COMPLEX SYSTEM DESIGN WITH DESIGN LANGUAGES: METHOD, APPLICATIONS AND DESIGN PRINCIPLES[1]

S. Vogel[1], S. Rudolph[2]

Institute for Aircraft Design, University of Stuttgart, Stuttgart, Germany
[1]Samuel.Peter.Vogel@gmail.com, [2]Rudolph@ifb.uni-stuttgart.de

Abstract

Graph-based design languages are presented as a method to encode and automate the complete design process and the final optimization of the product or complex system. The Unified Modeling Language (UML) is used to represent the design language which models the design process. A design language consists of a vocabulary (i.e. the digital building blocks) and a set of rules (i.e. the digital composition knowledge) along with an executable sequence of the rules (i.e. the incremental digital encoding of the design process). The rule-based mechanism instantiates a central and consistent global product data structure (the so-called design graph). Upon the incremental generation of the abstract central model, the domain-specific engineering models are automatically generated, remotely executed and their results are fed-back into the central design model for subsequent design decisions or optimizations. The design languages are manually modeled and automatically executed in a so-called design compiler. Up to now, a variety of product designs in the areas of aerospace (satellites, aircraft), automotive (space frame structures, automotive cockpits), machinery (robots, digital factory) and consumer products (coffeemakers, exhaust systems) have been successfully accelerated and automated using graph-based design languages. Different design strategies and mechanisms have been identified and applied in the automation of the design processes. Approaches ranging from the automated and declarative processing of constraints, through fractal nested design patterns, to mathematical dimension-based derivation of the sequence of design actions, are used. The existing knowledge for a design determines the global design strategy (i.e. top-down vs. bottom-up). Similarity-mechanics in the form of dimensionless invariants are used for evaluation to downsize the solution for an overall complexity reduction. Design patterns, design paradigms (i.e. form follows function, or function follows form) and design strategies (divide and conquer) from information science are heavily used to structure, manage and handle the design complexity.

Key words: *design languages, design automation, design method, design principles, design ontology.*

Citation: *Vogel S., Rudolph S.* Complex System Design with Design Languages: Method, Applications and Design Principles. *Ontology of designing*. 2018; 8(3): 323-346. - DOI: 10.18287/2223-9537-2018-8-3-323-346.

Introduction

The digitization of industrial processes, e.g. in the context of Industry 4.0, makes new design processes possible and necessary. The automation of the product development process promises a considerable increase in efficiency. Especially designs and decisions of the very early concept phase have a very large influence on the later life cycle costs of the product [1]. The development of modern and more competitive products requires to go even closer to the limits of what is physically feasible in order, for example, to squeeze the last bit of weight advantage or efficiency out of a product or system. Modern products are integrating typically multiple physical domains (mechanics, thermodynamics, electronics, logistics, …) as well as a lot of system levels consisting of sub-systems or parts that mutually build on each other. The combination of both, multiple domains together with a number of system entities, results in a high level of design and process complexity that has to be handled. Digitized design processes can be used to cope with this complexity and to find more optimal product designs in even earlier project phases. This digitization mainly comprises the comput-

[1] The article is published in expanded content on the recommendation of the Program Committee of the XX International Conference "Complex Systems: Control and Modeling Problems" (CSCMP-2018). Samara, Russia. September 3-6, 2018.

er-aided synthesis of designs (CAD) including the automated generation of functional validation calculations and simulations (structural mechanics, fluid mechanics, controls...). In fact, a virtual product design shall be automatically generated and optimized based on given product requirements to optimally meet the performance targets.

In this paper graph-based design languages are presented as a method to implement such digital and re-executable representations of (conceptual) design actions. At first, the method of graph-based design languages itself is explained. The method is proven for more than fifteen years and has been mainly developed in the Similarity Mechanics Group of the Institute for Statics and Dynamics, which moved now to the Institute of Aircraft Design at the University of Stuttgart. Second, scientific applications are shown as well as an early industrial stage application. Finally, a collection of design principles to handle the complexity in product design is presented that has been identified in the scientific work with design languages over the past years.

1 Method

The method of graph-based design languages [2] is a further evolution step of generative, computer-based design synthesis methods [3]. These design synthesis methods can be divided in to string-based, shape-based and graph-based design representations. From the viewpoint of the authors, graph-based design languages belong to the most generic and abstract means of knowledge representation across different domains due to its graph representation. Alternative computer-based synthesis methods such as L-Systems or Shape Grammars [3] define a rule set on elementary shapes (vocabulary) which is recursively called in a production system to generate more complex shapes. Graph-based design languages expand this concept by generalizing the vocabulary to conceptual objects together with an adaptive procedural rule sequence, the so-called production sequence. Along with these approaches, there are other solutions in formalizing the process of designing and creating automatic design systems based on formal knowledge worth noting [4-9].

1.1 Philosophical Motivation

Rudolph gives a philosophical motivation for the design language concept in [10]. First it is observed that during the product design process different areas of concept, each with different levels of knowledge, are traversed. It is distinguished between the first area called ‘believe’ which covers uncertain design targets as *simplicity*, *aesthetics* or *adequacy*. The second concept is ‘ability’ which covers more concrete but still not exact formulated design aspects as *Design for Manufacturing, Design for Assembly, Design for Recycling*. The third concept covers exact aspects and is called ‘knowledge’. It contains physical formulas and other reproducible, mathematically formalizable and provable laws and know-how. Figure 1 shows a schematic design process starting from an idea that is hosted in the concept of ‘believe’. During the iterative design process different solution concepts are derived from the given idea and product embodiments are synthesized as product variants. These variants are validated towards specified requirements. The iterative procedure is conducted until a variant meets the requirements and becomes the final product, see Figure 1.

In order to formalize and digitalize this design process all three aspects have to be represented in a single, unified description. Rudolph proposes a language-based representation that is closely related to natural languages which are a convenient candidate as they are able to cover all three conceptual areas presented above. This language-based representation is called a *graph-based design language*. The linguistic aspects of natural languages are reinterpreted here in the engineering application in the following way [2].

- *Syntax of the design language:*
  All designs that can be combinatorically represented by the classes in the class diagram.

- *Semantics of the design language:*
  All designs that are technically or physically meaningful (e.g. it may depend of the application whether a car with six wheels makes sense, no collisions between parts and components, …).
- *Pragmatics of the design language:*
  The designs that are optimal with respect to given requirements and boundary conditions.

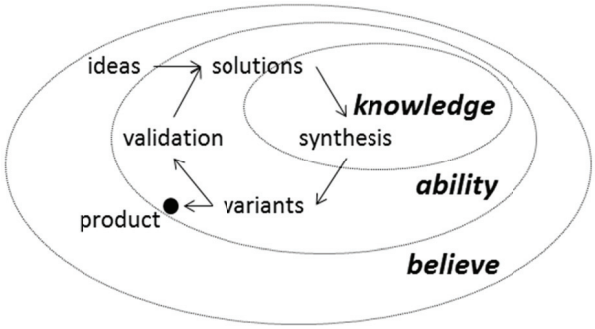

Figure 1 - Embedded areas of concept that are visited during an iterative product design process, starting from an idea. Figure reproduced from [10]

The linguistic aspects are embedded within each other as the optimal designs (pragmatics) are a subset of the technically and physically meaningful designs (semantics). These meaningful designs are at the same time a subset of the designs that can be created or represented in the design language. But this is in fact the underlying research hypothesis of design languages that *design is computable* and that the optimal designs as a subset within the generated meaningful designs may be found via a *design evaluation method* [11].

1.2 Graph-Based Design Language

The following section explains and defines the method of graph-based design language. The figure 4 shows the main components of a design language in its graph-based representation on one page.

1.2.1 Unified Modeling Language

For the formal representation and the concrete modelling of the design process the Unified Modeling Language (UML) is used [12]. The UML's origin is in object-oriented software engineering to graphically model and specify object oriented software. Subsets of the UML are used in systems engineering as the modeling language SysML. The UML provides ready-to-use modeling diagrams and tools for the representation of (engineering) ontologies (class diagram and instance diagram) as well as for representing sequential and branched processes (activity diagram) that are used in graph-based design languages to create a graphical model of a product's design process. The original naming of the design languages building blocks was based on linguistics as stated above as the UML was later introduced as modeling language [13]. The following headers reflect the duality between the design languages components' UML nomenclature and the 'original' linguistic based nomenclature in parenthesis.

Using the UML for the representation of the product design process allows a natural integration of the software domain into the product design process, which is present in most modern products. Modern products are typically made up of mechanical and electrical parts with a corresponding control unit that executes at least a product-specific controller software. Therefore software engineering becomes an integrated and important part of the product development process.

1.2.2 Class Diagram (Vocabulary)

The class diagram represents an ontology of the product that is to be designed. The product is decomposed in to its subsystems, components and even more granular entities that are assigned to classes. These classes are enriched with parameters that represent e.g. physical or cost variables. In this way, parametrized objects that are instantiated from classes form the vocabulary of a graph-based design language instead of words in natural languages. A class diagram is shown in figure 2 top for an exhaust aftertreatment system that reduces pollutants from a combustion engine's exhaust. The class *Catalyst* is a kind of *ExhaustSystem* through inheritance. Within the class diagram, associations can be drawn as links between classes to represent relationships between them. In the example in figure 2 top, the *ExhaustSystem* is connected to a *CombustionEngine* whose exhaust has to be cleaned and the *ExhaustSystem* itself is connected with the *Environment,* where the cleaned exhaust gas escapes to. This associations define which elements can be linked during an instantiation. Thereby the class diagram represents the (maximal combinatorial) template of the product that hosts all the information needed during the design process. Inheritance relationships can be defined between the classes, as is customary in object-oriented modeling. Abstract classes can be defined that cannot be instantiated. So components and entities can be mapped directly on classes on different levels of abstraction and detail.

Equations and constraints between the class parameters can be additionally modeled in the classes and are processed in an integrated solution path generator [14]. The equation and constraint network that is built on the instantiation of the classes is automatically solved in the solution path generator with an integrated computer algebra system. In the UML class definition physical dimensions can be assigned as data types to class parameters. This becomes especially important for the dimension analysis presented in the design principles section below.

1.2.3 Instance Diagram (Design Graph)

The classes from the class diagram can be instantiated into instances. The instantiated objects get an unambiguous name and the parameters defined in the class are provided with concrete values. The instances of associated classes can be linked with each other. The set of linked instances is called *design graph* in the context of graph-based design languages (figure 2 bottom). The instantiated objects form the nodes and the links form the edges of the graph. The specific values are stored in the parameters within each node. Thus, the topology of a product (an alternative name would be product architecture) can be mapped via the graph and the parametric of a product via the parameterization in the nodes. This design graph plays the role of the central data model in the virtual product design with design languages.

1.2.4 Rules (Grammar)

The engineering entities of the class diagram are rule-based instantiated into objects with specific parameter values. Graphical rules with a left-hand side (LHS) and a right-hand side (RHS) define the instantiation as manipulation on the instance diagram. Again, the design instances in the design graph are linked with each other according to the associations that are defined in the class diagram. This associations define the possible connection of instances, also called instance patterns, in the graphical rules. The instance pattern on the LHS of the graphical rule is looked-up in the design graph and replaced with the instance pattern on the RHS. The first rule is called 'axiom' and has an empty LHS as the design graph is empty in the beginning. The figure 2 center left shows the axiom rule that introduces the boundary conditions of an exhaust aftertreatment system, which comprises of a given combustion engine with its specific parameters (not all shown), the environment and the installation space as STEP geometry file that defines the available space for engineering the exhaust system. In the 'axiom' typically the requirements and given boundary conditions are

defined. The graphical rule in figure 2 center right shows the incremental design step that adds the main building blocks of an SCR system, *DEF_Injection* and *Catalyst*, to the initially created instance of the *CombustionEngine* class.

