Integrated Design and Process Technology, **IDPT-2002**
Printed in the United States of America, June, 2002

Grammar-Based Design and Analysis of Technical Systems*

André Schulz
DaimlerChrysler AG
Division GSP/SDF
D-71063 Sindelfingen

Benno Stein
Paderborn University
Dept of Computer Science
D-33095 Paderborn, stein@upb.de

ABSTRACT: Research on modeling and design of technical systems aims at a comprehensive system specification from different viewpoints and at different levels of granularity. Modern CASE tools (based on UML, SDL, MSC) provide support for the modeling process, which relates to model transformation, coupling of tools, code generation, etc. However, the starting point is a clear understanding of the structure and behavior of the system.

The contribution of this paper focuses on a step before: the generation of structure models from a functional description. Technically speaking, we define the space of possible structure models amongst which a solution is to be searched. Starting with an incomplete structure specification, graph grammars are used for the formulation of domain-specific modifications rules as well as for the operationalization of the search.

The paper introduces the functional abstraction paradigm as a starting point for the use of graph grammars in design, and an application where structure model generation and behavior model simulation have been coupled to solve synthesis problems in chemical engineering.

I. Introduction

Modeling and design of technical systems can be regarded under two substantially different assumptions:

(1) The system is specified *explicitly*, i. e., there is a clear understanding of the structure and behavior of the desired system. (2) The system is specified *implicitly*, e. g. in the form of the desired demands or function. In the former case we have a modeling problem. In the latter case we have a configuration or design problem that is characterized by a search space wherein an optimum solution is to be found. The contributions of this paper relate to the latter case: the use of graph grammars to describe and to explore a space of models within the functional abstraction paradigm.

A model designates some purposeful abstraction of the system of interest. All models we are dealing with here can be specified in the form of a graph. The graph may represent an UML model, a petri net, a topological description, an algebraic equation system, etc. Generally speaking, we consider a model to be some kind of labeled graph.

Note that grammar-based approaches to modeling and design must not be seen as another formalism amongst the existing paradigms such as signal flow diagrams, UML, or SDL. Instead, grammars provide a means to handle several models at the same time, to describe model variants, and to modify or to evolve models. Figure 1 illustrates this view. It shows the problem of configuration and design as search in a model space atop the technical and software-based modeling paradigms; the lower part of the figure is based on [12].

Our research is in the tradition of existing approaches in the field of grammatical and geometric representation for engineering design, such as [31, 22, 23, 1]. Much of this research has focus on the representation of shape and the application of rules. However, the application presented in this paper is tied-up with the role of structure models in engineering design; this abstraction level has also been chosen by Schmidt et al. who developed a grammar-based approach to the structure synthesis of mechanisms [23].

II. Design Tasks and Graph Grammars

The design of a system encompasses a variety of different aspects or tasks, each of which may require operations of varying complexity such as

- the insertion and deletion of single items,
- the change of specific items and connection types, or
- the manipulation of sets of items, e. g. for repair or optimization purposes.

The operations delineated above can be viewed as transformations on labeled graphs; they are of the form *target* $\rightarrow$ *replacement*. A precise specification of such graph transformation rules can be given with graph grammars. A central concept in this connection is bound up with the notions of matching and context, which, in turn, ground on the concept of isomorphism [10].

Existing graph grammar approaches are powerful, but lack within two respects. Firstly, the notion of context is not used in a clear and consistent manner, which is also observed in Drewes et al. [6, Page 97]. Secondly, graph grammars are seldom applied to solve synthesis and analysis problems in technical domains. Actually, graph grammar solutions focus on software engineering problems for the most part [7, 8, 16, 18, 26].

*Supported by DFG grants PA 276/23-1 and KL 529/10-1.

