On the Automated Design of Technical Systems *

André Schulz Benno Stein † Annett Kurzok ‡

October 15, 2001

Contents

*Supported by DFG grant PA 276/23-1

†Dept. of Computer Science / Knowledge-Based Systems, University of Paderborn, 33095 Paderborn, Germany
Email: {aschulz,stein}@upb.de

‡Dept. of Engineering / Chemical Engineering, University of Paderborn, 33095 Paderborn, Germany,
email: annett.kurzok@vt.upb.de

Abstract

At present design support of technical systems is typically connected with the configuration of the technical devices that realize a technical process and does not properly address other tasks such as structure definition or analysis. In fact, the configuration step is one of the last jobs within the entire design process, and several hurdles have been taken by the designer before.

This paper outlines both an idea and a methodology to realize a design support right from the start of the design process. Central element of the methodology is the abstraction of a technical system as a graph along with the use of graph grammars for structural manipulation, which encode an engineer's design knowledge. By applying graph transformation rules, an incompletely and coarsely defined design can be completed and refined towards a desired solution, and a given design can be verified to determine its feasibility.

In order to illustrate the presented methodology, detailed examples from the chemical engineering domain are presented, as well as short examples from other technical domains.

1 Introduction

The design of a system is a major challenge faced by designers at all times.

Many design processes contain the same principal tasks, such as analysis and optimization: design does not mean configuration alone, as is the general perception. Clearly, the complexity of a design process may vary with the domain.

When solving a design task, it is useful to look at a system from the viewpoints of structure and behavior [Stein, 2001], which, depending on the domain, are either loosely or tightly connected to each other. Modern design tools focus on one of these facets, which, in most cases, is behavior.

In contrast to these approaches, the purpose of the paper in hand is to provide foundations for a holistic support of the design process. Given a set of parameterized system building blocks and a description of the demands, e. g. in the form of inputs and outputs, both the selection and the connection of the necessary building blocks shall be derived. Moreover, the analysis and improvement of structure and behavior of a given design is also addressed in this paper.

The essence of our approach is: A technical system is viewed as a graph, the nodes of the graph describe system building blocks, the edges of the graph specify the connections between building blocks and enable information and energy exchange. Modifications of a technical system are defined as node-insertion and node-deletion operations on the graph. We use graph grammars as a proper means to precisely specify such modifications, say, to encode an engineer's design knowledge.

For illustrative purposes we will resort to the domain of chemical engineering throughout this paper. As an aside, in this domain computer-based design support exists in the form of configuration systems for particular devices such as mixers and agitators [Brinkop and Laudwein, 1993, Knoch and Bottlinger, 1993]. Moreover, there are tools concentrating around simulation, and yet other tools were developed as tailored CAD programs [Räumschüssel et al., 1993, Marquardt, 1992, Stephanopoulos et al., 1990, Piela et al., 1991, Pantelides, 1988].

This paper is organized as follows. Section 2 gives a description of chemical design tasks, which is exemplified by means of a realistic example. Section 3 examines the requirements imposed by design tasks and introduces the concept of design graph grammars. Section 4 applies the design graph grammar approach to structure analysis and briefly discusses the simulation of the underlying model. Section 5 applies the design graph grammar approach to structure synthesis. Section 6 sheds some light on different topics: design evaluation, repair, and optimization. Section 7 supplies a theoretical foundation related to the area of graph grammars. The appendix contains practical examples of design graph grammars used within different domains.

2 A Design Task from the Domain of Chemical Engineering

The approach presented in this paper may be suited to tackle various kinds of chemical design problems. However, within our project as well as our research we concentrate only on a particular part of chemical processes: The design of plants for the food processing industry.

A chemical plant can be viewed as a graph, where the nodes represent the devices, or unit-operations, and 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. Modifications of a chemical process include the insertion of devices, rearrangement of chains, removal or substitution of redundant devices etc. At the abstract level these modifications match the graph operations mentioned earlier.

The task of designing a chemical plant is defined, as in many other fields, by the given input and the desired output. The goal is to mix or transform various input substances in such a way that the resulting product meets the imposed requirements. In general, this process may also yield some by-products, but this will be neglected in this paper—we will restrict ourselves to the $n : 1$ case. Figure 1 illustrates the solution process followed in general practice.