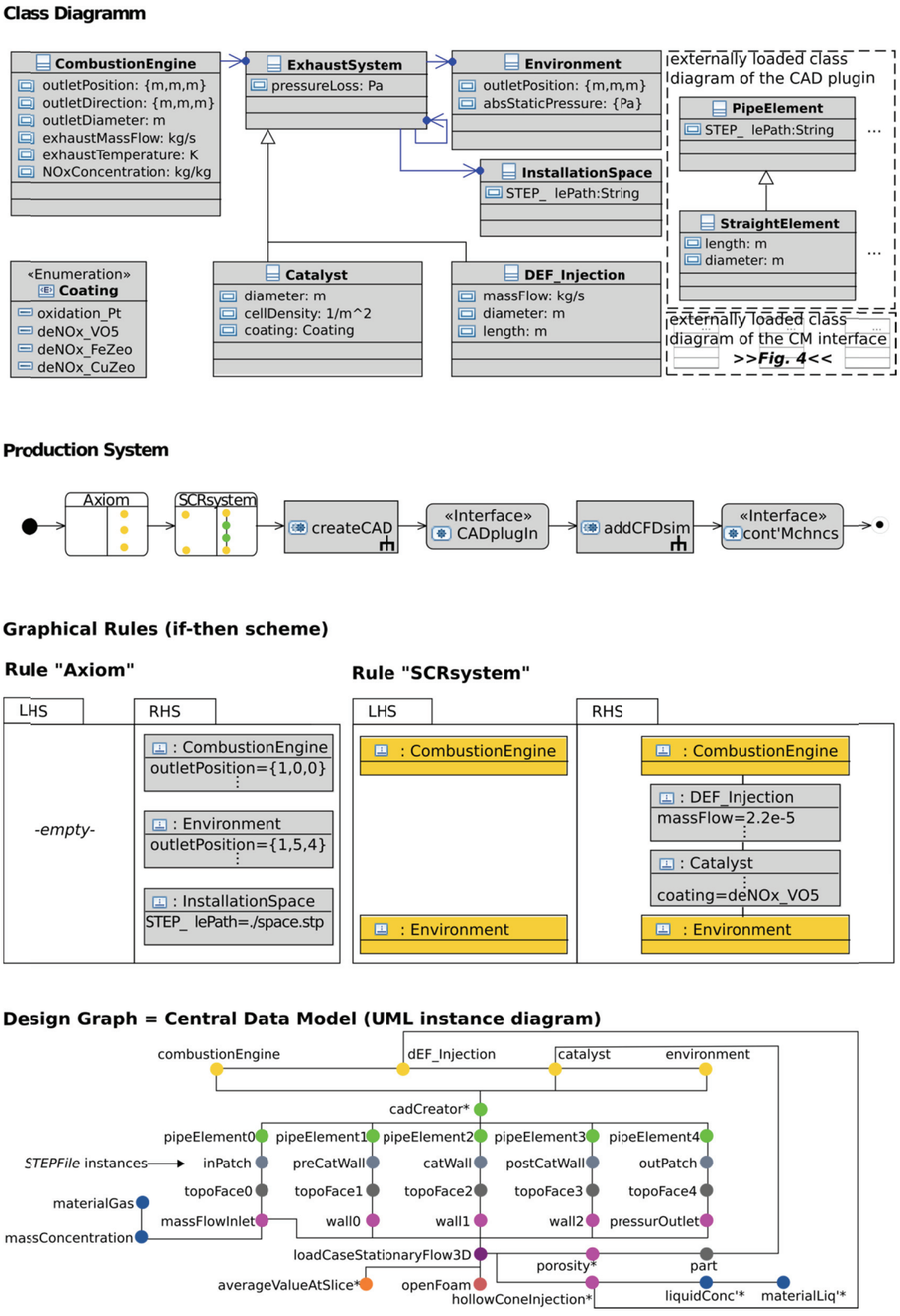

Figure 2 - Schematic graph-based grammar of an exhaust system and its main building blocks

1.2.5 Production System

The design process itself is then split into an incremental rule sequence. This predefined rule sequence in graphically modeled in an activity diagram and is called production system. Beside the graphical rules as building blocks an activity diagram can hierarchically host sub-activities which are activity diagrams themselves. As third object, interface calls can be modeled that trigger the execution of engineering applications in so-called process chains that are described in the following section. The activity diagram in figure 2 top center shows from the left two calls of graphical rules followed by a two times alternating sub-activity and interface call. As fourth element so-called decision nodes are available to branch the rule sequences in the production system in dependence of the state of the design graph (figure 3).

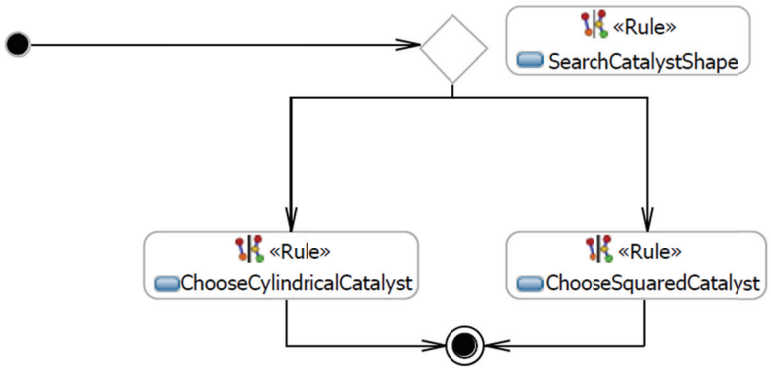

Figure 3 – Decision node (rhombus) in the activity diagram to trigger different rules based on the LHS pattern in rule *SearchCatalystShape*

With this feature a branching design process can be realized that adaptively and dynamically reacts on different model states and inputs as different rule sequences can be triggered. In figure 3 one of the two *Choose*Catalyst* rules is triggered, which differ in the geometric form of the catalyst that is added to the model. Which of the rules is chosen depends on a parameter in an instance on the LHS of the *SearchCatalystShape* rule on the top right in figure 3. In this way adaptive design sequences can be realized to create a wide variety of different product configurations.

1.2.6 Information Architecture

The figure 4 shows the information architecture that hosts the previous presented elements and shows their interaction. The class diagram (vocabulary), the rules, as well as the production system, represent a digital blueprint of the design process, wherein the design knowledge is encoded in the form of a design language (figure 4 left). The manual modeling of the class diagram, the rules and the production system is done in a so-called design compiler (further details in the following section). On the right side of figure 4 so-called process chains are shown. Process chains represent the automated creation of engineering models in the design language. This is realized with unidirectional model-to-text transformations between the abstract central model (design graph as UML instance diagram) and the product engineering models as CAD, simulation models, etc. The process chains extract the required information from the central design graph and create the CAD and simulation models automatically by executing the model-to-text transformations that are stored in the process chains. The results of the simulations runs are partially fed back to the design graph for influencing subsequent design rules and operations. Closed-loop optimization as well as adaptive and self-controlling design processes, using the decision nodes above, can be realized in this way.

This architecture with a central model ("*single source of truth*") has multiple advantages. There are no longer different and/or outdated model versions since upon a model update in the central

model the process chains can be triggered to get an automatic update of all engineering models. There is no need of model-to-model transformations between each engineering model as all model updates are done or triggered in the central model with an subsequent update via calling the process chains. This reduces the number of model-to-model transformations drastically, that are necessary to have valid models, and drastically reduces the need for tracing and managing model changes.

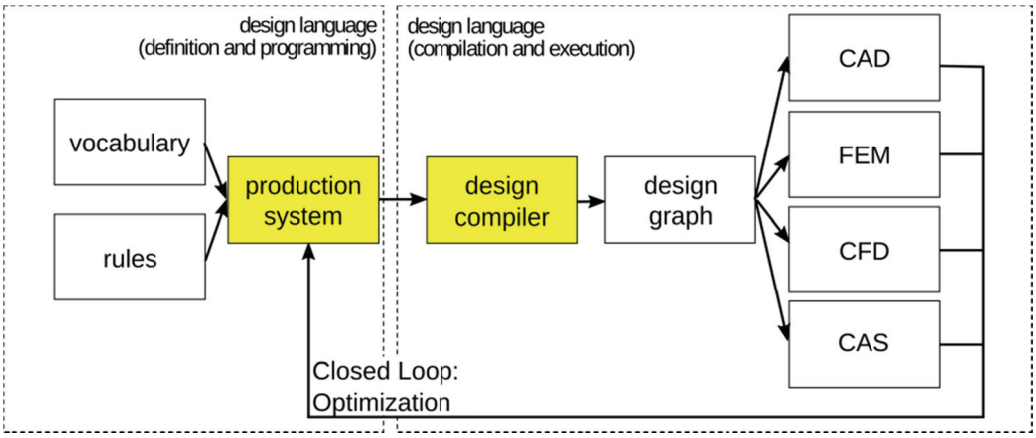

Figure 4 - Information architecture of graph-based design languages

1.2.7 Design Compiler

In analogy to programing languages a design compiler is used for modelling and execution of the graph-based design languages. The commercially available Design Compiler 43 (https://www.iils.de/) in its Version V2 was used in the applications projects presented below. The Design Compiler 43 is a standalone application based on the Eclipse IDE that provides the following functionalities and tools to model and execute graph-based design languages:

- Graphical editor to manually model the class diagram (figure 2 top).
- Process chains as engineering plugins in the form of class diagrams for modeling engineering tasks as CAD geometry creation, simulation model creation (CFD, FEM) and product integration problems (wiring, piping, packaging,…) that can be loaded into the central class diagram (figure 2 top right). In this way different ontologies from different engineering areas can be combined in the central class diagram.
- Graphical editor to manually model the production system including its sub elements as graphical rules, sub activities and interface calls (figure 2 center).
- Graphical view of the design graph with filter function to restrict the scope of view to sub patterns (figure 2 bottom).
- Design language debugging mode to execute the rules of the production system step by step with the possibility to stop during execution.
- *Java Rule* functionality to execute and use Java code on the design languages execution to manipulate the design graph code-based, especially useful for complex manipulations that contain difficult patterns and/or control structures as loops.

The solution path generator functionality is provided by the design compiler as plugin which solves, first, the equation system modeled in the class diagram in terms of a solution sequence and, second, passes the found solution sequence to a computer algebra system for solving the equation system symbolically or numerically if necessary. The bidirectional solution path generator writes the results back into the design graph as last execution step of the plugin.

2 Applications

A broad range of products has been designed using graph-based design languages. From applications in aerospace [15, 16] through consumer products [17] and off-road machinery components [18] up to automotive [19], the method has been successfully applied mainly in a scientific context. The scope of design languages has been prototypically extended up to downstream stages of the life-cycle by generating and designing the digital factory for a product in addition to the product itself [20]. Additional work has been done in the implementation of algorithms to automate engineering tasks as routing of cables and wires [16] and the automated creation of pipework in given installation spaces [21]. These intelligent wiring and piping algorithms become necessary as graph-based design languages fully automate the design process and therefore need to be able to create an intelligent integration and interaction of system components in given installation spaces for different product architectures.

