INTERNATIONAL DESIGN CONFERENCE - DESIGN 2010
Dubrovnik - Croatia, May 17 - 20, 2010.

DESIGN 2010

OBJECT-ORIENTED CONCEPTS FOR COMPUTATIONAL DESIGN SYNTHESIS

B. Helms and K. Shea

Keywords: computational design synthesis, graph grammars, object-oriented programming, implementation

1. Introduction

Computational tools have become routine means of support in the development of new, innovative and complex products in the last decades. The aim of approaches and tools in the area of computational design synthesis are to generate alternative solutions tailored to particular problems and to computationally describe solution spaces. As not only products are getting more and more complex, but also the environments in which they are developed and used, research in the field has proven it's capability to support engineers when dealing with these challenges. However, design synthesis tools need to extend their scope of applications and become more efficient, more intelligent and more flexible in order to encourage industry up-take.
A similar situation led in the sixties – after procedural programming has proven its great potential – to the development of the concepts of object-orientation and their implementation in the programming language Simula-67 [Meyer 1997]. This paper presents goals and concepts from object-oriented programming, points out their benefits and draws analogies to computational design synthesis in order to seize the same or similar benefits. More specifically, the realization of object-oriented techniques in design synthesis is presented in the context of graph-grammars. The aim of this paper is to present how the fundamentals of object-orientation are applicable in the computational design synthesis methodology and open the field for new promising applications. Hence, this paper concentrates on representation methods for synthesis. A major challenge is always efficient ways to encode and modify knowledge within computational design synthesis tools, e.g. through modifying grammar rule sets. This paper does not consider search or optimization.
First, the paper reviews related work and background on computational design synthesis, especially approaches based on graph-grammars. Next, the goals of object-oriented programming and their significance in the field of computational design synthesis are presented. After a short presentation of the process through a computational design synthesis study, the realization of object-oriented principles in computational design synthesis is illustrated. Afterwards, the definition part of the approach is presented in which the formal grammar is defined followed by the execution of the grammar in the application part. In the discussion the benefits of this approach and future work are presented.

2. Background

Formal methods for computational design synthesis are aimed at aiding designers in developing better products faster through rapid generation of spaces of feasible, optimized, and when appropriate simulation-driven designs. Grammar-based methods capture the engineer's knowledge or knowledge

stemming, e.g., from design catalogues in a formalized representation, an engineering design grammar, so that the knowledge can be used to generate solutions.
Conceptually, graph-grammars are a generative method that consists of the vocabulary and a set of rules. Whereas the vocabulary contains all valid elements represented by nodes and edges, the set of rules defines transformations that model how these elements can be created, deleted or modified. In the same way as a natural language is based on words and grammatical rules, it is also possible to develop a language of designs via design (or more specifically graph-) grammars. Starting with a legal construct, repeated application of different grammar rules generates new designs. The combinatorial expansion of all valid rule sequences is termed the design language. The advantage in using graphs is the fact that they can be used as foundation for almost any kind of formal modeling language in conceptual design (e.g. SysML diagrams, function models, geometric models etc.). Further, they provide a strong basis for computational representation and transformation.
Based on the implementation of graph grammars in mechanical engineering, Kurtoglu and Campbell [Kurtoglu 2006] transform a function structure into a configuration flow graph, Schmidt et al. [Schmidt 2000] synthesize mechanisms and epicyclic gear trains. Based on the combination of shape and graph grammars, automated simulation and multi-criteria search, Starling and Shea [Starling 2005] automate the design of gear systems.
Although these approaches are suitable means for exploring design spaces efficiently, their representations are not based on object-oriented techniques. For example, they do not have an explicitly upfront defined modeling space that classifies the design elements in an object-oriented way (the so-called meta-model) and serves as a foundation for the definition of a set of rules. However, this is often not necessary because of the fact that the modeling space is implicitly defined through the scope of application. For example, all nodes in the approach for the synthesis of gear systems by Starling and Shea [Starling 2005] define shafts and all of the edges correspond to gear pairs. Hence, no a priori and explicit definition of semantics through typing the elements is required. However, labels and attributes are applied to specify and detail the significance of the elements. When using purely labels and attributes, an implicit definition of the modeling space is achieved (probably in the head or on paper) but that approach does not take advantage of a strict separation between the definition of a modeling space and its execution the way it is realized in object-oriented programming.
However, the synthesis approach by Kerzhner and Paredis [Kerzhner 2009] that has strong influences from the field of computer-aided software engineering and is based on the formal modeling languages SysML and MOF, adopts certain principles from object-oriented programming. For example, the design elements are defined in advance in a meta-model on which the definition of rules is based. Concerning rule application, this approach is based on probabilistic selection of rules.
In prior work by one of the authors [Bolognini 2007] some object-oriented concepts, such as interpreting nodes in a graph as objects that define their own behavior, were realized in order to synthesize MEMS structures based on simulation and optimization. That approach allowed for a high degree of usability and modularity. However, a clear separation into a meta-model and its application - was not realized.
The authors previous work focused on synthesis methods for product development that are based on classic paper-based methodological foundations [Helms 2009]. As these approaches often involve several levels of abstractions or modularity and the re-use of knowledge from catalogues [Wölkl 2009], a powerful structuring scheme on class-level is crucial. Not only the definition of the modeling space, but also on an object level, the instantiation of element classes and the application of rules and their combination to rule sequences is facilitated and provides space for sophisticated features when utilizing the object-oriented programming paradigm.