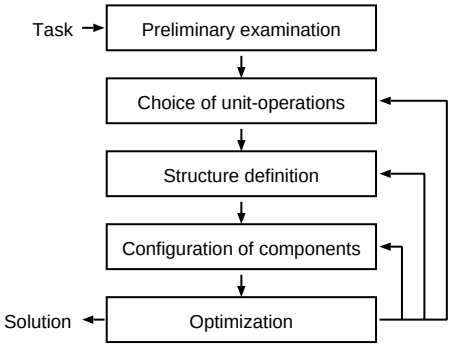


Figure 1: Steps in the design process of a chemical plant.

The steps depicted on Figure 1 can be described in more detail as follows:

1. *Preliminary examination*. This step comprehends any preparatory measures that must be taken prior to beginning with the design process. This includes examining the task

specification, i. e., the input substances and the desired output, from which possibly implicit information can be extracted. For instance, the input substances may differ in a certain essential property like "solubility in water", and, if there are more than two input substances, it might be necessary to process the ones belonging to the same solubility type before dealing with all input substances together. This can be done by grouping or clustering the substances according to prespecified properties.

2. *Choice of unit-operations*. After having examined the substances involved in the desired chemical process, abstract building blocks, so-called unit-operations, are chosen in compliance with certain rules. For example, if the output mixture should have a temperature that is substantially different from the temperature of the input substances, then the unit-operation *heat transfer* is needed. Moreover, using the example of the last step, if at least two substances have different solubility properties, then the unit-operation *emulgation* is necessary. Similarly, any other conclusions concerning the choice of unit-operations are drawn in a rule-based fashion.

   In practice, engineers choose concrete devices at this stage—the use of abstract building blocks is done implicitly.

3. *Structure definition*. The previous step produces a set of unit-operations devoid of any structure, i. e., the unit-operations are still "unconnected". To find an apt topology, different circuits are tried until one that meets the requirements is found. This well-known *propose-and-revise* behavior has been also applied to the field of chemical engineering Brinkop and Laudwein [1993].

4. *Configuration of devices*. The chosen devices, still represented by unit-operations, are instantiated. Beginning with the first unit-operation in the process chain, concrete devices are chosen from a database. Since different devices of the same class often produce outputs with slightly different properties, these changes must be propagated throughout the chain, thus influencing the choice of later devices.

   Alternatively, this step may also be performed before the structure is defined (but no propagation takes place at this point).

5. *Optimization*. The plant's functionality is tested whether it meets the imposed requirements. If the designed plant fails to fulfill any of these conditions, some changes have to be applied either to the structure or to the set of chosen devices.

   Even if the plant represents a solution to the problem, the engineer might still want to refine it to reduce energy consumption or to decrease mixing time. These optimizations or modifications also require some changes to the plant, making the return to a previous step obligatory.

2.1 Model Simplification

An automation of the steps listed in section 2 at the behavioral level would be very expensive—present systems limit automation to human-unfriendly tasks like simulation, and the effort involved there is high enough. Thus, we must apply some simplifications to

the design task in order to be able to provide some kind of automation for all steps. The following model simplification[1] steps lead to a more tractable problem:

- *Approximation*. Instead of using different functions and formulas that apply under different conditions, only one function or formula covering the widest range of restrictions is used in each case. For example, there are over 50 different formulas to calculate the viscosity of a mixture, most of which are very specialized versions and only applicable under very rare circumstances—the formula $ln(\eta) = \sum_i \varphi_i \cdot ln(\eta_i)$, however, is very often applicable and delivers a good approximation, even in the complicated cases.

- *Numeric representation*. Although the use of crisp values leads to exact results, fuzzy sets are used to represent essential value ranges. This simplification diminishes the combinatorial impact on our graph grammar approach, since substance properties are coded into edge labels and the use of crisp values would lead to an excessive number of rules.

- *Aggregation of cycles*. This structural simplification eliminates all cycles from the abstract plant design, thereby avoiding feedback within the design. Without cycles—and without feedback—simulation becomes simpler.

- *Hierarchical aggregation by structure*. The introduction of a component hierarchy within which properties are shared and inherited is not only profitable from a software design point of view, but also as far as calculations are concerned, since abstract devices and families can be used therein.