2.1 Aeronautics: Air Cabin Design

Figure 5 shows results of a graph-based design language that automates the layout design of an aircraft cabin [22, 16]. Beginning with the requirements, the designer can (manually) define the seating requirements based on a number of ratios. A ratio might define how many passengers share the same lavatory in a certain class. The aircraft main dimensions are as well given as a requirement. The proposed seating configuration within the aircraft's hull can be additionally manually edited in a graphical editor that appears during the execution of the design language. The subsequent design process is schematically shown in figure 5. At first, an initial CAD model of the aircraft cabin is created (figure 5 top left) based on the previously found or manually edited seating configuration. From this CAD model the available routing space for cable routing is rule-based extracted. Due to the previous automated CAD model generation, the information of the position, size and shape of the area, which is accessible for routing, is explicitly available in the central data model (figure 5 top mid). In the following step the equipment boxes are positioned in the available routing space. This is done via a parametrization along the aircrafts main dimensions which can be later varied in an optimization run (figure 5 top right). Afterwards, the equipment boxes are subtracted from the routing space and the remaining space is meshed for conducting collision detections for the subsequent cable routing (figure 5 left bottom). Then the search algorithm, a modified A*-algorithm which is available as engineering plugin in the design compiler, is executed to identify the cable routes within the routing space (figure 5 mid bottom). Finally, collision-free CAD models of the cables are created which are used for an evaluation of the aircraft cabin configuration. Evaluation metrics comprise of cable length and weight as well as electromagnetic compatibility. Additional constraints in the aircraft cable design, as minimum distances between the cables of redundant systems, can also be taken into account by the design compiler's integrated routing functionality.

Using the graph-based design language reduces the time needed for an aircraft cabin layout from many weeks and months to a few hours. The complexity of the interaction of the coupled systems and components is handled through the interplay of the production system and the design rules together with the intelligent algorithms to solve the integration tasks of positioning and wiring the electrical components. Different aircraft cabin designs can thus be automatically evaluated and optimized in terms of total weight, cable length and wiring compatibility and validity [22]. Figure 6 shows a final cable routing together with the comparison of the results of two cabling variants with a different number of distribution boxes (SPDB). The diagrams on the right are showing the different resulting cable weights for the different networks and subsystems.

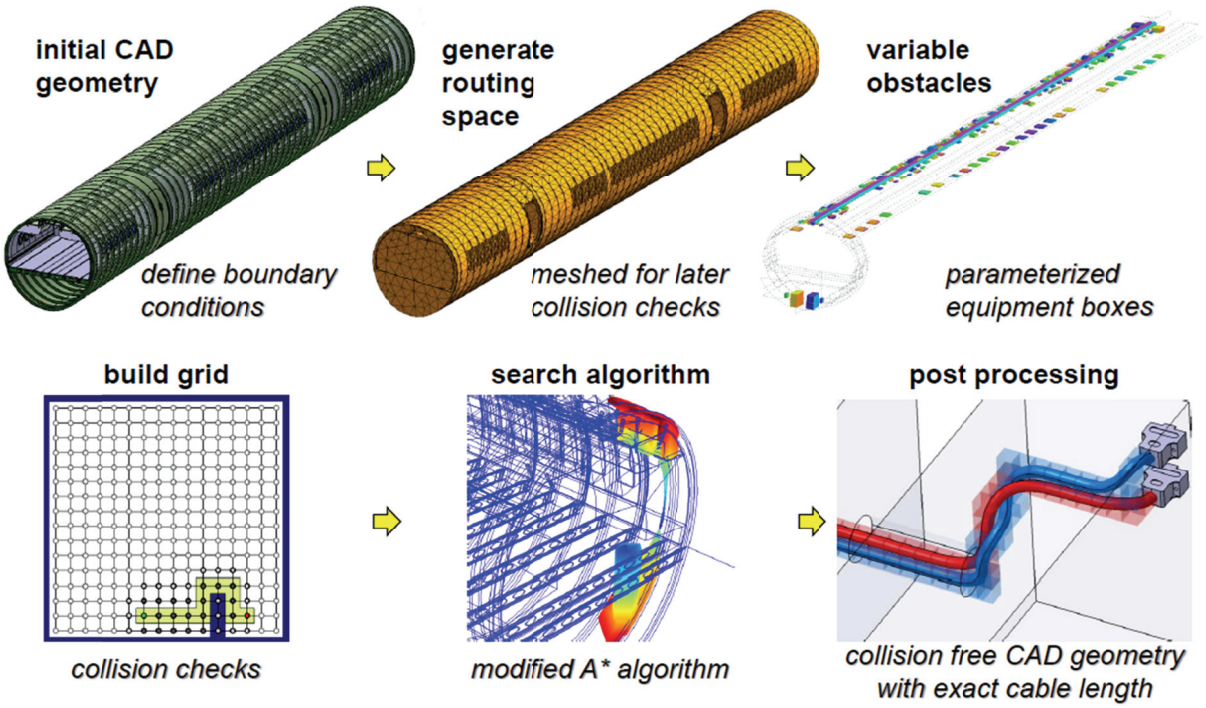

Figure 5 - Aircraft cabin design with automated generation of electric and climate system generated with a design grammar [22]

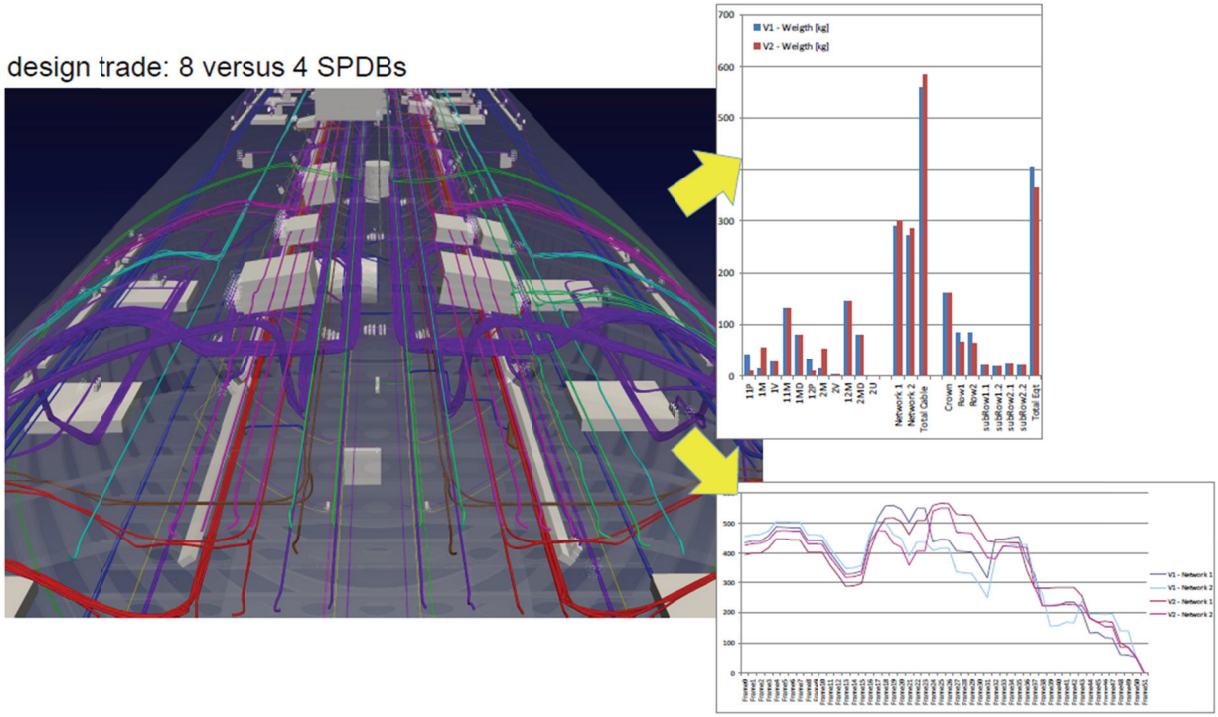

Figure 6 – Final aircraft cabin cable configuration in overhead area. Diagrams on the right show a schematic comparison of two cable variants (8 vs. 4 distribution boxes) in terms of cable weights [22]

2.2 Aerospace: Satellite Design

Figure 7 shows the result and intermediate steps of a graph-based design language for creating a small satellite [15]. The design language starts from a given mission that defines payload, target

orbit, energy and information demand of the payload as requirements. The design process of creating an optimal satellite for a given requirement set starts in [15] by solving the so called *enumeration problem.* In the enumeration problem different system topologies are synthesized and analyzed as different system topologies can match the requirements. The design for a given system topology is created rule-based in the design language presented in [15]. For each system topology the *configuration problem* has to be solved which comprises of integrating the subsystems and components, selected during solving the enumeration problem, in the available space (the so-called packaging). At last, the *integration problem* has to be solved, which means that the functionality of the whole systems has to be checked under all relevant loads and physical conditions. For conducting this integration test the corresponding simulation models, as well as the cables routes of the satellite, have to be generated. These steps have to be executed and repeated many times on different modeling levels of detail to properly resolve the couplings between the systems [15].

Figure 7 shows the synthesis of the FireSat satellite [15]. The rule set creates a complete virtual mockup based on the given requirements (figure 7 top row). The design language includes the automated creation and dimensioning of the control systems as well as the validation. Critical figures, such as the mass, energy and momentum balance, are calculated and balanced. A mission-related communication system is chosen based on the mission figures. All critical subsystems are selected and in the subsequent configuration step spatially arranged (i.e. packaged) and finally, with the previously mentioned routing algorithm, connected by wires (figure 7 bottom right). The integration of the components is validated for the defined orbits in terms of thermal loads arising from the incoming sun light as well as heat sources from components and systems (figure 7 bottom left).

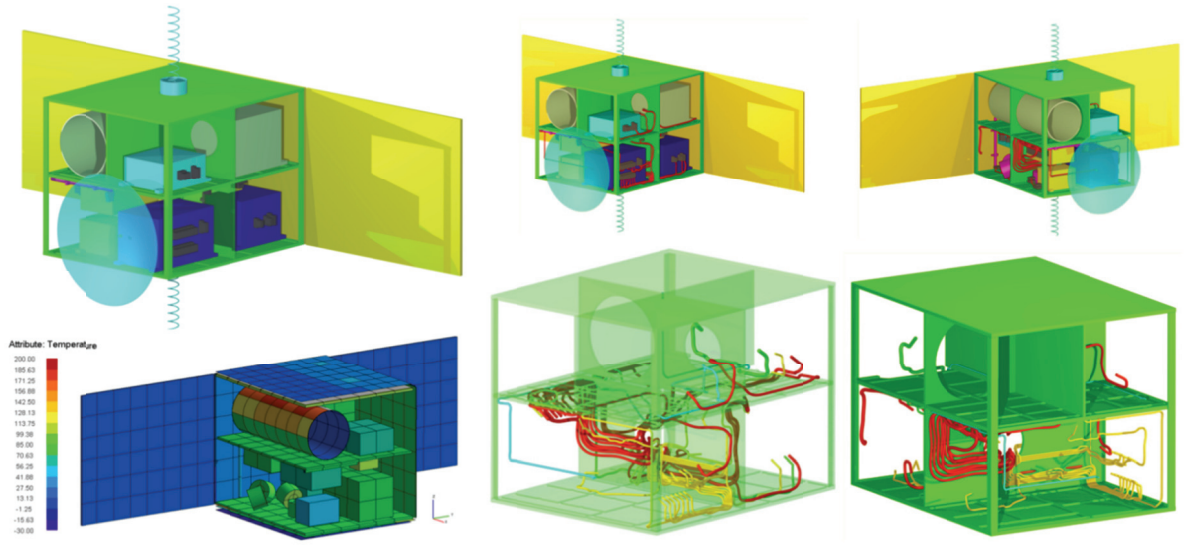

Figure 7 - Satellite design with graph-based design languages. Reproduced from [15]

Figure 8 shows the intermediate result of the FireSat design language for solving the enumeration problem of the communication system. For the given requirements transmission power and transmission data rate (speed) the best mass of each communication system topology can be seen.