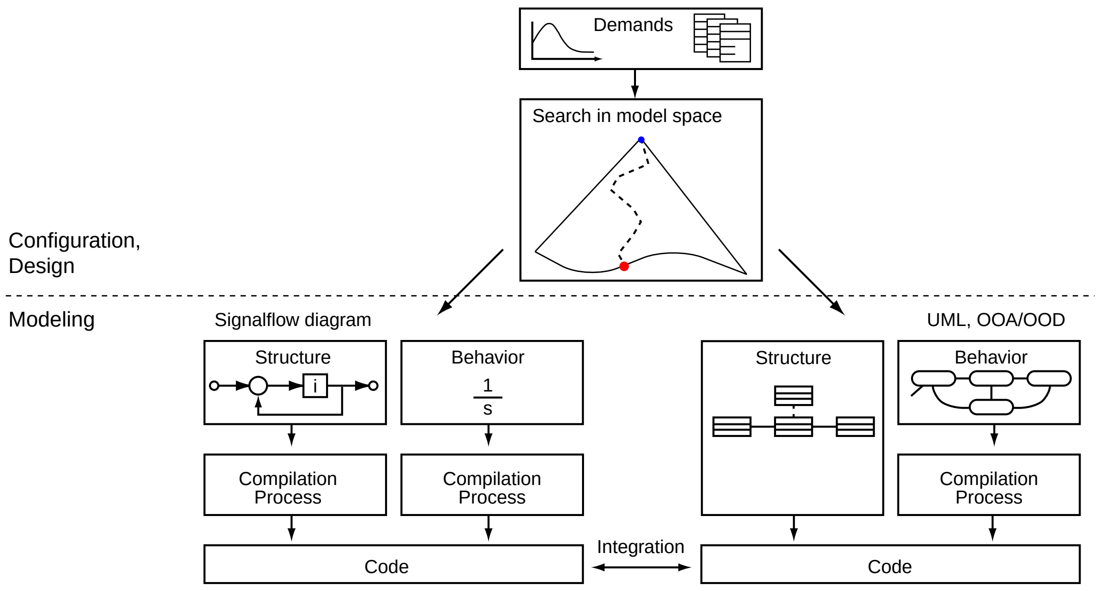

Fig. 1. Solving a design problem means to find a model in the space of possible models (upper part); grammars are a means to operationalize the search. The lower part of the figure shows the traditional modeling process including model coupling [12].

The systematics of design graph grammars introduced in [24] addresses some shortcomings. An important issue was to simplify and to enhance the use of grammars as a tool to define design knowledge in technical domains. Design graph grammars are closely related to node replacement graph grammars; among others, they allow for a straightforward specification of contexts, which is indispensable to describe the various kinds of manipulations at technical systems.

Figure 2 relates classical graph grammar terminology to typical design tasks. An analysis of the relationship between classical graph grammar families and design graph grammars is given in Section II-B.

A. Design Graph Grammars

A graph grammar is a collection of graph transformation rules each of which is equipped with a set of embedding instructions. What happens during a graph transformation is that a node t or a subgraph T in the original graph G, called the host graph, is replaced by a graph R. Say, R is embedded into G. The subsequent definition stems from [30].

Definition (Design Graph Grammar). A context-sensitive design graph grammar is a tuple $\mathcal{G} = \langle \Sigma, P, s \rangle$ where

- Σ is the label alphabet used for nodes and edges,
- P is the finite set of graph transformation rules and
- s is the initial symbol.

The graph transformation rules in P have the form $\langle T, C \rangle \rightarrow \langle R, I \rangle$ where

- $T = \langle V_T, E_T, \sigma_T \rangle$ is the target graph to be replaced,
- C is a supergraph of T, called the context,
- $R = \langle V_R, E_R, \sigma_R \rangle$ is the possibly empty replacement graph, and
- I is a set of embedding instructions that prescribe how R is connected to G.

Semantics of the embedding process: Firstly, a matching of the context C is searched within the host graph G. Secondly, an occurrence of T within the matching of C along with all incident edges is deleted. Thirdly, an isomorphic copy of R is connected to the host graph according to the embedding instructions $((h, t, e), (h, r, f)) \in I$ where

- $h \in \Sigma$ is a label of a node v in $G \setminus T$,
- $t \in \Sigma$ is a label of a node w in T,
- $e \in \Sigma$ is the edge label of $\{v, w\}$,
- $f \in \Sigma$ is another edge label not necessarily different from e, and
- $r \in V_R$ is a node in R.

If there is an edge labeled e connecting a node labeled h in $G \setminus T$ with a node labeled t in T, then a new edge with label f is created, connecting the node labeled h with the node r.

Remarks. (1) Labels in Σ can be used to specify node types, edge types, and variables. (2) Each graph is defined as a triple consisting of a node set, V, an edge set, E, and a labeling function σ that maps from Σ onto $V \cup E$. (3) The syntax of the rules in P and the instructions in I facilitate a uniform specification of different grammars types, such as node-based, graph-based, context-free, or context-dependent. E. g., a context-free rule is written as $T \rightarrow \langle R, I \rangle$; the graph T may consist of single node t; em-

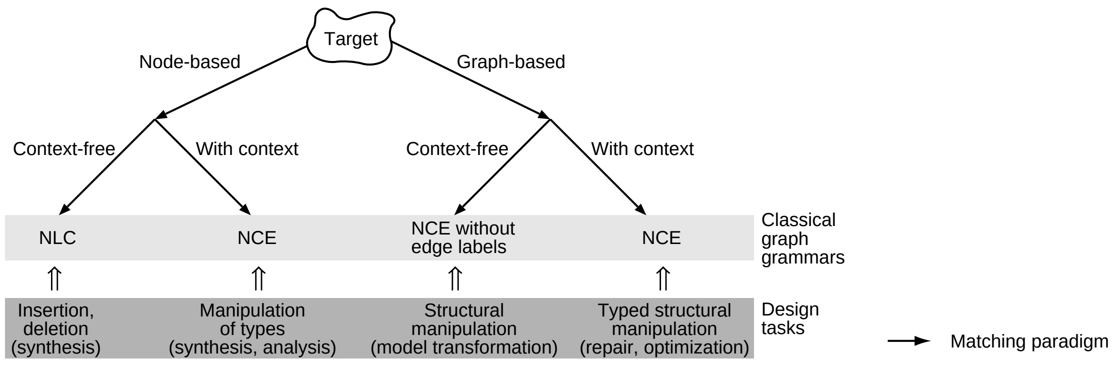
Target
Node-based
Graph-based
Context-free
With context
Context-free
With context
NLC
NCE
NCE without edge labels
NCE
Classical graph grammars
Insertion, deletion (synthesis)
Manipulation of types (synthesis, analysis)
Structural manipulation (model transformation)
Typed structural manipulation (repair, optimization)
Design tasks
Matching paradigm

Fig. 2. A graph grammar hierarchy for various design tasks. The abbreviations NLC and NCE denote the classical graph grammar families "node label controlled" and "neighborhood controlled embedding".

bedding instruction without edge labels may be abbreviated as $((h,t),(h,r))$.