3. Goals of object-oriented programming

As object-oriented programming became the standard paradigm in software development, Meyer [Meyer 1997] identified the following quality factors to which the object-oriented method has made significant contributions:

- *Extendibility* is the ease of adapting software products to changes of specification.

- *Reusability* is the ability of software elements to serve for the construction of many different applications.
- *Compatibility* is the ease of combining software elements with others.
- *Efficiency* is the ability of a software system to place as few demands as possible on hardware resources, such as processor time, space occupied in internal and external memories, bandwidth used in communication devices.
- *Portability* is the ease of transferring software products to various hardware and software environments.
- *Ease of use* is the ease with which people of various backgrounds and qualifications can learn to use software products and apply them to solve problems. It also covers the ease of installation, operation and monitoring.

Any implementation of computational design synthesis methods is a piece of software, hence, these qualities are also desirable. In order to broaden the scope of application in computational design synthesis, these quality factors can be interpreted as:

- *Extendibility* is the ease of adapting the method and the implementation to a wider range of applications or completely new applications that includes an extension of both vocabulary and rules. Since knowledge is always evolving, this point is of particular interest.
- *Re-usability* is the ability to define a grammar for multiple applications and re-use the vocabulary and rules.
- *Compatibility* is the ability to define different sets of vocabulary and rules, e.g. in different domains (design, manufacture), that can be interchanged for different applications or product generations.
- *Efficiency* means that the formalization and the encapsulation of knowledge are supported in a way that the demand of resources is as low as possible. This involves computational resources, but more importantly is to efficiently support the time consuming human task of encoding knowledge.
- As *Portability* relates solely to the implementation, it is not considered here.
- *Ease of use* means that the structure of the grammar can be understood as intuitively as possible and provides the foundation for the factor of extendibility.

Extending methods for design synthesis such that they include concepts from object-oriented programming provides a wide range of benefits. This is a promising approach to realize more efficient, flexible and extendable systems.

4. Application of object-oriented concepts in computational design synthesis

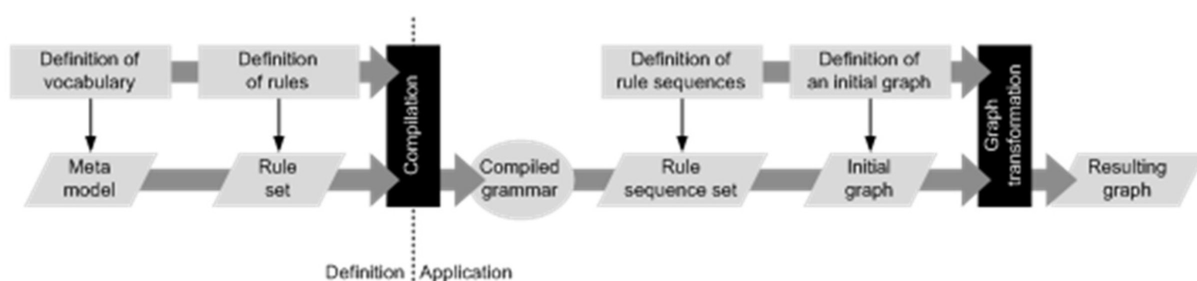

Figure 1. Definition of an object-oriented computational design synthesis study

Following an object-oriented approach, the procedure of defining a computational design synthesis study is subdivided into a definition and an application part, fig. 1,:

