




Towards a more complete object-orientation in graph-based design languages

Samuel Vogel¹  · Peter Arnold²Received: 12 December 2019 / Accepted: 27 May 2020 / Published online: 16 June 2020
© The Author(s) 2020 

Abstract

In this paper an extension of the design method graph-based design languages is proposed. This is realized by adding object-oriented class methods and interface mechanisms to the design method. Additionally, graphical mechanisms for modeling and calling the methods are proposed. This allows object-oriented design patterns to be transferred to the product design, where they improve the handling of complexity in the product engineering. As result, the proposed extension enables modularization and reuse of engineering knowledge, the integration of engineering domains is enhanced and multi-stakeholder collaboration with security access control (information security) becomes feasible.

Keywords Systems engineering · Engineering automation · Design grammar · Design language · Object-orientation

1 Introduction

The ongoing digital transformation in industry applies to all product life cycle's stages. The design decisions and dimensioning carried out in the early conceptual design stages determine a huge part of the product's life cycle costs (LCC) [1]. The automation of the conceptual design phase promises therefore huge gains in terms of LCC. Graph-based design languages encode design processes in production systems made up of rule sequences which automatically create an abstract central product model (central data model) from given requirements. Graph-based design languages use the unified-modeling-language (UML) to define the product entities (classes) supporting object-oriented inheritance. Graph rules, either graphically defined or code-based, instantiate the classes and iteratively assemble the central model. This interdisciplinary systems engineering approach tries to capture all aspects of a product (design) and helps to handle the complexity in the development of modern products [2].

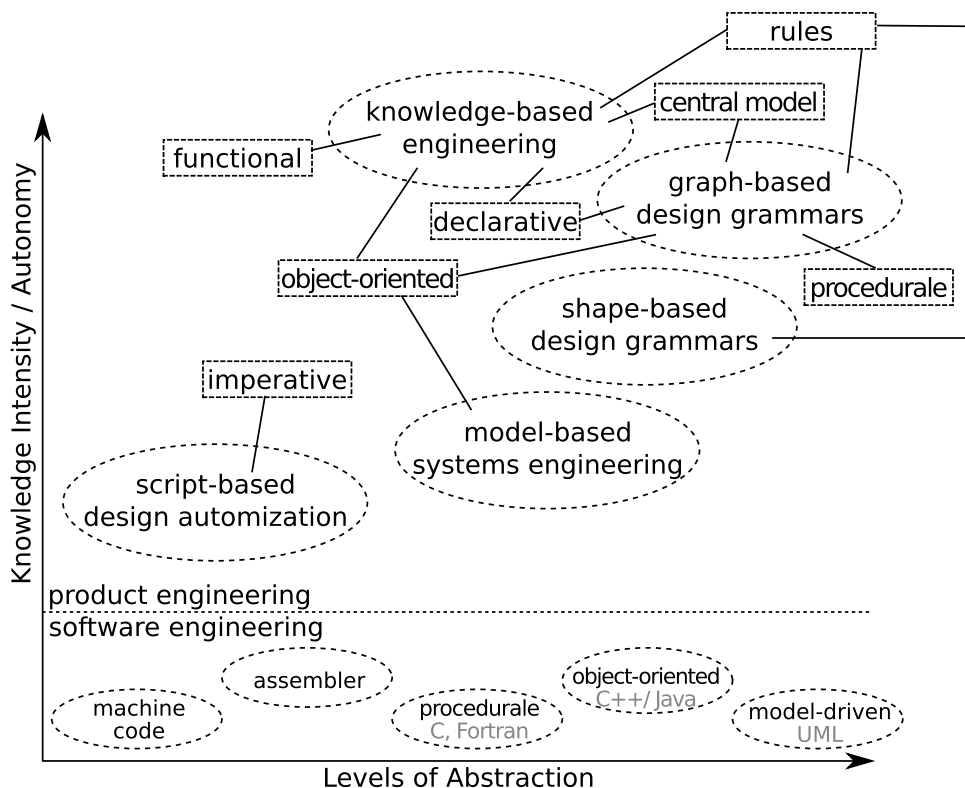
The goal of this work is to further improve the design method of graph-based design languages. It shall be shown that graph-based design languages only partially follow object-oriented modeling principles, but in some places (class methods and interfaces) the paradigm of object-orientation is not yet fully implemented. As part of the work, the graph-based design languages are now to be further developed methodically so that a more complete implementation of the principles of object-orientation is achieved. The work is intended to push the limits of the design method of graph-based design languages further towards a more compact and simpler formulation to gain an even better handling of the complexity immanent in (the design of) modern products and systems. Handling this is becoming even more important in the design of networked cyber-physical systems in the upcoming age of the Internet of Things.

The work is organized as follows: After the introducing section the current state of knowledge and previous work in the area of design grammars and graph-based design languages is presented. Then the problem setting

✉ Samuel Vogel, samuelpeter.vogel@rwu.de | ¹Technology and Management, RWU Ravensburg-Weingarten University of Applied Sciences, Doggenriedstraße, 88250 Weingarten, Germany. ²IILS mbH, Albstrasse 6, 72818 Trochtelfingen, Germany.



Fig. 1 The level of abstraction versus knowledge-intensity for different methods of engineering automation (ovals). The boxes assign the key properties of the approaches. Analogously, the different levels of abstraction in software engineering are shown below. Taken from [3]



is formulated, why graph-based design languages do not yet provide a complete realization of object-oriented principles. In the following method section approaches are proposed to solve this issue. Then a case study presenting a graph-based design language that creates an exhaust aftertreatment system is carried out. In the following results section the issues raised in the problem setting are addressed. Finally, the paper ends with discussion and conclusion.

2 State of knowledge and previous work

This section presents the state of knowledge relevant to the work carried out. The main aspects of automation of design processes as well as principles of object-orientation, especially known from software engineering, are introduced. The presentation of the relevant engineering design methods in the following subsection is based on the author’s introduction in the reference [3].

2.1 Engineering design methods

In the past, various approaches to automate design tasks have been proposed. Figure 1 gives a first overview of the approaches pursued. Typical properties and characteristics are also assigned. However, it should be noted that the presentation is simplified and that the boundaries of the

approaches and their typical characteristics are blurred and may overlap.

2.1.1 Multidisciplinary design optimization (MDO)

MDO is probably the most widely used approach to design automation in industrial practice at present. The article [4] shows applications of MDO methods in aerospace engineering. According to the industrial experiences of the author the main approach to implement MDO is by using a parameterized geometry in a CAD (computer-aided design) environment together with an (integrated) simulation workflow. The workflows are very often implemented using an imperative, script-based approach as shown in Fig. 1 bottom left. There exist a lot of different MDO architectures fitted to the specific design problem [5]. In contrast to the model-driven approaches described below, there is generally no explicit conceptual model in MDO besides the implicit CAD and simulation models used.

2.1.2 Knowledge-based engineering (KBE)

The much more knowledge-intensive KBE approaches promise a higher degree of automation and autonomy. According to [6] these are knowledge-based systems which are closely interwoven with a CAD core. On the one hand there are approaches in which the knowledge is modeled in corresponding (declarative) programming

languages like Lisp [7] and which then address a connected CAD application. On the other hand, there are CAD applications that provide KBE functionality in the form of programming options and interfaces [8]. The KBE approach is further characterized by features like runtime caching, dependency tracking as well as a demand-driven evaluation [7]. This allows purely practical designs to be generated and reconfigured online without the need for a complete new generation. However, the complexity of implementing KBE applications still stands in the way of practical application in an industrial context [9].

2.1.3 Model-based systems engineering (MBSE)

The main idea behind systems engineering is to decompose a complex product, as well as its requirements, into smaller systems that are made up of (sub)systems themselves [10]. Clear interfaces are defined between linked systems, that can be interchanged afterwards according to the interfaces' definitions. The decomposition leads to smaller system entities that can be handled more easily and enables therefore the handling of complex products and systems (divide-and-conquer).

MBSE now represents a form of implementing the systems engineering approach, which is less knowledge intensive and less automated compared to KBE. The MBSE is defined in [11] as: "The formalized application of modeling to support system requirements, design, analysis, verification and validation activities beginning the conceptual design phase and continuing throughout development and later life cycle phases". The model-based aspect means that a product is no more represented by classical documents but through object-oriented, abstract and hierarchical models using languages as the Systems Modeling Language (SysML). Usually there is still a lot of manual work involved: Either in a manually created central model of a product from which domain-specific models (e.g. simulations or geometries) are then automatically generated, or in the context of distributed domain-specific models which have to be mutually updated manually when changes occur [12, 13].

2.1.4 Design grammars

There exists a variety of different design grammars [14]. These approaches have a common language-like structure. A formal vocabulary is defined, which represents (parts of) the later product. Using this vocabulary, rules are defined how a valid model of the product is built from the individual 'words' (= parts of the product)—according to the grammar of a language. In a production system, these rules are called up when the precondition of the respective rule is fulfilled. From the combinatorics of the gradual

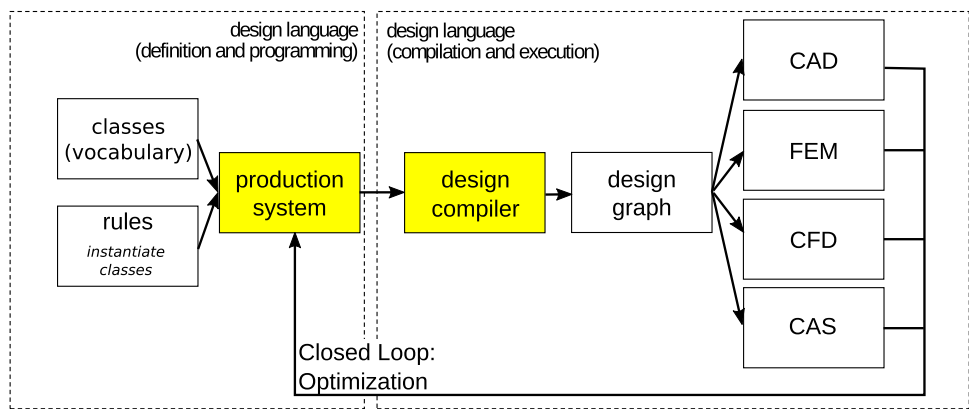
application of the applicable rules, whole trees (generative tree) of product designs result, which are then searched for target-oriented designs [15]. A reduction of the search spaces can be realized using first-order logic and Boolean satisfiability [16]. To gain further insight in to design grammars a representation of the (unique) designs in transition graphs is proposed. The transition graphs allow a systematic rule analysis which supports the human designers to gain a deeper understanding of the grammar they developed [17]. Other recent works propose to divide rules into groups of more and less abstraction to accelerate design synthesis [18]. More complex rules are proposed in [19] by grouping two or more rules into so called composite shape rules (*flows*) to get an algorithmic rule pattern.

