENBAUM 1982

Barr, A.; Feigenbaum, E. A. (Eds.): The Handbook of Artificial Intelligence, vol. 18. Los Altos, CA, USA: Kaufmann 1982.

BEIERLE & KERN-ISBERNER 2008

Beierle, C.; Kern-Isberner, G.: Methoden wissensbasierter Systeme. 4th edition, Wiesbaden, Germany: Vieweg+Teubner Verlag 2008. ISBN: 3834805041.

BERKOVICH ET AL. 2009

Berkovich, M.; Leimeister, J. M.; Krcmar, H.: An empirical exploration of requirements engineering for hybrid products. In: XVIIth European Conference on Information Systems, ECIS 2009. Verona, Italy 2009.

BETTIG & M. HOFFMANN 2011

Bettig, B.; M. Hoffmann, C.: Geometric Constraint Solving in Parametric Computer-Aided Design. Journal of Computing and Information Science in Engineering 11 (2011) 2, p. 021001. ISSN: 15309827.

BOLOGNINI ET AL. 2007

Bolognini, F.; Seshia, A. A.; Shea, K.: Exploring the Application of Multidomain Simulation-based Computational Synthesis Methods in MEMS Design. In: Bocquet, J.-C. (Ed.): International Conference on Engineering Design, ICED '07, pp. 81–82. Glasgow, United Kingdom: The Design Society 2007.

BOOLOS ET AL. 2007

Boolos, G. S.; Burgess, J. P.; Jeffrey, R. C.: Computability and Logic. 5th edition, New York, NY, USA: Cambridge University Press 2007. ISBN: 9780521701464.

BORGIDA ET AL. 1989

Borgida, A.; Brachman, R. J.; McGuinness, D. L.; Resnick, L. A.: CLASSIC: a structural data model for objects. ACM SIGMOD Record 18 (1989) 2, pp. 58–67. ISSN: 01635808.

BRACEWELL & SHEA 2001

Bracewell, R. H.; Shea, K.: CaeDRe: A product platform to support creation and evaluation of advanced computer-aided engineering tools. In: International Conference on Engineering Design, ICED '01, pp. 539–546. Glasgow, UK 2001.

BRACHMAN & SCHMOLZE 1985

Brachman, R. J.; Schmolze, J. G.: An overview of the KL-ONE Knowledge Representation System. Cognitive Science 9 (1985) 2, pp. 171–216. ISSN: 03640213.

BROENINK 1999

Broenink, J.: Introduction to Physical Systems Modelling with Bond Graphs. SiE Whitebook on Simulation Methodologies (1999), pp. 1–31.

BRUEGGE & DUTOIT 2010

Bruegge, B.; Dutoit, A. H.: Object Oriented Software Engineering. Internatio edition, Upper Saddle River, NJ, USA: Pearson Education, Inc. 2010. ISBN: 978-0138152215.

BRYANT ET AL. 2006

Bryant, C. R.; McAdams, D. A.; Stone, R. B.; Kurtoglu, T.; Campbell, M. I.: A validation study of an automated concept generator design tool. In: ASME 2006 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference, IDETC/CIE 2006, pp. 1–12. Philadelphia, USA 2006.

CAGAN 2001

Cagan, J.: Engineering Shape Grammars. In: Antonsson, E. K.; Cagan, J. (Eds.): Formal Engineering Design Synthesis, chap. 3, pp. 65–92. Cambridge, UK: Cambridge University Press 2001. ISBN: 0-521-79247-9.

CAGAN ET AL. 2005

Cagan, J.; Campbell, M. I.; Finger, S.; Tomiyama, T.: A Framework for Computational Design Synthesis: Model and Applications. Journal of Computing and Information Science in Engineering 5 (2005) 3, pp. 171–181. ISSN: 15309827.

CAGAN & VOGEL 2001

Cagan, J.; Vogel, C. M.: Creating Breakthrough Products: Innovation from Product Planning to Program Approval. Upper Saddle River, NJ, USA: Prentice Hall 2001. ISBN: 978-0139696947.

CAMPBELL ET AL. 2000

Campbell, M. I.; Cagan, J.; Kotovsky, K.: Agent-Based Synthesis of Electromechanical Design Configurations. Journal of Mechanical Design 122 (2000) 1, pp. 61–69. ISSN: 10500472.

CARBONELL 1983

Carbonell, J. G.: Learning by Analogy: Formulating and Generalizing Plans from Past Experience. In: Michalski, R. S.; Carbonell, J. G.; Mitchell, T. M. (Eds.): Machine Learning An Artificial Intelligence Approach, no. CMU-CS-82-126 in Machine learning: An artificial intelligence approach, pp. 137–161. Tioga 1983.

CHAKRABARTI 2002
Chakrabarti, A. (Ed.): Engineering Design Synthesis. London: Springer 2002. ISBN: 1852334924.

CHAKRABARTI & BLIGH 1994
Chakrabarti, A.; Bligh, T. P.: An Approach to Functional Synthesis of Solutions in Mechanical Conceptual Design. Part I: Introduction and Knowledge Representation. Research in Engineering Design (1994) 6, pp. 127 – 141.

CHAKRABARTI ET AL. 2011
Chakrabarti, A.; Shea, K.; Stone, R. B.; Cagan, J.; Campbell, M. I.; Hernandez, N. V.; Wood, K. L.; Vargas-Hernandez, N.: Computer-Based Design Synthesis Research: An Overview. Journal of Computing and Information Science in Engineering 11 (2011) 2, pp. 021003–1 – 021003–10. ISSN: 15309827.

CHOMSKY 1957
Chomsky, N.: Syntactic Structures. Language 33 (1957) 3, pp. 375–408. ISSN: 00978507.

CHRISTENSEN 2006
Christensen, J. F.: Wither Core Competency for the Large Corporation in an Open Innovation World? In: Chesbrough, H. W.; Vanhaverbeke, W.; West, J. (Eds.): Open Innovation, chap. 3. New York, NY, USA: Oxford University Press 2006.

CLANCEY 1985
Clancey, W. J.: Heuristic classification. Artificial Intelligence 27 (1985) 3, pp. 289–350. ISSN: 00043702.

COOPER & EDGETT 2005
Cooper, R. G.; Edgett, S. J.: Lean, Rapid, and Profitable New Product Development. Ancaster, Canada: Product Development Institute 2005. ISBN: 0973282711.

CORRADINI ET AL. 1997
Corradini, A.; Montanari, U.; Rossi, F.; Ehrig, H.; Heckel, R.; Löwe, M.: Algebraic approaches to graph transformation, Part I: Basic concepts and double pushout approach. In: Rozenberg, G. (Ed.): Handbook of Graph Grammars and Computing by Graph Transformation - Foundations, vol. 1, pp. 163–245. Singapore: World Scientific 1997. ISBN: 981-02-2884-8.

CRAWLEY ET AL. 2004
Crawley, E.; de Weck, O.; Eppinger, S. D.; Magee, C.; Moses, J.; Seering, W.; Schindall, J.; Wallace, D.; Whitney, D. E.: The influence of architecture in engineering systems, 2004.

DAHL ET AL. 1970
Dahl, O.-J.; Myhrhaug, B.; Nygaard, K.: The Simula-67 Common Base Language. Tech. Rep. S-22, Norwegian Computer Center, Oslo, Norway, 1970.

DESCOTTE & LATOMBE 1981
Descotte, Y.; Latombe, J. C.: GARI: A Problem Solver that Plans How to Machine Mechanical Parts. In: Proc Seventh International Joint Conf Artif Intel, pp. 766–772. 1981.

DYM & LEVITT 1991
Dym, C. L.; Levitt, R. E.: Knowledge-Based Systems in Engineering, Lecture Notes in Computer Science, vol. 5178. New York, NY, USA: McGraw-Hill 1991. ISBN: 0070185638.

EDER & HOSNEDL 2008

Eder, W. E.; Hosnedl, S.: Design Engineering a Manual for Enhanced Creativity. Boca Raton: CRC Press 2008. ISBN: 9781420047653.

EHRLENSPIEL 2009

Ehrlenspiel, K.: Integrierte Produktentwicklung. 4th edition, München, Germany: Hanser 2009. ISBN: 3446420134.

EHRLENSPIEL ET AL. 2007

Ehrlenspiel, K.; Kiewert, A.; Lindemann, U.: Kostengünstig Entwickeln und Konstruieren. 6th edition, Berlin, Germany: Springer 2007. ISBN: 978-3540742227.

EIGNER ET AL. 2012

Eigner, M.; Anderl, R.; Stark, R.: Interdisziplinäre Produktentstehung. In: Anderl, R.; Eigner, M.; Sendler, U.; Stark, R. (Eds.): Smart Engineering, chap. 2. Berlin, Germany: Springer 2012. ISBN: 978-3642293719.

EIGNER & ZAFIROV 2009

Eigner, M.; Zafirov, R.: Function modelling for efficient generation of mechatronic simulation models of automated production installations. In: International Conference on Collaborative Mechatronic Engineering, ICCME '09. Salzburg, Austria 2009.

ELCOCK 1990

Elcock, E. W.: Absys: the first logic programming language –A retrospective and a commentary. The Journal of Logic Programming 9 (1990) 1, pp. 1–17. ISSN: 07431066.

ERDEN ET AL. 2008

Erden, M.; Komoto, H.; van Beek, T.; D'Amelio, V.; Echavarria, E.; Tomiyama, T.: A review of function modeling: Approaches and applications. Artificial Intelligence for Engineering Design, Analysis and Manufacturing 22 (2008) 02, pp. 147–169. ISSN: 0890-0604.

FELFERNIG & SCHUBERT 2011

Felfernig, A.; Schubert, M.: Personalized diagnoses for inconsistent user requirements. Artificial Intelligence for Engineering Design, Analysis and Manufacturing 25 (2011) 02, pp. 175–183. ISSN: 0890-0604.

FELFERNIG ET AL. 2011

Felfernig, A.; Stumptner, M.; Tiihonen, J.: Special Issue: Configuration. Artificial Intelligence for Engineering Design, Analysis and Manufacturing 25 (2011) 02, pp. 113–114. ISSN: 0890-0604.