- In the first part, the definition of the grammar, consisting of vocabulary and rules, results – after compilation – in an executable grammar. In programming, this part is analogous to the implementation of classes (cf. vocabulary) and their functions (cf. rules) and remains static throughout the study.
- The second part describes the application of the compiled grammar in order to perform problem-solving tasks. Through the definition of a rule sequence, the logic, or strategy, behind

the application of rules can be encoded. Based on an initial graph, the transformation is carried out by applying the defined rule sequence and returning the resulting graph. Afterwards, this graph can be evaluated, e.g. through simulation, be used as an initial graph for a subsequent transformation or be stored in a design archive.

4.1 Definition

In programming, the definition, or static part, captures the assumptions the software developer has taken in order solve a given task and results in the implementation of classes and functions. The counterpart in this synthesis approach is the definition of a vocabulary (in a meta-model) and rules (in a rule set), i.e. the definition of a formal grammar, that contains the formalized engineering knowledge. Both, the vocabulary and the rules remain static throughout the application and define the scope of a generative design study.

4.1.1 Definition of a vocabulary

Classes are used in programming as a static description of elements that can be instantiated at run-time, i.e. when the program is executed. In formal modeling approaches, the definition of the modeling space is realized with a meta-model. The common understanding is that a meta-model is a model that defines a model, i.e. the modeling elements, valid combinations of them, etc. Hence, in order to avoid confusion, the terms class-model and meta-model are equivalent and should be used synonymously when adopting object-oriented techniques in computational design synthesis.
The example in Figure 2 depicts a meta-model that evolved throughout three studies. In this example, only the definition of node types is represented, edge types are disregarded but follow the same principle. A meta-model does not contain the objects (represented as ellipses and named with lower case letters) that can be used in the graph. It rather defines types (represented as rectangles and named with a first upper case letter) from which objects can be instantiated and used for modeling. To summarize, the meta-model defines a schema of model elements that provides, on the one hand, the basis for formulating rules and, on the other hand, serves as a basis for instantiating objects.

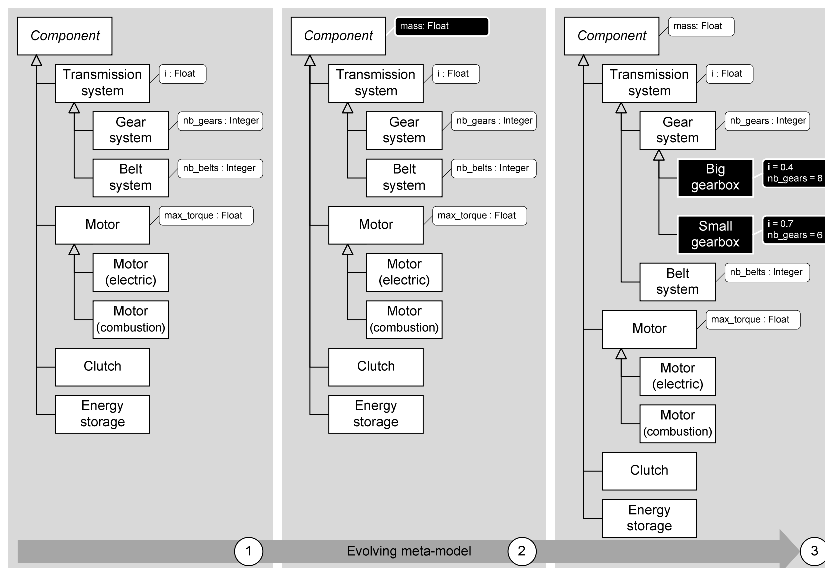

Figure 2. Example of an object-oriented meta-model for graph-grammars that evolves during three generative design studies