2.1.5 Graph-based design language

The so-called graph-based design languages, that are used and further developed in this work, are another kind of a design grammar. The approach shares the properties of having a vocabulary (*as classes*) and rules. But the production system that calls the rules is different: Graph-based design languages have an explicitly procedurally modeled rule sequence [20]. These sequences contains state dependent branches and are organized into hierarchical sub-sequences (*sub-activities*). Using this approach, no combinatory set of designs is created on execution, but one final design is created for a set of given starting conditions and requirements. Graph-based design languages are a generic approach that uses the object-oriented Unified-Modeling-Language (UML) and that is not limited to a specific (engineering) domain [21].

A specific product instance is iteratively expanded from a given set of requirements. This is practically done by the execution of the *production system* in a so-called *design compiler* [22] that creates a design graph as digital blueprint of the product. Figure 2 shows the information architecture of graph-based design languages schematically. The left part contains the implementation of the design process in the design compiler. Product building blocks are defined in terms of *classes*. The *production system* is made up of an adaptive rule sequence that instantiates the classes. The sequence can be branched, based on judgments rendered in *decision nodes*. *Graph Rules*, defined either graphically or textually as program code, are able to conduct manipulations of the *design graph*. The design graph acts as central data model (single source of truth). On the right side the validation part of the design process (process chains) is shown where domain-specific engineering models (CAD, structural mechanics, fluid mechanics, etc.) are derived from the central model by executing model-to-model (M2M) transformations. These transformations are implemented in plug-ins that are provided by

Fig. 2 Information architecture of a graph-based design language



the design compiler. The design graph is analyzed in the M2M-transformations within the plug-in calls, the domain-specific information is filtered and the domain- and application-specific models are generated and executed [3]. The results of the automatically executed and post processed validation calculations and simulations can be fed back into the production system to change requirements or start conditions. In this way outer optimization loops to explore design space can be realized.¹ Graph-based design languages have been used throughout different applications like automotive, aerospace and manufacturing [23–29]. An exemplary design language is presented in the following Sect. 3 and its existing disadvantages are discussed there.

2.2 Principles of object-orientation

Object-orientation is a method from software development to map and handle complex systems. A system is defined as a set of cooperating objects that perform a task within or in the form of an software application (compare with the systems engineering approach explained above). The following list shows the key features of object-orientation:

- **Abstraction** Representing only the essential features of an entity. Extraction of the essential features and interfaces of a system.
- **Encapsulation** Wrapping entities, that are defined by data and methods to conduct specific behavior, into units. Hiding system internals and implementations behind externally accessible and well defined interfaces.

- **Polymorphism** Sub-typing of entities through hierarchical inheritance. Behavior is abstractly defined by interfaces that are implemented and reused in sub-types.

Reusable design patterns are widely spread in software engineering. These patterns heavily rely on the object-oriented features listed above and follow the central idea of: “Programming to an Interface, not an Implementation” [30]. The interface mechanism is the central feature of flexible object-oriented software design which allows to easily couple, exchange and reuse “black-box” entities whose interaction is specified through interfaces and which is independent of the inner structure and the specific implementation of the entities. This fits perfectly to the systems engineering’s central idea to recursively couple encapsulated and hierarchically defined sub-systems to compose increasingly complex systems and products.

3 Problem setting

3.1 Challenges in product engineering

Modern systems engineering faces the challenge of designing highly complex cyber-physical systems that typically cover many physical and even logical domains (control, ...) simultaneously. This makes it necessary, in the design process, to bring together contributions from the *different domains* and departments on the one hand, but on the other hand the associated requirements for the protection of *intellectual property* and *data security* must also be taken into account. This is even more important when third-party suppliers are also involved in this process. Thus, a modern tool to support and automate the design process must provide both information security and encapsulation of potentially sensitive data, as well as functions for interdisciplinary and inter-company *collaboration*.

Furthermore, most modern products have an immense implicit complexity: This is due to the progressive

¹ In contrast to conventional design grammars each design is started anew.

development of the technologies themselves, the increasing implementation of product functions by software functions as well as ever more far-reaching regulatory requirements. Last but not least, the customer demands ever better products with ever more extensive functions. Imagine the development of the automobile over the last 50 years from a pure means of transport to a highly comfortable and already partly autonomous transportation system with integrated information and entertainment systems, which meets much stricter environmental requirements at the same time. This complexity has to be handled in the design process within product engineering. The systems engineering approach of dividing the product into a system-of-systems can support this handling. This can be realized by a *decomposition* and *modularization* of the product model description. To avoid double work and to realize an efficient design process, a *reuse* of already developed solutions shall be supported.

3.2 Current status of graph-based design languages

The graph-based design languages, as presented above, are modeled in an object-oriented modeling language (UML), but follow more or less a procedural programming paradigm [31]. This is exemplarily shown in the schematic design language² illustrated in Fig. 3.

A simple car model, consisting of a chassis with a defined number of wheels, is expanded and refined by introducing wheel suspensions. The objects that are put together by the rules are defined in the *UML class diagram* on the top. A graphically defined graph rule (if/then scheme, left-hand-side [LHS]/right-hand-side [RHS] scheme) to add *wheel suspensions* between the *wheels* and the *chassis* is shown below. The production system on the lower part shows the rule sequence for iteratively building up the car model. The *JavaRule* hosts a program code rule which iteratively adds the wheels in a loop. It is called *numberOfWheels* times as set in the *Chassis* object. The design graph (central data model) on the bottom of Fig. 3 is generated by the execution of the production system. It hosts the instances of the current product design with its specific design parameters. The rule sequence in the production system manipulates the design graph in a procedural manner as imperative commands work on a common program state (current design graph state) in a predefined sequence. The production systems in graph-based design languages have an additional entity called *Decision Node* that allows a branching of the rule sequence and the implementation of conditional switch or loop

statements in the design language (Fig. 4). Hierarchical *sub activities* can be modeled in the production system. They can embed sub production systems (Fig. 4). A sub activity can be seen as being equivalent to a sub routine that takes the central data model as sole argument.

In the world of the object-oriented software engineering this approach would be considered as “bad design”. Translated to object-oriented software engineering, the design language’s rule sequence can be interpreted as a sequence of static methods—without any explicit method parameters—that builds up the central data model. The lack of both, explicit interface definitions and methods that are coupled to data objects (classes), harms reusability and modularization. For bigger and complex design languages it gets difficult to maintain consistency and to debug the model as the whole design graph is exposed to every rule and sub activity. So the lack of a tight encapsulation in conjunction with the missing mechanism of abstractly defined interfaces can be seen as a central challenge of the current graph-based design language approach. This disadvantage also applies to numerous other expert systems, as rule-based or logical systems, that miss hierarchical modeling concepts [32].

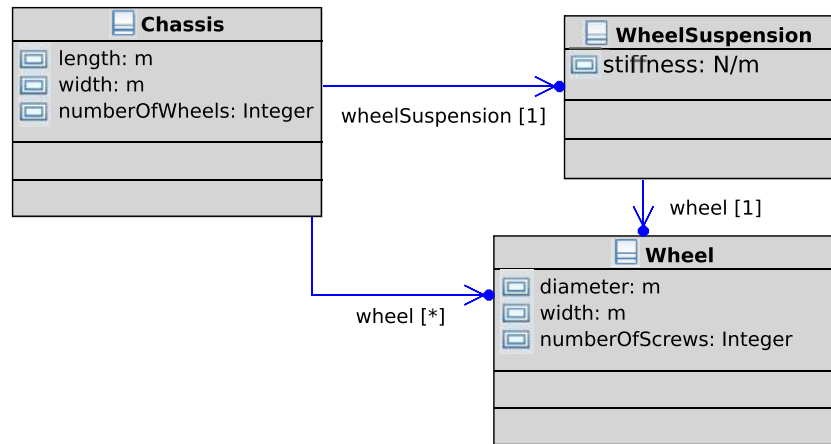
The following list summarizes briefly the shortcomings of the current design languages. These are addressed using the methods presented in the following Sect. 4. In the results Sect. 6, the individual points are then discussed again and the effectiveness of the proposed methods is discussed.

- *Modularization* Systems-of-systems aspect difficult to realize without explicit interface definitions as the sub systems are difficult to delimit mutually. The behavior of sub entities is detached from themselves as operations are not coupled to the data entity they apply to.
- *Reusability* Components can not be properly encapsulated into entities with explicit and self-explanatory interface definitions for later reuse.
- *Domain integration* The integration of domain-specific models via M2M transformations is not sufficient. Some domains can be easier integrated when granular class methods (with defined explicit parameter lists bounded to a class) are provided in the production system and the domain-specific models themselves are created iteratively.
- *Collaboration* Designing complex designs needs involvements from multiple domains and therefore involvement of multiple experts. Proper interfaces to clarify the requirements and responsibilities are a prerequisite for successful collaboration.
- *Information security* Modules with defined interfaces can be encrypted and hidden to allow collaboration

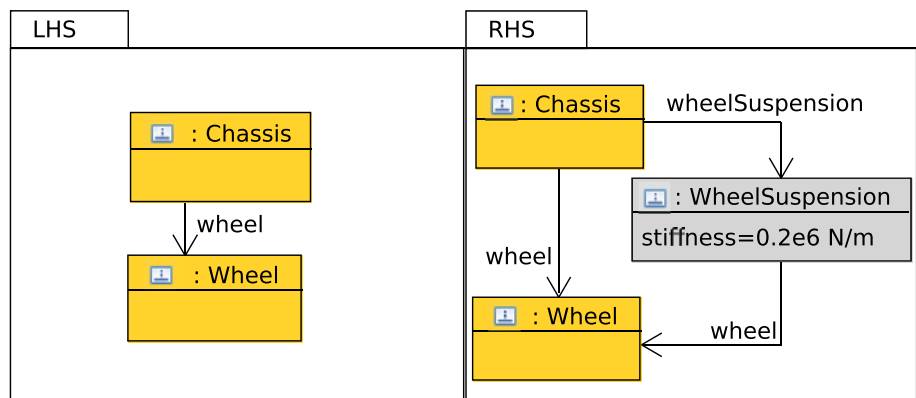
² The truncated term *Design Languages* always refers to *Graph-Based Design Languages*.

Fig. 3 Schematic design language of a simplified car design

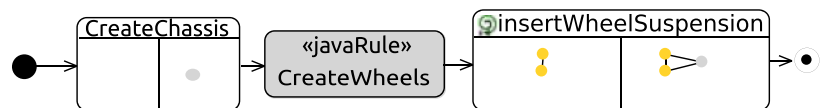
Class Diagram



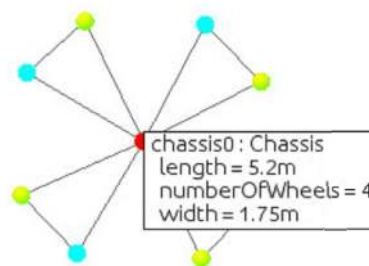
Rule "insertWheelSuspension" (LHS/RHS if/then Scheme)



Production System



Design Graph (Central Data Model)



between companies whilst respecting intellectual property.