FENVES ET AL. 2008

Fenves, S. J.; Foufou, S.; Bock, C.; Sriram, R. D.: CPM2: A Core Model for Product Data. Journal of Computing and Information Science in Engineering 8 (2008) 1, p. 014501. ISSN: 15309827.

FENVES ET AL. 2005

Fenves, S. J.; Sriram, R. D.; Subrahmanian, E.; Rachuri, S.: Product Information Exchange: Practices and Standards. Journal of Computing and Information Science in Engineering 5 (2005) 3, pp. 238–246. ISSN: 15309827.

FERREIRA & RIBEIRO 2003

Ferreira, A. P. L.; Ribeiro, L.: Towards object-oriented graphs and grammars. Formal Methods for Open Object Based Distributed Systems (2003), pp. 16–31.

FISCHER ET AL. 2000

Fischer, T.; Niere, J.; Torunski, L.; Zündorf, A.: Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. Theory and Application of Graph Transformations 1764 (2000), pp. 157–167.

FISHER 1987

Fisher, D. H.: Knowledge acquisition via incremental conceptual clustering. Machine Learning 2 (1987) 2, pp. 139–172. ISSN: 0885-6125.

FOWLER 2002

Fowler, M.: Patterns of Enterprise Application Architecture, vol. 48. 2nd edition, Amsterdam, The Netherlands: Addison-Wesley 2002. ISBN: 978-0321127426.

FOXVOG 2010

Foxvog, D.: Cyc. In: Poli, R.; Healy, M.; Kameas, A. (Eds.): Theory and Applications of Ontology, chap. 12, pp. 259–278. Dordrecht, The Netherlands: Springer 2010. ISBN: 978-90-481-8846-8.

FRENCH 1999

French, M. J.: Conceptual Design for Engineers. 3rd edition, London, United Kingdom: Springer 1999. ISBN: 1-85233-027-9.

FREUDER & MACKWORTH 2006

Freuder, E. C.; Mackworth, A. K.: Constraint Satisfaction: An Emerging Paradigm. In: Rossi, F.; Beek, P. V.; Walsh, T. (Eds.): Handbook of Constraint Programming, Foundations of Artificial Intelligence, chap. 2, pp. 13–27. Amsterdam, The Netherlands: Elsevier Science 2006. ISBN: 978-0444527264.

FRÜHWIRTH & ABDENNADHER 2006

Frühwirth, T.; Abdennadher, S.: Principles of constraint systems and constraint solvers. ARCHIVES OF CONTROL SCIENCE 16 (2006) 2, p. 131. ISSN: 12302384.

FUHS 2009

Fuhs, A.: Hybrid Vehicles and the Future of Personal Transportation. Boca Raton: CRC Press 2009. ISBN: 978-1-4200-7534-2.

GAMMA ET AL. 1995

Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: Design Patterns. Amsterdam, The Netherlands: Addison Wesley 1995. ISBN: 978-0201633610.

GEISS ET AL. 2006

Geiß, R.; Batz, G. V.; Grund, D.; Hack, S.; Szalkowski, A.: GrGen: A Fast SPO-Based Graph Rewriting Tool. In: International Conference on Graph Transformations, pp. 383–397. Natal, Brazil: Springer 2006. ISBN: 9783540388708.

GEISS & KROLL 2007

Geiß, R.; Kroll, M.: On Improvements of the Varro Benchmark for Graph Transformation Tools. Tech. Rep. ISSN 1432-7864, Universität Karlsruhe, Karlsruhe, 2007.

GELLE & FALTINGS 2003

Gelle, E. M.; Faltings, B. V.: Solving mixed and conditional constraint satisfaction problems. Constraints 8 (2003) 2, pp. 107–141.

GENNARI ET AL. 2003

Gennari, J. H.; Musen, M. A.; Fergerson, R. W.; Grosso, W. E.; Crubezy, M.; Eriksson, H.; Noy, N. F.; Tu, S. W.: The evolution of Protégé: an environment for knowledge-based systems development. International Journal of Human-Computer Studies 58 (2003) 1, pp. 89–123. ISSN: 10715819.

GERO 1990

Gero, J. S.: Design prototypes: A Knowledge Representation Schema for Design. AI Magazine 11 (1990) 4, pp. 26–36. ISSN: 07384602.

GERO 2004

Gero, J. S.: The situated function-behaviour-structure framework. Design Studies 25 (2004) 4, pp. 373–391. ISSN: 0142694X.

GIPS & STINY 1980

Gips, J.; Stiny, G.: Production systems and grammars: a uniform characterization. Environment and Planning B: Planning and Design 7 (1980) 4, pp. 399–408. ISSN: 0265-8135.

GOEL ET AL. 1996

Goel, A.; Bhatta, S. R.; Stroulia, E.: Kritik: An Early Case-Based Design System. In: Maher, M. L.; Pu, P. (Eds.): Issues and Applications of Case-Based Reasoning in Design, chap. 5, pp. 87–132. Mahwah: Lawrence Erlbaum Associates 1996. ISBN: 9780805823134.

GORBEA ET AL. 2010

Gorbea, C.; Hellenbrand, D.; Srivastava, T.; Biedermann, W.; Lindemann, U.: Compatibility Matrix Methodology Applied to the Identfication of Vehicle Architectures and Design Requirements. In: 11th International Design Conference, DESIGN 2010, pp. 733–742. Dubrovnik, Croatia 2010.

GORBEA ET AL. 2008

Gorbea, C.; Spielmannleitner, T.; Lindemann, U.; Fricke, E.: Analysis of Hybrid Vehicle Architectures Using Multiple Domain Matrices. In: 10th International Design Structure Matrix Conference, DSM '08. Stockholm, Sweden 2008.

GRUBER 1993

Gruber, T. R.: A translation approach to portable ontology specifications. Knowledge Acquisition 5 (1993) 2, pp. 199–220. ISSN: 10428143.

HARALICK ET AL. 1978

Haralick, R. M.; Davis, L. S.; Rosenfeld, A.; Milgram, D. L.: Reduction operations for constraint satisfaction. Information Sciences 14 (1978) 3, pp. 199–219. ISSN: 00200255.

HARALICK & SHAPIRO 1979

Haralick, R. M.; Shapiro, L. G.: The Consistent Labeling Problem: Part I. IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-1 (1979) 2, pp. 173–184. ISSN: 0162-8828.

HARALICK & SHAPIRO 1980

Haralick, R. M.; Shapiro, L. G.: The Consistent Labeling Problem: Part II. IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-2 (1980) 3, pp. 193–203. ISSN: 0162-8828.

HART 1982

Hart, P. E.: Directions for AI in the Eighties. SIGART Newsletter 79 (1982), pp. 11–16.

HELLENBRAND & LINDEMANN 2008

Hellenbrand, D.; Lindemann, U.: Using the DSM to support the selection of product concepts. In: Kreimeyer, M.; Lindemann, U.; Danilovic, M. (Eds.): 10th International Design Structure Matrix Conference, DSM '08. Stockholm, Sweden: Hanser 2008. ISBN: 9783446418257.

HELMS ET AL. 2011

Helms, B.; Schultheiß, H.; Shea, K.: Automated Assignment of Physical Effects to Functions Using Ports Based on Bond Graphs. In: ASME 2011 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference, IDETC/CIE 2011. Washington DC, USA 2011.

HELMS & SHEA 2010

Helms, B.; Shea, K.: Object-Oriented Concepts for Computational Design Synthesis. In: Marjanović, D.; Štorga, M.; Pavković, N.; Bojčetić, N. (Eds.): 11th International Design Conference, DESIGN 2010, pp. 1333–1342. Glasgow, United Kingdom: The Design Society 2010.

HELMS & SHEA 2012

Helms, B.; Shea, K.: Computational Synthesis of Product Architectures Based on Object-Oriented Graph Grammars. Journal of Mechanical Design 134 (2012) 2, pp. 021008-1 – 021008-14. ISSN: 10500472.

HELMS ET AL. 2009

Helms, B.; Shea, K.; Hoisl, F.: A Framework for Computational Design Synthesis Based on Graph-Grammars and Function-Behavior-Structure. In: ASME 2009 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference, IDETC/CIE 2009, pp. 841–851. San Diego, USA 2009. ISBN: 978-0-7918-4905-7.

HEPPERLE ET AL. 2009

Hepperle, C.; Eben, K.; Lindemann, U.: Elements and Ways of Integrated Product Development Education - Current and Future Challenges. In: International Conference on Engineering Design and Product Design Education, September. 2009.

HERB 2000

Herb, R.: TRIZ - Der systematische Weg zur Innovation. Landsberg/Lech, Germany: Moderne Industrie 2000. ISBN: 3-478-91980-0.

HIRTZ ET AL. 2002a

Hirtz, J.; Stone, R. B.; Mcadams, D. A.; Szykman, S.; Wood, K. L.: A Functional Basis for Engineering Deisgn : Reconciling and Evolving Previous Efforts (NIST Technical Note 1447), 2002.

HIRTZ ET AL. 2002b

Hirtz, J.; Stone, R. B.; McAdams, D. A.; Szykman, S.; Wood, K. L.: A functional basis for engineering design: Reconciling and evolving previous efforts. Research in Engineering Design 13 (2002) 2, pp. 65–82.

HIX & ALLEY 1958

Hix, C. F.; Alley, R. P.: Physical Laws and Effects. New York, NY, USA: John Wiley & Sons 1958.

HOFFMANN 2005

Hoffmann, C. M.: Constraint-Based Computer-Aided Design. Journal of Computing and Information Science in Engineering 5 (2005) 3, p. 182. ISSN: 15309827.

HOLT & PERRY 2008

Holt, J.; Perry, S.: SysML for Systems Engineering. Herts, UK: The Institution of Engineering and Technology 2008. ISBN: 0863418252.

HORROCKS 2010

Horrocks, I.: Implementation and Optimization Techniques. In: Baader, F.; Calvanese, D.; McGuinness, D. L.; Nardi, D.; Patel-Schneider, P. F. (Eds.): The Description Logic Handbook, chap. 9, pp. 329–373. Cambridge, UK: Cambridge University Press 2010.

