

"43" - A Generic Approach for Engineering Design Grammars

Rolf Alber and Stephan Rudolph

Similarity Mechanics Group
Institute for Statics and Dynamics of Aerospace Structures
University of Stuttgart, Germany

alber@isd.uni-stuttgart.de and rudolph@isd.uni-stuttgart.de

Abstract

In the past grammars have already been successfully applied for computational synthesis in several technical domains. A drawback of most approaches however has been the fact that the grammar rules were formulated and implemented for a specific application domain. This strategy results in a lack of generalisation and restricts the use of the grammar compiler to the application domain it has been developed for. To overcome this limitation a generic approach for engineering design grammars is implemented into a software tool for engineering design named "43". The generic character of the „43“ engineering design compiler software is achieved by separating the formal design process into two steps: The formulation of graph grammar rules and the transfer from the generic graph layer to a domain specific material layer by a interpretation step which allows the subsequent generation of several final model descriptions which can be processed by common engineering analysis tools.

Introduction

This paper describes the conceptual background and implementation ideas of the software tool "43" which has been developed in order to facilitate evolutionary design synthesis in a variety of engineering application domains. The concept of "43" relies on a combination of evolutionary strategies together with a rule-based grammar formalism as underlying representation (Rudolph and Alber, 2002). As one main point it has been attempted to develop a generic representation form which is suitable for many different technical applications. This means that „43“ has not been tailored for one special application domain but is intended to work as a generic design tool for automated synthesis in engineering.

Motivation

To a certain level of abstraction modern computational design synthesis can mainly be classified into two basic categories or approaches:

- In the bottom-up approach it is attempted to achieve a higher level functionality by systematically combining

basic building blocks into assemblies. This is basically corresponding to the "design philosophy" as observed in nature where a sophisticated combination - in the sense of growth and self-organization - of simple basic blocks like molecules/cells into entities of extraordinary complexity can be found. Such an approach requires rather few effort for a symbolic formal representation of the design object and its evolution as it is necessary for a computational integration. This feature makes this technique easy to use for evolutionary strategies as already shown successfully in different publications (Bentley 1999). Although the combinatorial freedom supports the creation of new and creative designs, computational complexity is still an issue as combinatorial possibilities grow exponentially with the number of building blocks.

- In the top-down approach design synthesis is addressed from an opposite viewpoint. Beginning with the requirements definition the evolution of the design object is constituted out of subsequent decompositions from abstract conceptual descriptions into more detailed functional representations which finally find an embodiment into material components. In contrary to the first approach this attempt requires a huge amount of a priori knowledge about the design domain and about how requirements and abstract concepts can be fulfilled and decomposed into lower level functionality respectively material structures. As a consequence of the constraints imposed by this synthesis knowledge the possibilities to evolve completely new solutions are reduced. Therefore this approach can mostly not come up with the same amount of creative design solutions as the first approach described here. On the other hand and in contrary to the bottom-up approach a successful top-down decomposition intrinsically assures the fulfilment of the design requirements and thus shows a certain efficiency in a computational context. This approach basically resembles the way human engineers address design synthesis and has found different specific realizations in design theory (Pahl and Beitz 1996, O'Sullivan 2002).

Since both the bottom-up and the top-down approach dispose of their specific advantages, the implementation chosen in the design tool "43" is trying to at least partially bridge the gap between the two described approaches while at the same time providing a representational mechanism which is formal and simple enough to be easily integrated into evolutionary strategies. This means that it is attempted

to basically allow a bottom-up technique of combining building blocks while at the same time being able to incorporate a priori design knowledge in order to face combinatorial complexity. This also allows the combination of building blocks to happen on different and/or more or less abstract/specific levels respectively.

The concept applied in the approach of „43“ is based on the use of a generative design description which focuses on the representation of construction rules and principles to stepwise evolve a design instead of a direct representation of the final design itself. This technique for describing a design in the sense of generation instructions especially shows its advantages when it comes to finding a compact but still structured and formal representation of a design space as shown in detail later.