Each combination of a transmitter and an antenna presents a unique topology and has its own mass to power to speed characteristic as shown in figure 8 in different colors in the central plot. During the execution of the design language this characteristic figures can be automatically created. Then the optimal communication system for a given requirements can be chosen based on this characteristic map. In cases where a chosen (sub-)system topology has still variable system parameters, a parametric heat map as in figure 9 can be used to find an optimal solution and to gain insight into

the system's behavior. The sensitivities of the calculated system characteristics values (vertical axis) are shown in dependence of given design parameters (horizontal axis).

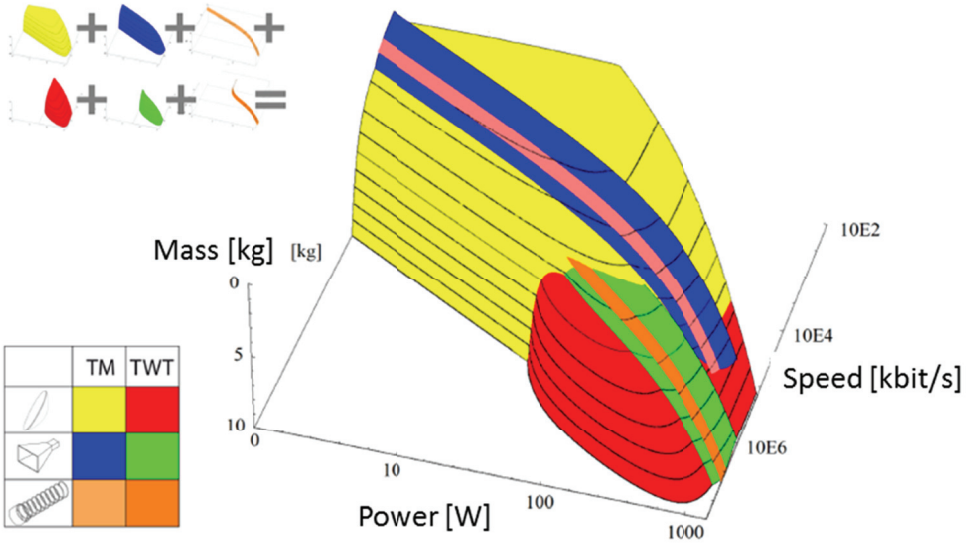

Figure 8 – Enumeration problem: The mass of different communication system topologies (colors) for required transmission power and speed. There are 3 antenna system variants (vertical) and 2 transmission amplifier variants (horizontal) [15]

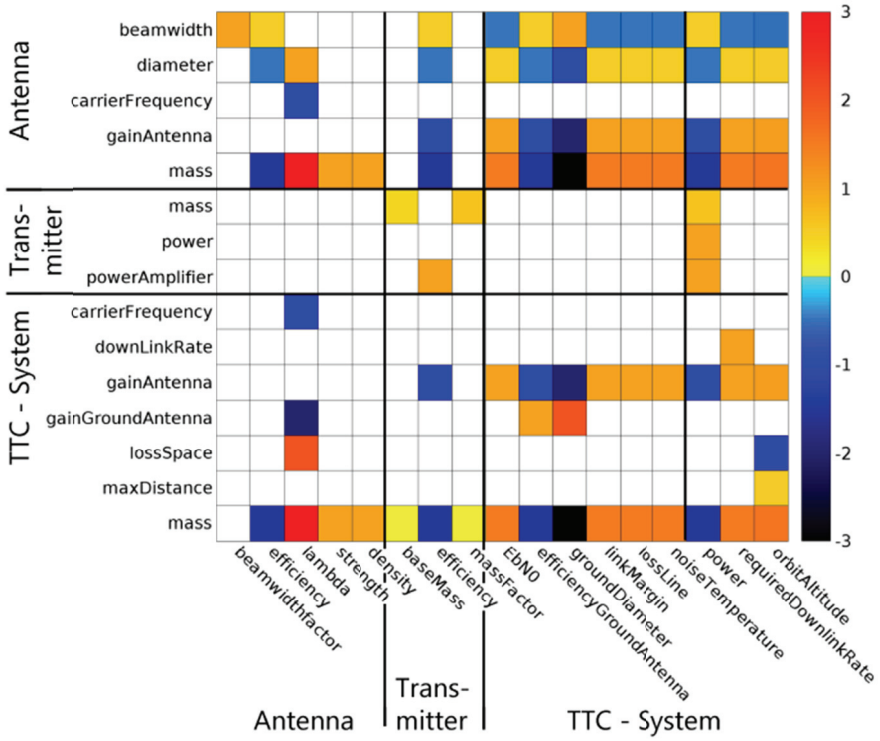

Figure 9 – Heat map showing the dependencies between the calculated communication system characteristics (vertical) and the given design parameters (horizontal) of a single communication system topology [15]

2.3 Manufacturing: From Design to Digital Factory

Graph-based design languages have also been proposed to bridge the gap between product design and manufacturing [20]. Design languages can support to close the current inconsistencies between the product design and the manufacturing engineering. Both disciplines are usually using different models, applications and data formats, even if both areas are closely coupled as changes in the product design can have a significant impact on the manufacturability in a given production infrastructure and vice versa. Arnold et al. implements a graph-based design language in [20] that automates the design of an aircraft panel. A plugin that automatically creates a digital factory for a given design is added. Using this approach in both, the design and the manufacturing planning, a central data model is shared in the form of the common design graph that extends over both interwoven domains.

Figure 10 shows four assembly configurations of an aircraft panel for finding the optimal configuration with the minimal turnaround time [20]. With this approach, the optimal digital factory setup can be found for a specifically designed aircraft panel to optimize the manufacturing process in the virtual reality before building the real factory. The design language in [20] provides a fully automated design stage of an aircraft panel based on given aircraft hull geometries and shapes. The simulation and optimization of the manufacturing process is realized with a corresponding design compiler plugin to the domain specific DELMIA digital factory software (www.3ds.com). The digital factory model is generated out of the central data model, the digital factory simulation is triggered and the turnaround times as well as information on the manufacturability of the panel are returned. Figure 11 shows the simulation of the coating process of the aircraft panel that is also created out of the design language's central data model. With this coating simulation critical coating parameters as local thickness and application efficiency can be optimized by a variation of the spray paths and spray configurations. It was thus possible to show in an exemplary manner that production processes and their process parameters can also be optimized with design languages [20].

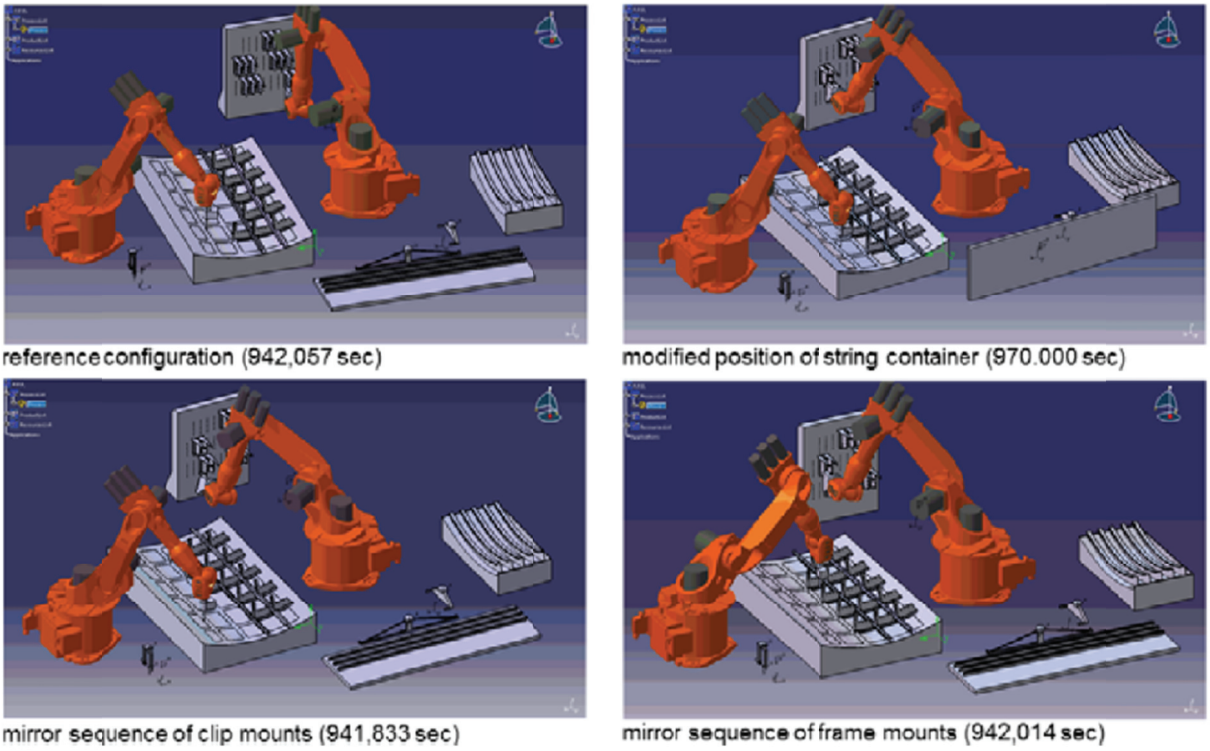

Figure 10 – Different digital factory assembly configurations with turnaround times [20]

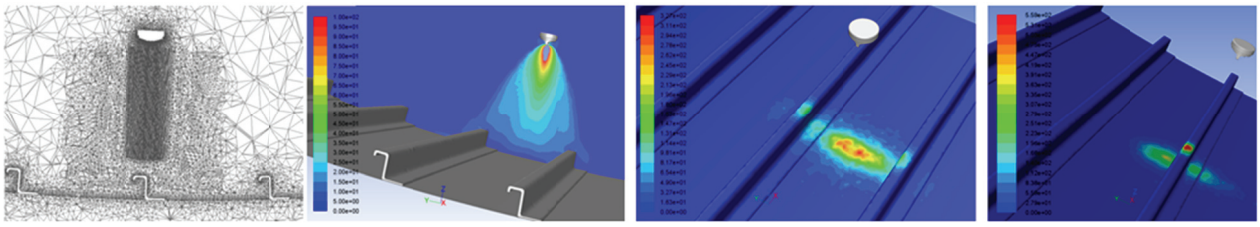

Figure 11 – Simulated coating process of the aircraft panel. Courtesy Fraunhofer IPA [20]

This approach is being further developed in a currently running project ("ZAFH Project Digital Product Lifecycle"). The project tries to expand the applicability of graph-based design grammars to the whole product life cycle. Among other things, production lines in automotive applications are created [23] especially with regard to costs, and their virtual commissioning is visualized within a virtual reality environment generated automatically in the design language [24]. The website of the project provides additional links to further publications (https://dip.reutlingen-university.de/).