HORROCKS ET AL. 2010

Horrocks, I.; Patel-Schneider, P. F.; Welty, C. A.: OWL : A Description-Logic-Based Ontology Language for the Semantic Web. In: Baader, F.; Calvanese, D.; McGuinness, D. L.; Nardi, D.; Patel-Schneider, P. F. (Eds.): The Description Logic Handbook, chap. 14, pp. 458–486. Cambridge, UK: Cambridge University Press 2010.

HUNDAL 1990

Hundal, M.: A Systematic method for developing function structures, solutions and concept variants. Mechanism and Machine Theory 25 (1990) 3, pp. 243–256.

HÜRSCH & LOPES 1995

Hürsch, W. L.; Lopes, C. V.: Separation of Concerns. Tech. Rep., College of Computer Science, Northeastern University, Boston, MA, USA, 1995.

INFINITI RESEARCH 2011

Infiniti Research: Global Computer Aided Engineering Market 2010-2014. Tech. Rep., 2011.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION 2004

International Organization for Standardization: Information technology - Metadata registries (MDR) - Part 1: Framework (ISO/IEC 11179-1), 2004.

ISHII ET AL. 1995

Ishii, M.; Sekiya, T.; Tomiyama, T.: A Very Large-Scale Knowledge Base for the Intensive Engineering Framework. In: Mars, N. J. I. (Ed.): Towards Very Large Knowledge Bases, p. 306. Amsterdam, The Netherlands: IOS Press 1995. ISBN: 9051992173.

ISHII ET AL. 1993

Ishii, M.; Tomiyama, T.; Yoshikawa, H.: A Synthetic Reasoning Method for Conceptual Design. In: Wozny, M. J.; Olling, G. J. (Eds.): IFIP TC5/WG5.3 Conference on Towards World Class Manufacturing 1993, IFIP Transactions, vol. B-17, pp. 3–16. Litchfield Park, Arizona, USA: North-Holland 1993. ISBN: 0-444-81850-2.

JAKUMEIT ET AL. 2010

Jakumeit, E.; Buchwald, S.; Kroll, M.: GrGen.NET. International Journal on Software Tools for Technology Transfer 12 (2010) 3-4, pp. 263–271. ISSN: 1433-2779.

Jung et al. 2004

Jung, H.-W.; Kim, S.-G.; Chung, C.-S.: Measuring Software Product Quality: A Survey of ISO/IEC 9126. IEEE Software 21 (2004) 5, pp. 88–92. ISSN: 07407459.

Kerzhner & Paredis 2009

Kerzhner, A. A.; Paredis, C. J. J.: Using Domain Specific Languages to Capture Design Synthesis Knowledge for Model-Based Systems Engineering. In: ASME 2009 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference, IDETC/CIE 2009. San Diego, USA 2009.

Kim et al. 2008

Kim, S.; Bracewell, R.; Wallace, K. M.: Some reflections on ontologies in engineering domain. In: 7th International Symposium on Tools and Methods of Competitive Engineering (TMCE 2008). Izmir, Turkey 2008. ISBN: 978-90-5155-044-3.

Kläger 1993

Kläger, R.: Modellierung von Produktanforderungen als Basis für Problemlösungsprozesse in intelligenten Konstruktionssystemen. Dissertation, Universität Karlsruhe (TH), 1993. Aachen: Verlag Shaker 1993.

Koller 1994

Koller, R.: Konstruktionslehre für den Maschinenbau. 3rd edition, Berlin, Germany: Springer 1994.

Koller & Kastrup 1998

Koller, R.; Kastrup, N.: Prinziplösungen zur Konstruktion technischer Produkte. 2nd edition, Berlin, Germany: Springer 1998. ISBN: 3540630600.

Kolodner 1992

Kolodner, J. L.: An introduction to case-based reasoning. Artificial Intelligence Review 6 (1992) 1, pp. 3–34. ISSN: 0269-2821.

Komoto & Tomiyama 2010

Komoto, H.; Tomiyama, T.: Computational Support for System Architecting. In: ASME 2010 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference IDETC/CIE 2010, pp. 1–10. Montreal, Canada 2010.

Komoto & Tomiyama 2012

Komoto, H.; Tomiyama, T.: A framework for computer-aided conceptual design and its application to system architecting of mechatronics products (manuscript accepted for publication). Computer-Aided Design (2012). ISSN: 00104485.

Kruse et al. 2012

Kruse, B.; Münzer, C.; Wölkl, S.; Canedo, A.; Shea, K.: A Model-Based Functional Modeling and Library Approach for Mechatronic Systems in SysML. In: ASME 2012 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference, IDETC/CIE 2012. Chicago, USA 2012.

KURTOGLU & CAMPBELL 2006

Kurtoglu, T.; Campbell, M. I.: A Graph Grammar Based Framework for Automated Concept Generation. In: Marjanovic, D. (Ed.): 9th International Design Conference, DESIGN 2006, pp. 61–68. Glasgow, United Kingdom: The Design Society 2006.

KURTOGLU ET AL. 2010

Kurtoglu, T.; Swantner, A.; Campbell, M. I.: Automating the conceptual design process: "From black box to component selection". Artificial Intelligence for Engineering Design, Analysis and Manufacturing 24 (2010) 01, p. 49. ISSN: 0890-0604.

KWASNIK 1999

Kwasnik, B. H.: The role of classification in knowledge representation and discovery. Library Trends 48 (1999) 1, pp. 22–47. ISSN: 00242594.

LANGER & LINDEMANN 2009

Langer, S.; Lindemann, U.: Managing Cycles in Development Processes - Analysis and Classification of External Context Factors. In: International Conference on Engineering Design, ICED '09, pp. 539–550. Stanford, USA 2009.

LANGTANGEN 2011

Langtangen, H. P.: A Primer on Scientific Programming with Python. Texts in Computational Science and Engineering, 2nd edition, Heidelberg, Germany: Springer 2011. ISBN: 978-3-642-18365-2.

LEEMHUIS 2004

Leemhuis, H.: Funktionsgetriebene Konstruktion als Grundlage verbesserter Produktentwicklung. Dissertation, Fakultät V für Verkehrs- und Maschinensysteme, Technische Universität Berlin, 2004.

LEHMANN 1992

Lehmann, F.: Semantic Networks in Artificial Intelligence. In: Lehmann, F. (Ed.): The Handbook of Brain Theory and Neural Networks. Oxford, United Kingdom: Pergamon Press 1992. ISBN: 978-0080420127.

LEHMER 1957

Lehmer, D. H.: Combinatorial problems with digital computers. In: Proceedings of the Fourth Canadian Mathematical Congress, pp. 160–173. 1957.

LIANG & PAREDIS 2004

Liang, V.-C.; Paredis, C. J. J.: A Port Ontology for Conceptual Design of Systems. Journal of Computing and Information Science in Engineering 4 (2004) 3, pp. 206–217. ISSN: 15309827.

LIN ET AL. 2009

Lin, Y.-s.; Shea, K.; Johnson, A. L.; Coultate, J.; Pears, J.: A Method and Software Tool for Automated Gearbox Synthesis. In: ASME 2009 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference, IDETC/CIE 2009, pp. 111–121. San Diego, USA 2009. ISBN: 978-0-7918-4902-6.

LINDEMANN ET AL. 2008

Lindemann, U.; Maurer, M.; Braun, T.: Structural Complexity Management: An Approach for the Field of Product Design. Berlin: Springer 2008. ISBN: 3540878882.

Lindsay et al. 1993

  Lindsay, R. K.; Buchanan, B. G.; Feigenbaum, E. A.; Lederberg, J.: DENDRAL: A case study of the first expert system for scientific hypothesis formation. Artificial Intelligence 61 (1993) 2, pp. 209–261. ISSN: 00043702.

Lucas 1891

  Lucas, M. E.: Récréations Mathématiques. Paris, France: Gauthier-Villars 1891.

Luger 2005

  Luger, G. F.: Artificial Intelligence: Structures and Strategies for Complex Problem Solving. 5th edition, Harlow, England: Addison Wesley 2005. ISBN: 978-0-1320-900-18.

Mackworth 1977

  Mackworth, A. K.: Consistency in networks of relations. Artificial Intelligence 8 (1977) 1, pp. 99–118. ISSN: 00043702.

Maher et al. 1995

  Maher, M. L.; Balachandran, M. B.; Zhang, D. M.: Case-Based Reasoning in Design. Mahwah, NJ, USA: Lawrence Erlbaum Associates 1995. ISBN: 978-0805818321.

Mao et al. 2008

  Mao, M.; Peng, Y.; Spring, M.: Neural Network based Constraint Satisfaction in Ontology Mapping. In: Fox, D.; Gomes, C. P. (Eds.): Twenty-Third AAAI Conference on Artificial Intelligence, pp. 1207–1212. AAAI Press 2008. ISBN: 9781577353683.

McGuinness 2010

  McGuinness, D. L.: Configuration. In: Baader, F.; Calvanese, D.; McGuinness, D. L.; Nardi, D.; Patel-Schneider, P. F. (Eds.): The Description Logic Handbook, chap. 12, pp. 417–435. Cambridge, UK: Cambridge University Press 2010, 2nd edition.

McKay et al. 2012

  McKay, A.; Chase, S.; Shea, K.; Chau, H. H.: Spatial grammar implementation: From theory to useable software. Artificial Intelligence for Engineering Design, Analysis and Manufacturing 26 (2012) 02, pp. 143–159. ISSN: 0890-0604.

Meyer 1997

  Meyer, B.: Object-oriented software construction, vol. 352. 2nd edition, Englewood Cliffs, NJ, USA: Prentice Hall International 1997. ISBN: 0136291554.

Minsky 1975

  Minsky, M. L.: A Framework for Representing Knowledge. In: Winston, P. H. (Ed.): Psychology of Computer Vision, pp. 211–277. New York, NY, USA: McGraw-Hill 1975. ISBN: 978-0070710481.

Mittal & Frayman 1989

  Mittal, S.; Frayman, F.: Towards a generic model of configuration tasks. In: Eleventh International Joint Conference on Artificial Intelligence, IJCAI-89, vol. 2, pp. 1395–1401. San Mateo, CA, USA: Morgan Kaufmann 1989. ISBN: 978-1558600942.

Montanari 1974

  Montanari, U.: Networks of constraints: Fundamental properties and applications to picture processing. Information Sciences 7 (1974), pp. 95–132. ISSN: 00200255.