The design representation in “43” is therefore utilizing a rule-based design grammar formalism which shall be described in the following.

Generative representation with grammars

Grammars - originally an expression from the field of linguistics - play an important role when it comes to the definition of a set of data by providing specific structuring rules. Especially the ability to access a large set of elements by a rather dense description encoded in the underlying rules has made grammars an interesting technique in information processing which has led to the definition of a theory of formal languages and to the terminology of formal language grammars in computational science (Hopcroft et al. 2001).

Formal language grammars

A formal language grammar is a generative description which provides rules that can be applied to control the subsequent generation of a sentence. A set of possible rule-combinations - a so-called a production system - therefore represents the generative description of a syntactically correct sentence belonging to the language of that grammar.

From a computational point of view it is straightforward to generate all valid production systems defined over a grammar G . In this way the grammar formalism represents a means for accessing every sentence in the language $L(G)$ and can therefore be used as complete definition.

Such grammars can be found in several areas. Besides in natural languages grammars are especially used to define programming languages like C, Pascal, Fortran etc. (Aho et al. 1986). Special L-system grammars have been developed in biology to even simulate cell epigenesis and describe the branching topology in plants for a broad spectrum of species (Lindenmayer and Prusinkiewicz 1996).

In classical formal languages a sentence which has been derived by such a formal generation so far simply represents a set of syntactical placeholders. In order to give the structured symbol set - the sentence - a meaning, or in other words to add semantic content to the so far pure syntactical description a mapping is applied which assigns a unique

meaning to each symbol. This mapping is called an interpretation.

By carefully defining such an interpretation, grammars can be used to define entities which are far from what is normally associated to the term „language“ (either spoken language or programming language). This means grammars can be generally applied to entities which can be formally encoded in a symbolic representation. In this way grammars have been already successfully applied in different technical areas as for example in architecture and product design. Known applications can be found p.ex. for the design of shape patterns (Stiny 1980), in product-shape, function and cost estimates of coffee makers (Agarwal and Cagan 1998) as well as in mechanical design applications (Antonsson and Cagan 2001).

A major obstacle for a wider use of grammars in automated engineering design is in our view the fact that each engineering domain has its own traditions and requirements concerning the vocabulary and the syntax in which the rules can be easily formulated. This has led in the past to the development of several domain specific grammars, each especially tailored for the purposes of the specific application domain but therefore suffering a lack in generalization which made it hard or impossible to apply the developed grammar formalism and its computational implementation to other areas.

The still ongoing research effort reported here has therefore the objective to provide a generic grammar formalism for engineering design together with a ready-to-use implementation which has so far lead to the grammar compiler and engineering design tool „43“.

The basic idea in the approach is to remember that the classical technique in formal language theory makes a clear distinction between a pure syntactical structuring of a symbol set through the rules of the grammar and the following interpretation step which maps this symbol arrangement to the physical world. Following this 2-step sequence the idea is to apply a technique for the formal part which is common and well-suited for a manifold of design objects and then subsequently map the obtained results to the specific domain by an appropriate interpretation where further application specific post-processing can be undertaken. In this way the formal part and its implementation can remain as a common core for grammars in all different application areas (Alber, Rudolph and Kröplin 2002).

A common formal fundament for engineering design grammars

For the development of a generic design grammar it is crucial to find a formalism which serves two purposes:

First it is necessary to be easily approachable by a stepwise rule-based inference in order to allow a description of the *generative* procedure underlying the design object.

Second it is necessary to represent the information which is needed to directly define a design object in a way which makes afterwards the necessary mapping from the symbolic representation to a semantic processing of the description as easy and straightforward as possible. Since the focus of

the grammar is on the engineering design domain this results in the following requirements:

- representation of material entities, features, procedures, and abstraction concepts in the sense of qualitative and quantitative values,
- representation of relationships in the sense of ordering (hierarchical/sequential), common and similar elements, information exchange and interdependency.