2.4 Mechanical Engineering: Exhaust Aftertreatment Design

Finally, figure 12 shows results of a design language for the automated creation and functional optimization of SCR (selective catalytic reaction) exhaust aftertreatment systems that reduce the emissions of internal combustion engines [18, 21]. At the beginning, a combustion engine is given in terms of mass flow and exhaust temperature as well as raw emissions and emission target. An installation space is also given, where the exhaust aftertreatment has to fit in (blue box figure 12 top right) as 3D CAD STEP model. First, the catalyst is analytically dimensioned. Then a CAD model of the catalyst housing is created based on the previously calculated catalyst size. The catalyst box is automatically positioned in the installation space. This is done with a Dijkstra shortest path algorithm that identifies a path between the engine's exhaust outlet and the exhaust target tail pipe position at the interface to the environment. This path is searched within the installation space under the constraint of having a maximum distance to the installation space's walls. Installation positions for the catalyst box, that have a sufficient distance to the wall, are filtered from this path. In the next step a geometrical constraint pipework, made-up of only standardized pipe bends with predefined bend angles (eg. 45° and 90°), is created to connect the exhaust system components with the engine and the environment [21]. This pipework (figure 12 left) is identified within its own optimization runs that are nested within the production system of the overall design language that creates the whole exhaust system (figure 12 top right). The pipework can be created including CAD detailed mountings and connectors between the pipe elements.

Based on the pipework and the finally positioned catalyst box the overall CAD model is merged. This CAD model is automatically meshed and a fluid simulation is created to determine the emission reduction efficiency as well as the pressure loss of the system (figure 12 center bottom). A finite element simulation to evaluate the thermal expansion of the system is created and executed as well (figure 12 center top). The whole design language is integrated into a design of experiments (DoE) and optimization framework to conduct a design space exploration and to determine the Pareto-front as best possible trade-off between competing design targets (e.g. pressure loss and emission reduction efficiency). Figure 12 shows the results of a DoE run at bottom right. Each point in the plot represents one synthesized exhaust aftertreatment system with its characteristic evaluation figures pressure loss (horizontal axis) and outlet emissions (vertical axis) that both have to be minimized. The Pareto front is drawn as a line through those configurations for which there are no better configurations in both optimization targets. The system was proven to scale very well on high performance clusters. Thousands of variants could be created and evaluated within a few days up to

weeks. With this approach the physical limits of a given exhaust aftertreatment requirement set could be determined as Pareto front which is a very useful and valuable information, especially in an early project phase. The Pareto front gives an indication of the maximum system efficiency that can be achieved due to the physical limits for a given set of boundary and starting conditions (installation space, engine, ...).

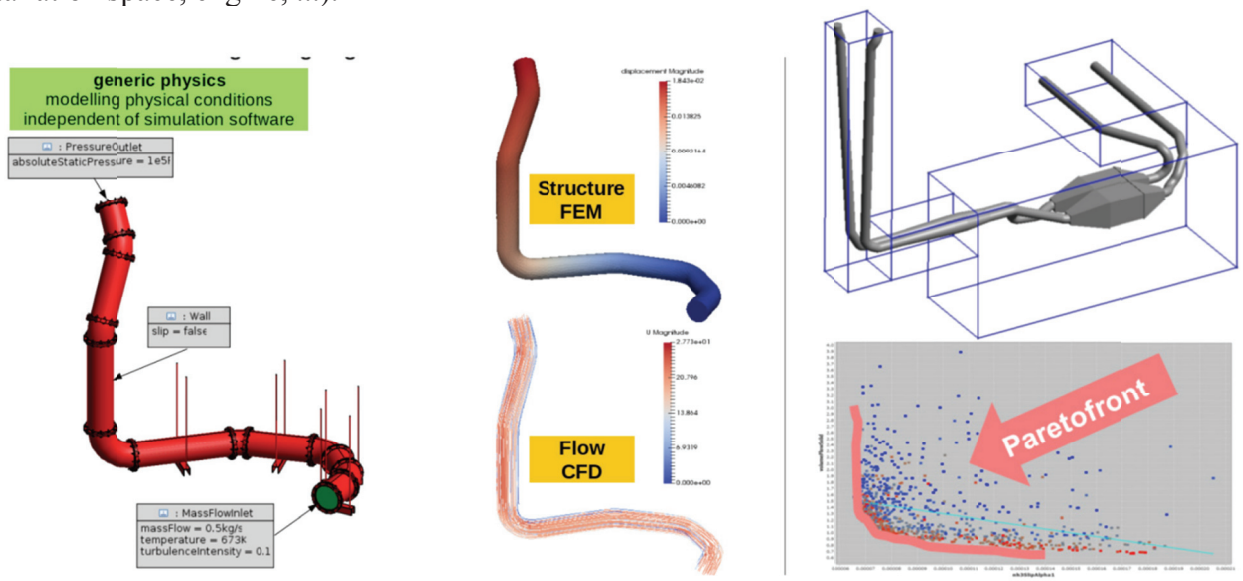

Figure 12 – Results of a graph-based design language for developing SCR exhaust aftertreatment systems in given installation spaces. Bottom right: results of DoE optimization runs with two conflicting optimization targets (horizontal: pressure loss, vertical: outlet emissions) with the resulting Pareto front. Reproduced from [18, 21].

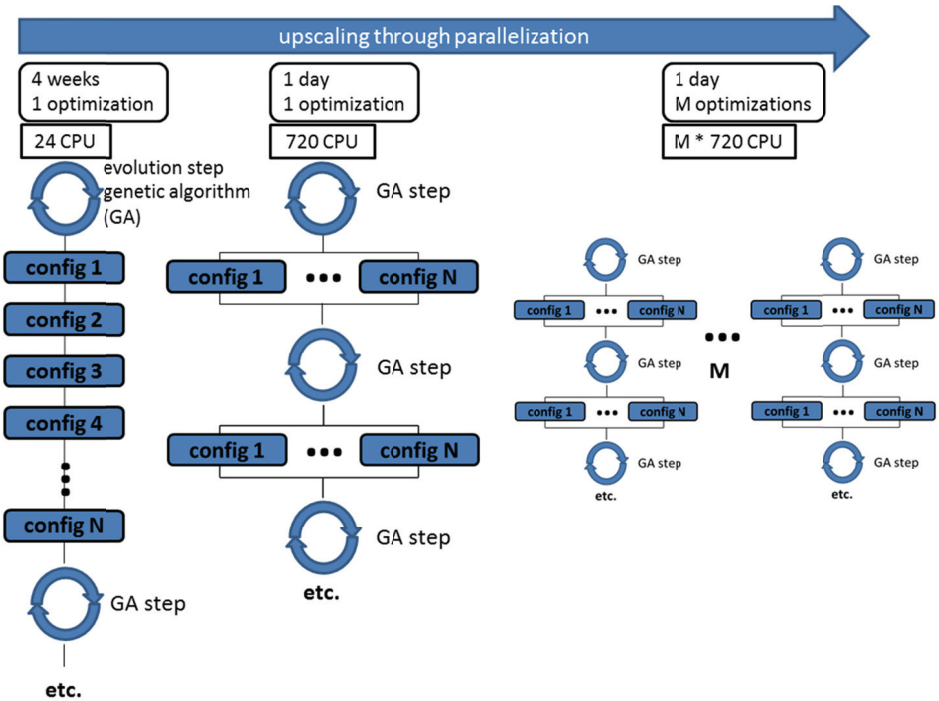

Figure 13 – Upscaling of design languages integrated into DoE/optimization runs.

The very good scalability of the design languages on HPC (high-performance computing) infrastructures could be demonstrated using the example of the design of exhaust aftertreatment systems. Figure 13 schematically shows the principle of upscaling the design process by executing the

genetic optimization(s) in parallel. The design language, which calculates a configuration based on a set of given requirements, is executed within a genetic optimizer. The configurations of the generation - one design language execution per configuration - can now run in parallel on an appropriate number of CPUs.

In the example in figure 13, this allows the optimization process to be reduced from several weeks to one day if the CPU number is correspondingly increased. There are no parallelization losses because each design language calculates an isolated configuration with corresponding input parameters, such as pipe diameter or dosing positions. This can be even further scaled up by a parallel execution of different optimization scenarios. The maximum runtime of the optimizations depends then only on the available computing power, since the entire design and validation process of the product is digitally mapped in the graph-based design language and can be re-executed as required without any manual intervention.

3 Design Principles

Different design strategies and mechanisms have been identified and applied in the automation of the design processes. Some principles have been identified by working on the explicit scientific problem of handling the complexity of engineering design and working on an engineering design theory. Others just emerged in application-centered design language projects as by-product.

3.1 Top-Down and Bottom-Up

The work in [25] proposes to distinguish between top-down and bottom-up designs. The definitions in [25] can be directly quoted: *"In the bottom-up approach it is attempted to achieve a higher level functionality by systematically combining basic building blocks into assemblies."* A bottom-up design occurs especially in domains where only limited knowledge is available in advance. This phenomena often occurs in designs with non-linear physics, as fluid flows in the exhaust system example presented above, where small changes in geometry can produce significant changes in the overall physical behavior (e.g. flow separation). Figure 14 shows this approach schematically on the left side. The basic building blocks are combined to form assemblies that are able to fulfill higher-order functions and requirements. The systematical recombination is usually done in a kind of evolutionary approach, for example using a genetic optimization algorithm. This approach has a very high level of computational complexity, but it is able to produce new and creative solutions.

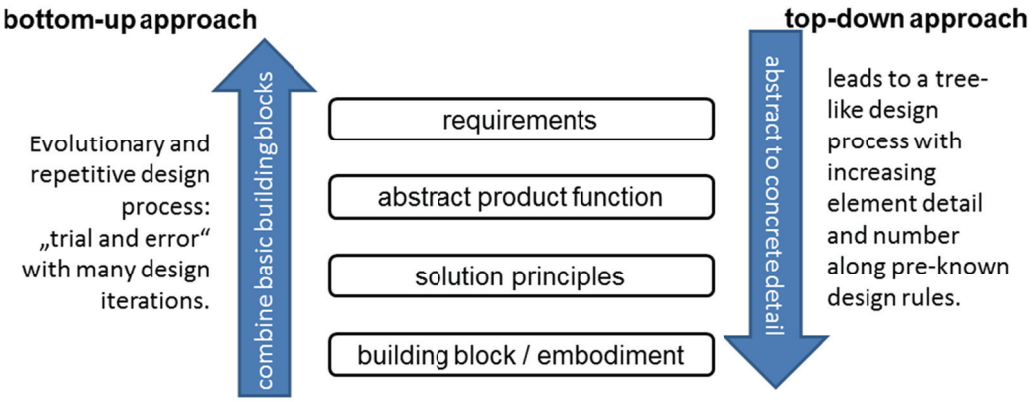

Figure 14 – Different design approaches according to [25]. The top-down approach as well as the design process steps in the center correspond to the design theory presented in [1]

For the counterpart of the bottom-up design approach the following is stated in [25]: *"In the top-down approach design synthesis ... is beginning with the requirements definition, the evolution of the design object is constituted out of subsequent decompositions from abstract conceptual descriptions into more detailed functional representations which finally find an embodiment into material components."* The presented satellite example is an example for a top-down design where a priori knowledge is used to synthesize designs directly from the requirements. The right side of figure 14 shows this approach were the abstract requirements are stepwise decomposed into a final concrete embodiment. This approach follows the popular design theory that was presented by Pahl and Beitz in [1]. In a practical and realistic design process both approaches have to be combined as usually not all engineering problems can be solved using a-priori knowledge. At least the interplay between the enumeration, integration and configuration problem, as shown in the satellite example above, requires a bottom-up approach for non-trivial design tasks. This is true even when the system selection in the enumeration problem could be done in a top-down manner based on available knowledge. Graph-based design languages can implement both: bottom-up and top-down approaches.