A fully object-oriented graph-based design language should therefore realize the object-oriented paradigms

presented above. An abstract interface mechanism would address the issues of modularization, reusability and maintainability. Such an interface mechanism would require class methods on the other hand to bind

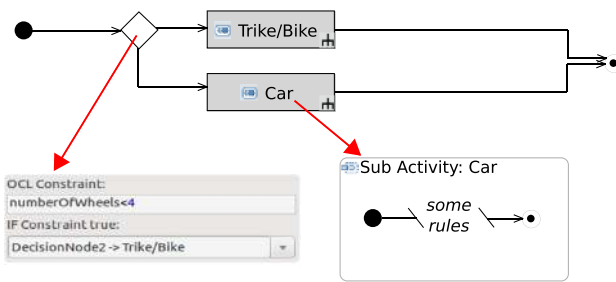


Fig. 4 Decision node for branching the rule sequence (left) based on model constraints and sub activity (right) to model hierarchically embedded rule sequences

together data and functions as claimed in object-oriented software engineering.

4 Methods

In this work modifications to the graph-based design languages are suggested to increase the level of object-orientation to tackle the diagnosed disadvantages above. The proposed solutions have been mainly implemented in the current release V3 of the Design Compiler 43. The following proposed graphical calling mechanisms of the class methods and constructors remain in a conceptual state as its implementation in the design compiler requires additional efforts.

The foundation of the following solution approaches is a duality of graphical and textual (code-based) modeling options provided by the design compiler. The target in each step is to have either a textual-based (source-code) or a graphical definition of the rules and newly added object-oriented features. This graphical definition is interpreted internally in the design compiler that modifies the design graph correspondingly. Then it is up to the creator of the design language which representation is preferred.

For each UML class a Java class source file is internally in the design compiler created which hosts the class parameters as well as the setter and getter methods. This enables the user to model the design process either graphically or code-based, depending on the personal preferences or on the specific problem. The graphical and code-based approaches are reduced to just two different views on the same modeling task.

4.1 Rules

Graphically defined manipulation rules of the design graph, as shown in Fig. 5 top, can be equivalently implemented as Java code, Fig. 5 bottom. Both sorts of rules can be called in the production system as shown in Fig. 3

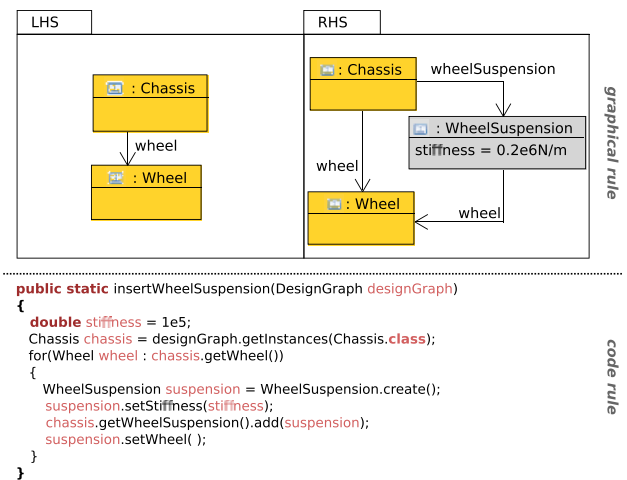


Fig. 5 Rule *insertWheelSuspension*: Equivalence of graphically defined rule (top) and code rule (bottom). Both implementation techniques execute equivalent graph manipulations on the design graph

where the textually defined, code-based rule is marked by the term *javaRule* and a graphically defined rule is shown in its iconified LHS/RHS representation. This feature was already supported in earlier versions and is just mentioned for a better understanding.

4.2 Class methods

As shown in Fig. 6, class methods are added to the classes below each class' properties fields to enable *encapsulation*. The introduction of classes' methods realizes the object-oriented foundation principle of a tight coupling between data and methods. Constructors of the classes, identified by the class name, are added in the same way. These constructors are executed in the creation of the classes' instances. The methods are executed on selected instances of the class in the production system. Both, constructors and class methods can be called from source code or within the graphically represented rules as proposed in Sect. 4.4. According to the graphical and textual duality the constructors and methods can be implemented either as source code or again in the graphical representation as suggested in Sect. 4.5.

4.3 Modeling abstract interfaces

An object-oriented interface mechanism is implemented by the introduced class methods. The abstract behavior of components is defined through interface elements as abstractly defined methods (empty methods with defined signatures). These interfaces are implemented by classes that realize the interfaces' abstract behavior.

Fig. 6 Extended class diagram with object-oriented class methods and class constructors (compare with 'classical' class diagram in Fig. 3 that lacks the methods and constructors). The data types of the methods' parameters are not shown for the sake of simplicity. This applies to all figures

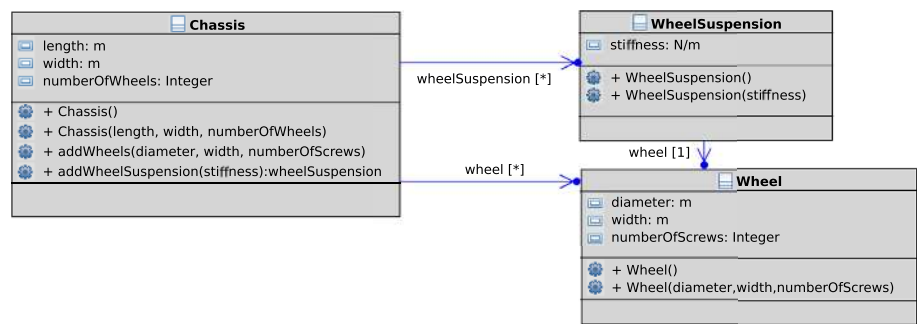


Fig. 7 The classes that realize an interface have to implement the methods defined in the interface

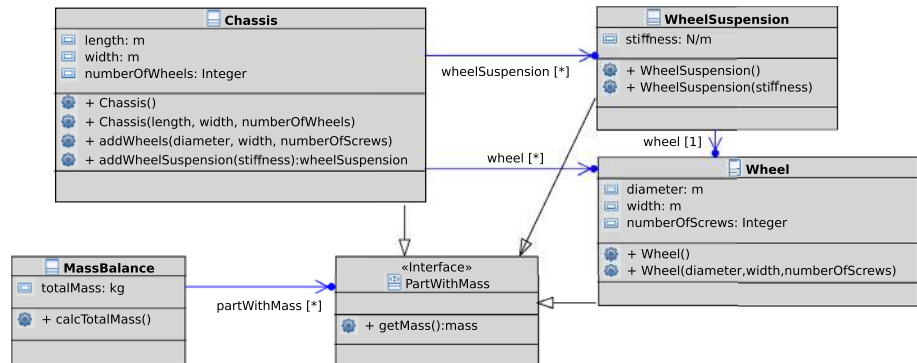
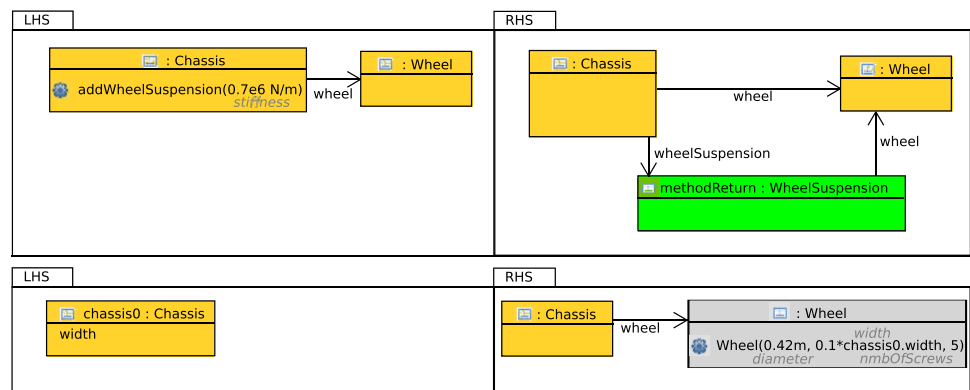


Fig. 8 Top: Calling method *addWheelSuspension* that returns an instance *methodReturn* which is integrated in the model on the RHS. Bottom: An instance of type *Wheel* is created and linked to the existing *Chassis* instance on the RHS by calling the constructor

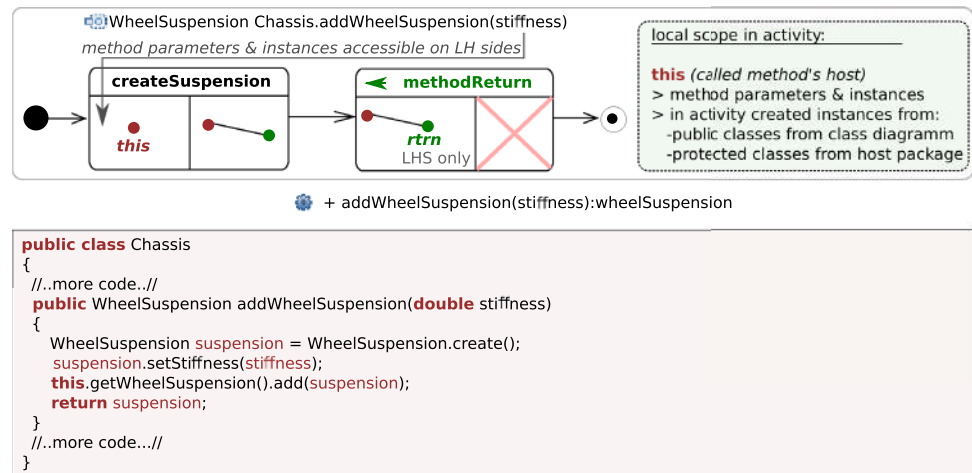


The interface mechanism shall provide *abstraction* and *polymorphism*. The behavior of components is modeled in an abstract way and sub-typing of entities through hierarchical inheritance and/or implementation relations in the class diagram is supported. Figure 7 shows an interface mechanism for calculating mass balances of mechanical parts. These parts inherit the behavior/property of carrying a mass from an interface. Each of these parts has to implement the *getMass()* method that returns the value of its mass. This method, required by the interface, is called by an instance of the *MassBalance* class that calculates the total mass of the linked classes by calling and summing-up the values returned by the attached parts' *getMass()* methods.