The following example defines a graph transformation rule that operationalizes repair knowledge in fluidic engineering: The insertion of a by-pass throttle, tv, to improve an insufficient damping (see Figure 3). The formal specification of the rule $\langle T, C\rangle \rightarrow \langle R, I\rangle$ is as follows.

$$\begin{aligned}
T &= \langle V_T, E_T, \sigma_T\rangle = \langle\{1,2\},\ \ \{\{\}\},\ \ \{(1,p),(2,p)\}\rangle\\
C &= \langle V_C, E_C, \sigma_C\rangle = \langle\{3,4,5,6\},\ \ \{\{3,4\},\{3,5\},\{4,6\},\{5,6\}\},\\
&\qquad \{(3,w),(4,p),(5,p),(6,cv)\}\rangle\\
R &= \langle V_R, E_R, \sigma_R\rangle = \langle\{7,8,9,10,11,12,13\},\ \ \{\{7,9\},\{8,11\},\\
&\qquad \{9,10\},\{10,11\},\{9,12\},\{11,13\}\}\\
&\qquad \{(7,pw),(8,pw),(9,tri),(10,tv),\\
&\qquad (11,tri),(12,p),(13,p)\}\rangle\\
I &= \{((w,p),(w,pw)),\ \ ((cv,p),(cv,p))\}
\end{aligned}$$

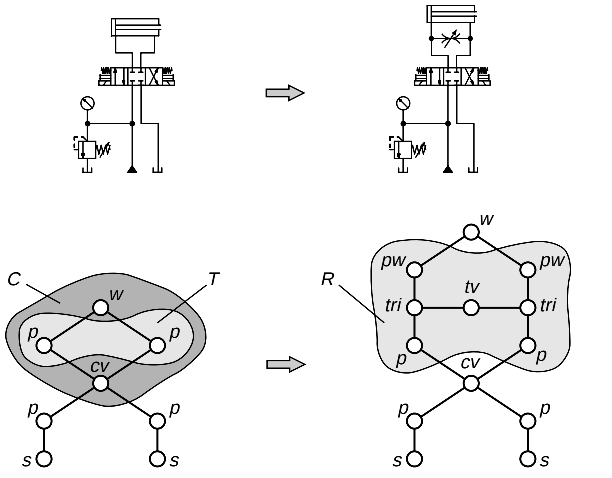

Fig. 3. Application of the graph grammar definition at a fluidic circuit where a by-pass throttle is inserted (top). At the bottom of the figure the related graph transformation rule is illustrated.

B. Relation to Classical Graph Grammars

The world of graph grammars is divided into two inherently different approaches: The connecting approach and the gluing approach [see 21, pp. 3-4]. The connecting approach is a node-centered concept that has given rise to numerous graph grammars usually called node replacement graph grammars. The gluing approach, on the other hand, is a hyperedge-centered approach on which various hypergraph grammars are based.

The most prominent representative of the connecting approach is the neighborhood controlled embedding (NCE) graph grammar family[1] [9]. NCE graph grammars perform graph transformations on labeled graphs. A graph transformation step is based on node and edge labels, which are used both to increase the discerning power and as some sort of simple context. Each graph transformation rule also has a set of instructions that dictate the embedding of the transformation.

Hyperedge replacement (HR) grammars represent the most popular grammar family following the gluing approach. HR grammars replace hyperedges, which are identified through labels, by hypergraphs. Embedding is realized by identifying so-called attachment nodes of hyperedges with external nodes of a host hypergraph.

Apart from the above described grammars there also exist hybrid approaches that try to combine the features of the connecting and the gluing approach. A widely known approach is the handle hypergraph (HH) grammar [5, 21], which rewrites handles, e. g., hyperedges together with their attachment nodes. The embedding is performed according to the connecting approach. Other hybrid approaches work similarly [cf. 13, 14].

Figure 4 gives an overview of the different graph grammar families and their concepts. Design graph grammars are direction preserving grammars. Obviously, they are closely related to NCE grammars of the connecting approach and possess similar theoretical properties [cf. 24].

[1] In the literature it is often distinguished between NCE, eNCE, dNCE, and edNCE graph grammars. For the sake of simplicity, we refrain from doing so here.

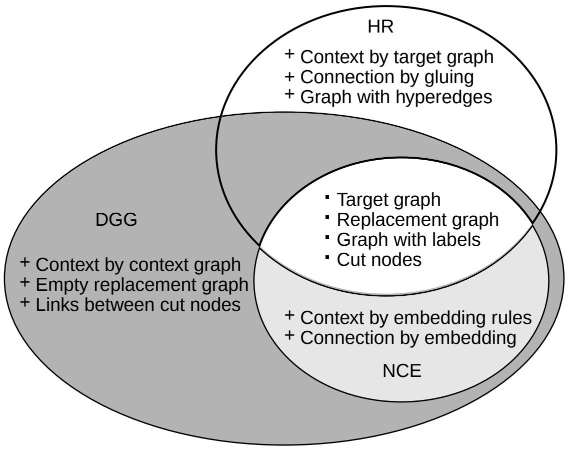
HR
+ Context by target graph
+ Connection by gluing
+ Graph with hyperedges
DGG
+ Context by context graph
+ Empty replacement graph
+ Links between cut nodes
· Target graph
· Replacement graph
· Graph with labels
· Cut nodes
+ Context by embedding rules
+ Connection by embedding
NCE

Fig. 4. Relation of design graph grammars (DGG) to classical grammars with respect to underlying concepts.