The definition of a meta-model follows a hierarchical approach whereas the upmost class *Component* is an abstract class (represented through italic letters) which cannot be instantiated. This means that no objects of the type *Component* can appear in the model. The concept of inheritance provides that all attributes of an element are inherited by its descendants. That means, for example, that the node types *Gear system* and *Belt system* automatically possess the attribute *i* (containing the transmission ratio as a float) although it is defined by their parental node type *Transmission system*. The fact that only

additional features, e.g. the number of gears, had to be added and that subsequent changes are forwarded – via inheritance – to one's class descendants, contributes strongly to the goal of efficiency and also in terms of ease of use: A logical, hierarchical structure is more intuitive and understandable than unstructured vocabulary
Also in terms of extendibility, a hierarchy of inheritance provides huge advantages. For example, as the consideration of the design's mass became necessary, the attribute *mass* has been added in the second study (Figure 2). As all descendants of the node type *Component* are considered as mass-carrying elements, it is sufficient to only add the attribute once to this node type. All child elements are automatically equipped with the new property via inheritance. Further, inheritance supports extendibility by enabling the extension of the inheritance hierarchy without creating incompatibilities. For example, the two elements *Big gearbox* and *Small gearbox* are added in the third study as a further differentiation of gear systems became necessary. Those two elements became descendants of the class *Gear system,* which means that all attributes of the super ordinate elements (transmission ratio *i*, number of gears *nb_gears*) are inherited; only the further detailing by assigning concrete values to the new node type attributes was necessary. The application of object-oriented concepts provides a rich foundation for the formalization of engineering knowledge. In [Helms 2009] the realization of ports has been presented that capture the compatibility of design elements based on a taxonomy of flows (energy, material, signal).

4.1.2 Definition of rules

The definition of grammar rules has its counterpart in programming as the definition of functions. Both, functions and rules are means for encoding the manipulation of. Although rules codify operations that involve objects, their definition is based on the prior definition of a meta-model and remains unchanged during the program execution. For this reason, the definition of rules is independent from their application on specific graphs. Due to this separation of knowledge and application, the approach presented in this paper is in accordance with the same concept known from KBE systems.
Rules in graph-grammars always follow the same principle: A left-hand side pattern is searched for and replaced with a right-hand side pattern. The mode of operation and the formalization of knowledge shall be illustrated by the problem-specific rule *Insert clutch:* In case a clutch has to be added in a powertrain configuration, usually, several positions are conceivable, e.g. before or after the transmission system. But, in order to minimize size and cost of the clutch, the position should be where the torque is minimal and this can be figured out through the transmission ratio *i*: If *i* is lower than one, the torque is lower before the *Transmission system* and vice versa. Thus, on the left-hand side a subgraph is searched for that contains the node *a* of type *Motor* and the node *b* of type *Transmission system* with a transmission ratio lower than 1. The nodes *a* and *b* are local variables that are only valid within the scope of the rule and can be any nodes of the corresponding types. These local variables are required in order to model the correspondence between elements on the left-hand side and their occurrence on the right-hand side to identify the position of insertion of the clutch with a mass of 21.4 (SI units are assumed). Another advantage of the object-oriented meta-model becomes apparent here: As the definition of this rule is based on the node types *Motor* and *Transmission system,* any of their descendants will also be found on the left-hand side. Hence, this rule remains valid and applicable even when new node types, e.g. in study 3, Figure 2, are added.
Beyond the allocation of elements between the left-hand side and the right-hand side, local variables – same as in functions in programming – can be used to capture the logic and the processing of data within rules. In the case that a components' mass needs to be adapted, e.g. more robust components are required due to an increase of power, a very flexible rule (*Change mass*, Figure 3) is required that applies either to one specific element or to a whole class of elements and should be able to handle different factors of mass change. The way to realize this flexibility is to equip the rule with the parameters *e* (defined as the node type *Components* or one of its descendants) and *factor* (defined as floating number). When the rule is called within a rule sequence at run-time these parameters have to be defined: The parameter *e* defines the left-hand side and may contain either a node type, so any element of that type, or one of its descendants, would be found, or one very specific instance of a node

to which the change of mass would be applied. The amount of mass change is captured by the parameter *factor* that is used on the right-hand side to recalculate the mass of the object *e*. Finally, the new mass value becomes an output of this rule and might be subject to further operations. The combination of operations within rules, based on local variables, and these kind of parametric rules allow a very generic rule definition that provides a huge potential for realizing logical rule concatenations for capturing complex knowledge. The application of parametric rules in rule sequences will be further detailed in section 4.2.1.