Muller 2007

Muller, G.: A multidisciplinary research approach, illustrated by the Boderc project, project website: http://www.gaudisite.nl (accessed on May 5, 2012), 2007.

Münzer 2010a

Münzer, C.: Fulfillment of requirements based on graph-grammars and constraint solving. Unpublished semester thesis (SA2579), Institute for Product Development, Technische Universität München, 2010.

Münzer 2010b

Münzer, C.: Integration modellbasierter und regelbasierter Wissensformalisierung. Unpublished diploma thesis (DA1194), Institute for Product Development, Technische Universität München, 2010.

Münzer et al. 2012

Münzer, C.; Shea, K.; Helms, B.: Automated Parametric Design Synthesis Using Graph Grammars and Constraint Solving. In: ASME 2012 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference, IDETC/CIE 2012. Chicago, USA 2012.

Nardi & Brachman 2010

Nardi, D.; Brachman, R. J.: An Introduction to Description Logics. In: Baader, F.; Calvanese, D.; McGuinness, D. L.; Nardi, D.; Patel-Schneider, P. F. (Eds.): The Description Logic Handbook, chap. 1, pp. 1–40. Cambridge, UK: Cambridge University Press 2010. ISBN: 0521781760.

Neema et al. 2005

Neema, S.; Gray, J.; Picone, J.; Porandla, S.; Musunuri, S.; Mathews, J.: Hybrid Powertrain Design Using a Domain-Specific Modeling Environment. In: 2005 IEEE Vehicle Power and Propulsion Conference, pp. 6–12. Chicago, USA 2005. ISBN: 0-7803-9280-9.

Object Management Group 2007

Object Management Group: Unified Modeling Language: Infrastructure (2007), p. 230.

Pahl et al. 2007

Pahl, G.; Beitz, W.; Feldhusen, J.; Grote, K.-H.: Engineering Design: A Systematic Approach. 3rd edition, London, United Kingdom: Springer 2007. ISBN: 1846283183.

Paredis et al. 2001

Paredis, C. J. J.; Diaz-Calderon, A.; Sinha, R.; Khosla, P. K.: Composable Models for Simulation-Based Design. Engineering with Computers 17 (2001) 2, pp. 112–128. ISSN: 0177-0667.

Parr 2007

Parr, T.: The Definitive ANTLR Reference. Raleigh, North Carolina: Pragmatic Bookshelf 2007. ISBN: 978-0978739256.

Partridge 1987

Partridge, D.: The scope and limitations of first generation expert systems. Future Generation Computer Systems 3 (1987) 1, pp. 1–10. ISSN: 0167739X.

Patel & Campbell 2010

Patel, J.; Campbell, M. I.: An Approach to Automate and Optimize Concept Generation of Sheet Metal Parts by Topological and Parametric Decoupling. Journal of Mechanical Design 132 (2010) 5, p. 051001. ISSN: 10500472.

PATEL-SCHNEIDER & SWARTOUT 1993

Patel-Schneider, P. F.; Swartout, B.: Description-Logic Knowledge Representation System Specification from the KRSS Group of the ARPA Knowledge Sharing Effort. Syntax And Semantics (1993), pp. 1–19.

PAYNTER 1961

Paynter, H. A.: Analysis and Design of Engineering Systems. Cambridge, MA, USA: MIT Press 1961. ISBN: 0262160048.

PONN & LINDEMANN 2011

Ponn, J.; Lindemann, U.: Konzeptentwicklung und Gestaltung technischer Produkte. 2nd edition, Berlin, Germany: Springer 2011. ISBN: 978-3642205798.

PUPPE 1990

Puppe, F.: Problemlösungsmethoden in Expertensystemen. Studienreihe Informatik, Berlin, Germany: Springer 1990. ISBN: 3540532315.

RECTOR & NOWLAN 1994

Rector, A.; Nowlan, W. A.: The galen project. Computer Methods and Programs in Biomedicine 45 (1994) 1-2, pp. 75–78. ISSN: 01692607.

REDDY & CAGAN 1995

Reddy, G.; Cagan, J.: An Improved Shape Annealing Algorithm for Truss Topology Generation. Journal of Mechanical Design 117 (1995) 2, pp. 315–321. ISSN: 10500472.

REENSKAUG 1979

Reenskaug, T.: Models - Views - Controllers. Tech. Rep., Xerox PARC, 1979.

RIHTARŠIČ ET AL. 2010

Rihtaršič, J.; Žavbi, R.; Duhovnik, J.: SOPHY - Tool for Structural Synthesis of Conceptual Technical Systems. In: 11th International Design Conference, DESIGN 2010, pp. 1391–1398. Dubrovnik, Croatia 2010.

ROSENBERG & KARNOPP 1983

Rosenberg, R. C.; Karnopp, D. C.: Introduction to Physical System Dynamics. New York, NY, USA: McGraw Hill 1983. ISBN: 0070539057.

ROTH 2001

Roth, K.: Konstruieren mit Konstruktionskatalogen II - Konstruktionskataloge. 3rd edition, Berlin, Germany: Springer 2001. ISBN: 3-540-67026-2.

ROZENBLIT & HU 1992

Rozenblit, J. W.; Hu, J.: Integrated knowledge representation and management in simulation-based design generation. Mathematics and Computers in Simulation 34 (1992) 3-4, pp. 261–282. ISSN: 03784754.

RUDE 1998

Rude, S.: Wissensbasiertes Konstruieren. Habilitation, Universität Karlsruhe (TH), 1998. Aachen, Germany: Shaker 1998.

RUSSELL & NORVIG 2003

Russell, S. J.; Norvig, P.: Artificial Intelligence: A Modern Approach. 2nd edition, Upper Saddle River: Pearson Education 2003. ISBN: 0-13-790395-2.

SABIN & WEIGEL 1998

Sabin, D.; Weigel, R.: Product Configuration Frameworks-A Survey. IEEE Intelligent Systems 13 (1998) 4. ISSN: 1541-1672.

SCHÄFER & RUDOLPH 2005

Schäfer, J.; Rudolph, S.: Satellite design by design grammars. Aerospace Science and Technology 9 (2005) 1, pp. 81–91. ISSN: 12709638.

SCHMIDT & CAGAN 1998

Schmidt, L. C.; Cagan, J.: Optimal Configuration Design: An Integrated Approach Using Grammars. Journal of Mechanical Design 120 (1998) 1, pp. 2–9. ISSN: 10500472.

SCHMIDT ET AL. 2000

Schmidt, L. C.; Shetty, H.; Chase, S. C.: A Graph Grammar Approach for Structure Synthesis of Mechanisms. Journal of Mechanical Design 122 (2000) 4, pp. 371–376.

SHEA 1997

Shea, K.: Essays of Discrete Structures: Purposeful Design of Grammatical Structures by Directed Stochastic Search. Dissertation, Carnegie Mellon University, 1997.

SHEA ET AL. 2005

Shea, K.; Aish, R.; Gourtovaia, M.: Towards integrated performance-driven generative design tools. Automation in Construction 14 (2005) 2, pp. 253–264. ISSN: 09265805.

SHEA & STARLING 2003

Shea, K.; Starling, A. C.: From Discrete Structures to Mechanical Systems: A Framework for Creating Performance-Based Parametric Synthesis Tools. In: AAAI 2003 Spring Symposium. Palo Alto, USA 2003.

SHORTLIFFE ET AL. 1975

Shortliffe, E. H.; Davisa, R.; Axlinea, S. G.; Buchanan, B. G.; Greena, C. C.; Cohena, S. N.: Computer-based consultations in clinical therapeutics: Explanation and rule acquisition capabilities of the MYCIN system. Computers and Biomedical Research 8 (1975) 4, pp. 303–320. ISSN: 00104809.

SIMON 1996

Simon, H. A.: The Sciences of the Artificial. 3rd edition, Cambridge, MA, USA: The MIT Press 1996. ISBN: 9-780-262193740.

SPATH 2001

Spath, D.: Vom Markt zum Produkt - Impulse für die Innovationen von morgen. Stuttgart: LOG_X Verlag 2001. ISBN: 3932298187.

SRIDHARAN & CAMPBELL 2004

Sridharan, P.; Campbell, M. I.: A Grammar for Function Structures. Proceedings Of The Asme Design Engineering Technical Conference 2004 (2004) 46962a, pp. 41–55.

SRIRAM 1997

Sriram, R. D.: Intelligent Systems for Engineering. London: Springer 1997. ISBN: 3540761284.

STANKOVIC 2011

Stankovic, T.: Grammatical evolution of technical processes. Dissertation, Faculty of Mechanical Engineering and Naval Architecture, University of Zagreb, 2011.

STANKOVIC ET AL. 2009

Stankovic, T.; Shea, K.; Storga, M.; Marjanovic, D.: Grammatical Evolution of Technical Processes. In: ASME 2009 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference, IDETC/CIE 2009, pp. 895–904. San Diego, USA: ASME 2009.

STARLING & SHEA 2005

Starling, A. C.; Shea, K.: A parallel grammar for simulation-driven mechanical design synthesis. In: ASME 2005 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference, IDETC/CIE 2005. Long Beach, USA 2005.

STINY & MITCHELL 1978

Stiny, G.; Mitchell, W. J.: The Palladian grammar. Environment and Planning B 5 (1978) 1, pp. 5–18. ISSN: 02658135.

STOKES & MOKA CONSORTIUM 2001

Stokes, M.; MOKA Consortium: Managing Engineering Knowledge: MOKA - Methodology for Knowledge Based Engineering Applications. London, United Kingdom: Professional Engineering Publication 2001. ISBN: 978-1860582950.

STONE & WOOD 2000

Stone, R. B.; Wood, K. L.: Development of a functional basis for design. Journal of Mechanical Design 122 (2000) December, p. 359.

STONE ET AL. 2000

Stone, R. B.; Wood, K. L.; Crawford, R. H.: A heuristic method for identifying modules for product architectures. Design Studies 21 (2000) 1, pp. 5–31. ISSN: 0142694X.

STRUSS & PRICE 2003

Struss, P.; Price, C.: Model-Based Systems in the Automotive Industry. AI Magazine 24 (2003) 4, pp. 17–34. ISSN: 07384602.