- *Derived relationships*. Some fields of chemical engineering still remain unveiled and are dealt with as black boxes. In such cases one has to resort to look-up tables and interpolation, as far as sufficient information is available.

- *Parameter elimination*. Some model aspects of minor consequence are ignored, thereby simplifying the model as a whole. One good example are pipes used to transport the substances within the plant: In general, the length and the material of a pipe determine the degree of thermal loss of a substance being conveyed. However, these factors play a minor role and can therefore be neglected.

  Another simplification that belongs to parameter elimination is the restriction of the number of relevant substance properties at the design generation stage. During design generation, decisions are taken based on the abstract values of a small set of substance properties, such as temperature, viscosity, density, mass and state. Properties such as heat capacity, heat conductivity or critical temperature and pressure are neglected at this point.

2.2 Caramel Syrup Example—Structure

We now present a concrete example for the design process described in section 2. Here the emphasis is laid on the structure of the design; section 2.3 will address the behavior of the

[1] A model construction theory was developed in [Stein, 2001], where *model simplification* plays an important role.

design.

The following task specification excerpt shall help illustrate the usual design procedure performed by an engineer.

Name	State	Mass	Temperature	Viscosity
sugar	solid	47.62%	20°C	–
water	liquid	15.75%	20°C	0.0010012 Pas
starch syrup	liquid	36.63%	20°C	0.2-1.6 Pas
caramel syrup	liquid	100.00%	110°C	?

The goal is to produce caramel syrup, which is necessary for the production of caramel bonbons, using water, starch syrup and sugar.

The following table containing viscosity values of sugar solution, a possible intermediate product, is also available, although it does not belong to the task specification per se:

Temperature	Viscosity (71% solution)
0°C	5000 Pas
10°C	1000 Pas
20°C	500 Pas
30°C	250 Pas
40°C	130 Pas
50°C	80 Pas
60°C	50 Pas
70°C	30 Pas
80°C	20 Pas

Based on this task specification, the following steps pertaining to the structure are performed in compliance with the general procedure depicted in section 2:

1. *Preliminary examination*. The first observation made by the engineer is that one of the substances, sugar, is a solid and must be dissolved within one of the other input liquids. Since water has a lower viscosity than starch syrup, it will be better to mix sugar and water first and then add the starch syrup to the solution. Depending on the mass ratios the water may have to be heated beforehand to increase solubility.

2. *Choice of unit-operations*. The comparison of the mass ratios of sugar and water leads to the conclusion that heating is necessary; thus, a heat transfer unit-operation is needed to heat the water. The heated water and the sugar are then mixed—for this purpose a mixing unit-operation for lower viscose substances is appropriate. To avoid recrystallization, the starch syrup should also be heated, thereby making another heat transfer unit-operation necessary. Finally, the heated sugar solution and the heated starch syrup are mixed. In order to reach the required temperature of 110°C, another heat transfer unit-operation will be needed. Furthermore, pump unit-operations are required to transport the substances between devices.

3. *Structure definition*. The choice of unit-operations, although having no direct impact on the structure, allows for certain conclusions pertaining to the ordering of the unit-operations. In this case this ordering is relatively evident; Figure 2 shows the chosen topology.

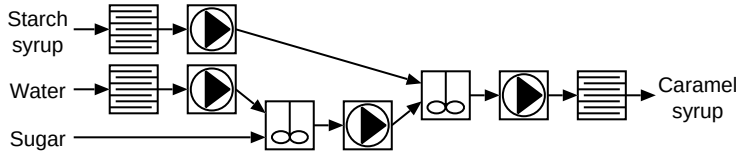


Figure 2: The first design of the example process.

2.3 Caramel Syrup Example—Behavior

The example presented in the previous section shows how an abstract design can be generated, using a simplified task description. The result of the generation process is a feasible structure complying with the simplified demands. This, however, does not represent a *complete* design, since all devices remain abstract. Thus, this abstract model must be enhanced with concrete device data—static and dynamic parameters.

The following steps determine the behavior of the design and make some corrections, if applicable:

1. *Configuration of devices*. Based on the mass, the volume, and the other properties of the involved substances, matching devices are chosen from databases or data sheets. For the sake of simplicity we will refrain from a detailed description here and refer to Figure 3, where the abstract design with additional data from the underlying model is shown.