Graph based representation. Considering the features listed above, graphs have been chosen as a well suited fundament for a formal representation which is able to fulfill the major needs for representing technical design objects. Graphs already serve as modeling formalisms in a manifold of engineering applications (Otter et al. 1996) in form of network graphs, petri nets, part-occurrence trees and class-diagrams in software engineering etc.

Furthermore, graphs are a common means for structured knowledge representation where with the field of graph grammars already techniques for a generative rule-based description similar to the techniques in formal languages have been invented (Nagl 1979). From a mathematical viewpoint a graph $G(N, E)$ consists of a set of nodes N and a set of relations $E \subseteq N \times N$ called edges, whereby the graph nodes as well as the edges can furthermore have a label assigned each. For our purpose two modifications which are also featured by some of the application examples given above are undertaken:

- each node shall be represented by a data structure containing besides its label a set of attributes in order to carry quantitative (i.e. numerical) as well as qualitative design information
- the relations between nodes are „moderated“ which means that the set of possible relations $N \times N$ is reduced and constrained to a set of 'meaningful' relations. For this purpose each node has to provide additional information about valid relations towards other nodes. This leads to the concept of ports, with each port representing a connection point with constraints for a relation that has to be matched. Thus the ports represent the interfaces which each node provides for the connections to other nodes.

The introduced changes from the general graph concept represent a trade-off between the need for a formal technique feasible for computational approaches as provided by graphs and the need for an easy to formulate interpretation when it comes to transforming the results to an engineering design model.

Fig. 1 shows an example for the graph formalism which has been chosen for the engineering design grammar. There the ports are symbolized by the smaller circles which in case a port is unused resides inside a node or in case it contributes to a relation is aligned to the respective edge.

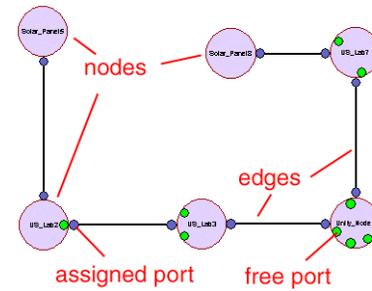


Figure 1: graph structure for design representation

Graph grammars. In an analogy to grammars for formal languages graph grammars are composed of a vocabulary V representing the building blocks of the design, a set of rules $R = \{r_1, r_2, \dots, r_n\}$ defining the assembly steps and transformations a design object can undergo, and a set of initial states $S = \{s_1, s_2, \dots, s_n\}$. The vocabulary $V = \{v_1, v_2, \dots, v_n\}$ consists of labeled and attributed nodes. An initial state or axiom can be any structured combination of the vocabulary elements $v_i \in V$, thus representing a graph itself. A specific axiom or initial state $s_i \in S$ together with an ordered set of rules $r_i \in R$ defines a production system and thus is the generative description of one specific graph which can be constructed following this program. During the execution of a production system the initial graph s_i is extended or modified by the graph rules, thereby evolving in several stages and forming the graph evolution sequence $\{G_0, G_1, \dots, G_n\}$ with $G_0 = s_i$.

The main difference between grammars for formal languages and graph grammars lies in the definition of the production rules which, due to the extended structuring capabilities of attributed graphs, are more complex to handle. On the other hand this additional complexity allows more freedom to express the necessary operations needed for constructing an engineering design object. In order to provide the necessary flexibility to structure attributed graphs, a graph rule has to provide several features for its conditional and its generative part.

The conditional part of the graph rule:

- requires the existence or non-existence of certain nodes and node topologies
- requires numerical conditions concerning the attributes of nodes

The generative part of the graph rule:

- creates new and/or removes existing nodes
- connects nodes by edges and/or removes existing edges
- modifies labels and attributes of existing nodes or edges

Furthermore the need to seamlessly fit in modified parts into the actual graph in each evolutionary stage does require additional embedding information for each rule and deserves special care when it comes to the application and execution of a graph operation.