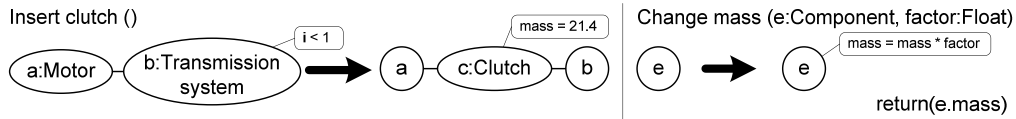

Figure 3. Problem-specific rules

Due to the meta-model, all modeling elements are known when the rules are formulated and the knowledge about these components can be used to create generic rules, Figure 4. Hence, all rules for simply adding nodes to the graph can be entirely derived out of the meta-model. For instance, the rule *Add clutch* has on its left-hand side the symbol of an empty set (∅) which signifies that nothing has to be matched. Consequently, an object of the type *Clutch* is directly inserted into the graph without edges to any other elements. In the other examples, *Add motor* and *Add transmission system*, the information from the meta-model about the node type's attributes (mass, max_torque, i) is used in order to define the corresponding rule parameters. Same as in the rule *Change mass*, local variables (here: *x*, *y*) are used to transfer the information from the rule parameters to the assignment of the node's attribute. Because of the *return(a)* statement, the created node becomes an output of the rule and can be used for further processing in the definition of the rule sequence (cf. section 4.2.1.). In the same manner, rules for adding a node and relating it directly to another node (e.g. *Add transmission system and relate*) or for converting between node types (*Convert to motor (electric)*) can be created automatically.

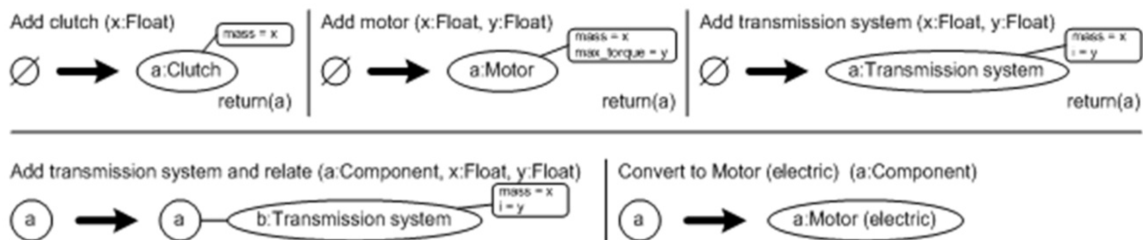

Figure 4. Generic rules

Based on a definition of compatibility, e.g. through ports in the meta-model, the automated generation can be even extended to rules that accomplish design-specific tasks, as for example a functional decomposition or the assignment of behaviors to functions [Helms 2009]. As graph transformation rules are able to model a plethora of possible operations on nodes and edges, they are used in the conceptual phase of Software Engineering [Göttler 1988] in order to model the functionality of a program and to eventually generate code automatically.

4.2 Application

While the definition contains the meta-model and the set of rules, those two components of a formal grammar come to life in the application, or dynamic part. The process of making the meta-model and the rule set executable is called compilation, Figure 1. At this point, the grammar operates with real data objects.

The easiest way to imagine what the object level signifies in programming is to imagine a human user working with standard office software (Figure 5): Usually, such applications are subdivided in different windows that display information of specific objects. The instantiation and modification of these objects is achieved by calling functions, e.g. by clicking on the icons on the menu bar. If, for

example, the icon "*Save as…*" is clicked, a new window object is instantiated and pops up in which the user can specify the path and the file name. Many programs allow the user to automate tasks by programming macros that control the handling of the object flow. The definition of rule sequences can be seen in exactly the same fashion, as they allow automating the instantiation of nodes and edges and, moreover, to apply graph transformation rules on them.