C. Relation to Programmed Graph Replacement Systems

When comparing design graph grammars to programmed graph replacement systems (PGRS) one should keep in mind that the former is located at the conceptual level while the latter emphasizes the tool character. PGRS are centered around a complex language allowing for different programming approaches. PROGRES, for instance, offers declarative and procedural elements for data flow oriented, object oriented, rule based and imperative programming styles [27, 25, 28]. A direct comparison between PROGRES and the concept of design graph grammars is of restricted use only and must stay at the level of abstract graph transformation mechanisms.

However, it is useful to relate the concepts of design graph grammars to PGRS under the viewpoint of operationalization. PGRS are a means to realize a design graph grammar by reproducing its concepts.

III. Grammar-Based Design in Chemical Engineering

“Grammar-based design systems have the potential to both automate the design process and allow greater exploration of design alternatives.”

Scott C. Chase [4]

This section outlines a grammar-based approach to conceptual design in chemical engineering, which developed from a cooperative project with the Chemical Engineering Group at Paderborn University. The approach may be applied to various kinds of chemical design problems, but by now we focus on a particular type of chemical processes only, the design of plants for the food processing industry [17].

A chemical plant can be viewed as a graph where the nodes represent the devices, or unit-operations, while the edges correspond to the pipes responsible for the material flow. Typical unit-operations are mixing (homogenization, emulsification, suspension, aeration etc.), heat transfer, and flow transport. The task of designing a chemical plant is defined by the given demands D, in the form of properties of various input substances, along with the desired output substance. The goal is to mix or to transform the input substances in such a way that the resulting output substance meets the imposed requirements.

The design happens by passing through (and possibly repeating) the following five steps: Preliminary examination of the demands, choice of unit-operations, structure definition, component configuration, and optimization. An automation of the steps at a behavioral level would be very expensive—if possible at all. Present systems limit design support to isolated subjobs; they relieve the human designer from special simulation or configuration tasks, and the effort involved there is high enough [3, 15].

A. Underlying Paradigm

Our primary concern was the investigation of possibilities to support the design process as a whole, with the long-term objective to operationalize step-spanning optimization knowledge. The result of our research can be comprised as a four step approach to design (cf. Figure 5): (1) The properties of the input and output substances, D, are abstracted towards linguistic variables, $\widehat{D}$. (2) At this functional level a structure model S is synthesized that fulfills $\widehat{D}$ and that is used as a solution candidate for D. (3) S is completed towards a tentative behavior model B, (4) which is then repaired, adapted, and optimized.

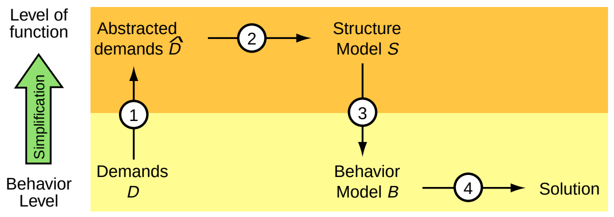

Fig. 5. The paradigm of functional abstraction in design problem solving [29].

The procedure in Figure 5 resembles the paradigm of functional abstraction [29]. The key idea of this paradigm is a systematic construction of candidate solutions within a very simplified design space, which typically is some structure model space. Design by functional abstraction makes heuristic simplifications at least at two places: The original demand specification is simplified (Step 1), and, the structure model is transformed locally into a behavior model (Step 3). Note that this paradigm enables us to automate the entire design procedure. Clearly, the mentioned steps will not be performed at the highest level of behavioral and structural details:

(1) Manual abstraction of the demands.

(2) Structure model synthesis by design graph grammars.

(3) Enhancement of the structure model into a behavior

model.

(4) Behavior model optimization based on the evaluation of simulation results.

Figure 6 reflects the described steps with respect to the design of food processing plants. The next subsection, III-B, deepens the grammar-based operationalization of Step 2.

The concept of separating the structure model from the behavior model for synthesis purposes is not new. The VEDA/MODKIT system, for instance, follows the same philosophy and treats chemical plants as systems consisting of basic building blocks and connections [19]. However, in VEDA/MODKIT the synthesis of a plant flow chart is performed manually by a human designer.

B. Structure Model Synthesis

The synthesis of a structure model (cf. Step 2 in Figures 5 and 6 respectively) is a creative job which is efficiently solved by human experts, who devise solutions based on their extensive knowledge and experience. However, they are not capable of generating and checking all possible solutions to a given task systematically—the probability of missing the optimum solution is considerable. This gap can be closed by automated design with, for instance, graph grammars. Graph grammars allow for knowledge modeling, manipulation, and systematic search of the solution space, which are essential requirements for a successful synthesis of structure models [22].