In order to fulfill these requirements a rule syntax scheme similar to the X-scheme which is common in the field of graph grammars (Göttler 1988) is used. It does however feature some extensions which were necessary for this purpose and makes graph rules even slightly more intuitive to define. The syntax for a graph modification rule can be represented and explained by a geometrical arrangement of graph elements (nodes and edges) in a 2D plane which is divided into the 4 quadrants Q1...Q4 as illustrated in fig. 2.

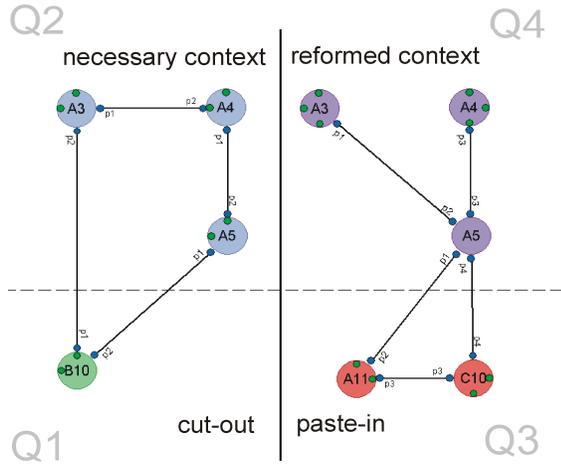


Figure 2: 4-quadrant rule-syntax scheme

The two quadrants Q1 and Q2 contain a graph G_{cond} which represents the conditional part of the graph rule. Prior to rule execution a search is performed to test if there exists any subgraph isomorphism between G_{cond} and the actual graph G_i . In case such an isomorphic subgraph $G_{match} \subseteq G_i$ can be identified, the generative part of the rule is executed and applies modifications on G_{match} . The modification of the graph is described by the elements contained in Q1, Q3, and Q4, thus, giving the elements in Q1 a role in the conditional and the generative part of the rule.

Nodes lying in Q1 mark deletions and will be removed from G_{match} together with all edges leading to these nodes. The graph nodes contained in Q2 and Q4 play the role of a context in order to specify the embedding of nodes which will be cut out of or pasted into G_{match} . Hereby the contents of Q2 and Q4 are identical in their nodes but not necessarily in their edges. In this way Q4 can be used to define a rearrangement in the connection topology of G_{match} by removing or adding edges between the identified nodes.

Concerning the attributes associated to each node, a rule can have additional features of which the definition can not directly be expressed in the above 4-quadrant scheme and therefore has to be described in a text-based formulation:

- Each node in G_{cond} can have additional constraints concerning the label and values of the attributes in that node, which have to be fulfilled by the corresponding nodes in order to be identified as an isomorphism. In this sense the term isomorphism denotes more than an isomorphism only in a topological sense.

- Attributes of newly pasted or already existing nodes can be set or modified in the generator part of the rule by directly supplying a numerical value or by definition through a mathematical equation (constraint) which expresses the dependency of an attribute on other attributes and which is evaluated during rule execution.