STUBBLEFIELD & LUGER 1996

Stubblefield, W. A.; Luger, G. F.: Source selection for analogical reasoning an empirical approach. In: AAAI'96 Proceedings of the thirteenth national conference on Artificial intelligence - Volume 1, pp. 696–702. AAAI Press 1996. ISBN: 0-262-51091-X.

SUSSMAN & STEELE 1980

Sussman, G. J.; Steele, G. L. J.: A Language for Expressing Almost-Hierarchical Descriptions. Artificial Intelligence 14 (1980) 1, pp. 1–39.

SZYKMAN ET AL. 2000

Szykman, S.; Racz, J.; Bochenek, C.; Sriram, R. D.: A web-based system for design artifact modeling. Design Studies 21 (2000) 2, pp. 145–165. ISSN: 0142694X.

TOMIYAMA ET AL. 1989

Tomiyama, T.; Kiriyama, T.; Takeda, H.; Xue, D.; Yoshikawa, H.: Metamodel: A key to intelligent CAD systems. Research in Engineering Design 1 (1989) 1, pp. 19–34. ISSN: 0934-9839.

TSANG 1993

Tsang, E.: Foundations of Constraint Satisfaction (Computation in Cognitive Science). London, United Kingdom: Academic Press 1993. ISBN: 0127016104.

ULRICH 1995

Ulrich, K.: The role of product architecture in the manufacturing firm. Research Policy 24 (1995) 3, pp. 419–440. ISSN: 00487333.

ULRICH & EPPINGER 2008

Ulrich, K.; Eppinger, S. D.: Product Design and Development. 4th edition, New York, NY, USA: McGraw Hill 2008. ISBN: 978-007-125947-7.

UMEDA & TOMIYAMA 1995

Umeda, Y.; Tomiyama, T.: FBS modeling: Modeling scheme of function for conceptual design. In: 9th International Workshop on Qualitative Reasoning, pp. 271–278. 1995.

VDI 1987

VDI: Systematic approach to the development and design of technical systems and products (2221), 1987.

VDI 2005

VDI: Classification and evaluation of description methods in automation and control technology (3681), 2005.

ŽAVBI & RIHTARŠIČ 2009

Žavbi, R.; Rihtaršič, J.: Synthesis of elementary product concepts based on knowledge twisting. Research in Engineering Design 21 (2009) 2, pp. 69–85. ISSN: 0934-9839.

WELCH & DIXON 1994

Welch, R. V.; Dixon, J. R.: Guiding conceptual design through behavioral reasoning. Research in Engineering Design 6 (1994) 3, pp. 169–188. ISSN: 0934-9839.

WHITNEY 1996

Whitney, D. E.: Why mechanical design cannot be like VLSI design. Research in Engineering Design 8 (1996) 3, pp. 125–138. ISSN: 0934-9839.

WILKES 1964

Wilkes, M. V.: Constraint-type statements in programming languages. Communications of the ACM 7 (1964) 10, pp. 587–588. ISSN: 00010782.

WÖLKL & SHEA 2009

Wölkl, S.; Shea, K.: A Computational Product Model for Conceptual Design Using SysML. In: ASME 2009 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference, IDETC/CIE 2009. San Diego, USA 2009.

WU ET AL. 2008

Wu, Z.; Campbell, M. I.; Fernandez, B. R.: Bond Graph Based Automated Modeling for Computer-Aided Design of Dynamic Systems. Journal of Mechanical Design 130 (2008) 4, pp. 041102–1 – 041102–11. ISSN: 10500472.

WYATT ET AL. 2012

Wyatt, D. F.; Wynn, D. C.; Jarrett, J. P.; Clarkson, P. J.: Supporting product architecture design using computational design synthesis with network structure constraints. Research in Engineering Design 23 (2012) 1, pp. 17–52. ISSN: 0934-9839.

WYNN ET AL. 2010

Wynn, D. C.; Wyatt, D. F.; Nair, S. M. T.; Clarkson, P. J.: An Introduction to the Cambridge Advanced Modeller. In: International Conference on Modelling and Management Engineering, July, pp. 19–20. Springer 2010. ISBN: 1849961980.

YANER & GOEL 2007

Yaner, P.; Goel, A.: Understanding drawings by compositional analogy. In: International Joint Conference on Artificial Intelligence, IJCAI-2007, pp. 1131–1137. Hyderabad 2007.

YOSHIOKA ET AL. 2004

Yoshioka, M.; Umeda, Y.; Takeda, H.; Shimomurad, Y.; Nomaguchi, Y.; Tomiyama, T.: Physical concept ontology for the knowledge intensive engineering framework. Advanced Engineering Informatics 18 (2004) 2, pp. 95–113. ISSN: 14740346.

YOUNG 2003

Young, R. R.: The Requirements Engineering Handbook. Norwood, MA, USA: Artech House 2003. ISBN: 1-58053-266-7.

9 Appendix

9.1 Illustrative example for constraint-based representations

In the following, the application of the theoretical foundation of CSPs for modeling constraints is illustrated using a simplified automotive gearbox as an example[50], illustrated in Figure 9-1. The 6-speed gearbox and the other elements are characterized by the tuple of variables X. The algorithm for solving the CSP aims to assign values to the variables x_i with respect to the tuple D defining the valid value ranges. The variable i[51] contains a list of the six transmission ratios, a reasonable value range is between 0.8 and 5. The gears can either be spur gears or helical gears; this is modeled with the variable *type*. The physically meaningful range of the *efficiency* is between 0 and 1. In the context of automotive powertrains, the input of a gearbox can be either linked to a clutch or to the motor (variable *input*). To prevent confusion and facilitate implementation, variables are attached to the components using a dot ".", e. g. the gearbox' efficiency is represented as *gbox.efficiency*. GELLE & FALTINGS (2003) differentiate three types of constraints:

6-Speed Gearbox (gbox)	
i[1..6]	: [0.8, 5]
efficiency	: [0, 1]
type	: {"spur gears", "helical gears"}
supplier	: {"ZF", "Getrag"}
input	: {"cl", "ice", "ele"}
output	: {"cl"}
...	: ...
X	*D*

Internal combustion engine (ice)	
input	: {"tank"}
output	: {"gbox", "cl"}
...	: ...

Electrical engine (ele)	
input	: {"battery"}
output	: {"gbox", "cl"}
...	: ...

Clutch (cl)	
input	: {"gbox", "ice", "ele"}
output	: {"gbox"}
...	: ...

Figure 9-1: Constrained-based representation of components for a hybrid automotive powertrain

- *Numeric constraints* represent mathematical relations as equations or inequalities and only relate to numeric variables, such as the transmission ratios in i. Usually, transmission ratios are defined in descending order. These relations can be represented as five numeric inequality constraints:

$$c_1 = gbox.i[1] > gbox.i[2] \tag{9.1}$$

$$\vdots \tag{9.2}$$

$$c_5 = gbox.i[5] > gbox.i[6] \tag{9.3}$$

---

[50]This example is adapted and extended from the semester thesis of MÜNZER (2010a))

[51]The letter i is typically used in the engineering domain as variable for transmission ratios and is not to be confused with the control variable $_i$ used as index, such as in x_i.

- *Discrete constraints* only apply to discrete variables such as *gbox.type, gbox.supplier* or *gbox.input*. In this example discrete constraints define relations between distinct values of discrete variables. The gearbox' type and the supplier directly depend on each other:

$$c_6(gbox.supplier, gbox.type) = (\text{"}ZF\text{"}, \text{"}planetary\ gear\text{"}), (\text{"}Getrag\text{"}, \text{"}spur\ gear\text{"}) \quad (9.4)$$

- *Mixed constraints* involve the use of both – numeric and discrete – variable types. For example here, mixed constraints represent the dependency of the gearbox' efficiency and gear type:

$$c_7(gbox.type, gbox.efficiency) = (\text{"}planetary\ gear\text{"}, 0.95), (\text{"}spur\ gear\text{"}, 0.98) \quad (9.5)$$

The constraints $c_1 \dots c_7$ represent only parametric relations of the modeled components. In the scope of this thesis, these constraints are termed *parametric constraints*. Contrarily, *topological constraints* represent topological relations between components. Rozenblit & Hu (1992) use the term *coupling constraints* and state that they "impose a manner in which components [...] can be connected together"[52]. For example, the gearbox' input can be connected to the output of the clutch (cl), the internal combustion engine (ice) or electrical engine (ele); they all have compatible mechanical rotational interfaces. The compatibility of the components can be represented with discrete constraints. To achieve this, the constraint c_8 is defined relating the input variable of the gearbox *gbox.input* to all potential output variables using an OR-operator, represented by the symbol $\vee$. If two components are connected, their input- (or output-)variables contain the name of the allocated component.

$$c_8(gbox.input, ele.output \vee cl.output \vee ice.output) =$$
$$(\text{"}ele\text{"}, \text{"}gbox\text{"}), (\text{"}cl\text{"}, \text{"}gbox\text{"}), (\text{"}ice\text{"}, \text{"}gbox\text{"}) \quad (9.6)$$

For every component with an input a topological constraint has to be defined that represents the compatibility of that component with all potential outputs. Consequently, the number of constraints and number of interrelated variables within the constraints increases significantly with an increase of compatibility and a larger number of elements. Depending on the degree of compatibility, this representation can become convoluted.

The concept of *ports* that is introduced in more detail in Section 3.3.1 provides a clearer representation of topological constraints. Instead of defining the individual input-output-compatibility of every component, ports represent classes of compatibility, i.e. interfaces. The significance and suitability of ports in conceptual design has been described by Liang & Paredis (2004): "The port connections represent interactions consisting of the exchange of energy, matter or signals [...]. Representing design alternatives as configurations of port-based objects is useful at the conceptual design stage when the geometry and spatial layout is still ill-defined." Figure 9-2 depicts the representation of topological constraints based on ports. First, ports have to be defined; three types of energy flow ports are required: mechanical rotational, chemical and electrical. Second, ports can be assigned to the components and represent their compatibility. Thirdly, based on this representation the set of valid system architectures can be derived. In this example, four valid powertrain configurations are depicted.