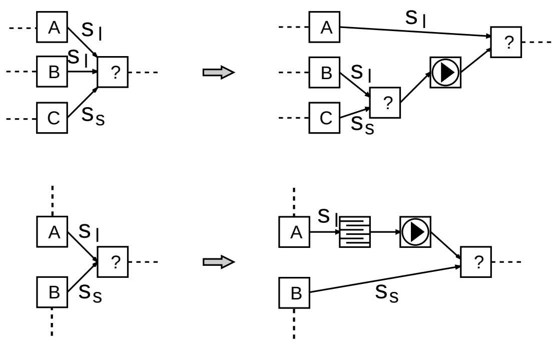

Fig. 7. Synthesis rules: splitting of mixing jobs (top), insertion of a heating chain to improve dissolution (bottom).

Graph grammars generate graphs by applying transformation rules on, initially, some start symbol or graph. Here, the start graph consists of a “?”-node representing the unknown chemical plant where the abstracted demands in the form of input and output substance properties are connected to. The successive application of transformation rules corresponds to the application of domain knowledge with the goal of refining the “?”-node into a concrete design fulfilling all demands. Note that domain knowledge also applies to other tasks, such as analysis or optimization.

The Figures 7, 8, and 9 illustrate the use of graph grammars in design. The figures show graphical representations of graph transformation rules $\langle T, C\rangle \rightarrow \langle R, I\rangle$ from our chemical engineering knowledge base. The rules, which have been developed in a close cooperation with domain experts, are used for synthesis, refinement, and optimization purposes when designing plants for food processing.

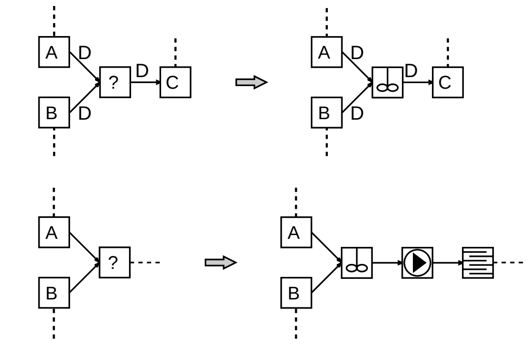

Fig. 8. Refinement rules: Refinement of a generic mixer into a propeller mixer (top), refinement of a generic mixer into a propeller mixer with trailing heating chain (bottom).

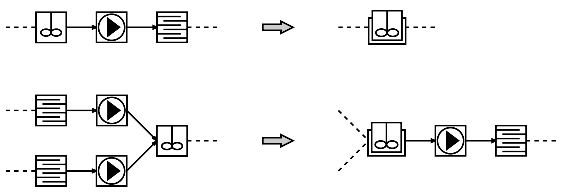
Fig. 9. Cost reduction rules: Substitution of a heating chain with a mixer with built-in heat transfer (top), combination of two identical partial chains by relocation (bottom).

Graph grammars can be seen as a collection of rules that define some search space. Note, however, that they cannot provide a fitting search mechanism to *navigate* within this search space; each domain requires a dedicated search method exploiting domain knowledge. Thus, the synthesis algorithm integrating domain-specific search (function SELECT-RULE) can be written as follows.

SYNTHESIS-STEP
Input: A design graph grammar $\mathcal{G}$ and an initial graph G.
Output: A graph consisting of terminal nodes or **fail**.

```
(1)  SYNTHESIS-STEP(DGG $\mathcal{G}$, GRAPH $G$) {
(2)   **if** TERMINAL-P($G$) **then** res := $G$;
(3)   **else** {
(4)     res := **fail**;
(5)     rules := CHECK-MATCHINGS($\mathcal{G}$, $G$);
(6)     **while** (rules $\neq$ {} **and** res = **fail**) {
(7)       rule := SELECT-RULE(rules, $G$);
(8)       rules := rules \ {rule};
(9)       res := SYNTHESIS-STEP($\mathcal{G}$, APPLY-RULE(rule, $G$));
(10)     }
(11)   }
(12)   **return** res;
(13) }
```