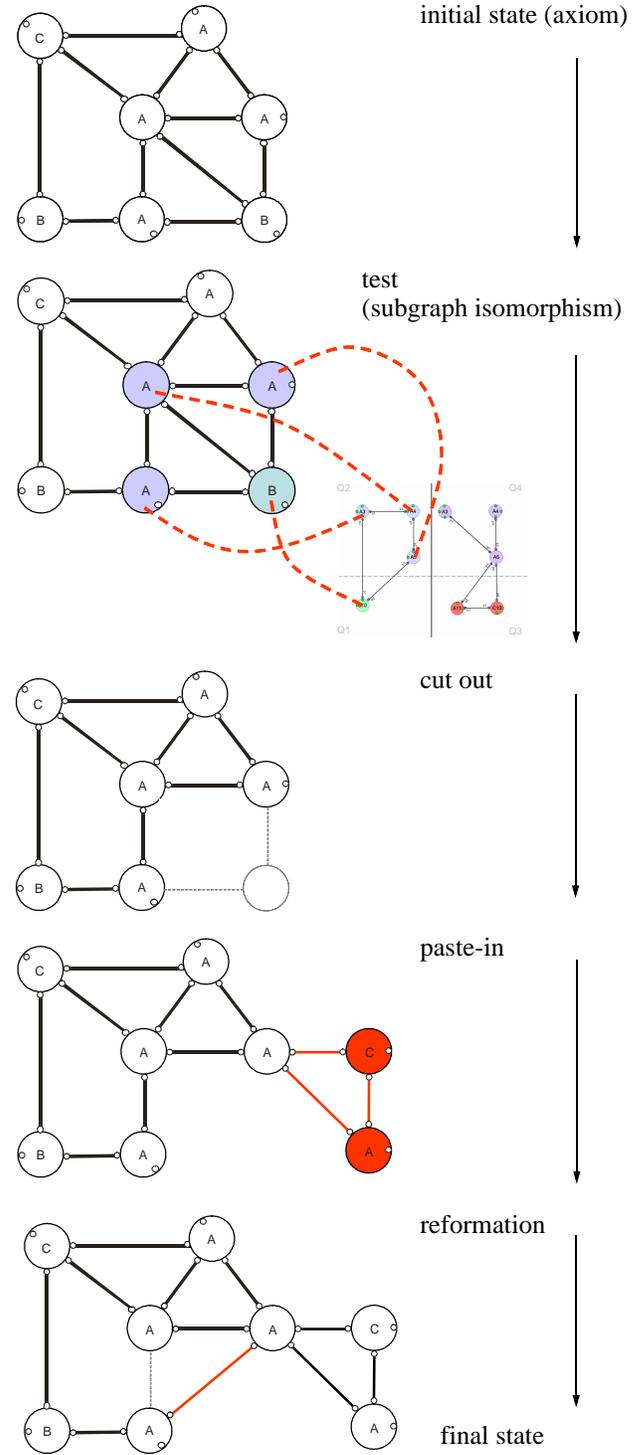


Figure 3: Application of the graph rule in fig. 2

Figure 3 shows exemplarily the four intermediate steps (test, cut-out, paste-in, reform) during the application of the graph rule from fig. 2 on an exemplary chosen initial graph.

Operational range of graph grammar rules

The formalism of graph substitution rules as used in this approach defines the operational range for expressing single construction steps during the synthesis of a design object. This operationability can be basically grouped into the 3 categories (addition/embedding, decomposition/refinement and modification/adaptation) which stand for different synthesis techniques which can be seen as originating of either the top-down or the bottom-up design approach as described in the introduction.

Addition / Embedding. Addition and embedding are the most basic operations which can be expressed by the graph-rule formalism. They correspond to the creation and integration of material parts/components in a bottom-up design approach. For a consistent integration of the new parts it is useful to apply the moderated connection via ports and edges between the corresponding graph nodes and at the same time adjust the parametric information in the attributes of the new or the neighboring nodes with respect to constraints associated with the rule explicitly.

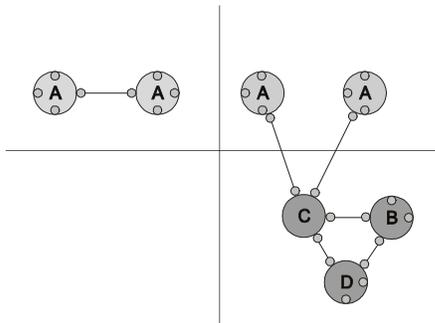


Figure 4: Graph rule example for addition and embedding

Fig. 4 shows an example for a rule which creates a graph structure (C,B,D) and embeds it between two existing Nodes (A,A) .

Decomposition / Refinement. This category of operations serves the purpose to resemble the subsequent splitting of higher level modules / categories into more detailed units / functions as used in top-down synthesis approaches. Concerning the rule-logic this category shows similar expressions as the “addition and embedding” category with the slight difference that the created and embedded nodes here do not correspond to material entities. Therefore the meaning of the graph rule is strongly dependent on the interpretation of the vocabulary elements involved.