2. *Optimization*. The computed properties of the plant design usually represent feasible values, but improvement may still be possible. With this goal in mind, the parameterization process is repeated and parameters adjusted accordingly. In our case the last heat transfer unit-operation of the process chain represents an overkill—the last mixing unit-operation is then slightly changed so that only devices with a built-in heat transfer unit are considered. This change shortens the process chain, thereby reducing costs and mixing time. The final design is depicted by Figure 3.

Alternatively, another design with fewer devices is conceivable. For instance, water and starch syrup could be mixed first, and the resulting solution used to dissolve sugar. This structure choice would require one heat transfer unit-operation less than the proposed design because both water and the starch syrup have to be heated to the same temperature, which is best done if both substances are mixed together beforehand. However, this alternative would cause a longer mixing time, since the sugar must be dissolved in a more viscose solution (compared to pure water). Figure 4 shows this alternative solution.

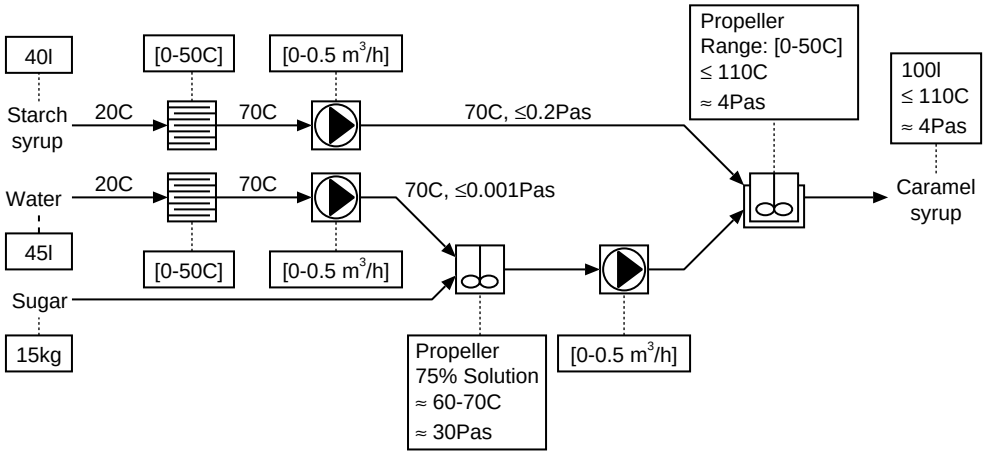


Figure 3: Design showing part of the underlying model.

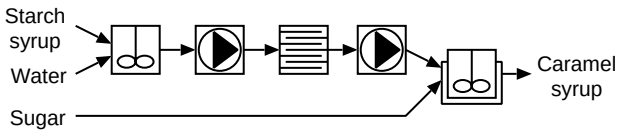


Figure 4: Alternative design of the example process.

3 Graph Grammar Model for Design

Each of the steps depicted in Figure 1 can be automated in an isolated fashion. However, a separate processing may lead to loss of information, since the choice of a unit-operation often affects the structure and vice versa. For example, the choice of a certain mixer might influence the decision whether a heat transfer device is necessary or not, thereby possibly causing a change to the topology. Likewise, a certain ordering of the devices within the plant structure can make one of them superfluous.

Due to the intertwined nature of these steps, it is strongly desirable to combine the choice of unit-operations and the structure definition to make use of all information available. One way of tackling both tasks simultaneously is to use a graph grammar to generate feasible designs in a controlled manner, thus allowing for an incremental execution of the mentioned steps. The graph grammar will not only be used for controlled generation, but also for analysis tasks, optimization and repair tasks, and also for dynamic visualization purposes.

In the following we analyze the requirements imposed by the various design aspects and introduce suitable graph grammar models to fulfill them. Finally, special issues concerning the semantics of design graph grammars are addressed.

3.1 Design Tasks and Graph Transformation Rules

A technical system can be described by a labeled graph. The nodes of the graph designate the system's items, the graph's edges define relations between the items, labels specify the

types of nodes and edges. The following definition introduces this concept formally.