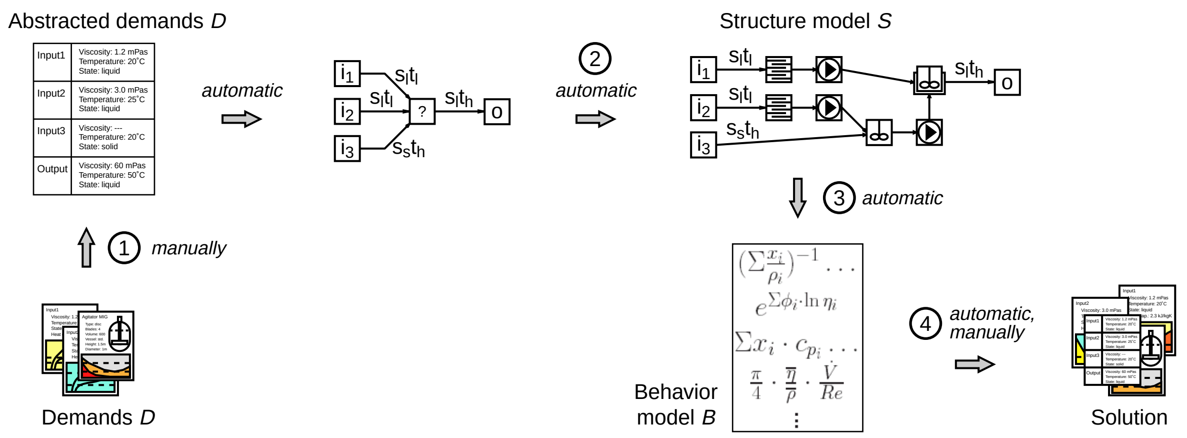
Abstracted demands D
Input1 Viscosity: 1.2 mPas Temperature: 20°C State: liquid
Input2 Viscosity: 3.0 mPas Temperature: 25°C State: liquid
Input3 Viscosity: --- Temperature: 20°C State: solid
Output Viscosity: 60 mPas Temperature: 50°C State: liquid
automatic
i_1 i_2 i_3 $s_l t_l$ $s_l t_l$ $s_s t_h$? $s_l t_h$ o
2 automatic
Structure model S
3 automatic
1 manually
Demands D
Behavior model B
$(\sum \frac{x_i}{\rho_i})^{-1} \ldots$
$e^{\sum \phi_i \cdot \ln \eta_i}$
$\sum x_i \cdot c_{p_i} \ldots$
$\frac{\pi}{4} \cdot \frac{\overline{\eta}}{\overline{\rho}} \cdot \frac{\dot{V}}{Re}$
4 automatic, manually
Solution

Fig. 6. Design of food processing plants according to the functional abstraction paradigm: The properties of the input and output substances are abstracted (Step 1), a structure model candidate S is generated by graph grammars (Step 2), S is completed towards a behavior model (Step 3), which then is repaired or adapted (Step 4).

As observed by Chase [4], "*a well-defined grammar will generate a set of designs which adhere to a specific set of user defined constraints*".

Put in other words, the quality of a generated solution is strongly dependent of the granularity and quality of the knowledge applied, which is reflected by the search space spanned by the graph grammar as shown in Figure 10. Furthermore, the imposed simplifications also influence the attainable solution quality, and, solutions found by the search process will often be suboptimum.

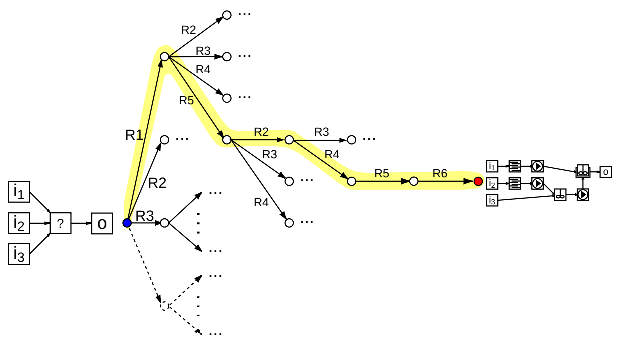

Fig. 10. Heuristic search space navigation using domain knowledge.

For synthesis tasks in chemical engineering the concepts described above were and are being implemented within a prototypical tool, called DIMOD (domain-independent modeler). Figure 11 shows a screen snapshot. The core of the system consists of a generic graph grammar engine, used for modeling and application of knowledge, and a domain-specific module used to guide the search process.

Within DIMOD the simulation is realized as follows. A generated structure model is completed towards a tentative behavior model by attaching behavior model descriptions to the components of the structure model (cf. Step 3 in Figure 6). The behavior model is then validated by simulation. For this purpose, the ASCEND IV simulator is used [20]; the attached model descriptions stem from the ASCEND IV model library or from custom models.

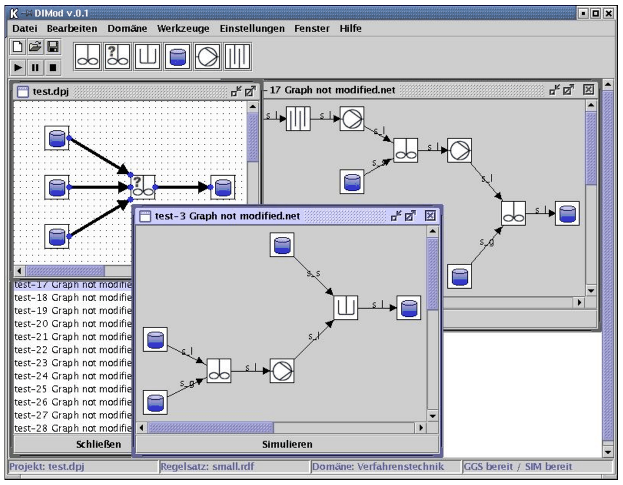
DIMod v.0.1
Datei Bearbeiten Domäne Werkzeuge Einstellungen Fenster Hilfe
test.dpj
17 Graph not modified.net
test-3 Graph not modified.net
test-17 Graph not modifie
test-18 Graph not modifie
test-19 Graph not modifie
test-20 Graph not modifie
test-21 Graph not modifie
test-22 Graph not modifie
test-23 Graph not modifie
test-24 Graph not modifie
test-25 Graph not modifie
test-26 Graph not modifie
test-27 Graph not modifie
test-28 Graph not modifie
Schließen
Simulieren
Projekt: test.dpj
Regelsatz: small.rdf
Domäne: Verfahrenstechnik
GGS bereit / SIM bereit

Fig. 11. The DIMOD system. Upper left window represents the abstracted demands, the windows to the right and center show two generated structure models.