Decomposition can be expressed either by connecting nodes which represent lower level concepts to the nodes representing the „Parent-Concept“ as illustrated in fig. 5 or by simply replacing the higher level nodes whereby the information about the decomposition relations gets lost.

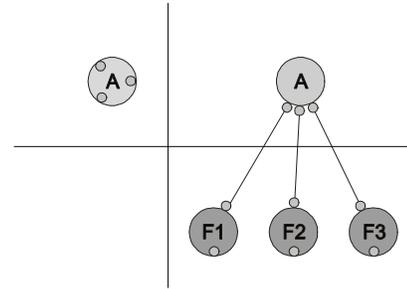


Figure 5: Graph rule example for refinement of „concept“ (A) into 3 „functions“ $(F1,F2,F3)$

Modification / Adaption. Operations belonging to this category are not intended to extend the graph model obtained so far but rather can perform necessary modifications concerning the attributes of nodes or relations among nodes. In this way the consistency of interdependent entities can be checked and, if necessary, be adapted in order to again fit into a context which might have changed during previous generation steps. Fig. 6 shows a graph rule which removes a node (D) and at the same time modifies the topological relations of the embedding graph structure (A,C,B) .

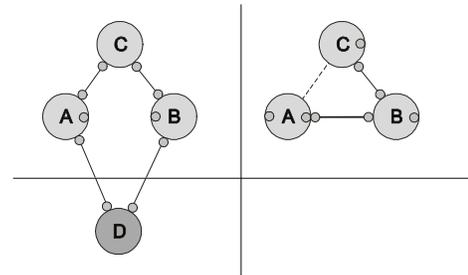


Figure 6: Graph rule example for modification

It should be noted that the classification of the rules into the 3 categories as shown above is only a virtual interpretation in order to show the connection to common expression categories from different synthesis approaches. This classification is however not directly reflected in the rule syntax which means that this distinction does not exist from a formal viewpoint. Therefore it is possible to express a graph rule containing elements of all the 3 categories at the same time which furthermore underlines the bridging between a top-down and a bottom-up approach as it has already been explained for the rule-based formalism within „43“.

Domain specific post processing

The graph derived by a production system is a data structure which applies relations to organize attributed symbols and represents the complete information about the desired engineering object. The transfer from the abstract graph-level to a physical design model which can undergo further steps for analysis and visualization makes an interpretation step necessary. This is described in the following.

This interpretation is domain specific and has to be designed individually for each area of application, whereas the graph grammar formalism is on an abstract level and thus remains common for all domains.

During the interpretation further model descriptions well suited for a post-processing with the respective domain specific tools are generated, based on the information encoded in the derived graph. In this way the design tool „43“ is intended to work as a front-end solely for synthesizing and organizing the necessary information which unequivocal describes a design. Any further steps like analysis and evaluation which especially in connection with evolutionary design play a crucial role, are performed by connecting this front-end to further third party analysis-tools through specific interfaces as illustrated in fig. 7.

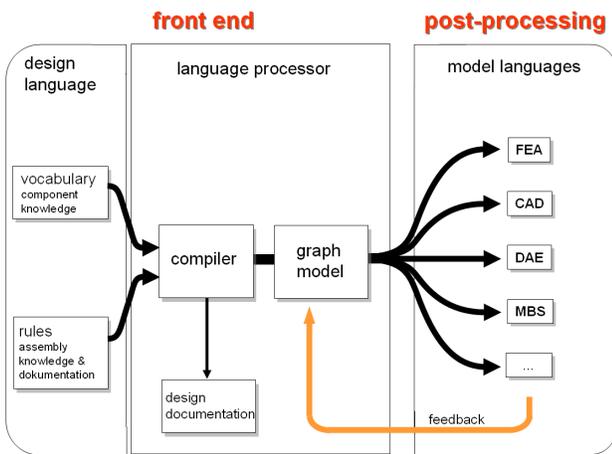


Figure 7: Process chain for a rule based synthesis

In the current state of the design tool „43“ interfaces to model descriptions for post-processing in CAD programs (like CATIA, Microstation) and formats (VRML), network equation solving / constraint processing / computer algebra tools (like Maple and Microsoft Excel) are supported.