3.2 Dimensionless Evaluation

Rudolph identifies the evaluation of engineering objects as crucial challenge in the (automated) design of products [11] as the chosen evaluation method directly influences and determines the outcome of the product optimization process. Using the Pi-theorem [26] with its dimensionless invariants addresses the three main problems of evaluation: "How can the *evaluation* of parts and components be found and represented? How are partial results *aggregated* into a single evaluation? How are the goal criteria *structured*, arranged and are they complete? [11]. In this sense, a complete description of a product entity in terms of design parameters combined to dimensionless invariants forms a valid evaluation: *"Any minimal description in the sense of the Pi-theorem is an evaluation"* [11]. This leads to the evaluation pattern shown in figure 15. The description of the engineering problem (physical equations) has a physical dimension and forms the parameter set $(x_1,\ldots,x_n)$. The physical equations form a description function $X = X(x_1,\ldots,x_n)$ that is defined on these dimensional parameters and describe the behavior of the system. Each component and subsystem contributes to this equation system which is assembled during the design process as components and subsystems are put together with an increasing level of detail to form the final product. From the fact that fully similar designs must get the same evaluation, it is concluded that the evaluation has to take place in dimensionless invariants $(\pi_1,\ldots, \pi_m)$. This transition into dimensionless space, according to the Pi-theorem, ensures that fully similar designs fall within the same evaluation point. The description function is transformed to the evaluation function Π in the same manner. Since all information and variables, including physical dimensions, are stored in the design languages in the central data model, dimensionless evaluation parameters can in principle be obtained automatically.

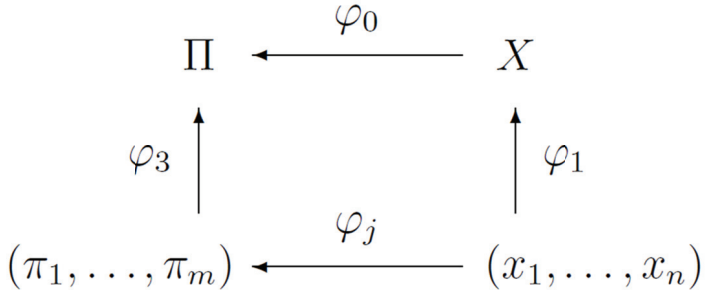

Figure 15 – Evaluation pattern in engineering based on PI-theorem [26]: The product parameters (x_i) are (physically) coupled through the description function X. In the equivalence of description and evaluation, the dimensionless evaluation parameters π_i derived from the x_i are forming the evaluation function Π [10]. Taken from [11]

3.3 Declarative Constraint Processing

In the process of designing complex products, consisting of many components, systems and spanning over multiple physical domains, large equation systems arise from the aggregation of analytical models that are part of each subsystem and component. Within the classes of the design languages equations between the class' parameters as well as between associated classes can be defined. On the execution of the production system the classes are instantiated and the class parameter values are set during the rule executions. So the equation system is created automatically alongside and it is stored in the design graph. For getting a fully automated design process the solution of this equation system needs to be automated. A solution using a so called solution path generator was proposed in [27]. The solution path generator identifies a sequence along which the equations of the equation system can be solved. This leads to a declarative processing of the equation system as only the problem has to be stated in form of the equation system and the solution sequence is internally determined. As an initial requirement, the given variables must be marked as constant in the design graph. This is also done in the production system. So the declarative processing of the equation system in the solution path generator frees the designer from providing a solution sequence. Different design scenarios with different set of given variables ("Who determines what?") can be realized in this way.

The resulting ordered equation system is automatically solved in a computer algebra system that is integrated into the design compiler (see figure 16). In addition, sensitivity analysis can be automatically executed on the equation system to determine the critical design drivers and main dependencies within a product design [14]. This design compiler feature was used in figure 9 in the satellite example presented above.

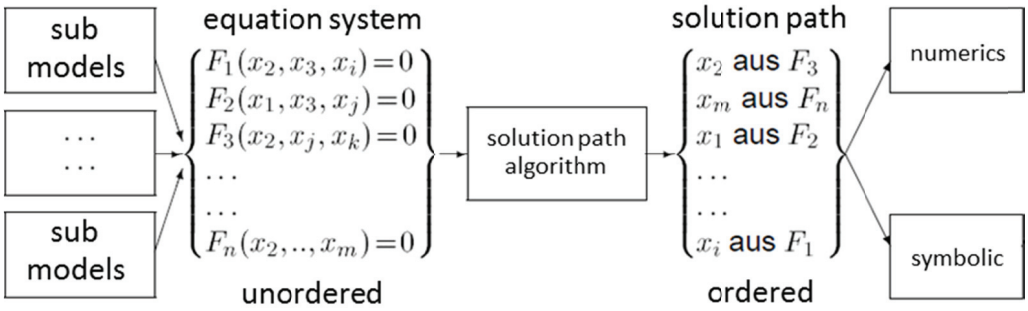

Figure 16 – The algebraic equations from sub models aggregate to a equation system. The solution path generator determines the sequence of equation solution algorithmically. The ordered equation sequence is solved numerically or symbolically in a computer algebra system. Reproduced from [10]

3.4 Self-Similar, Nested Design Patterns

Kormeier identifies in [28] a generic design pattern as observation from recurring design patterns in the application of design languages. In a first step a design process is identified that is an *"…iterative sequence of design synthesis, analysis and evaluation…"* [28]. This reflects the fact that during the design process usually a design of a component or a subsystem is created, analyzed using simulations or calculations and finally evaluated based on this results. Then it is checked whether the requirements, broken down to the component, are fulfilled. If not, then the cycle is repeated and, if necessary, changes must be made in an external cycle enclosing the loop under consideration in order to arrive at a solution. Thus, these design cycles are nested within each other and can be found at the most diverse abstraction and different levels of detail of the design. In this sense the pattern can be regarded as being self-similar [28]. The generic pattern is called integrated design pattern and refines the step of design synthesis of the process explained above. It is defined in the following way: "…*The design synthesis itself is subdivided into the definition of requirements, the*

decomposition, the formulation of functional solutions and the combinatorial exploration process." Figure 17 shows the integrated design pattern graphically. It unifies the two design approaches presented above, the top-down and bottom-up approaches, into one pattern. The red line divides the step of combination (finding solutions) into a bottom-up approach like part and a top-down approach like part. The red dividing line now shifts in one direction or the other depending on which design type prevails. The pattern thus provides a template that can be used in the implementation of design languages as a kind of ideal guideline for structuring the design process.

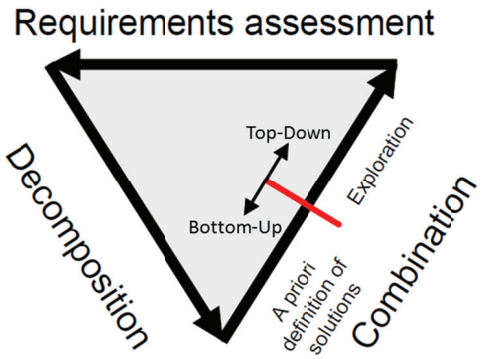

Figure 17 – Integrated design paradigm. Design synthesis as iterative pattern: "…definition of requirements, the decomposition, the formulation of functional solutions and the combinatorial exploration process..." [28]. Taken from [28] and extended.

3.5 Dimension-Based Design Sequence

The paper [18] deals with the problem of splitting the design process into a sequential step-by-step process. It is shown that in order to find an optimal design the dimensioning and integration of sub-systems and components has to occur at once within one design step. But this implies that the design parameters of the components and subsystems that are to be integrated have to be determined at once. This leads to a very high-dimensional problem which practically can't be solved in reasonable time. In general, the parameters have a non-linear behavior and subsets of parameters are strongly coupled with each other. Usually subsystem dimensioning has to be at least partially conducted in a bottom-up approach as the optimal parameter values cannot be determined in advance. Therefore it is necessary to split the design process into sequential steps to find a good design in reasonable time.

Since there are many sequences into which the design process can be divided, a preferred order of the sequential design tasks is proposed based on the mathematical dimension of the involved subsystems [18]. When two systems are sequentially integrated, the (common) design parameters of the lower dimensional system have to be fixed before fixing the design parameters of the higher dimensional system ("Begin with the system that has less degrees of freedom"). In figure 18 the concept of design parameters having different degrees of freedom associated with a different mathematical dimension is exemplarily illustrated. To explain the concept, the simplified problem of the installation or integration of the red square box in an installation space with an existing component (grey square box) is used. The left column a) just shows the two dimensional parameter space of the installation positions of the red box in the plane. The red shaded area in column b) shows the possible parameter values without caring for intersections of the two boxes. If the possible parameters are restricted to the technical meaningful installation positions where the boxes touch each other one gets the parameter set in c) which has now a lower mathematical dimension. Another step down to a point-shaped parameter set is done in column d) where the set is restricted to the installation position where the boxes are aligned which is regarded as being particularly advantageous in this

example. This example shall visualize that in the product design parameter sets with different mathematical dimensions occur, depending on their number of parameters as well as on restrictions due to the context of the design (technical meaningful, …).

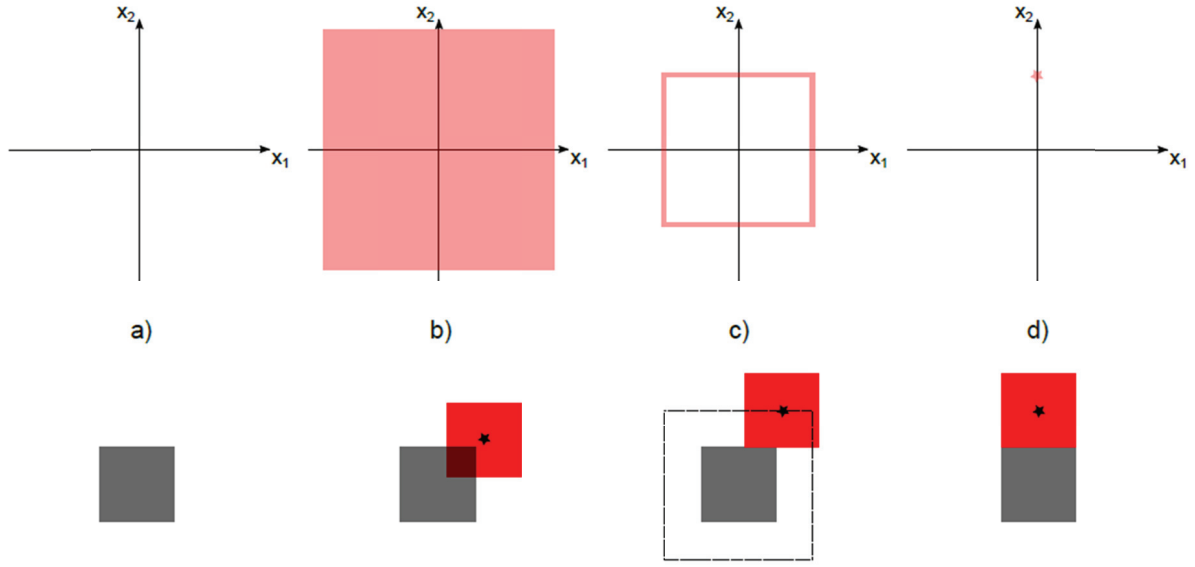

Figure 18 – Lower half: Assembly of two parts (red square and fixed grey square). Upper half: Installation positions (light red) of the red square as x_1,x_2-parameter set for different levels of integration: a) x_1,x_2-parameter space, b) combinatorial possible installation positions, c) technical meaningful installation positions (no intersection), d) optimal installation position. The different parameter sets have different mathematical dimensions [18].