4.4 Calling methods and constructors in the activity diagram

According to the initially mentioned spirit, the design compiler shall provide graphical and code-based representations of the design language's components in parallel. Calling constructors and methods within Java code is self-explanatory, whereas a graphical way of calling constructors and methods needs to be defined to maintain the dual representations. Figure 8 shows a mechanism to call a constructor (bottom) and a method (top) within graphically defined rules. The instance whose method shall be called, as well as the specific method, are defined on a graphically defined rule's LHS. Parameters in the method can be

Fig. 9 Duality of method modeling: Either graphical implementation of method *addWheelSuspension* (method call in Fig. 8) as sub activity (top). The method's return instance or value is specified in a LHS search context which is the sub activity's final element. Or the method is implemented as code rule within a specified method body (bottom)



explicitly provided by typing in the value as shown in the top or referenced from other LHS instances' parameters as explained for the constructor call below. Possible return objects of the called method are created on the RHS of the graphically defined rule. Using this approach, the methods calls are used in the same spirit as conventional graphically defined rules. The context³ of a method call is defined on the LHS. The design graph change resulting from the call is defined on the RHS within the mapped LHS context.

Constructor calls are defined solely on the RHS, as a constructor creates an instance whose target context is searched on the LHS and the connection to the mapped context is defined on the RHS. Parameters are defined as in the method call, either explicitly or via instances' parameters from within the rule context as shown in the *Wheel* constructor's parameter list, where the parameter value of the LHS *chassis0* instance is accessed via the expression *chassis0.width*.

4.5 Modeling methods

In the dual spirit the graphical definition of methods needs to be enabled. Filling the methods hull textually with Java code is again self-explanatory, as shown in Fig. 9 bottom. Filling the methods hull graphically is explained in the following and shown in Fig. 9 top: A sub activity is set as method hull and used to define the methods behavior.

The methods input parameters (instances or variables) are available within the sub activity's namespace. Passed-over instances (via the method's parameters) shall be graphically inserted on the LHSs of the graphical

rules either per-default or on-demand via the context-assist function of the rule editor. The instances' fields in the rules' context can be accessed by the instances' and fields' names in the same way, using a corresponding *instanceName.fieldName* expression, as the constructor parameter in the previous subsection has been set. Pure value parameters (eg. number or strings) in the methods parameter list could be accessed via a *self.valueParameter-Name* expression.

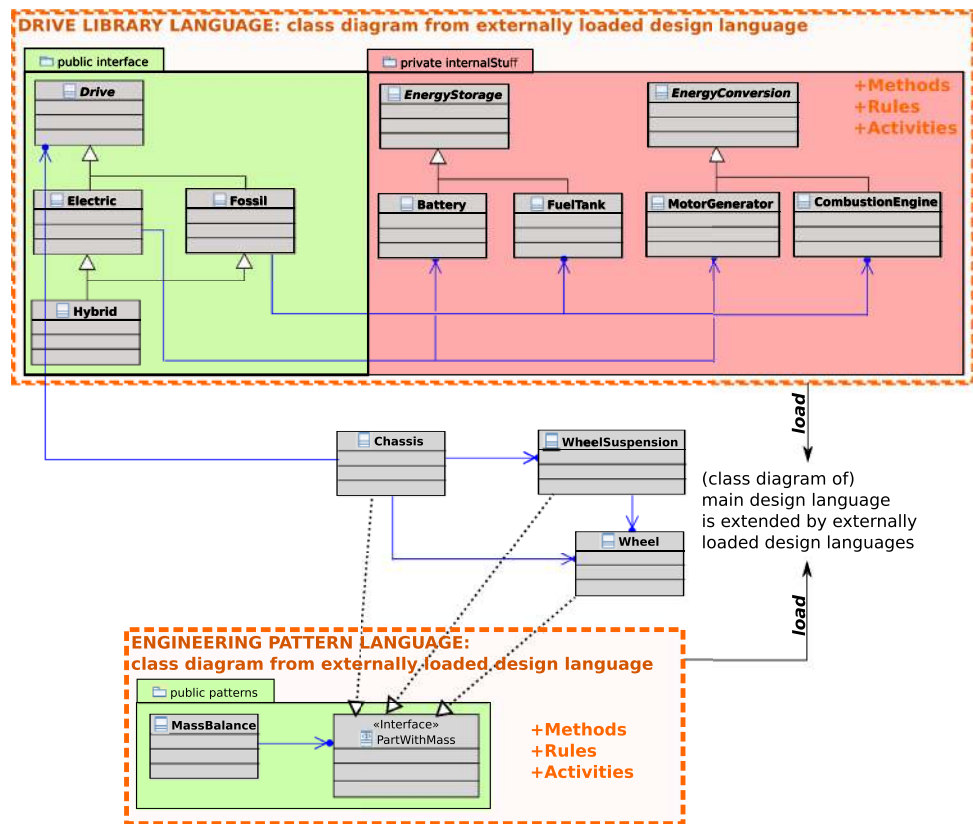
To model the method's return values the last element of the sub-activity is proposed to be a fixed return rule with LHS only that is used for defining the method's return which can be possibly in a further return context. This return context is defined by the context that has been mapped to the RHS of the graphical method call presented in the previous Sect. 4.4 in Fig. 8 top right. In this way different graphical method calls with their specific return context require an individual return rule. Following the presented approach graphically defined methods need to be called graphically and the code-based methods need to be called textually in a *JavaRule* where the return context of the executed method can be explicitly coded (eg. to which instances of the design graph a returned method shall be attached).

The accessibility of instances and variables in the method implementing sub activity complies with Java's namespace definition, which shall account for *encapsulation* and *information hiding*. This local namespace is further explained in Fig. 9 right.

The code-based representation of the rules directly changes the internal representation of the design graph in the design compiler when the rules are executed. In contrast, the graphically modeled rules are interpreted by the design compiler during execution and converted into corresponding manipulations of the design graph.

³ A *context* shall be the graph pattern defined on the LHS of a graphically defined rule as this pattern is searched for in the design graph when executing the rule. This pattern defines the location(s) in the design graph where the rule's manipulations are applied.

Fig. 10 Encapsulation and modularization: Extending a design language (‘s class diagram) through externally loaded design languages (dotted frame) that are encapsulated modules. Only *public* packages of the external design languages are accessible by the main design language in the center



4.6 Encapsulating design languages

The classes in the class diagram can be assigned to packages with different access modifiers. Only the classes and interfaces in the *public* packages are shown and accessible from outer design languages. This allows the creation of complex design tasks in hierarchically encapsulated sub design languages. The internal implementation details are hidden from the outer design language that calls and uses the *public* methods, classes and interfaces provided by the sub languages. Implementing this mechanism in the design languages shall realize both *encapsulation* and *information hiding*.

Figure 10 shows the proposed accessibility concept on the level of (re-)using and integrating multiple class diagrams from multiple design languages. The class diagram of the central design language is schematically shown in the middle. The class diagrams of the externally loaded design languages are shown in the orange dashed boxes.

The classes in the loaded *Drive Library Language* are assigned to two packages (Fig. 10): The left one with the green background is a *public* one, the right package with the red background is the *private* one. This means that in the class diagram of the central design language only the classes in the *public* package of the loaded drive library can be seen and used. The private classes are only used

internally within the drive library when calling methods of the public classes.

Using the external *Engineering Pattern Language* in Fig. 10 bottom is different. The interface *PartWithMass* from the *public* package is implemented by classes in the central design language. Then the *MassBalance* class of the pattern language can be loaded and called to calculate the mass balance in the central language.

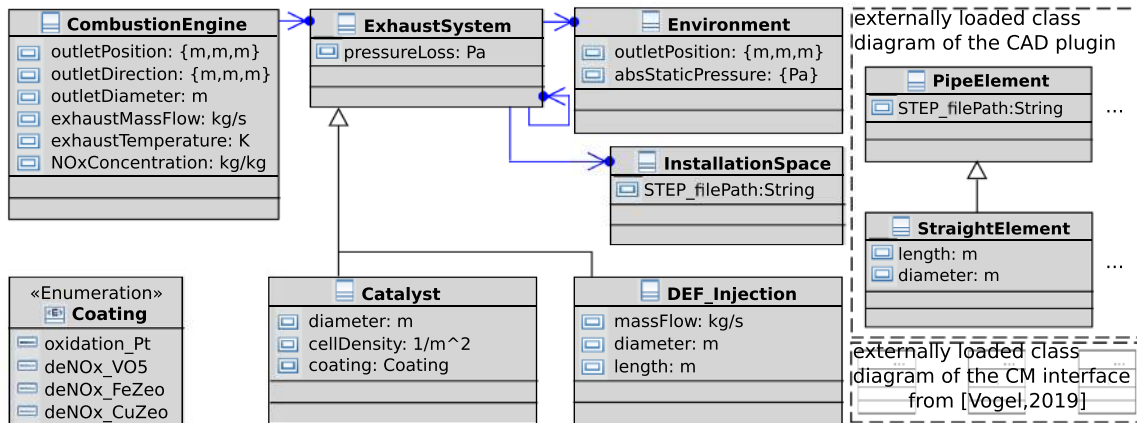
5 Case study: designing SCR system

In this case study a simplified design language taken from [3] shall be introduced as example for the ‘classical’ design language approach without using class methods or interfaces. Then the proposed approach using class methods and interfaces is transferred to this example. And the approaches are finally compared with each other.

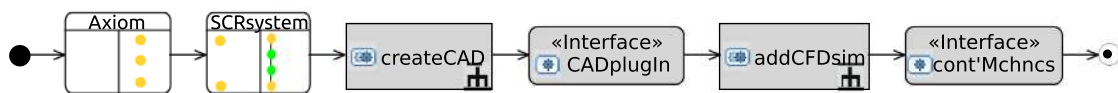
5.1 Classical approach without class methods and interfaces

The example in Fig. 11 shows a graph-based design language of a SCR exhaust aftertreatment system. The design language represents the essential components of a SCR system: an exhaust gas piping system with addition of

Class Diagram

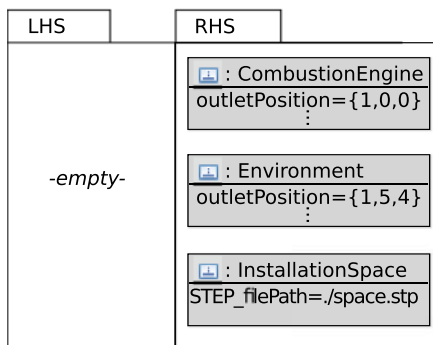


Production System

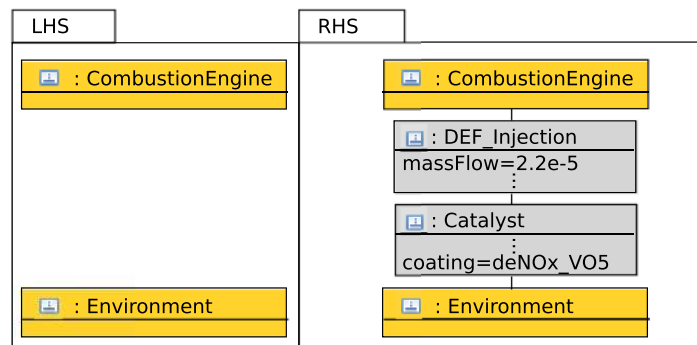


Graphical Rules (if-then scheme)

Rule "Axiom"



Rule "SCRsystem"



Design Graph = Central Data Model (UML instance diagram)