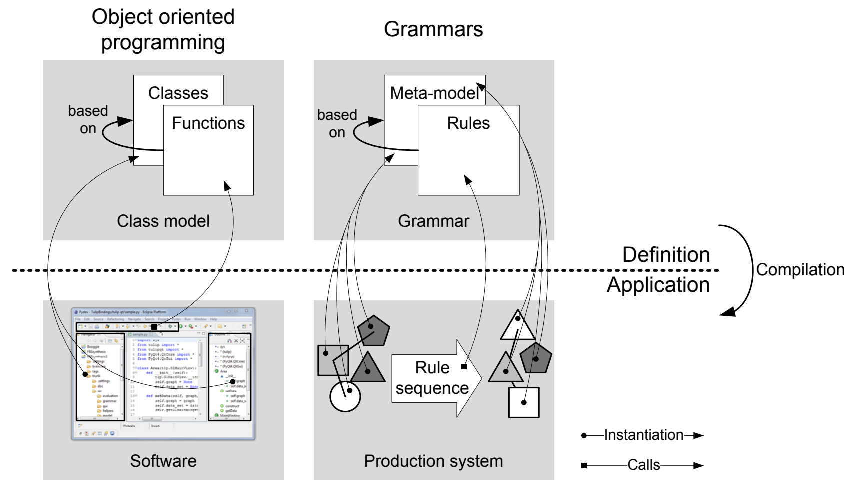

Figure 5. Interrelation between the definition and the application in object-oriented programming and grammars

4.2.1 Definition of rule sequences

```
1  xgrs motor_1 = add_motor(83.8, 48.2)
2  xgrs add_trans-syst_relate(motor_1, 29.4, 0.9) && insert_clutch(21.4)
3  xgrs change_mass(1.5)
4  new battery_1:EnergyStorage
   battery_1.mass = 12.9
5  xgrs convert_motor_el(motor_1) | relate(battery_1, motor_1)
```

Listing 1. Example rule sequence

The capability of formulating rule sequences goes far beyond the simple sequencing of rules and rather compares to the programming of macros. For this purpose, an already existing rule sequence language has been used [Geiß 2006], which is illustrated by the example in Listing 1 based on the previous examples of a meta-model and a rule set:

1. Firing rules is always initiated by the command xgrs (extended graph rewrite sequences). The rule add_motor is fired in order to create a new node of the type Motor. With the two rule parameters (that were defined as local variables x an y in the rule definition, fig. 2) the mass of the motor of 83.9 and its maximum torque of 48.2 (SI units are assumed) are defined. The return value of the rule, containing the new node, is stored in the variable motor_1.
2. The operator && allows to execute several rules consecutively whereas the execution of the latter depends on the successful execution of the former. That means in this case that the clutch is only inserted (rule: Insert clutch) in case the rule for adding a transmission system and relating it was successfully executed. The rule Add transmission system and relate accepts the previously created node motor_1, the mass (29.5) and the transmission ratio (0.9) as parameters.
3. Increasing the mass of all existing elements in the graph by a factor of 1.5 is achieved with the rule Change mass. Due to the * operator, this applies to all possible elements.
4. This rule sequence language also allows executing simple tasks, like adding or setting attributes manually without having to formulate rules. In this example the node battery_1 is created and a mass of 12.9 is assigned.
5. The simplest way of applying several rules consecutively is through the | operator that concatenates the rule application without considering whether the application was successful

or not. In this example, the node motor_1 is converted into the type Motor (electric) while preserving its attributes and afterwards related to the previously created node battery_1.

Due to the expressiveness of the rule sequence language, the execution of rules becomes quite similar to procedural programming. As simple values, graph elements and even (sub-)graphs, can be stored in variables and be further processed either by rules or directly within the rule sequence, complex, knowledge-intensive processes can be realized. For example, the dimensioning of machine parts could be subdivided in several rules that carry out distinct modifications, whereas the interplay of all those rules is realized one rule sequence level.

Because rule sequences are defined after the compilation, they contain the control directives for the graph transformation as plaintext. For this reason, they provide an ideal interface to automate the synthesis process as they could also be parsed by software. This provides a huge potential for realizing a global search process based on the application of a graph grammar. Current research by the authors focuses on how rule sequences can be created automatically and be embedded in a more global search process.