It has been shown in [18] that in a sequential engineering process, where components and subsystems are integrated stepwise, the preferred order is given by the mathematical dimensions of the parameter sets that have to be combined. These parameter sets are not restricted to geometrical parameters as in figure 18. The concept also applies to the non-geometric parameters of the components and systems which must be matched and integrated with one another. This can be applied to the example in figure 18. The grey box has a fixed position and the mathematical dimension of its installation parameter set is therefore zero. The red box, on the other hand, initially has a two-dimensional parameter set. If the installation of the boxes should now happen sequentially, it makes no sense to start with the installation of the red box and to position it somewhere within its parameter set, since the probability of having it installed in exactly such a preferably way at the fixed position as in column d), goes towards zero. However, if the grey box, with the lower dimensional parameter set, is first installed, the degrees of freedom of the red box can be used to find a meaningful installation position in relation to the grey box.

3.6 Design Patterns and Paradigms

In the creation of a graph-based design languages it is possible to adopt design patterns [29] from object-oriented software engineering [29]. The use of object-oriented design patterns from software engineering in design languages is made possible by the extension of the design language classes with methods and interfaces proposed in [30]. With this extension the design languages are adopting central elements of object-oriented software engineering. The design patterns are solutions for recurring kind of problems in programming as for object creation, object composition (structural) and for object interaction (communication). They are typically made of two or more classes that are associated with each other and/or inherit behavior from each other, together with abstract methods in the classes and/or interfaces and a defined interplay and usage of these methods to solve a specific problem.

Figure 19 shows two patterns that are used in design grammars. The builder pattern on the top can be used within the design compilers plugin mechanism. The abstract information to trigger a process chain is extracted from the central model by a translator class. An engineering simulation application is set by calling the corresponding method of the translator class to select the application which shall be executed in the plugin. For example, it is possible to switch between different simulation programs, using the builder pattern shown, without having to change anything in the design language. The information needed to run a simulation is stored abstractly in the central data model [30, 31]. The lower part of figure 19 shows the composite pattern which is used to model a systems-of-systems relation that is the basic structure of all systems engineering activities where a system itself is made up of further systems or of components on the lowest level. The pattern provides methods to add and remove subsystems and components as well as to call operations or actions on all system elements in a recursive manner.

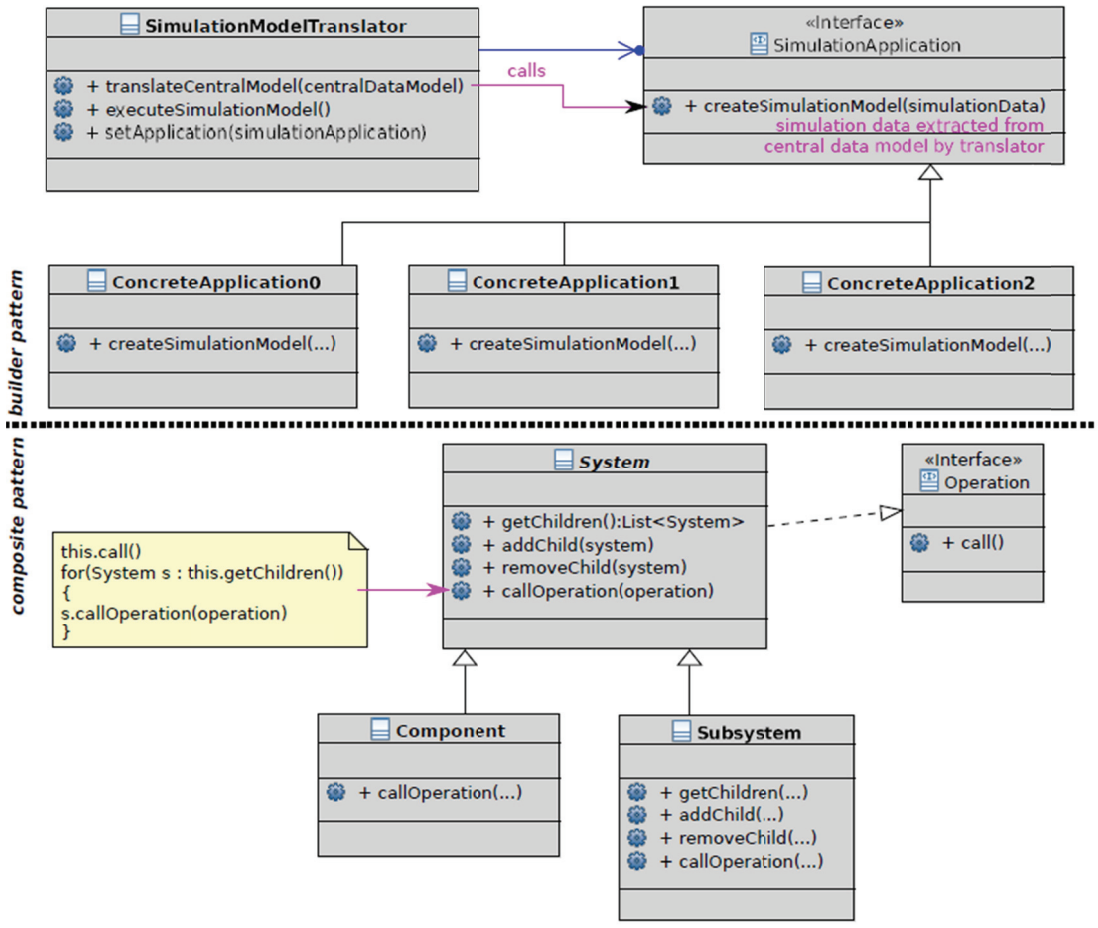

Figure 19 – Two design patterns from object-oriented software engineering [29]. Top: Builder pattern for calling different applications in a engineering plugin that shares a common central model. Bottom: Composite pattern for modeling a system-of-systems relation in systems engineering [30]

The ‘Design for X’ paradigms [10], mentioned above in the philosophical motivation, can also be seen as a kind of design pattern. As these paradigms are less specific than the previously presented design patterns from object-oriented software engineering they are part of the ‘ability’ concept area. This paradigms help the designer to structure the design process, at least mentally, when designing a graph-based design language. Nevertheless, it is often possible to transform the less specific paradigms in to rule-based heuristics when implementing a design language.

4 Discussion

In this paper an overview on graph-based design languages was given. The method allows a graphical programming of design processes that helps to manage the complexity in modern product design. The data model is separated from the operational procedural aspects in an object-oriented way. The object-oriented UML modeling language is used for an open, re-useable, compact and non-proprietary knowledge representation of the product entities and design process. Modeling redundancy is avoided through inheritance between classes and decomposition into classes, modules and plugins. Storing engineering knowledge in incremental rules and adaptive rule sequences allows a hierarchical decomposition of the engineering process itself into smaller chunks that can be more easily captured and overviewed by the engineer even for complex products and systems. This approach allows an easy reuse of knowledge in further design languages as external classes and packages can be easily loaded. The design languages are preferably used to automate recurring design tasks as the implementation of a design language in the design compiler takes some upfront effort.

The presented example applications show that the method of graph-based design languages is able to solve substantial real-world engineering problems in a fraction of the time that would be necessary in manual engineering. In fact, the design time collapses to the addition of the (potentially concurrent) run-times of the algorithms which is close to the lower theoretical limit. Furthermore, it is our observation that a graph-based design language, when embedded in an optimization framework, is often able to find more optimal engineering solutions as can be found in a conventional manual engineering process. The presented language is able to capture engineering knowledge digitally which leads to a highly scalable and re-executable digital blueprint of recurring design tasks. When executing design languages in optimization or DoE runs on HPC or cloud infrastructure, the available hardware power becomes the only limiting factor in accelerating the design tasks. The practical execution of optimizations with design languages on HPC environments shows that new challenges and questions arise from the resulting flood of results: How are result outliers to be treated and how are they caused? How can structures in the results be interpreted? Are these structures caused by properties of the design language or by the physics of the problem? Can generally valid technological laws be derived from this? Nevertheless, a significant upfront invest is necessary as implementing a product's generic design process is apparently more expensive than creating a few product designs by hand. The wide acceptance of this kind of modeling method in industry is at the moment therefore still often hindered by the traditional 'silo mentality' present in today's companies which look on the short-term profitability of the individual business unit instead on the mid- or long-term profitability of the company overall.

The presented design principles provide various patterns, templates and aids to handle the occurring complexity in product design. They can be used on different levels and stages during the implementation of engineering processes. Some of the shown principles as the design approaches, generic design patterns and the dimension-based design sequence help to understand the process of designing itself to simplify the implementation of design languages. Others, as the declarative constraint processing and the dimensionless evaluation scheme support an unambiguous evaluation of products which is a prerequisite for an unbiased optimization of the product itself. At last, design patterns from software engineering can be reused in design languages due to the close relation of the design languages and the design compiler with tools from object-oriented software engineering. All the shown design principles help to handle the complexity in product design. This is an important task as the complexity of the products themselves is steadily increasing and even more domains and disciplines have to be considered to get a holistic evaluation, validation and optimization of a product's life cycle from cradle to grave. In this sense, graph-based design languages have

shown to be a successful method to push the limits of the still controllable complexity in industrial engineering further away.

References

[1] ***Pahl G, Beitz W, Feldhusen J, Grote KH.*** Engineering Design: A Systematic Approach. Series: Solid mechanics and its applications, Springer London, 2006.

[2] ***Kröplin B, Rudolph S.*** Entwurfsgrammatiken - Ein Paradigmenwechsel? Der Prüfingenieur, 26:34-43, 2005.

[3] ***Antonsson E, Cagan J.*** Formal Engineering Design Synthesis. Cambridge University Press, 2001.

[4] ***Borgest NM, Vlasov SA, Gromov AlA, Gromov AnA, Korovin MD, Shustova DV.*** Robot-designer: on the road to reality. *Ontology of Designing*. – 2015; 5(4): 429-449. - DOI: 10.18287/2223-9537-2015-5-4-429-449.

[5] ***Borgest N, Gromov An, Gromov Al, Korovin M.*** The concept of automation in conventional systems creation applied to the preliminary aircraft design. In: Wiliński A., et al. (eds.) *Soft Computing in Computer and Information Science*, Part II. AISC, Springer, 2015; vol. 342: 155–165. – DOI: 10.1007/978-3-319-15147-2_13.

[6] ***Borgest N, Korovin M.*** Ontological approach towards semantic data filtering in the interface design applied to the interface design and dialogue creation for the "robot-aircraft designer" informational system. *Advances in Intelligent Systems and Computing.* Springer, 2017; vol. 534: 93-101. – DOI: 10.1007/978-3-319-48429-7_9.

[7] ***Jaroslaw Sobieszczanski-Sobieski, A J Morris, M J L van Tooren.*** Multidisciplinary Design Optimization Supported by Knowledge Based Engineering. - Chichester, UK : John Wiley and Sons, 2015.

[8] ***Muenzer C.*** Constraint-Based Methods for Automated Computational Design Synthesis of Solution Spaces, ETH Zuerich, Diss., 2015. - 119 p. - https://www.research-collection.ethz.ch/handle/20.500.11850/113711.

[9] ***La Rocca, Gianfranco.*** Knowledge based engineering: Between AI and CAD. Review of a language based technology to support engineering design. Advanced Engineering Informatics 26 (2012), P.159–179. - DOI: 10.1016/j.aei.2012.02.002.

[10] ***Rudolph S.*** Übertragung von Ähnlichkeitsbegriffen. Habilitationsschrift, Universität Stuttgart, 2002.