IV. GRAMMAR-BASED ANALYSIS

The use of graph grammars is not restricted to synthesis tasks: Analysis tasks, such as structural feasibility tests, or model transformation tasks, such as the conversion of models, are also conceivable.

Structural feasibility means that a structure model is consistent with the design knowledge encoded within a given design graph grammar. This is equivalent to saying—in terms of formal languages—that a structure model is a valid "word" of a design graph grammar language. The following algorithm yields a statement concerning the membership of a structure model to a given design graph grammar.

ANALYSIS-STEP
Input: An inverted design graph grammar $\mathcal{G}$ and a graph G.
Output: **true**, if the initial symbol could be derived, otherwise the resulting graph is returned.

```
(1)  ANALYSIS-STEP(DGG $\mathcal{G}$, GRAPH $G$) {
(2)   if INITIAL-SYMBOL-P($G$) then res := true;
(3)   else {
(4)     res := false;
(5)     rules := CHECK-MATCHINGS($\mathcal{G}$, $G$);
(6)     while (rules $\neq$ {} and res = false) {
(7)       rule := SELECT-RULE(rules, $G$);
(8)       rules := rules \ {rule};
(9)       res := ANALYSIS-STEP($\mathcal{G}$, APPLY-RULE(rule, $G$));
(10)    }
(11)  }
(12)  return res;
(13) }
```

Remarks. Context-free graph grammars containing rules with empty right-hand sides cannot be used for structural feasibility tests, since the rules of the graph grammar have to be inverted. However, this drawback can be avoided by adding a context to the rules with empty right-hand sides.

Note that a graph-based analysis means to solve the membership problem for a given graph G and a graph grammar $\mathcal{G}$. This job requires to find a derivation of G from the start symbol s; a derivation in turn is based on the application of graph transformation rules defined in $\mathcal{G}$; and, to fire a rule it is necessary that a matching of the left-hand side be found within the host graph. In the general case, the membership problem is PSPACE-complete [2]. However, special properties or concepts such as confluence, boundary, leftmost, precedence graph, and flowgraph may decisively reduce the complexity of the graph membership problem [21, 11, 18].

V. Conclusion

Grammars provide a means to handle several models at the same time, to modify models, or to evolve models—say, they can be used to define a model space as well as to operationalize the search within this space. Design graph grammars, as proposed in this paper, can help to encode design knowledge in complex engineering tasks: They have been created as an instrument to formulate very different kinds of structure knowledge while still providing a well-defined semantics.

In our working group there is a long tradition in solving design and configuration tasks in engineering domains. The design graph grammar approach introduced here is a result of the analysis of several projects involving structure model manipulations. The paper in hand delineated a project from the chemical engineering domain. Note that our primary concern is to get a grip on search spaces: Current research concentrates on the automatic derivation of heuristics that guide the search in sophisticated design situations. Other design graph grammar applications, theoretical results, as well as an in-depth comparison respecting classical graph grammars can be found in [24, 29].

References

[1] M. Agarwal and J. Cagan. On the use of Shape Grammars as Expert Systems for Geometry-Based Engineering Design. *Artificial Intelligence in Engineering Design, Analysis and Manufacturing*, 14:431–439, 2001.

[2] Franz J. Brandenburg. On the Complexity of the Membership Problem of Graph Grammars. In Manfred Nagl and Jürgen Perl, editors, *Graphtheoretic Concepts in Computer Science*, pages 40–49, Linz, 1983. Trauner.

[3] Axel Brinkop, Norbert Laudwein, and Rüdiger Maassen. Routine Design for Mechanical Engineering. In *Proceedings of the Sixth Annual Conference on Innovative Applications of AI (IAAI 94)*, Seattle, August 1994.

[4] Scott C. Chase. User Interaction Models for Grammar-Based Design Systems. In *Proceedings of the Online Conference Design Computing on the Net, DCNet '98*, Sydney, December 1999.

[5] B. Courcelle, J. Engelfriet, and G. Rozenberg. Handle-rewriting Hypergraph Grammars. *Journal of Computer and System Sciences*, 46:218–270, 1993.

[6] Frank Drewes, Hans-Jörg Kreowski, and Annegret Habel. Hyperedge Replacement Graph Grammars. In Grzegorz Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation*, pages 95–162. World Scientific, Singapore, 1997.

[7] Marko van Eekelen, Sjaak Smetsers, and Rinus Plasmeijer. Graph Rewriting Systems for Functional Programming Languages. Technical report, Computing Science Institute, University of Nijmegen, 1998.

[8] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2 Applications, Languages, and Tools. World Scientific, 1999.

[9] J. Engelfriet. Context-free NCE Graph Grammars. In J. Csirik, J. Demetrovics, and F. Gécseg, editors, *Proc. Fundamentals of Computation Theory*, volume 380 of *Lecture Notes in Computer Science*, pages 148–161. Springer-Verlag, 1989.