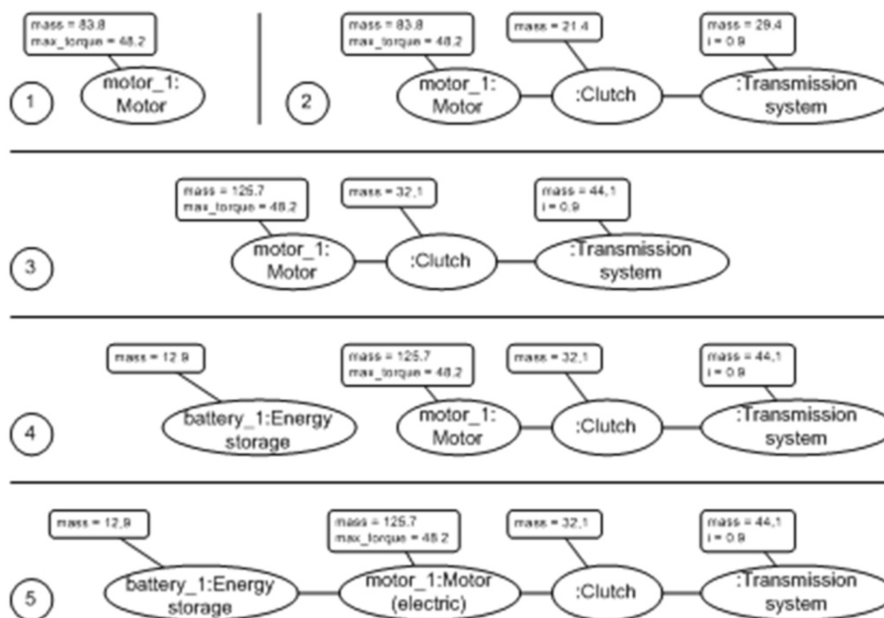

Figure 6. Resulting graph after each step of the rule sequence shown in Listing 1

4.2.2 Definition of an initial graph

From a formal point of view, a grammar transforms an initial graph (or initial symbol) into a resulting graph. Concerning the graph transformation, it makes no difference

1. if the initial graph is defined upfront in a different tool and imported before a transformation is applied
2. if an initial graph is defined in the rule sequence itself
3. if no initial graph is defined at all and the first rule has an empty left-hand side and starts creating a graph, like the rule *Add motor*.

However, the possibility to define a graph upfront and to apply different rule sequences on it or, vice-versa, to apply different rule sequences on one initial graph provides a strong foundation for carrying out what-if-studies or to synthesize design alternatives.

4.2.3 Graph transformation

This final step is carried out by an interpreter that interprets the directives in the rule sequence definition and executes – based on the meta-model and the rule set – the transformation. The graph transformation library that is used by the authors [Geiß 2006] provides additional features, such as a debug mode that highlights during the rule execution the sub-pattern in the actual graph that matches the left-hand side.

Figure 6 depicts the five states of the graph after each transformation. The nodes *:Transmission system* and *:Clutch* have been created by rules and not, like the other two elements, through the rule sequence. As the rules did not assign variable names to them the space before the colon is empty.
In a synthesis process, the resulting graph provides the basis for evaluating the quality of the design. In the authors work, the evaluation is divided in two categories. While quantitative methods estimate whether functional requirements are met, quantitative methods evaluate how well they are met.

5. Discussion

Approaches for computational design synthesis that aim to supporting the product development process in a general and efficient way often involve multiple levels of abstraction (e.g. function, behavior and structure) or different levels of granularity (e.g. components and modules). Knowledge representations based on graph-grammars are, because of their universality, very well suited to serve as a base representation. The pure use of labels and attributes, as in previous approaches, is too restricted for graph-grammar based implementations of sophisticated product development methods.
This paper contributes to the further development of methods for computational design synthesis by drawing an analogy to object-oriented programming and depicting means of integrating these concepts in graph-grammars. Quality factors for programming and their correspondence in design synthesis methods were introduced; references to these points were made throughout the paper.
The implementation of the twofold rationale in object-oriented programming of separating representation and execution into a static definition and a dynamic application part has been presented. Whereas the former defines the graph-grammar, thus vocabulary and rules, the latter contains the rule sequence definition in which real data objects are instantiated and manipulated by the execution of graph transformation rules.
The upfront definition of the modelling space by means of a hierarchical meta-model, contributes to the goal of *extendibility* by enabling the subsequent addition and detailing of elements in a structured way. Moreover, this kind of type hierarchy is intuitively understandable and increases the *ease of use*.
Due to the concept of inheritance, the definition of overlapping or redundant attributes or elements is abolished. Furthermore, the class taxonomy of the meta-model supports the automated creation of generic rules but also the definition of knowledge intensive rules that apply not only to one type of element but also to all of it subtypes. These features facilitate the encapsulation of engineering knowledge and support hence the goal of *efficiency*.
The separation of representation and program execution enhances the *reusability* of meta-models for multiple applications and domains. Furthermore, parts of the meta-model can be detached, further detailed and re-integrated.
The *compatibility* of different meta-models and rule sets can be assured as long as naming conventions, e.g. for functional modeling, are maintained. This comprises in particular rules that reason on different levels of granularity.
The rule sequence language provides the basis for controlling and manipulating the objects resulting in the synthesis of graph-based product models. Because of its expressiveness, the definition of rule sequences is very flexible and comparable to the automated control of software using macro programming. Due to the power of this language, an intuitive use is not possible because the user has to acquire a profound knowledge on the language constructs. For that reason, future work comprises the investigation of means to enable the user to interact with the graph transformation system in an intuitive manner based on a graphical user interface, i.e. an interpreter for engineering applications.
Furthermore, no global search process is implemented that controls the synthesis process. In the authors' opinion, the automated selection and execution of rule sequences, e.g. based on genetic algorithms, is a promising approach for future work that also requires coupling with quantitative evaluation methods. Therefore, the integration with simulation tools is in the scope of actual work.
The implementation of this research will be made available to the public in an open-source software framework called ***booggie*** (brings object-oriented graph-grammars into engineering) at http://www.booggie.org.