---

[52]Other common terms are *structural constraints* or *network structure constraints* such as used in (Wyatt et al., 2012)
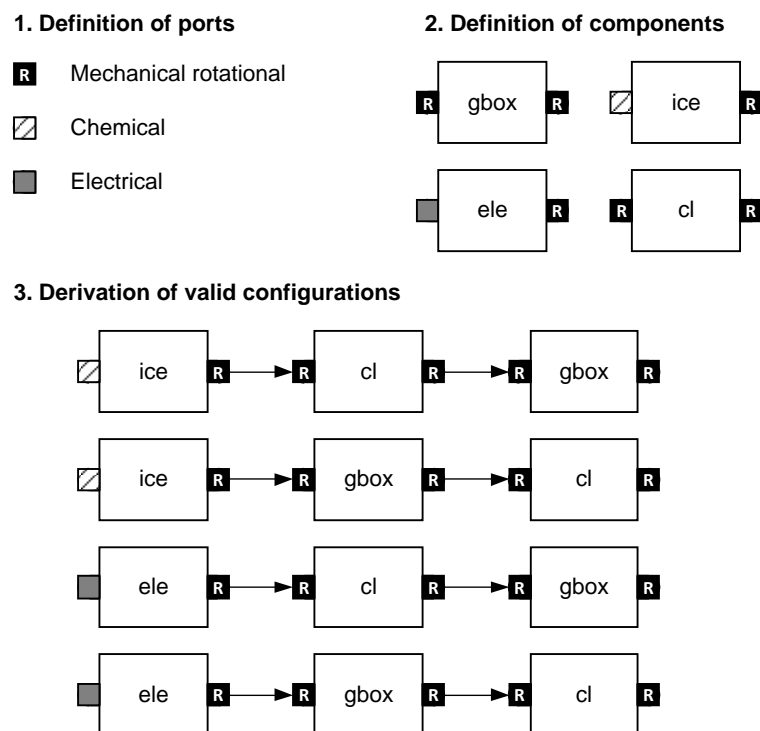
Figure 9-2: Port-based representation of topological constraints of components for a hybrid automotive powertrain

9.2 Illustrative example for description logics

In the following, the domain of road vehicles is used as an example. The syntax and terminology of the abstract language by Nardi & Brachman (2010) is used. The initial step of building a DL-based knowledge base is to declare atomic concepts and atomic roles. They can be seen as the fundamental building blocks providing the foundation for the definition of new concepts. For that purpose, logical operators, termed symbols, are available. One fundamental symbol is the intersection of concepts ($\sqcap$) that is used to restrict the regarded set of concepts and their individuals. For example, the declaration of the concept *Car* (using the equivalence operator $\equiv$) can be seen as the intersection of the atomic concepts *RoadVehicle* and *FourWheeled*:

$$Car \equiv RoadVehicle \sqcap FourWheeled \tag{9.7}$$

Roles provide for the restriction of concepts and the definition of new concepts as they characterize the relationships among concepts. Using the operator for an existential quantification ($\exists$) the role *usesEnergy* can be defined. It allows, by that, to link the atomic concept *Electric* to the concept of *RoadVehicle* and to define the new concept *ElectricVehicle*. Hence, the existence of any individual of *ElectricVehicle* requires this relationship to be fulfilled.

$$ElectricVehicle \equiv RoadVehicle \sqcap \exists usesEnergy.Electric \tag{9.8}$$

While the concept *FourWheeled* contains all subconcepts having four wheels – this might also include train wagons with four wheels – cars should have tires as wheels. For that purpose, the definition of the concept *Car* in equation (9.7) can be extended by an intersection with the set of concepts that fulfill the role *hasWheels.Tire*. This role establishes a relation between the concepts *Tire* and *Car*:

$$Car \equiv (RoadVehicle \sqcap FourWheeled) \sqcap \exists hasWheels.Tire \tag{9.9}$$

Based on the concepts *Car* and *ElectricVehicle* the concept of *ElectricCar* is defined as their intersection:

$$ElectricCar \equiv Car \sqcap ElectricVehicle \tag{9.10}$$

Through the previous definitions the terminology, i. e. the TBox, of this exemplary domain of interest is set. In contrast, the ABox contains assertions about individuals of these concepts. For example,

$$ElectricCar(Mitsubishi\_i - MiEV) \tag{9.11}$$

states that the individual *Mitsubishi_i − MiEV* is an electric car. As *ElectricVehicle* is derived from the intersection of the concepts *Car* and *ElectricVehicle*, the role

$$hasWheels.Tire(Mitsubishi\_i - MiEV, BridgeStone175/55\_R15) \tag{9.12}$$

can be applied and relates one specific *Tire* individual described as *BridgeStone*175/55_*R*15, to one specific individual described as *ElectricVehicle*. Assertions of the former kind are termed *concept assertions*; the latter are called *role assertions*.

To conclude, the combination of TBox and ABox provides a formal representation for defining concepts, categories of concepts, their relationships and instances. Based on this representation, reasoning tasks can be accomplished. One important and commonly applied reasoning problem is to "determine whether a description is *satisfiable*" (Baader, 2010). This means that assertions in the ABox are checked against the concept definitions in the TBox. Russell & Norvig (2003, p. 353) term this task "classification" and mention another main task for description logics: "Subsumptions" check whether a concept can be a subset of another through comparison of their definitions.

Subsumptions can be seen as "the basic reasoning service for the TBox" (Nardi & Brachman, 2010). They are helpful to determine the meaningfulness and consistency of a knowledge base. For example, one might want to ensure that any individual of the concept *ElectricCar* is contained in the set of individuals of the concept *Car*. Hence, any electric car would be equipped automatically with four wheels. Through comparison of the considered sets, reasoners are able to infer that the concept *Car* is subsumed by *ElectricCar*, which can be expressed as:

$$ElectricCar \sqsubseteq Car \tag{9.13}$$

9.3 Functional Basis

Table 9-1: Functional operators of the Functional Basis (Hirtz et al., 2002b)

Class (Primary)	Secondary	Tertiary	Correspondents
Branch	Separate		Isolate, sever, disjoin
		Divide	Detach, *isolate*, release, sort, split, disconnect, subtract
		Extract	Refine, filter, purify, percolate, strain, *clear*
		Remove	Cut, drill, lathe, polish, sand
	Distribute		Diffuse, dispel, disperse, dissipate, diverge, scatter
Channel	Import		Form entrance, *allow*, input, *capture*
	Export		Dispose, eject, *emit*, empty, *remove*, destroy, eliminate
	Transfer		Carry, deliver
		Transport	Advance, lift, move
		Transmit	Conduct, convey
	Guide		Direct, shift, steer, straighten, switch
		Translate	Move, relocate
		Rotate	Spin, turn
		Allow DOF	*Constrain*, unfasten, unlock
Connect	Couple		Associate, connect
		Join	Assemble, fasten
		Link	Attach
	Mix		Add, blend, coalesce, combine, pack
Control	Actuate		Enable, initiate, start, turn-on
	Regulate		Control, equalize, limit, maintain
		Increase	*Allow*, open
		Decrease	Close, delay, interrupt
Magnitude	Change		Adjust, modulate, *clear*, demodulate, invert, normalize, rectify, reset, scale, vary, modify
		Increment	Amplify, enhance, magnify, multiply
		Decrement	Attenuate, dampen, reduce
		Shape	Compact, compress, crush, pierce, deform, form
		Condition	Prepare, adapt, treat
	Stop		End, halt, pause, interrupt, restrain
		Prevent	Disable, turn-off
		Inhibit	Shield, insulate, protect, resist
Convert	Convert		Condense, create, decode, differentiate, digitize, encode, evaporate, generate, integrate, liquefy, *process*, solidify, transform
Provision	Store		Accumulate
		Contain	*Capture*, enclose
		Collect	Absorb, consume, fill, reserve
	Supply		Provide, replenish, retrieve
Signal	Sense		Feel, determine
		Detect	Discern, perceive, recognize
		Measure	Identify, *locate*
	Indicate		Announce, show, denote, record, register
		Track	Mark, time
		Display	*Emit*, expose, select
	Process		Compare, calculate, check
Support	Stabilize		Steady
	Secure		*Constrain*, hold, place, fix
	Position		Align, *locate*, orient

Overall increasing degree of specification from left to right

Table 9-2: Functional flows of the Functional Basis (HIRTZ ET AL., 2002b)

Class (Primary)	Secondary	Tertiary	Correspondents
Material	Human		Hand, foot, head
	Gas		Homogeneous
	Liquid		Incompressible, compressible, homogeneous
	Solid	Object	Rigid-body, elastic-body, widget
		Particulate	
		Composite	
	Plasma		
	Mixture	Gas–gas	
		Liquid–liquid	
		Solid–solid	Aggregate
		Solid–liquid	
		Liquid–gas	
		Solid–gas	
		Solid–liquid–gas	
		Colloidal	Aerosol
Signal	Status	Auditory	Tone, word
		Olfactory	
		Tactile	Temperature, pressure, roughness
		Taste	
		Visual	Position, displacement
	Control	Analog	Oscillatory
		Discrete	Binary
Energy	Human		
	Acoustic		
	Biological		
	Chemical		
	Electrical		
	Electromagnetic	Optical	
		Solar	
	Hydraulic		
	Magnetic		
	Mechanical	Rotational	
		Translational	
	Pneumatic		
	Radioactive/Nuclear		
	Thermal		

Overall increasing degree of specification from left to right

9.4 Software architecture

9.4.1 Model

The entities introduced in Chapter 3, such as *element type* or *port*, are computationally represented within booggie. This section gives a high-level overview of the most important entities that are implemented as classes[53] and introduces their properties. All of these classes can be found in the booggie source code within the respective Python modules. The classes depicted in Figure 9-3 are crucially important for the fundamental functionality of booggie.
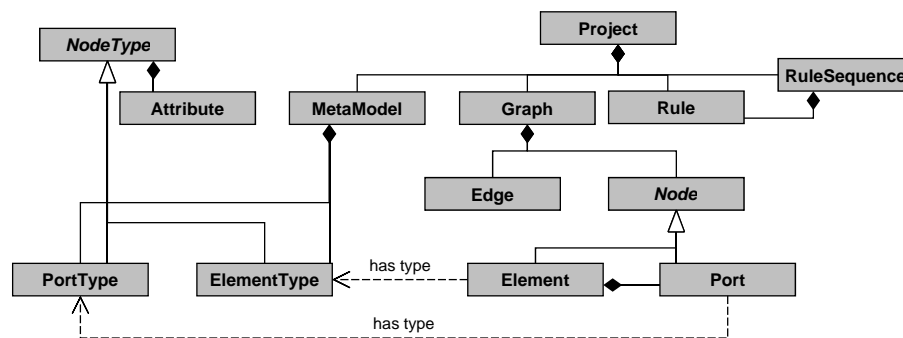
Figure 9-3: UML class diagram of the booggie classes

- *Project* represents a snapshot at run time of all instantiations of the following classes. It contains the Metamodel as defined by the user, the Rules and RuleSequences and the Graphs.