[10] Dieter Jungnickel. *Graphen, Netzwerke und Algorithmen*. BI Wissenschaftsverlag, Wien, 1990.

[11] Manfred Kaul. *Syntaxanalyse von Graphen bei Präzedenz-Graph-Grammatiken*. Dissertation, Department of Mathematics and Computer Science, University of Passau, Passau, 1986.

[12] Markus Kühl, Clemens Reichmann, Bernhard Spitzer, and Klaus D. Müller-Glaser. Eine durchgehende Entwurfsmethodik für das Rapid Prototyping von eingebetteten Systemen. *Workshop Modelltransformation und Werkzeugkopplung*, 2001.

[13] Changwook Kim and Tae Eui Jeong. HRNCE Grammars – A Hypergraph Generating System with an eNCE Way of Rewriting. In J. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, *Graph Grammars and Their Application to Computer Science*, number 1073 in Lecture Notes in Computer Science, pages 383–396, Berlin, 1996. Springer-Verlag.

[14] Renate Klempien-Hinrichs. Node Replacement in Hypergraphs: Simulation of Hyperedge Replacement and Decidability of Confluence. In J. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, *Graph Grammars and Their Application to Computer Science*, number 1073 in Lecture Notes in Computer Science, pages 397–411, Berlin, 1996. Springer-Verlag.

[15] Achim Knoch and Michael Bottlinger. Expertensysteme in der Verfahrenstechnik – Konfiguration von Rührapparaten. *Chem.-Ing.-Tech.*, 65(7):802–809, 1993.

[16] Martin Korff. Application of Graph Grammars to Rule-Based Systems. In Hartmut Ehrig, editor, *Graph Grammars and Their Application to Computer Science*, number 532 in Lecture Notes in Computer Science, LNCS, pages 505–519, Berlin Heidelberg New York, 1991. Springer.

[17] Annett Kurzok, Manfred H. Pahl, and André Schulz. Software zur wissensbasierten Prozeßmodellierung – WIP. *Chemie Ingenieur Technik*, 73(9), 2001.

[18] Ulrike Lichtblau. Recognizing Rooted Context-Free Flowgraph Languages in Polynomial Time. In Hartmut Ehrig, editor, *Graph Grammars and Their Application to Computer Science*, number 532 in Lecture Notes in Computer Science, LNCS, pages 538–548, Berlin Heidelberg New York, 1991. Springer.

[19] Ralf Molitor. *Unterstützung der Modellierung verfahrenstechnischer Prozesse durch Nicht-Standardinferenzen in Beschreibungslogiken*. PhD thesis, Rheinisch-Westfälische Technische Hochschule Aachen, 2000.

[20] P. C. Piela, T. G. Epperly, K. M. Westerberg, and A. W. Westerberg. ASCEND: An Object-Oriented Computer Environment for Modeling and Analysis: The Modeling Language. *Computers Chemical Engineering*, 15(1):53–72, 1991.

[21] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1 Foundations. World Scientific, Singapore, 1997.

[22] L. C. Schmidt and J. Cagan. Configuration Design: An Integrated Approach Using Grammars. *ASME Journal of Mechanical Design*, 120(1):2–9, 1998.

[23] L. C. Schmidt, H. Shetty, and J. Cagan. A Graph Grammar Approach for Structure Synthesis of Mechanisms. In *Proceedings of the ASME 1998 Design Engineering Technical Conference*, Atlanta, GA, September 1998.

[24] André Schulz and Benno Stein. On Automated Design of Technical Systems. Notes in Computer Science tr-ri-00-218, Department of Mathematics and Computer Science, University of Paderborn, Germany, December 2000.

[25] A. Schürr. PROGRES: A VHL-Language Based on Graph Grammars. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proc. 4th Intl. Workshop on Graph Grammars and zheir Application to Computer Science*, volume 532 of *LNCS*, pages 641–659. Springer, 1991.

[26] A. Schürr, A. Winter, and A. Zündorf. Visual Programming with Graph Rewriting Systems. In *Proceedings of the Eleventh International IEEE Symposium on Visual Languages*. IEEE Computer Society Press, 1995.

[27] Andy Schürr. Introduction to PROGRESS, an Attribute Graph Grammar Based Specification Language. In M. Nagl, editor, *Proc. 15th Intl. Workshop on Graph-Theoretic Concepts in Computer Science*, volume 411 of *LNCS*, pages 151–165. Springer-Verlag, 1989.

[28] Andy Schürr. Developing Graphical (Software Engineering) Tools with PROGRES. In *Proc. ICSE*, pages 618–619. IEEE Computer Society Press, 1997.

[29] Benno Stein. *Model Construction in Analysis and Synthesis Tasks*. Habilitation thesis (submitted), University of Paderborn, Department of Mathematics and Computer Science, 2001.

[30] Benno Stein and André Schulz. Modeling Design Knowledge on Structure. In Gregor Engels, Andreas Oberweis, and Albert Zündorf, editors, *Proceedings of the Workshop "Modellierung 2001"*, volume P-1 of *Lecture Notes in Informatics, LNI*, pages 38–48, Bonn, March 2001. Gesellschaft für Informatik. ISBN 3-88579-330-X.

[31] G. Stiny. Introduction to shape and shape grammars. *Environment and Planning*, 7:343–351, 2001.