Definition 1 *(Labeled Graph)*

A labeled graph is a tuple $G = \langle V_G, E_G, \sigma_G \rangle$ where V_G is the set of nodes, $E_G \subseteq V_G \times V_G$ is the set of directed edges, and σ_G is the label function, $\sigma_G : V_G \cup E_G \to \Sigma$, where Σ is a set of symbols, called the label alphabet.

Notation: (v_1, v_2, l) represents a directed edge with tail v_1, head v_2 and label l. $\{v_1, v_2, l\}$ denotes an undirected edge with label l, which can be viewed as two directed edges, (v_1, v_2, l) and (v_2, v_1, l). Edges without labels will be written as (v_1, v_2) or $\{v_1, v_2\}$.

The *design* of a system encompasses a variety of different aspects or tasks and not only the traditional construction process with which it is usually associated. For each of these tasks different operations of varying complexity are required:

- Insertion and deletion of single items in a system
- Change of specific item and connection types
- Manipulation of sets of items, e. g., for repair or optimization

The operations delineated above can be viewed as transformations on graphs; they are of the form *target* $\to$ *replacement*. A precise specification of such "graph transformation rules" can be given with graph grammars, which are distinguished by the complexity of their rules' left-hand sides. A central concept in this connection is bound up with the notions of matching and context, which, in turn, build up on the concept of isomorphism (see, for example, [Jungnickel, 1999]).

Definition 2 *(Isomorphism, Isomorphism with labels)*

Let $G = \langle V_G, E_G \rangle$ and $H = \langle V_H, E_H \rangle$ be two graphs. An isomorphism is a bijective mapping $\varphi : V_G \to V_H$ for which holds: $\{a, b\} \in E_G \Leftrightarrow \{\varphi(a), \varphi(b)\} \in E_H$, for any $a, b \in V_G$. If such a mapping exists, G and H are called isomorphic.

G and H are called isomorphic with labels, if G and H are labeled graphs with labeling functions σ_G and σ_H, and the following additional condition holds: $\sigma_G(a) = \sigma_H(\varphi(a))$ for each $a \in V_G$, and $\sigma_G(e) = \sigma_H(\varphi(e))$ for each $e \in E_G$, where $\varphi(e) = \{\varphi(a), \varphi(b)\}$ if $e = \{a, b\}$.

Figure 5 shows an example of isomorphic and non-isomorphic graphs.

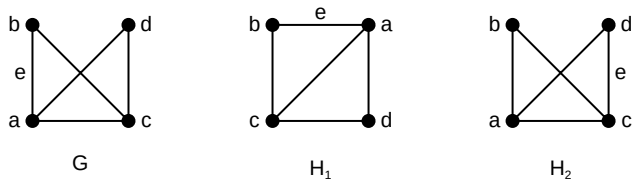


Figure 5: A graph G, a graph H_1 that is isomorphic to G, and a graph H_2 that is not isomorphic to G.

Definition 3 *(Matching, Context)*

Given are a labeled graph $G = \langle V, E, \sigma \rangle$ and another labeled graph, C. Each subgraph $\langle V_C, E_C, \sigma_C \rangle$ in G, which is isomorphic to C, is called a matching of C in G. If C consists of a single node only, a matching of C in G is called node-based, *otherwise it is called* graph-based.

Moreover, let T be a subgraph of C, and let $\langle V_T, E_T, \sigma_T \rangle$ denote a matching of T in C. A matching of C in G can stand in one or more of the following relations to $\langle V_T, E_T, \sigma_T \rangle$:

1. *$V_T \subset V_C, V_T \neq \emptyset$. Then the graph $\langle V_C, E_C, \sigma_C \rangle$ is called a context of T in G.*
2. *$\langle V_C, E_C, \sigma_C \rangle = \langle V_T, E_T, \sigma_T \rangle$. Then T is called context-free.*

A matching of a graph T in G is denoted by $\widetilde{T}$. In general, we will not differentiate between a graph T and its isomorphic copy.

Figure 6 illustrates the notions of matching and context.