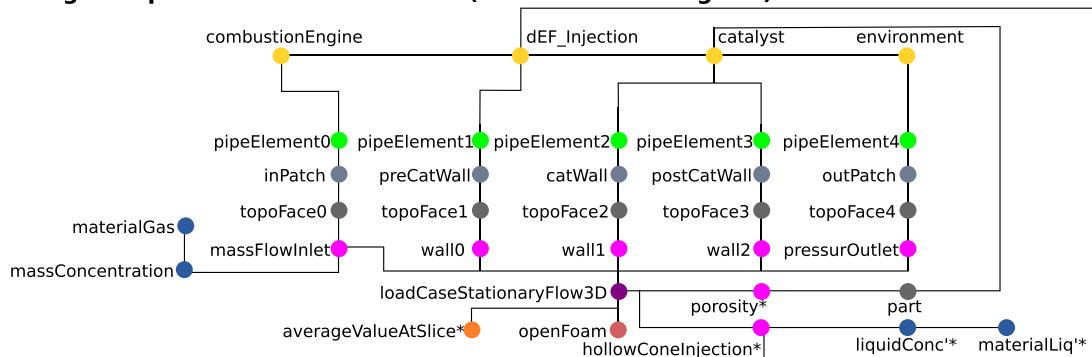


Fig. 11 Case study: Simplified graph-based design language to create the CAD and CFD model of a SCR exhaust aftertreatment system. The schematically shown approach follows the 'classic' design languages style without class methods and interfaces

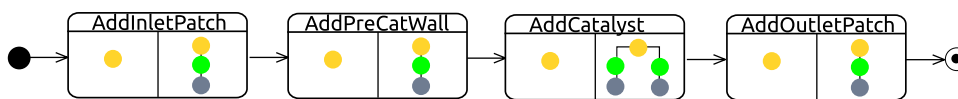


Fig. 12 The sub activity *createCAD* that attaches to each system component instance (yellow) the corresponding CAD instances for hosting the CAD parameters (green) and the CAD representation (grey)

reducing agent by means of an injector (dosing section), up to the catalyst on which the chemical reaction of the conversion of harmful nitrogen oxides into harmless gases takes place as well as the flow section after the catalyst.

The class diagram in Fig. 11 shows the classes of this main components. The CAD and CFD classes are loaded externally as indicated on the right in the dashed box. The production system below the class diagram shows the top-level design sequence: In the first rule *Axiom* the instances of the engine, the exhaust system and the environment are created. In the second rule the SCR system components' instances are added. The sub activities *createCAD* and *addCFDsim* host further hierarchical rule sequences to create the CAD elements as well as to add the CFD related instances. The sub activity to attach the CAD instances to the system component instances is exemplarily shown in Fig. 12. In this sub-activities there are rules for each SCR system component type to instantiate and link the corresponding CAD respectively CFD instances. The result can be seen in Fig. 11 in the bottom where the specific CAD and CFD instances have been created and linked to the system component instances. For each additional system component class, corresponding rules to attach the corresponding CAD and CFD instances have to be modeled, which is very time-consuming. If changes to the system components classes occur, one has to change these rules accordingly in the sub activities. Since the class definitions (class diagram) and the corresponding rules (production system) are now located at different places in the design language, it can very quickly lead to errors due to the missing encapsulation. The following subsection shows that these rules in the production system become unnecessary if the proposed class method and interface mechanisms are used.

These sub activities are followed by a call to the domain specific plugin. The first two graphical defined rules are shown in the mid of Fig. 11. At the bottom of the figure the resulting design graph is shown where the instances are collapsed into colored dots. The CAD related instances are shown in green. The yellow ones are representing the instances of the main components. And the remaining points represent the instances of the CFD simulation.

Figure 13 shows the corresponding CAD model in the background as well as a snapshot of the CFD simulation at the bottom right. The design language covers the

domains of geometry creation (CAD) and computational fluid dynamics (CFD) simulation. In the foreground again the design graph is shown. The instances are now shown in detail and also contain the parameter fields and values as initialized and set in the production system.

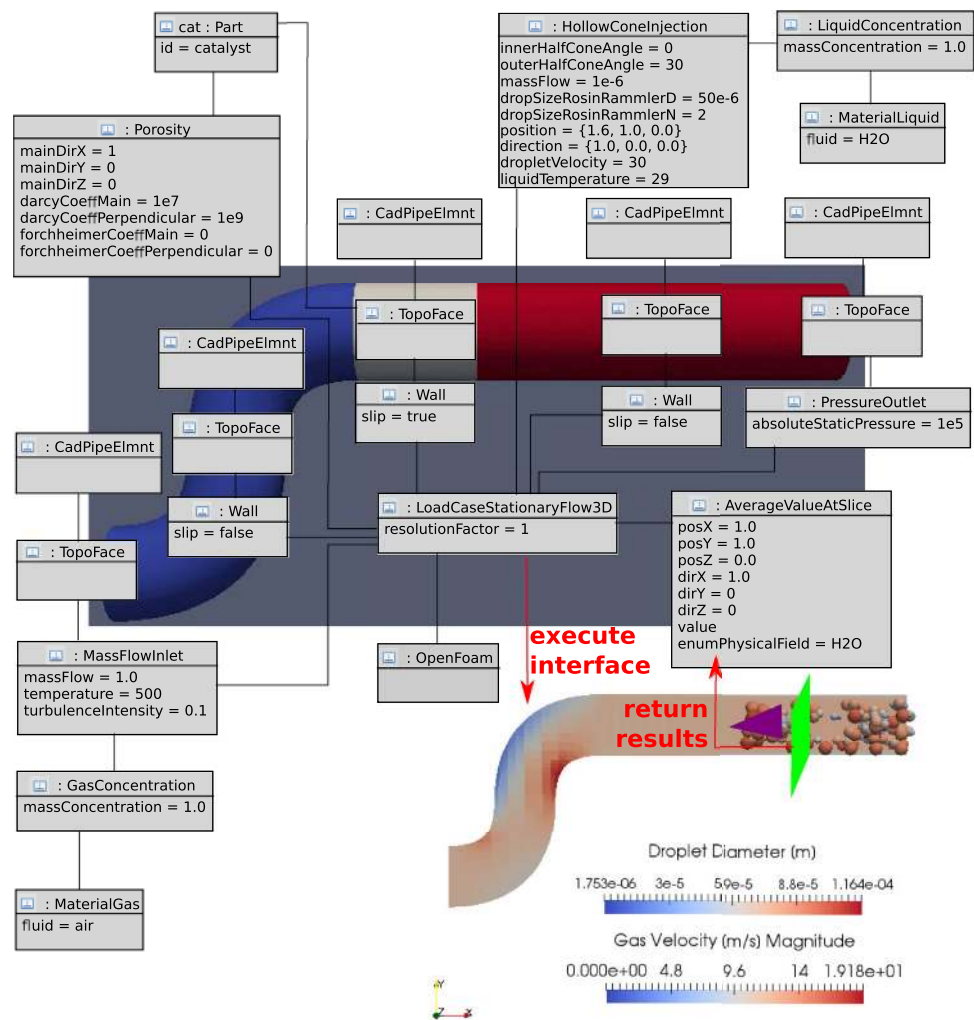
5.2 Proposed approach with class methods and interfaces

In this subsection the previous 'classical' design language is exemplarily adapted to the proposed method using interface and class method mechanisms. Figure 14 shows the modified class diagram at the top as well as the modified production system in the mid. In comparison to the previous approach, the domain-specific contributions of the individual system components of the emission control system (gray classes) can now be implemented directly in the latter. This is realized by implementing domain-specific interfaces (*InternalFlow* and *CADGeo*). These interfaces now require that the implementing classes must return CAD representing instances of *CAD_Represent* and CFD representing instances of *CFD_Represent*. The implementation of the creation of this representations is thus forced to be directly integrated in the component classes which realizes a strong encapsulation.

A schematic inheritance scheme of the CFD representation is shown in magenta at top left in Fig. 14. The specific CFD representations stand for typical boundary types of CFD simulations and they can be provided by a correspondingly physics plug-in as proposed in [3]. In this way they are reused in a generically manner. The only work that remains is the instantiation and parameterization of these reused representation classes in the implementation of the above mentioned interfaces within the component classes.

As a result, the design graph is simplified: The design graph no longer contains the explicit CAD and CFD related instances which are 'polluting' the design graph as they are on the same representation level as the system component instances. These are now hidden within the component classes in the corresponding implementation of the *getCADdata()* and *getCFDrepresentation()* methods requested by the interfaces. In this way, an abstract system component gets different domain-specific "faces".

Fig. 13 Design graph in instance diagram form of the 'classic' design language of the SCR exhaust aftertreatment system. The CFD boundary conditions are added to the CAD topology of an exhaust system. The system contains the catalyst (white) and an upstream hollow cone injector to inject the reduction agent. The system component instances are not shown for space reasons. Reproduced from and more details in [3]



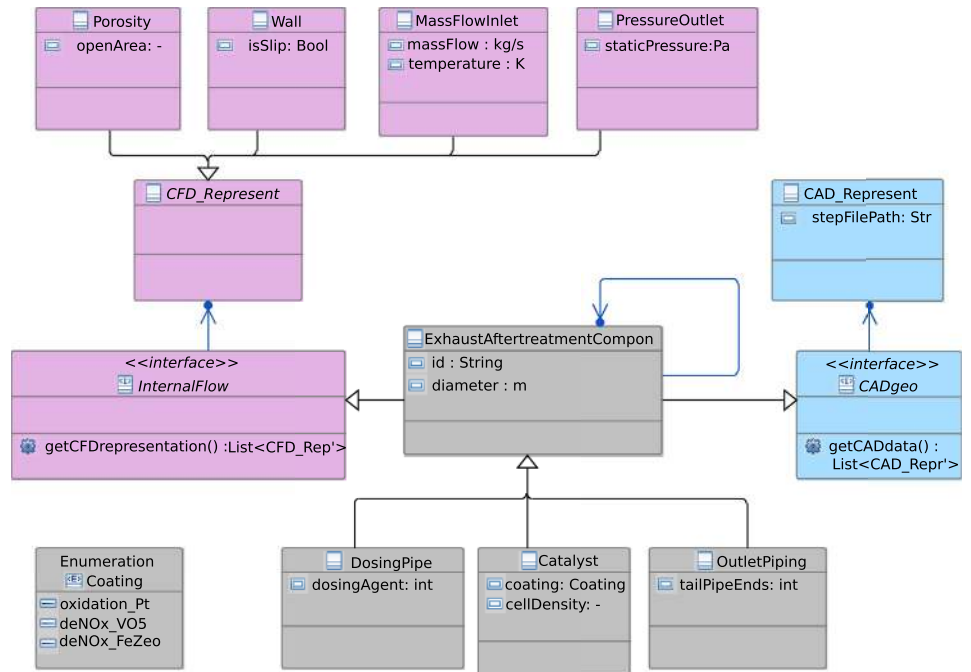
A similar simplification is achieved in the production system. Now it is no longer necessary to model a separate rule for each CAD or CFD data type as well as for each system component class to instantiate and attach the corresponding CAD or CFD instances - as required in the previous design language and shown in Fig. 12. The two Java rules shown at the bottom of Fig. 14, that read the CAD and CFD data to pass it over to the CAD and CFD engine of the specific plug-ins, become simple. In the first step one initializes the domain-specific engine. Then the instances that implement the specific interfaces are easily filtered and added to a list. Next, this list is passed over to the engines. Finally, the engines are executed and trigger the creation of the CAD model or the creation and execution of the CFD simulation.