It shall be noted at this point that it is not necessary and sometimes even not useful to apply a semantic interpretation for every node in the graph. In some cases it can be advantageous to have graph nodes representing abstract entities without structural counterparts or as placeholders to trigger the execution of other rules which perform further generations. This is especially common in decomposition steps like they are used in a top-down approach where abstract entities like requirements have to be expressed.

Evolutionary Design

In the following section the effort and possible influences of integrating the design tool „43“ as part of an evolutionary design system is outlined.

Evolutionary design transfers principles inspired from biological evolution to technical domains in order to provide a means for developing and optimizing technical solutions. The contribution which „43“ is intended to give to this area of design lies in the combination of an evolution-

ary strategy with a generative description through the use of the graph-grammar formalism. Due to the generic tailoring of the representation formalism this combined implementation is intended to become a ready-to-use tool for a manifold of different specific application domains.

In terms of evolutionary computation the genotype of our algorithm is represented by a production-system which in the sense of our grammar based design controls the generative steps during the synthesis of a design object. The thereby evolving graph represents the phenotype of our design object. After being interpreted and thereby transformed from a formal to a semantic model representation this phenotype can be analyzed and evaluated to provide a fitness value as needed in an evolutionary cycle.

Computational complexity

Concerning computational complexity the use of generative descriptions instead of a direct object description as genotype is expected to show several advantages.

The transformation from the production system to a graph model and furthermore to an analysis model carrying the semantics of a special domain as illustrated in fig. 8 is clearly deterministic - in a sense that the production system already carries the necessary information to unequivocal unfold the design object.

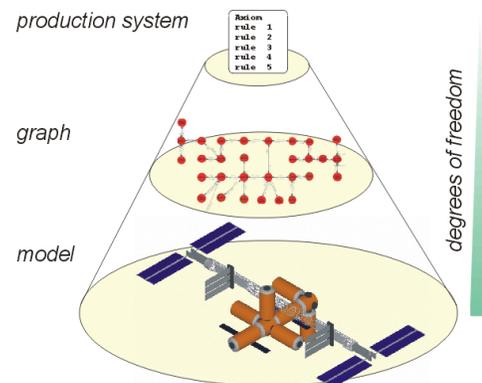


Figure 8: transformation of a generative to a semantic description (example of a space-station design grammar)

However, the number of possible degrees of freedom in fig. 8 which have to be defined for our design object increases steadily during this transformation. The therefore necessary information is provided by the rules respectively the interpretation which have been defined a priori in the grammar for one design domain. Knowledge which is common among individuals of the same design domain is not needed for uniquely identifying one specific individual. This means that such information has not to be provided in the genotype and therefore can be held in the background which leaves the production system to remain a very compact representation for our evolutionary cycle. Therefore one can await that the computational complexity which can be regarded as directly related to the number of degrees of freedom in the genotype description can be strongly reduced through the use of generative design descriptions.

Integration of expert knowledge

The ability to provide generation rules for the development of a design can be used to integrate external expert knowledge into the evolutionary process. Thereby this very generic exploration method can be individually adapted and constrained to the specific aspects of the design domain where it is intended to be used for. With the representation formalism in „43“ the influence of expert knowledge into the evolutionary cycle can happen in three ways:

- rules which only allow certain assembly steps under clearly defined conditions can be used to filter/restrict the combinatorial possibilities to such combinations which make sense from an a priori point of view. Here especially knowledge stemming from top-down design approaches can lead the evolutionary cycle to meaningful paths by considering requirements and by p.ex. providing the correct functional decomposition for the following development.
- As the outcome of a sequence of production steps is not only a result of the kind of rules which are triggered but also strongly depend on their order of appearance. It might be useful - again from an a priori viewpoint - to identify certain rule-*sequences* as meaningful. Those can be collected in sub-production systems which can then be called from the main production system which represents the genotype. Such a sub-production plays a comparable role to procedures in programming languages. In this way it is imaginable that one provides a kind of expert knowledge library containing several such procedures which can be used during the evolutionary cycle.
- Besides the a priori knowledge there might also exist certain a posteriori knowledge which can be utilized *after* the creation of a design object. This can be the case if we have knowledge about how certain aspects of a design can be modified in order to better match the design requirements. Rules or production systems which carry such knowledge can be regarded as „repair-rules“ respectively productions. In special cases it can also be possible that a low fitness of a design can already be detected directly from the topology and attributes of the model graph. Here the conditional part of a graph-rule can be used to detect such cases and as reaction skip the otherwise costly evaluation by directly setting the fitness for this individual to a low value. Such rules and productions which carry a posteriori knowledge could be packed in a special production system for post-processing which is not underlying any variations during the evolutionary cycle and which will be executed each time after the execution of the main genotype production system.

Summary

The basic conception behind the generative description as implemented in the design tool „43“ and its integration into an evolutionary engineering design strategy have been outlined. The feasibility for the application of this approach in engineering design has been recently validated with the

development of grammars for aircraft surface models, space-station configuration design and for systems design of satellites (Schäfer 2002). However, the complete integration of this approach into evolutionary strategies is still part of ongoing work. Further investigations will focus on the influence on the evolutionary cycle arising from the integration of expert knowledge as outlined in this paper.

References

- Aho, A., Sethi, R. and Ullman, J. 1986. Compilers - principles, techniques, and tools. Sydney: Addison-Wesley.
- Alber, R., Rudolph, S. and Kröplin, B. 2002. On Formal Languages in Design Generation and Evolution. Proc. 5th World Congress on Computational Mechanics (WCCM V), University of Vienna, Vienna, Austria, July 7-12th, 2002.
- Antonsson, K., and Cagan, J. eds. 2001. Formal Engineering Design Synthesis. Cambridge University Press.
- Bentley, P. 1999. Evolutionary design by computers. San Francisco, Calif. : Morgan Kaufmann Publ.
- Cagan, J. and Agarwal, M. 1998. A Blend of Different Tastes: the Language of Coffeemakers. Environment and Planning B: Planning and Design 25: 205-226.
- Göttler, H., 1988, Graphgrammatiken in der Softwaretechnik. Berlin: Springer Verlag.
- Hopcroft, E., Motwani, R. and Ullman, D. 2001. Introduction to Automata Theory, Languages, and Computation, Addison Wesley Longman
- Lindenmayer, A., and Prusinkiewicz, P. 1996. The Algorithmic Beauty of Plants. Berlin: Springer Verlag
- Nagl, M. 1979. Graph-Grammatiken. Braunschweig: Friedr. Vieweg & Sohn
- Otter, M., Elmquist, H. and Cellier F. 1996. Modeling of multibody systems with the object-oriented modeling language dymola. In Nonlinear Dynamics 9: 91-112. Dordrecht: Kluwer Academic Publishers
- O'Sullivan, B. 2002. Constraint Aided Conceptual Design. London: Professional Engineering Publishing Ltd.
- Pahl, G. and Beitz, W. 1996, Engineering Design: A Systematic Approach. London: Springer Verlag
- Rudolph, S. and Alber, R. 2002. An Evolutionary Approach to the Inverse Problem in Rule-Based Design Representations. Proc. 7th Intern. Conf. on Artificial Intelligence in Design (AID'02), Cambridge University, Cambridge, UK, July 15-17th, 2002, Kluwer Academic Press.
- Schäfer, J. 2002. Entwurfsgrammatiken im Systementwurf von Satelliten. Masters thesis, Institute for Statics and Dynamics of Aerospace Structures. Universität Stuttgart.
- Stiny, G. 1980. Introduction to Shape and Shape Grammars. Environment and Planning B: Planning and Design 7:343-351.