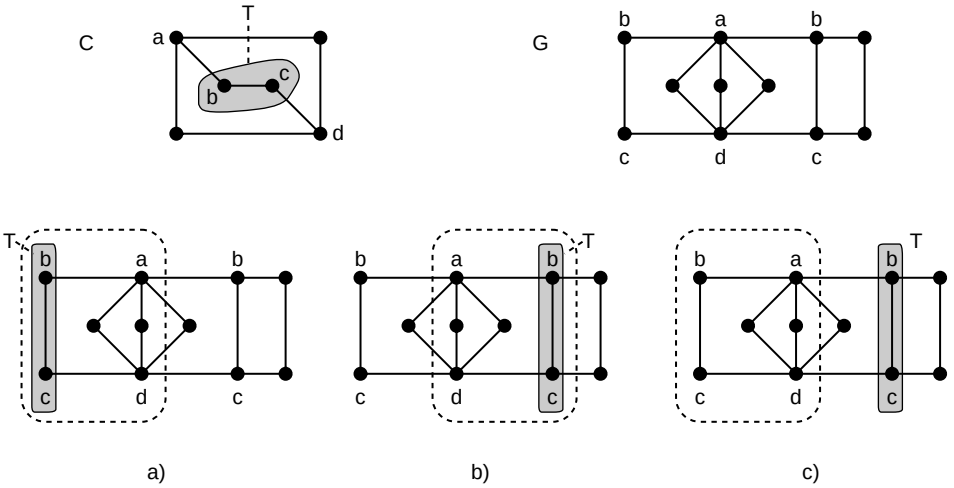


Figure 6: Above, a context graph C including a target graph T, and a host graph G. Below: a) strict degree matching of T in G; b) matching of T in G; and c) matching of C in G, but no context of T.

Remarks. A matching $\widetilde{T}$ of a graph T within another graph G represents a subgraph of G, which means that potentially every node of $\widetilde{T}$ may be connected to the remainder of G by arbitrarily many edges. This matching concept may be sufficient for most purposes, but the domain of technical systems requires more flexibility. Thus, the concept matching is refined to allow the matching of nodes with a precise number of edges. This type of matching is called *strict degree matching*; in practice, the use of this type of matching will be indicated by an asterisk appended to a node instance, as in $T = \langle \{1^*, 2\}, \{(1, 2)\} \rangle$.

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., 1997], page 97. Secondly, graph grammars have not been applied seriously in order to solve synthesis and analysis problems in the area of technical systems—graph grammar solutions focus mainly on software engineering problems [Rozenberg, 1997,

Ehrig et al., 1999a,b, van Eekelen et al., 1998, Kaul, 1986, 1987, Korff, 1991, Lichtblau, 1991, Rekers and Schürr, 1995, Schürr et al., 1995, Schürr, 1997a].

The systematics of design graph grammars introduced here addresses these shortcomings. Figure 7 relates classical graph grammar terminology to typical design tasks; the following list presents examples for the differently powerful rule types. A precise analysis of the relationship between classical graph grammar families and design graph grammars can be found in chapter 7.

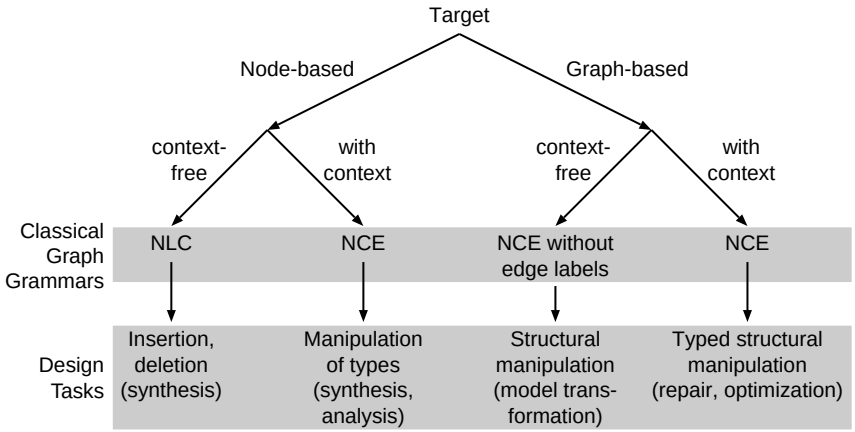
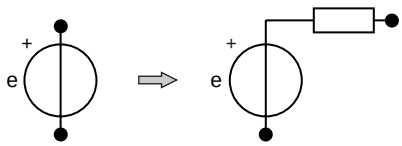


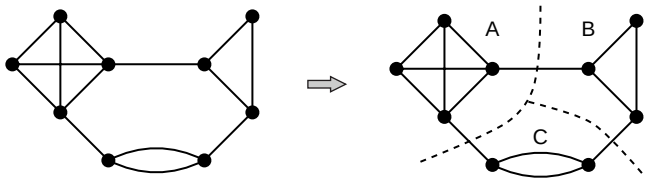
Figure 7: Graph grammar hierarchy for the various design tasks. The abbreviations NLC and NCE denote classical graph grammar families.