5.3 Comparison of classical and proposed approach

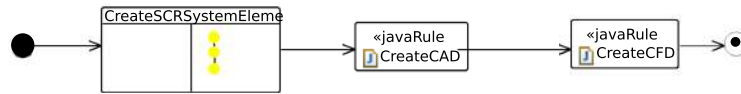
The application of *polymorphism* by requiring the implementation of the CAD and CFD interfaces in the common abstract superclass of the system components results in an improved encapsulation as well as reduced efforts in the production system modeling. The common behavior is defined in the *ExhaustAftertreatmentComponent* superclass which is then specifically implemented in the subclasses. The further processing of these classes' instances can take place, as in the Java rules in Fig. 14 shown, again on the superclass types which means that lengthy case separations for each system component type, as required in the 'classical' design language implementations, are omitted.

Fig. 14 Case study: Simplified graph-based design language to create the CAD and CFD model of a SCR exhaust aftertreatment system. The schematically shown approach follows the newly proposed design languages style using class methods and interfaces

Class Diagram



Production System



Java Rule: CreateCAD

```

public static CreateCAD(DesignGraph designGraph)
{
    CAD_Engine cadEngine = CAD_Plugin.getEngine();
    List<CADgeo> cadGeos = designGraph.getAllInstances(CADgeo.class);
    for(CADgeo cadInterface : cadGeos)
    {
        List<CAD_Represent> cadRepresentations = cadInterface.getCADdata();
        cadEngine.addAll(cadRepresentations);
    }
    cadEngine.createCAD();
}
    
```

Java Rule: CreateCFD

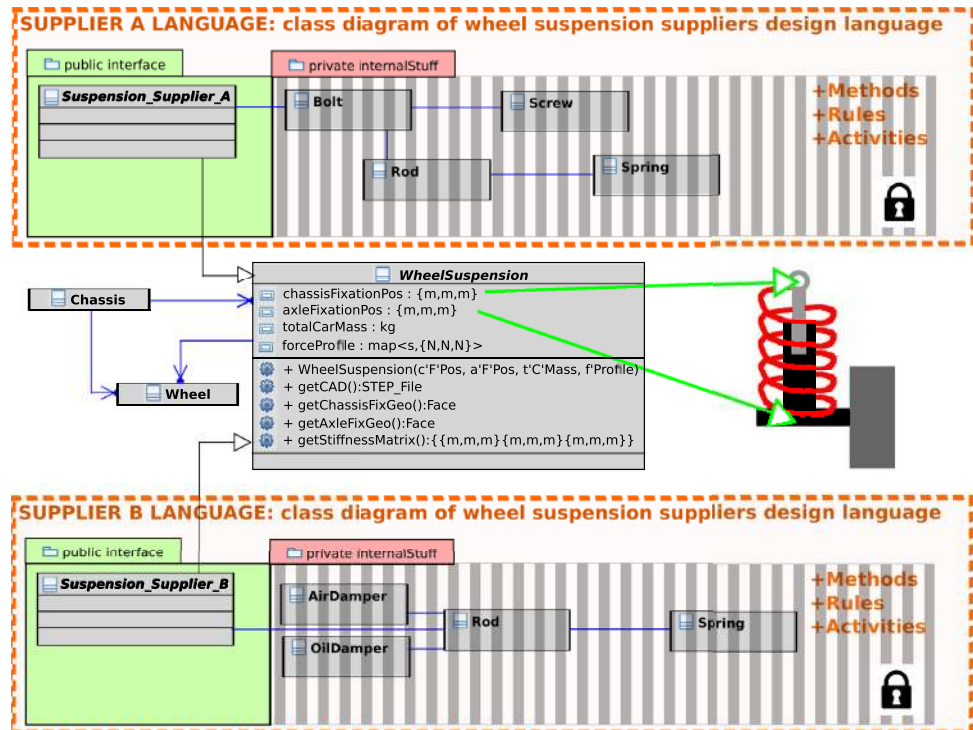
```

public static CreateCFD(DesignGraph designGraph)
{
    CFD_Engine cfdEngine = CFD_Plugin.getEngine();
    List<InternalFlow> internalFlows = designGraph.getAllInstances(InternalFlow.class);
    for(InternalFlow flowInterface : internalFlows)
    {
        List<CFD_Represent> cfdRepresentations = flowInterface.getCFDrepresentation();
        cfdEngine.addAll(cfdRepresentations);
    }
    cfdEngine.createAndRun_CFD();
}
    
```

Using the proposed mechanisms enforces that the methods and the data are implemented within the class

in the same location. Thus the paradigm of object-orientation is fulfilled and the corresponding advantages in the

Fig. 15 Collaboration and division of labor by an embedding and composition of external design languages. The public interfaces to the external design languages are defined by interfaces or abstract classes. The activities, rules and classes in the private packages can be encrypted to ensure information security and know-how protection



treatment of complex system are realized and transferred to product engineering.

6 Results

This section presents generalized findings in applying the suggested methods from Sect. 4 in product engineering. Emphasis is put on addressing the wide-ranging issues formulated in Sect. 3 with small schematic examples, without going as deep into the details as in the case study.

6.1 Information security and collaboration

The presented methods allow an encapsulation and modularization of sub modules in sub design languages as shown in Sect. 4.6. Together with the access modifiers from Sects. 4.5 and 4.6, this enables the encryption of (private) parts of sub design language modules with a distinct separation between the public (interface) elements, that are accessible by third parties, and the critical encrypted parts within the private packages. Suppliers can share their design languages without explicitly sharing their know-how and intellectual property.

Figure 15 shows again a central design language in the mid that shall implement a design process of a carmaker. This exemplary design language shall made up a simplified car model consisting of a chassis, wheels and a third party wheel suspension. In this scenario the suppliers A and B

provide design languages that contain the design process of the suppliers' wheel suspensions. They hide their engineering know-how within the white/black shaded parts of this design languages in the *private* packages. The content of this packages can now be potentially encrypted as symbolized by the lock in the figure. For the central design language of the car only the suppliers' classes in the *public* packages (green background) are visible.

An integrated virtual blueprint of a product development process, consisting of both OEM manufactured and third party components, can be realized this way. Figure 19 shows a scheme where the central product structure of the car is defined by the carmaker in a central design language within the green dashed boxes. The parts of the car, that are potentially provided and engineered by third-party suppliers, are modeled as abstract classes (italic class names). These abstract classes define the interfaces to the suppliers in form of their property fields (eg. for the chassis the number of doors and the material etc.). The property fields shall contain the parameters for both directions: from the carmaker to the supplier (requirements and specifications) as well as from the supplier to the carmaker (realized and guaranteed technical specifications for the part at hand). The method stubs of the abstract classes define the expected behavior and functionality. In the example just the methods that call the creation of the third party parts which finally fills the suppliers property fields. This behavior has now to be implemented by the suppliers. Together with the information security aspect of

Fig. 16 Using the methods and interface mechanism to explicitly model an iterative CAD model creation (following operations use return values of preceding method calls as parameters). Top: class diagram; Mid: activity diagram with schematic graphical method and constructor calls (see Fig. 8); Bottom: created CAD-Model of the spring damper with logic sequence

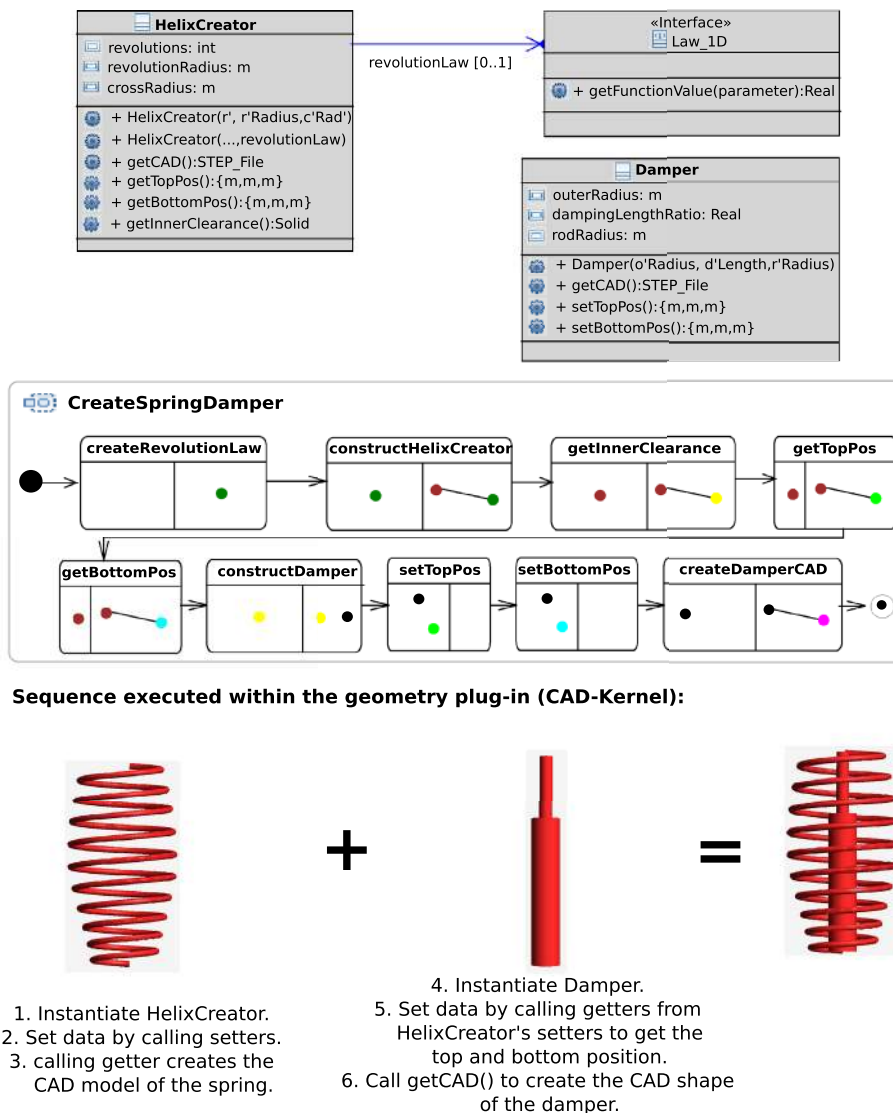


Fig. 15 a hierarchical composition of the car’s development process can be realized including third party knowledge whose intellectual property can be respected by encryption. In this example the interface mechanism is realized through abstract classes, that are made up of method stubs, just like in interfaces, together with property fields to host a defined data set.