- A *Rule* consists of two Graphs, the left-hand side and right-hand side, and a textual representation of the rule code according to the applied rule language.

- A *RuleSequence* contains a string-based description of the concatenation of Rules according to the applied rule sequence language.

- *Graph* is a collection of Elements and their respective Ports that are interconnected using Edges. Graphs that are built on this class are termed in the following *booggie graphs*.

- The *Metamodel* is the collection of all ElementTypes and PortTypes and represents them in a hierarchical way reflecting their hierarchy of inheritance.

- *NodeType* is an abstract class to describe nodes as general modeling elements of graphs. It encapsulates common functionality and is inherited by PortType and ElementType. Each type has a unique identifier, e. g. its name, a parent type and a list of attributes.

- *Port* is an instantiation of a PortType. It inherits all attributes from its type but allows the modification of attributes limiting the visibility of the changes to the scope of the instance.

- *Element* is, similarly to Port, an instantiation of an ElementType.

- *Attributes* are used to annotate ports and elements with additional information. Each attribute has an identifier through which it can be accessed in graph transformation rules.

---

[53]Classes are typed with leading capital letter.

With the implementation of these classes, the required support of the knowledge formalization process can be realized while remaining solution-independent, cf. SWReq 1.

9.4.2 View

The graphical user interface (GUI) builds on five perspectives that are part of the five individual software components (see Figure 5-1). Thereby, each perspective maps to one working step of the knowledge formalization process and aims to support the user to perform each task as demanded by SWReq 2. In general, a perspective is composed of these three main constituents (see Figure 9-4):

- *Docklets* are reusable view components that can be freely arranged at the margin of each perspective. They are multi-purpose elements offering additional perspective-relevant information or functionalities.

- *Toolbars* are displayed on the top of the window to provide shortcuts to often needed functionality that can be either required in all perspectives, e. g. saving a project, or be specific to one perspective, e. g. interaction with graphs.

- *Center Views* are the main part of a perspective and contain the primary GUI elements needed for the working steps. Additionally, a center view has to handle actions triggered by docklets. For instance, an "open graph"-action triggered by a graph docklet leads to a graphical representation of the selected graph in the center view.
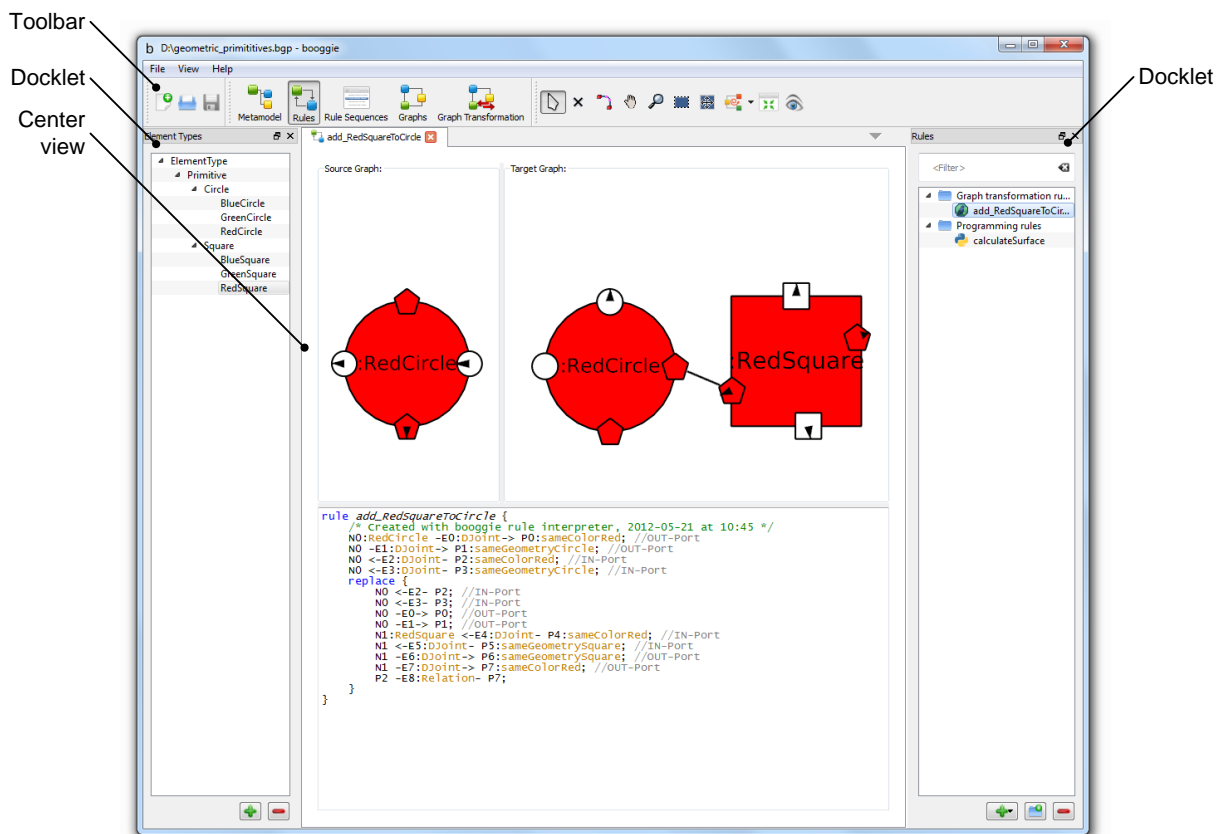
Figure 9-4: Ggeneral constituents of perspectives

Perspectives can be customized according to the user preferences as docklets and toolbars can be placed freely in the respective areas.

9.4.3 Controller

MVC separates the model from the views. If the model changes its state the views have to adapt to the changed model to reflect the correct data. booggie contains a variety of controller components for handling input and determining the result and relaying that information to the user via views. In this section, two of these controller components are presented: the observer pattern as implemented in booggie and the persistence controller allowing to save the runtime object to the file system.

The observer pattern provides a technique to synchronize the model with multiple views. The working principle is illustrated with an example as depicted in Figure 9-5:

1. The user creates a metamodel and defines an ElementType "gearbox" that has the color white.

2. This ElementType is used to create a rule that adds another element to the gearbox as defined in the right-hand side of the rule. To detect isolated gearbox elements, a left-hand side graph with only one gearbox element is created in the graph perspective. This rule can be applied on a host graph on which the left-hand side of the rule matches. Hence, multiple occurrences of this ElementType can be identified in various graph visualizations instances.

3. The user changes the view color property of the "gearbox" in the metamodel to grey.

4. All visual representations of the element have to change accordingly.
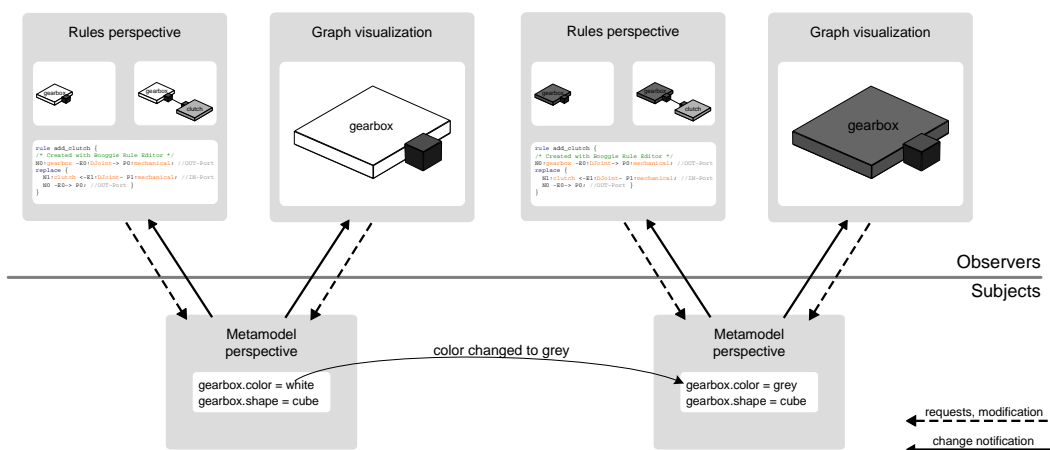
Figure 9-5: Example for the application of the observer pattern

In the case that the user changes the color of the gearbox, all perspectives that refer to this ElementType should redraw their graphs. However, the decoupling of MVC implies that graphs and elements have no dependencies between them and do not know about each other. The observer pattern describes a mechanism to establish a communication between independent components. This pattern consists of a *subject* and an *observer*. If the state of a subject changes, all observers get notified. Subsequently the observer queries the subjects to synchronize its own state with the subject's state. This interaction is known as *publish-subscribe*. Thereby, the subject publishes all state changes using notifications to which an observer might subscribe if its subject is affected by these changes (GAMMA ET AL., 1995)

In booggie, the observer pattern is implemented using an Event class enabling each class to trigger an Event or listen to Events fired by other classes. If a class subscribes to a certain Event it has to implement the related method that is called if the Event is fired. Whenever a class publishes a notification by triggering an Event, all listeners execute their associated methods.