6. Conclusion

In this paper, an extension of methods for computational design synthesis is presented that is based on the application of graph grammars and the implementation of concepts from object-oriented programming. The benefits in the field of programming and their correspondence in design synthesis are presented and exemplified. The approach subdivides the computational design synthesis process in two parts: A static part that contains the definition of a grammar and a dynamic part that provides for the grammar application in order to solve design tasks. The main contributions of this separation are a higher flexibility in the definition of rules, an increased extendibility and comprehensibility of the set of building blocks and the ability to generate expressive rule sequences through scripting. Thereby, this research responds to the need for extension of the scope of application, increasing the efficiency and intelligence and providing more flexibility. Future work includes integrating intuitive user interfaces and extending the approach to incorporate qualitative and quantitative evaluation methods.

Acknowledgement

We thank the German Research Foundation (Deutsche Forschungsgemeinschaft – DFG) for funding this project as part of the collaborative research center 'Sonderforschungsbereich 768 – Managing cycles in innovation processes – Integrated development of product-service-systems based on technical products'.

Literature

Bolognini, F., Seshia, A.A., Shea, K., "Exploring the Application of Multidomain Simulation-based Computational Synthesis Methods in MEMS Design", Proceedings of the International Conference on Engineering Design, ICED '07, Paris, 2007.

Geiß, R., Batz, G., Grund, D., Hack, S., Szalkowski, A., " GrGen: A Fast SPO-Based Graph Rewriting Tool", Graph Transformations, Springer, Berlin, 2006.

Göttler, H., „Graphgrammatiken in der Softwaretechnik", Springer, Berlin, 1988.

Helms, B., Shea, K., Hoisl F., "A Framework for Computational Design Synthesis Based on Graph-Grammars and Function-Behavior-Structure", Proceedings of the ASME 2009 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference, San Diego, 2009.

Kerzhner, A.A., Paredis, C.J., "Using Domain Specific Languages to Capture Design Synthesis Knowledge for Model-Based Systems Engineering", Proceedings of the ASME 2009 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference, San Diego, 2009.

Kurtoglu, T., Campbell, M.I., "A Graph Grammar Based Framework for Automated Concept Generation", Proceedings of the International Design Conference – DESIGN 2006, Cavtat, Croatia, 2006.

Meyer, B., "Object-oriented software construction", Prentice Hall International, Englewood Cliffs, NJ, USA, 1997.

Schmidt, L.C., Shetty, H., Chase, S.C., "A Graph Grammar Approach for Structure Synthesis of Mechanisms," Journal of Mechanical Design, Vol. 122, 2000, pp. 371-376.

Starling, A.C., Shea, K., "A parallel grammar for simulation-driven mechanical design synthesis", Proceedings of the ASME 2005 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference, Long Beach, USA, 2005.

Wölkl, S., Shea, K., "A Computational Product Model for Conceptual Design Using SysML", Proceedings of the ASME 2009 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference, San Diego, 2009.

Bergen Helms
PhD student
Institute of Product Development
Technische Universität München
Boltzmannstr. 15, 85748 Garching, Germany
Telephone: +49-89-259 15136
Telefax: +49-89-259 15144
Email: helms@pe.mw.tum.de
URL: http://www.pe.mw.tum.de