6.2 Domain integration

The method call mechanism of Sect. 4.4 supports the integration of different (physical) domains in the multi-disciplinary product design. The CAD model creation is a central task in product design. In general, it is difficult to create a CAD geometry in one step, as it was necessary in the traditional implementation of design languages. One step means, modeling and running a rule sequence that defines the expansion of the CAD geometry in the

central model which is then translated to a CAD geometry within *one CAD plug-in call* in the process chain.

An iterative creation of the CAD model is much more suitable to create complex CAD models, as subsequent geometry manipulations depend on the result of preceding geometry manipulations. For example are sketches extruded to bodies, whose faces are used as sketch planes for subsequent sketches and extrusions. But this requires that a CAD generation is executed iteratively in several steps. This iterative modeling is generically realized with the presented class methods: They can be used to modularize and subtype behavior, like higher level CAD methods. For example, an extrusion method that takes an extrusion vector as a parameter can return the extruded end face as a return parameter. This returned surface can now be used for further operations depending on the properties of the same.

Fig. 17 Two design patterns from object-oriented software engineering. Top: *Builder* pattern to translate the domain-relevant information in the central data model to different simulation applications (implementation of process chains in Fig. 2). Bottom: *Composite* pattern to model systems-of-systems relationships in systems engineering

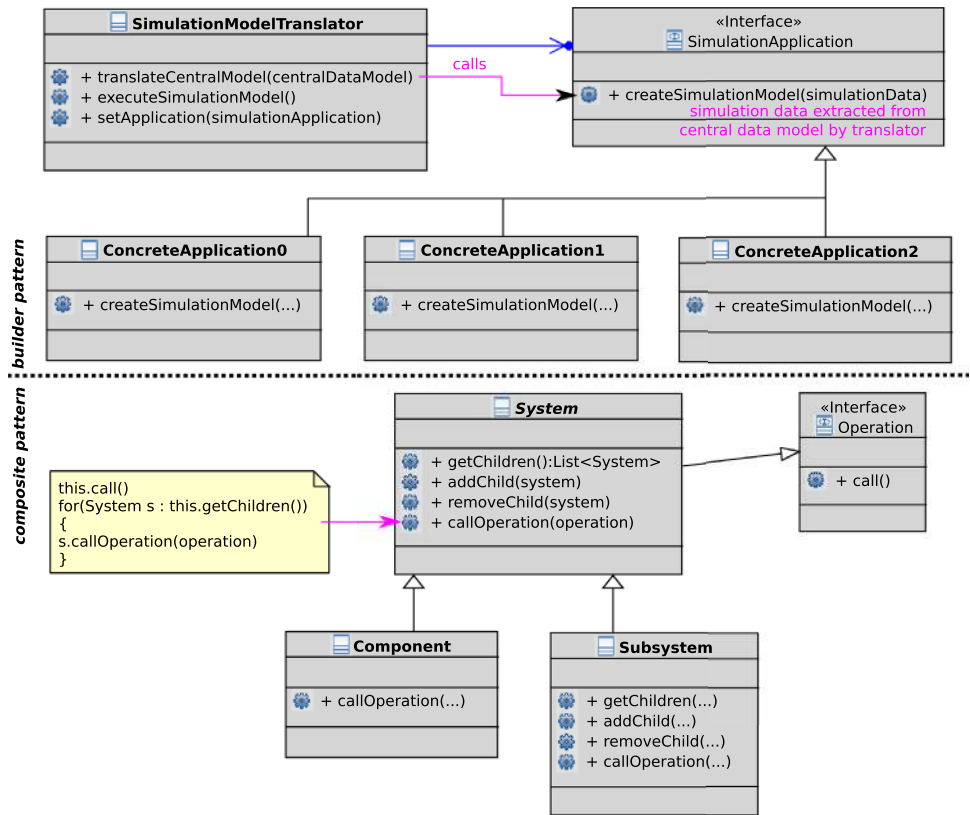


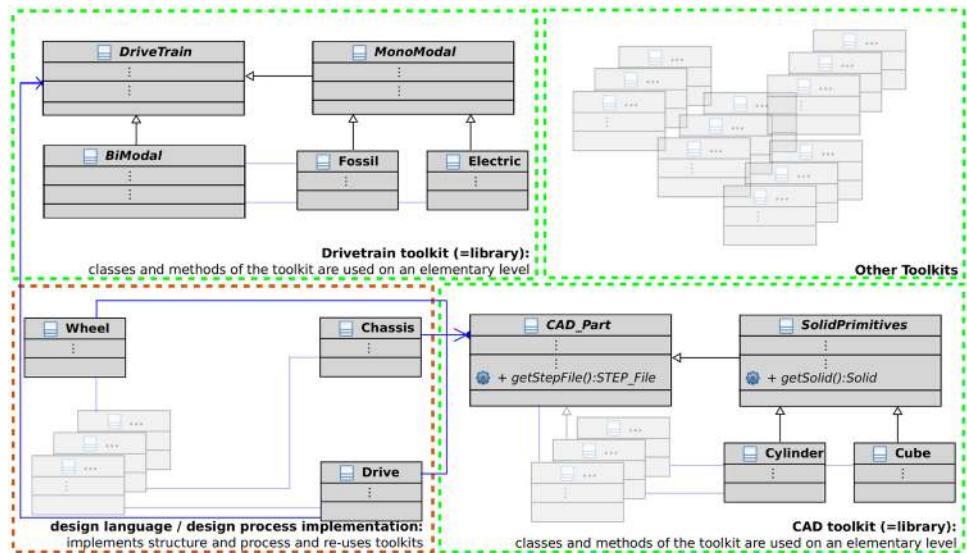
Figure 16 shows the described stepwise generation of the CAD model of a spring-damper as a simplified example. In the mid of the figure a graphical sequence of subsequent CAD operations is shown. In the first rule the *Law1d* revolution law instance shown in the class diagram top right in Fig. 16 is instantiated. In the subsequent rule *constructHelixCreator* the previously created instance is looked up on the LHS and fed to the constructor of the *HelixCreator* class in the RHS. The subsequent three rules are getter calls on the previously created *HelixCreator* instance. In this getter calls the CAD model of the helix is created and the returned artifacts are stored in the newly attached instances on the RHS. The *constructDamper* rule now looks up this returned instance(s) that provide the (getter-)data of the *HelixCreator* on the LHS and feds this data again into the *Damper* constructor. Additional data can be added in the same way by the following *set** rules that look up input data and the *Damper* instance on the LHS and pass the input data to the *Damper* instance by calling the setter methods on the LHS with the input data filled into the methods' parameters list. Finally the *createDamperCAD* rule is called: In the LHS the *getCAD()* method of the looked up *Damper* instance is called and the resulting CAD instance is attached on the RHS. Correspondingly, the creation and simulation of more complex geometries has been already demonstrated by the author [3, 33] and shall be not part of this method-oriented work.

6.3 Reusability

Design patterns are a well known strategy to create reusable software applications and modules [30] in the area of object-oriented software engineering. One can observe that all proposed patterns use *interfaces* realizing the paradigm to "...program to an interface not an implementation...". Interfaces allow the definition of abstract and mandatory behaviors of components and modules that in turn can be reused in many different contexts.

Figure 17 now shows two exemplary applications of software engineering design patterns in the class diagram of a design language. The top pattern is the application of the software engineering *builder* pattern in a design process application. This pattern is used to implement translation processes from a source description, in the shown example the *centralDataModel* of the design language (=design graph), into different target representations, in the example the input decks of different simulation applications. Reading the source description is implemented once in the *SimulationModelTranslator* class which extracts and transfers the simulation data into an intermediate data container instance *simulationData*. The specific builder classes *Concrete** now implement the *SimulationApplication* interface that defines the data transfer of the *simulationData* instance from the translator. The specific builders contains the code to create the specific models out

Fig. 18 Modeling the product structure and design process in a central design language. External toolkits for special purposes are embedded on an elementary level (single classes and methods) to re-use existing engineering solutions (toolkits=engineering libraries)



of the *simulationData* instance. In this way the read out of the data in the first step can be reused for the creation of different target models. This can be eg. used for implementing a plug-in that reads out domain specific data (eg. geometry data) from the central data model and translates this into different input file formats for different target CAD applications.

The bottom of Fig. 17 shows the composite pattern transferred to a system engineering product decomposition. An abstract *System* class has two realizations that inherit from system: *Component* and *Subsystem*. A *Subsystem* instance can hosts further subsystems as well as *Component* instances. The components can host no further instances as they are the lowest level entity in the resulting hierarchical system-of-systems tree. Now the *System* class children implement the *Operation* interface which then can be called hierarchically by once calling the top-level *callOperation* method according to the schematic source code shown in the figure on the left. With this mechanism operations can be called on all systems in an easy way. For example to update the system states or to execute a geometry creation operation on all system instances hanging under a given system instance.

Additionally, the *mass balance* engineering pattern presented in Fig. 10 exemplarily illustrates a potential application of the *interface* mechanism to enable *reusable product design patterns*. In this way reusability patterns from software engineering can be directly applied and the principle itself transferred to engineering processes.

6.4 Modularization

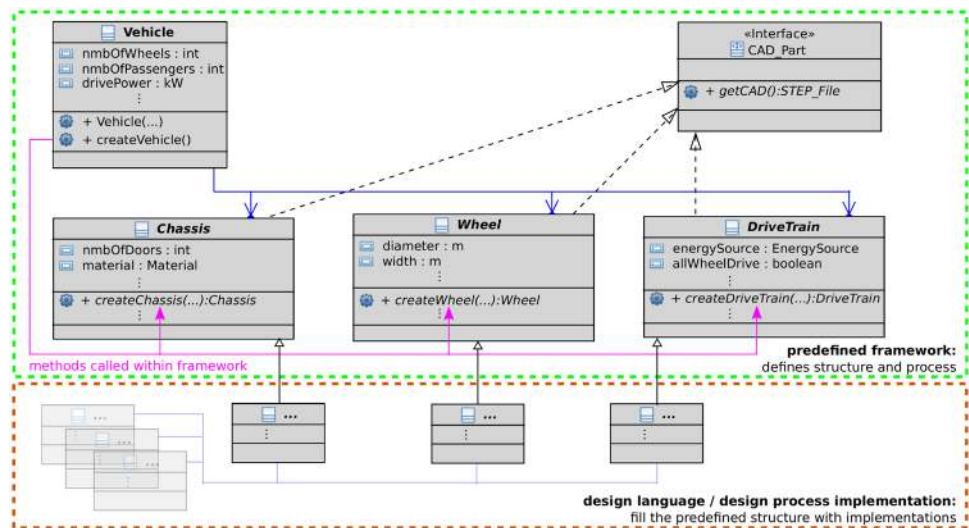
The introduction of interfaces and methods in the classes leads to an additional structuring and standardization on

an even higher abstraction level. Well known top-level software design concepts as *toolkit structure* or *framework structure* can now be realized in virtual engineering [30].