Saving a booggie project file is handled by the persistence controller. This controller uses the Project class to determine if changes to the project have been made and exports this data to a persistent storage. booggie uses the eXtensible Markup Language (XML) to encode the entities defined by the user. Entities are aggregated into groups (metamodel, rule set, rule sequences and graphs) and exported into single files. These files are then archived into a ZIP file container to enable the exchange of a single file. ZIP is chosen as a container as it is a widespread format supported by a wide range of libraries and can be opened transparently by many operating systems. Encoding the data in XML offers two key benefits:

- Interoperability: Similarly to the ZIP format, XML is a widely used industry standard. A large number of libraries and toolkits for handling XML encoded files are available. This enables any project to import booggie project files if a XML parser is available. Further, the declarative language XSLT (Extensible Stylesheet Language Transformations) can be used for transformation between XML schemata, e. g. for supporting interoperability with other tools.

- Human readable: XML is a human readable file format which allows the user to edit the file manually. Secondly, it supports the developer during test and debug processes because it simplifies the location of mistakes.

9.4.4 Plugin architecture

booggie implements the plugin architecture that is defined in (FOWLER, 2002). Similar implementations of this pattern can be found in various other software projects like Eclipse or Firefox. This design pattern allows the flexible extension of an application during runtime, cf. SWReq 13.

In general, this architecture consists of extension points and plugins that are using these extension points to provide additional functionality. The plugins themselves can provide extension points based on which other plugins can be developed. The functionality of extension points and how plugins can attach to them are described in files that are located either in the respective plugin folder or in the core implementation. Using these descriptions files and the source code of the plugins, the main application locates all plugins and is then calling specific plugins at the defined point during the execution of the program.
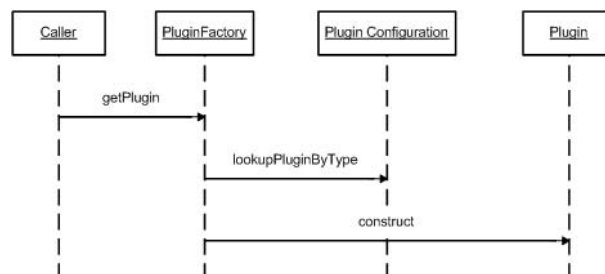
Figure 9-6: Sequence for calling a plugin

The call of a plugin within this design pattern is explained with sequence diagram in Figure 9-6. The actors and their roles in this diagram are:

- *Caller* is an object of the main application that wants to use a functionality provided by a plugin through accessing an extension point that must be defined in the PluginConfiguration.

- The *PluginFactory* possesses the information on the existence and the kind of all installed plugins. It parses all plugin configurations at startup, then loads the concrete plugin implementations at runtime and provides its functionality within booggie.

- The *PluginConfiguration* contains a description of the plugin and its extension points in the file plugin.xml located in the plugin package.

- *Plugin* contains the concrete implementation of a plugin. For examples on how to implement concrete plugins, refer to the booggieExamplePlugins folder located in the plugins folder.

This plugin architecture is not only used to extend booggie's functionality but also to modularize the core of booggie, i. e. the components as illustrated in Figure 5-1. This means that the entire software is built around a core that is extended either by standard plugins that are mandatory for assuring the key functionality or by optional plugins. All plugins have to be implemented according to the MVC convention. Same as in engineering design processes (cf. Section 2.2), this division of the entire system into subsystems allows parallelizing the development efforts to some extent.

Based on this architecture, several plugins were successfully developed. For example, a Matlab plugin allows to control Matlab out of booggie and to include thereby advanced calculations in the synthesis approach. Another plugin was developed, as a first proof of concept, to trigger in scripts the execution of simulations in Modelica using the Modelica solver JModelica[54].

---

[54]Project website: http://www.jmodelica.org

9.5 Notation for the definition of the booggie modeling language (bgML)

Table 9-3: Overview over the used sets and functions for the definition of bgML

G	Typed and attributed multigraph
N	Set of nodes
E	Set of edges
I_N	Graph describing the inheritance hierarchy of nodes types
I_E	Graph describing the inheritance hierarchy of edges types
A_N	Set of attributes of nodes
A_E	Set of attributes of edges
N_{El}	Set of elements
N_P	Set of ports
$attr$	Function for assigning an attribute to an element or a port
D	Set of the direction values a port can have {*in*, *out*, *undirected*}
$element$	Function returning the element a port belongs to
s	Function returning the port at the source of an edge
t	Function returning the port at the target of an edge
dir	Function returning the direction of a port
$type$	Function returning the type of a node or an edge from I_N or I_E
$clan$	Function returning all ancestors of a node or an edge type
G_{bg}	booggie graph

9.6 Final assessment of the cabin configurator project by EADS

EADS Deutschland GmbH · 81663 München

Dipl.-Ing. Bergen HELMS

Technische Universität München
Lehrstuhl für Produktentwicklung
Boltzmannstrasse 15
D-85748 Garching

EADS Innovation Works
81663 München, Deutschland
Ansprechpartner: Milton Amador
Telefon: +49 (0) 89.607-21245
Telefax: +49 (0) 89.607-21240
e-Mail: milton_amador@eads.net

Datum: 20.12.2011

Projektresümee: Automatische, regelbasierte Layoutsynthese zur Ermittlung und Bewertung der Inneneinrichtung von Flugzeugkabinen

Die European Aeronautic Defence and Space Company (EADS) ist ein führendes Unternehmen der Luft- und Raumfahrt sowie der Verteidigungsindustrie. Das Geschäft des Konzerns hängt stark von der Entwicklung und Integration von Hochtechnologie in EADS-Produkten ab, die dem Konzern den nötigen Wettbewerbsvorsprung in seinen Märkten sichern. EADS Innovation Works betreibt die Forschungs- und Technologie-Labore des Unternehmens. Diese sichern durch ihre langfristige Ausrichtung das technologische Innovationspotenzial des Unternehmens.

Im Rahmen dieses Projekts sollte basierend auf dem Ansatz der objekt-orientierten Graphgrammatiken sowie der Software booggie eine automatische, regelbasierte Layoutsynthese für die Bewertung der Inneneinrichtung von Flugzeugkabinen entwickelt werden. Es sollte unter anderem geprüft werden, ob der entwickelte Ansatz bei der EADS-Tochter Airbus zur Erzeugung und Bewertung von Konfigurationsalternativen eingesetzt werden kann.

Ausgangslage und Motivation

- Die Konfiguration der Innenausstattung von Flugzeugkabinen ist ein wissensintensiver Prozess, der bislang manuell von Experten durchgeführt wird.

- Das Ziel ist der Entwurf einer optimalen Zusammensetzung der Kabinenelemente im Passagierbereich (Sitze der diversen Klassen, Stauraum, etc.) und im Servicebereich (WC, Trolleys, Flugbegleitersitze, etc.).

- Es gilt, geometrische Zwangsbedingungen sowie airlinespezifische, fertigungsspezifische und sicherheitsrelevante Anforderungen zu berücksichtigen.

EADS

- Der Entwurf ist gemäß diverser Designparameter (z.B. Verhältnis Sitze/Trolleys) zu optimieren.

- Der Raum gültiger Konfigurationen ist zu komplex und umfangreich, als dass er vom Entwickler manuell vollumfänglich untersucht werden kann. In der Folge werden Konfigurationen erzeugt, die die nötigen Zwangsbedingungen erfüllen, aber keine optimale Lösung darstellen.

- Eine systematische Betrachtung des Lösungsraums erfordert vom Entwickler ein hohes Maß an Erfahrung und Expertenwissen.

- Die Erzeugung und Bewertung von Konfigurationsalternativen ist sehr aufwändig, somit ist ein schnelles Reagieren auf veränderte Kundenanforderungen nur begrenzt möglich.

- Basierend auf dem Ansatz der objektorientierten Graphgrammatiken sollen automatisiert Konfigurationen erzeugt, validiert und bewertet und in die Toolkette integriert werden.

Lösungsansatz und Methodik

- Die Kabinenelemente der Airbus A320-Familie werden mit ihren Schnittstellen in einem hierarchischen Metamodell definiert.

- Das Wissen über die Einzelschritte zur Kabinenkonfiguration und Validitätsprüfung wird regelbasiert formalisiert.

- Die logische Verknüpfung der Einzelschritte zu einem Gesamtvorgehen erfolgt über Regelsequenzen.

- Vordefinierte Teilkonfigurationen können über eine XML-Schnittstelle importiert werden.

- Die erzeugten Konfigurationsmodelle können über Designparameter miteinander verglichen und bewertet sowie in einer interaktiven 3D-Darstellung visualisiert werden.

Fazit und Ausblick

- booggie ermöglicht die automatisierte Konfiguration der Innenausstattung von Flugzeugkabinen unter der Berücksichtigung der gegebenen Zwangsbedingungen und Designparameter. Wesentliche Aspekte der Lösungsraumbetrachtung und Workflowautomatisierung können mit booggie umgesetzt werden.

- Es konnten Lösungsräume mit bis zu ca. 20.000 Konfigurationen generiert und zur Laufzeit exportiert werden. Eine umfassende Lösungsraumbetrachtung war nicht möglich, da hier weitaus

EADS

mehr Konfigurationen erzeugt werden müssten. Aufgrund von Systemabstürzen ist dies im derzeitigen, prototypischen Stadium von booggie nicht möglich.

- Die booggie zugrunde liegende Methodik stellt einen vielversprechenden Ansatz zur Lösung multikriterieller Konfigurationsprobleme dar. Der jetzige Reifegrad der Software erlaubt keinen Einsatz im Entwicklungsalltag oder als Konfigurationswerkzeug im Vertrieb. Die Zuverlässigkeit, Benutzbarkeit und Effizienz müssen dafür noch verbessert werden. Außerdem muss der Nachweis erbracht werden, dass große Lösungsräume umfassend betrachtet werden können. Des Weiteren ist derzeitig die semantische Expressivität bei der Definition von Metamodellen zu gering, um mittels logischen Schließens Konfigurationen zu validieren.

- EADS verfolgt mit Interesse die Weiterentwicklung der Methodik und der Software, da derzeitig keine kommerzielle Alternativlösung für diese Art von Problemen bekannt ist. Die bisherigen Ergebnisse und Daten stehen dem Gründerteam zur Untersuchung zur Verfügung. Sollten die jetzigen Problemfelder ausgeräumt sein, steht EADS dem Gründerteam als Diskussionspartner zur Verfügung und wird eine Prüfung der Technologie erneut in Betracht ziehen.

Index