[11] ***Rudolph S.*** Eine Methodik zur systematischen Bewertung von Konstruktionen. PhD thesis, Universität Stuttgart, 1995.

[12] **Object Management Group, OMG** Unified Modeling Language. www.uml.org. Version: 2014.

[13] ***Reichwein A.*** Application-specific UML Profiles for Multidisciplinary Product Data Integration. PhD thesis, Universität Stuttgart, 2011.

[14] ***Bölling M.*** Lösungspfadbasierte Analysen im Entwurf komplexer Systeme. PhD thesis, Universität Stuttgart, 2013.

[15] ***Gross J.*** Aufbau und Einsatz von Entwurfssprachen zur Auslegung von Satelliten. PhD thesis, Universität Stuttgart, 2014.

[16] ***Rudolph S, Beichter J, Eheim M, Hess S, Motzer M, Weil R.*** On multi-disciplinary architectural synthesis and analysis of complex systems with graph-based design languages. In 62. Deutscher Luft- und Raumfahrtkongress (DGLR 2013), Stuttgart, September 10-12, 2013.

[17] ***Tonhäuser C, Rudolph S.*** Individual Coffee Maker Design Using Graph-Based Design Languages. In: Gero J. (eds.) Design Computing and Cognition '16, pp 513-533, 2016. Springer, Cham. – DOI: 10.1007/978-3-319-44989-0_28.

[18] ***Vogel S.*** Über Ordnungsmechanismen im wissensbasierten Entwurf von SCR-Systemen. PhD thesis, Universität Stuttgart, 2016.

[19] ***Haq M, Rudolph S.*** A design language for generic space-frame structure design. In: Int. J. Comput. Appl. Technol., Band 30(1/2): S. 77-87, 2007. - DOI: 10.1504/IJCAT.2007.015699.

[20] ***Arnold P, Rudolph S.*** Bridging the gap between product design and product manufacturing by means of graph-based design languages. In: TMCE, 2012.

[21] ***Vogel S, Rudolph S.*** Automated Piping with Standardized Bends in Complex Systems Design. In: Proceedings of the Seventh International Conference on Complex Systems Design & Management, CSD&M Paris, 2016.

[22] ***Rudolph S, Arnold P, Eheim M, Hess S, Motzer M, Riestenpatt M, Schmidt J, Weil R.*** Design languages for multi-disciplinary architectural synthesis and analysis of complex systems in the context of an aircraft cabin, CEAS Conference, Toulouse, November 25-27, 2014.

[23] ***Breckle T, Kiefer J, Rudolph S, Manns M.*** Engineering of assembly systems using graph -based design languages, 21st International Conference on Engineering Design (ICED 17), Vancouver, Canada, 21-25.08.2017.

[24] ***Kiesel M, Klimant P, Beisheim N, Rudolph S, Putz M.*** Using Graph-based Design Languages to Enhance the Creation of Virtual Commissioning Models, 27th CIRP Design Conference on Complex Systems Engineering and Development, 2017. - DOI: 10.1016/j.procir.2017.01.047.

[25] ***Alber R, Rudolph S.*** A Generic Approach for Engineering Design Grammars. In: AAAI Spring Symposium Technical Report SS-03-02, March 24-26, 2003.
[26] ***Buckingham E.*** On Physically Similar Systems: Illustration of the Use of Dimensional Equations. Phys. Review 4, 345-376, 1914. - DOI: 10.1103/PhysRev.4.345.
[27] ***Bölling M, Rudolph S.*** Multi-disciplinary airship design using a graph-based design language. In: DGLR Jahrestagung, Band I, 2005.
[28] ***Kormeier T, Rudolph S.*** On self-similarity as a design paradigm. In: Proceedings of IDETC/CIE, International Design Engineering Technical Conference and Computers and Information in Engineering Conference, September 24-28, 2005.
[29] ***Gamma E.*** Design Patterns: Elements of Reusable Object-oriented Software. Pearson Education, 2004 (Addison-Wesley Professional Computing Series).
[30] ***Vogel S, Arnold P.*** Object-orientation in graph-based design grammars. Computing Research Repository, abs/1712.07204, 2017.
[31] ***Vogel S.*** An application-independent continuum mechanics interface for virtual engineering, Engineering with Computers, 2018.

ПРОЕКТИРОВАНИЕ СЛОЖНЫХ СИСТЕМ ПРИ ПОМОЩИ ЯЗЫКОВ ПРОЕКТИРОВАНИЯ: МЕТОДЫ, ПРИМЕНЕНИЕ И ПРИНЦИПЫ ПРОЕКТИРОВАНИЯ

С. Фогель[1], С. Рудольф[2]

Институт проектирования самолетов Штутгартского университета, Штутгарт, Германия
[1]Samuel.Peter.Vogel@gmail.com, [2]Rudolph@ifb.uni-stuttgart.de

Аннотация

Языки проектирования, основанные на графах, представлены как способ преобразования информации, автоматизации процессов проектирования и оптимизации продукта или сложной системы. Унифицированный язык моделирования (UML) используется для создания языка, моделирующего процесс проектирования. Язык проектирования состоит из терминологии («цифровых строительных блоков») и набора правил («знаний цифровой композиции») выполнения последовательности действий (т.е. последовательного преобразования в цифровой вид процесса проектирования). С использованием основанного на правилах метода создаётся обобщённая центральная согласованная схема данных об объекте проектирования (так называемый граф проектирования). После генерации абстрактной центральной модели автоматически генерируются инженерные модели, отражающие специфику конкретной предметной области, и после удалённого выполнения их результаты вносятся в центральную модель проектирования для принятия последующих проектных решений или оптимизаций. Языки проектирования моделируются вручную и автоматически выполняются в так называемом компиляторе проектирования. Языки проектирования, основанные на графах, успешно применяются при создании разнообразных изделий аэрокосмической (космические аппараты, самолеты), автомобильной (пространственные конструкции, кабины автомобилей), машиностроительной (роботы, цифровые производства) отраслей и потребительских товаров (кофе-машины, вытяжные системы) для повышения эффективности процесса проектирования и степени его автоматизации. Рассмотрены различные стратегии и механизмы проектирования с целью применения их к автоматизации процесса проектирования. Используются подходы, начиная с автоматизированной и декларативной обработки ограничений, фрактальных вложенных шаблонов проектирования до математического определения последовательности действий проектирования. Имеющиеся знания определяют общую стратегию проектированию (т.е. нисходящее или восходящее проектирование). С целью снижения размерности и общей сложности задачи используется построение безразмерных инвариантов на основе теории подобия. Шаблоны проектирования, парадигмы проектирования (т. е. форма следует за функцией или функция следует за формой) и стратегии проектирования (разделяй и властвуй) из информатики широко используются для структурирования и управления сложностью проекта.

Ключевые слова: *языки проектирования, автоматизация проектирования, метод проектирования, принципы проектирования, онтология проектирования.*

Сведения об авторах

Dr. Samuel Vogel (b. 1982) is currently external habilitation candidate in the group of similarity mechanics at the Institute of Aircraft Design at the University of Stuttgart. He is currently working at MTU Friedrichshafen GmbH (subsidiary of Rolls Royce Holding plc) in the research and technology development. Samuel Vogel studied physics at the University of Stuttgart and finished in 2007. After that he worked for five years at a development service provider as team lead and head of the department "Predevelopment, Methods and Analytics". In 2013, he moved to his current position at MTU. Between 2009 and 2015, Samuel Vogel did in parallel his doctorate on the automated design of exhaust systems with design languages in Stephan Rudolph's group. From 2016 to 2018 he also did research in the same research group as a postdoctoral student in part time. His research interests include the interplay between the (automated) design process and complex systems science as well as design optimization with design languages especially in conjunction with computational fluid dynamics. Automated 3D CAD model and geometry creation for autonomous optimization and validation workflows are also an important research interest of Samuel Vogel.

Д-р Самюэль Фогель (1982 г.р.) в настоящее время является кандидатом на внешнюю хабилитацию в группе механики подобия в Институте проектирования самолетов Штутгартского университета и работает в MTU Friedrichshafen GmbH (дочерняя компания Rolls Royce Holding plc). Самюэль Фогель окончил Штутгартский университет в 2007 году, после этого пять лет работал в качестве руководителя команды и начальника отдела «Предварительная разработка, методы и аналитика». В 2013 году он перешёл на свою нынешнюю должность в MTU. В период с 2009 по 2015 годы Самуэль Фогель параллельно выполнил свою докторскую работу по автоматизированному проектированию выхлопных систем с использованием языков проектирования в группе Стефана Рудольфа. С 2016 по 2018 годы он проводил исследования в той же исследовательской группе в качестве постдокторанта. Его исследовательские интересы включают взаимодействие между (автоматизированным) процессом проектирования и теорией сложных систем, а также оптимизацией проектирования с использованием языков проектирования. Автоматизированное 3D-моделирование в САПР и создание геометрии для автономных процессов оптимизации и валидации являются важным исследовательским интересом Самуэля Фогеля.

PD Dr. Stephan Rudolph (b. 1961) is currently head of similarity mechanics group at the Institute of Aircraft Design at the University of Stuttgart. Stephan Rudolph studied, earned his doctorate and habilitated at the Faculty of Aerospace Engineering and Geodesy at the University of Stuttgart. Additional one-year study visits at a French Grande Ecole for Aerospace (ENSICA) in Toulouse, France, and at the Massachusetts Institute of Technology (MIT) in Cambridge, USA. After obtaining his PhD in 1995 from the University of Stuttgart, he spent six months as a PostDoc in the Systems and Design Group at the MIT. Habilitation and Lecturer in the field of "Design Methodology" at the Faculty of Aerospace Engineering and Geodesy at the University of Stuttgart in 2002. Head of the research group "Similarity Mechanics" since 1996. Dr. Rudolph's research interests include formal methods of Model-Based System Engineering (MBSE) and formal design synthesis with graph-based design languages, automatic model generation and design evaluation methods as well as applications of similarity mechanics in engineering and artificial intelligence. Within the focus of graph-based design languages, the theoretical focus is on questions of uniqueness, consistency, validation and verification of design languages, as well as the practical focus on the language development of graph-based design languages for automatic product design of satellites, aircraft, vehicle structures and their digital factories.

Д-р Стефан Рудольф (1961 г.р.) возглавляет группу механики подобия в Институте проектирования самолетов Штутгартского университета. Стефан Рудольф учился и получил докторскую степень на факультете аэрокосмической техники и геодезии Штутгартского университета (1995). Получив степень PhD, прошёл стажировки в Высшей школе аэрокосмической техники (ENSICA) в Тулузе (Франция) и в течение шести месяцев в качестве постдоктора в группе «Системы и проектирование» Массачусетского технологического института (MIT) в Кембридже (США). В 2002 году получил хабилитацию как доктор и лектор в области методология проектирования на факультете аэрокосмической техники и геодезии в Штутгартского университета. Руководитель исследовательской группы механики подобия с 1996 года. Научные интересы доктора Рудольфа включают формальные методы системного инжиниринга на основе моделей (MBSE) и формальное проектирования с использованием языков, основанных на графах, автоматическое создание и оценки проектных моделей с применением механики подобия в проектировании и искусственного интеллекта. В языках проектирования, основанных на графах, наибольший интерес для доктора Рудольфа в теоретическом аспекте представляют вопросы уникальности, согласованности, валидации и проверки языков проектирования. Наибольший практический интерес в этой области представляет разработка языков проектирования, основанных на графах, которые бы позволили автоматизировать проектирование космических аппаратов, самолётов, конструкций машин и их цифровых производств.