In a *toolkit* setup existing modules are reused as (software) libraries to achieve specific tasks. Transferring this approach to design languages is exemplarily shown in Fig. 18. In this example a simplified car shall be created again. The toolkit scenario now assumes that existing design languages for subtasks are readily available (green dashed boxes). These represent the toolkits. The task of creating a central design language hosting the design process of the car now consists of using these toolkits and calling up the process steps, provided by the toolkits, in the correct sequence. Shown here are a powertrain toolkit in the form of a corresponding design language and a CAD toolkit that is called up from the central design language (orange dashed frame) triggering the class methods and interfaces.

In contrast Fig. 19 shows a *framework*-based strategy of modularization as already explained in Sect. 6.1. In the framework architecture, the global structure and decomposition is predefined by abstract *interfaces* that have to be realized by the specific implementations. Transferred to product engineering, the design process is predefined (compare systems engineering [34]) in a framework which facilitates and structures the design process and enables the exchange and the reusability of existing components. The example shows the predefined structure of the car in the class diagram hosted in the green dashed box. Now the decomposed abstract product modules (*Chassis*, *Wheel* and *DriveTrain*) need to be implemented in the orange dashed section. In this way the *framework* is just the opposite approach as the *toolkit* structure where the

Fig. 19 A existing framework defines the product's structure and the product's design process through *abstract interfaces* and *abstract classes* (italic labels). The concrete design language implements the predefined methods and classes of the framework



implementations of the modules have been assumed to be given.

7 Discussion

The lack of object-orientation in the classical design language approach has been addressed and class methods as well as interfaces have been added to overcome this issue. A prototypical implementation within the Design Compiler 43 was created and graphical method modeling and calling mechanisms have been proposed, that fully support the dual textual (code-based) and graphical modeling approach. It was shown that a more complete implementation of object-oriented modeling approaches results in improvements in the following design-relevant areas:

- Modularization
- Reusability
- Domain integration
- Collaboration
- Information security

These points represent important requirements in modern product design to handle complex design processes. The support of these requirements represents a significant improvement of the graph-based design languages and facilitates the application of the method by reducing the upfront modeling effort and modeling complexity. Divide-and-conquer based simplifications are enabled by the introduced extensions and systems engineering approaches can be realized more generically.

Compared to the UML-profile based approach presented in [35] the proposed extension provides a much more natural integration and modeling of domains

(Sect. 5) that does not require cumbersome language extensions to the UML metamodel. Hierarchical rule structures, as recently presented for shape grammars in [18], can be now easily realized in graph-based design languages by the enabled and improved modularization. Recently proposed graph-based design languages [36] show crowded and complicated design graphs. These can be simplified by shifting portions of the design process into the class (method) implementation itself as enabled by the present work.

On the other hand, moving artifacts of the design process into the classes leads to a more black-box-like representation. Although this is precisely what information hiding in the object-orientation aims at, it leads to the fact that processes, previously explicitly modeled in the production system, are now hidden in the class methods.

The code-based class methods and interfaces have been already introduced in the current design compiler release. The graphical method modeling and calling proposed in this paper is not yet fully implemented as this takes more effort. Further work needs to be done on how the encryption of externally used and loaded design languages can be practically implemented in the safest manner.

8 Conclusion

The introduction of class methods and interface mechanisms in graph-based design languages fills the lack of object orientation in the previously used methodology. The proposed extensions enables the adoption of object-oriented design methods, as design patterns and toolkit/framework architectures, to product engineering. Proven concepts as *abstraction*, *encapsulation*

and *polymorphism* are transferred to the virtual product design with design languages. The graph-based design languages are brought to the next level in terms of supporting and handling complex product design processes.

For the future, cleanly modularized design processes, formally defined by their interface specifications, may enable the implementation of self-organized design processes. Such a self-organized process could run without an explicitly defined execution sequence in the central activity diagram. It seems possible to derive an execution sequence from predefined modules in a self-organized way, based solely on the interface signatures of the modules, that aims to fulfill given product requirements. Replacing the procedural production system by a self-organized process, the graph-based design languages would formally move more in the direction of classic design grammars, but with much more powerful rule equivalents in the form of modularized complex design sequences.

Acknowledgements Open Access funding provided by Projekt DEAL. This article is based on the arXiv preprint: S. Vogel, P. Arnold: Towards a More Complete Object-Oriented Design Grammars, [arXiv:1712.07204](https://arxiv.org/abs/1712.07204)[cs.SE], last revision: 2020. The work was partially supported by the project “digital product life-cycle (ZaFH) funded by the European Regional Development Fund and the Ministry of Science, Research and the Arts of Baden-Württemberg, Germany (www.rwb-efre.baden-wuerttemberg.de).

Compliance with ethical standards

Conflict of interest Corresponding author Samuel Vogel declares no conflict of interest. Co-author Peter Arnold is employed at ILLS mbH which is the vendor of the Design Compiler 43 software used in the presented work.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

- Pahl G, Beitz W, Feldhusen J, Grote KH (2007) Engineering design: a systematic approach. Springer, London
- Walden D, Roedler G (2015) Systems engineering handbook: a guide for system life cycle processes and activities, 4th edn. INCOSE/Wiley, Hoboken
- Vogel S (2019) An application-independent continuum mechanics interface for virtual engineering. *Eng Comput* 35:551–565
- Goetzendorf-Grabowski T (2017) Multi-disciplinary optimization in aeronautical engineering. *Proc Inst Mech Eng Part G J Aerosp Eng* 231:095441001770699
- Martins JRR, Lambe AB (2013) Multidisciplinary design optimization: a survey of architectures. *AIAA J* 51:2049–2075. <https://doi.org/10.2514/1.J051895>
- Cooper S, Fan I, Li G (2001) Achieving competitive advantage through knowledge based engineering: a best practice guide. Cranfield University, Bedford
- Sobieski J, Morris A, van Tooren M (2015) Multidisciplinary design optimization supported by knowledge based engineering. Wiley, Hoboken
- La Rocca G (2012) Knowledge based engineering: between AI and CAD. Review of a language based technology to support engineering design. *Adv Eng Inform* 26:159–179
- Verhagen WJ, Bermell-Garcia P, van Dijk RE, Curran R (2012) A critical review of knowledge-based engineering: an identification of research challenges. *Adv Eng Inform* 26:5–15
- Komoto H, Tomiyama T (2011) A theory of decomposition in system architecting. In: Proceedings of ICED, international conference on engineering design
- Adcock R (2017) The guide to the systems engineering body of knowledge (SEBoK). The Trustees of the Stevens Institute of Technology, BKCASE Editorial Board, Hoboken. www.sebok.wiki.org. Accessed 10 April 2018
- Estefan J (2008) A survey of model-based systems engineering (MBSE) methodologies. Technical Report. INCOSE-TD-2007-003-02, International Council on Systems Engineering
- Wang T, Truptil S, Benaben F (2017) An automatic model-to-model mapping and transformation methodology to serve model-based systems engineering. *Inf Syst e-Bus* 15(2):323–376. <https://doi.org/10.1007/s10257-016-0321-z>
- Antonsson E, Cagan J (2001) Formal engineering design synthesis. Cambridge University Press, Cambridge
- Brown KN (1997) Grammatical design. *IEEE Expert Intell Syst Appl* 12:27–33
- Muenzer C (2015) Constraint-based methods for automated computational design synthesis of solution spaces. Ph.D. thesis, ETH Zuerich
- Königseder C, Stanković T, Shea K (2016) Improving design grammar development and application through network-based analysis of transition graphs. *Des Sci* 2:e5. <https://doi.org/10.1017/dsj.2016.5>
- Puentes L, McComb C, Cagan J (2018) A two-tiered grammatical approach for agent-based computational design. In: 44th Design automation conference, Proceedings of the ASME design engineering technical conference. American Society of Mechanical Engineers (ASME). <https://doi.org/10.1115/DETC2018-85648>. ASME 2018 international design engineering technical conferences and computers and information in engineering conference, IDETC/CIE 2018; Conference date: 26-08-2018 through 29-08-2018
- Stouffs R, Hou D (2019) Composite shape rules. In: Gero JS (ed) Design computing and cognition '18. Springer, Cham, pp 439–457
- Kröplin B, Rudolph S (2005) Entwurfsgrammatiken - Ein Paradigmenwechsel? *Der Prüflingenieur* 26:34–43
- Hertkorn P, Reichwein A (2007) On a model driven approach to engineering design. In: Proceeding of ICED, the 16th international conference on engineering design

22. The design compiler 43v2 (2005). www.iils.de. Accessed 15 Dec 2017
23. Arnold P, Rudolph S (2012) Bridging the gap between product design and product manufacturing by means of graph-based design languages. In: TMCE
24. Haq M, Rudolph S (2004) Ews-car: A design language for conceptual car design. In: VDI-Berichte 1846, Conference on numerical analysis and simulation in vehicle engineering, Würzburg, Germany, pp 213–237
25. Irani M, Rudolph S (2005) Space station design rules. SAE Aerosp Eng Mag 25:43–46
26. Schaefer J, Rudolph S (2005) Satellite design by design grammars. Aerosp Sci Technol 9:81–91
27. Vogel S, Danckert B, Rudolph S (2012) Knowledge-based design of scr systems using graph-based design languages. MTZ Motortechnische-Zeitschrift 73:702–708
28. Gross J, Rudolph S (2012) Dependency analysis in complex system design using the firesat example. In: INCOSE international symposium, Rom
29. Tonhaeuser C, Rudolph S (2017) Individual coffee maker design using graph-based design languages. In: Gero J (ed) Design computing and cognition' 16. Springer, Cham
30. Gamma E, Helm R, Johnson R, Vlissides J (1994) Design patterns: elements of reusable object-oriented software. Pearson Education, Delhi
31. White G, Sivitanides M (2005) Cognitive differences between procedural programming and object oriented programming. Inf Technol Manag 6(4):333–350
32. Jackson P (1998) Introduction to expert systems, 3rd edn. Addison-Wesley Longman Publishing Co., Inc., Boston
33. Vogel S, Rudolph S (2016) Automated piping with standardized bends in complex systems design. In: Proceedings of the seventh international conference on complex systems design and management
34. Walden D, Roedler G (2015) INCOSE systems engineering handbook: a guide for system life cycle processes and activities, 4th edn. Wiley, Hoboken
35. Reichwein A (2012) Application-specific UML profiles for multi-disciplinary product data integration. Ph.D. thesis
36. Ramsaier M, Breckle T, Till M, Rudolph S, Schumacher A (2019) Automated evaluation of manufacturability and cost of steel tube constructions with graph-based design languages

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.