- Node → node: Context-free transformation based on node labels. Graph grammars with rules of this type are called *node label controlled* graph grammars (NLC grammars). The following figure illustrates a type modification of a mixing unit, which can be realized by this class of rules.

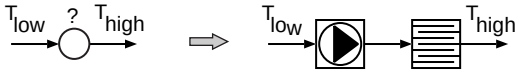
- Node → graph: Context-free transformation based on node labels (NLC grammars). The following figure shows the replacement of an ideal voltage source by a resistive voltage source (synthesis without context).



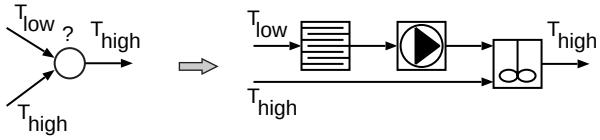
- Node with context → node: Node-based transformation based on node labels and edge labels. Graph grammars with rules of this type are called *neighborhood controlled embedding* graph grammars (NCE grammars). The clustering of graphs (analysis and synthesis with context) is an example for this type of transformation and is depicted in the following figure.



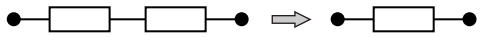
- Node with context → graph: Node-based transformation based on node labels and edge labels (NCE grammars). The following figure shows the replacement of an unknown unit by inserting heat transfer and pump units to fulfill the temperature constraints (synthesis with context):



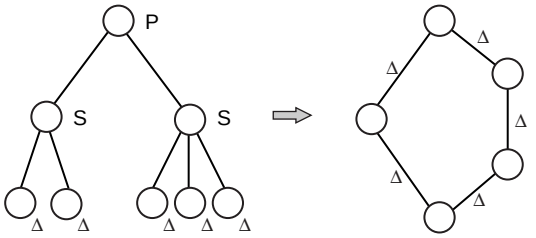
The following figure illustrates the replacement of an unknown unit by inserting a heat transfer unit, a pump unit, and a mixing unit (synthesis with context):



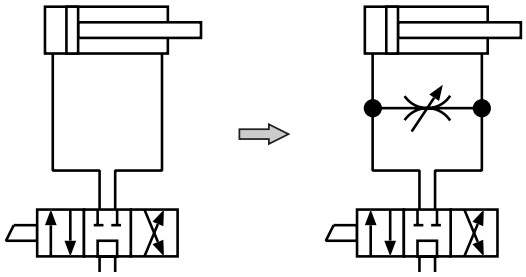
- Graph → graph: Context-free transformation based on graphs without edge labels (NCE grammars without edge labels). The replacement of two resistors in series with one resistor, an example for structural manipulation, is depicted below.



Another example for a transformation of this type is given by the conversion of a structure description tree into a parallel-series graph (model transformation):



- Graph with context → graph: Context-sensitive transformation based on graphs with edge labels (NCE grammars). The following figure shows the insertion of a bypass throttle (repair, optimization), which represents such a transformation.



3.2 Context-free Design Graph Grammar

What happens during a graph transformation is that a node, t—or a subgraph, T—in the original graph G is replaced by a graph R. Put another way, R is embedded into G.

A graph grammar can be regarded as the equivalent of a Chomsky grammar for the area of graphs, and as such it has similar properties. In the following we will provide a formal basis for the illustrated graph transformations.

Definition 4 *(Host Graph, Context Graph, Target Graph, Replacement Graph, Cut Node)*

Within the graph transformation context a graph can play one of the following roles:

- Host graph G. *A host graph represents the structure on which the graph transformations are to be performed.*
- Context graph C. *A context graph represents a matching to be found in a host graph G. The graph C is part of the left-hand side of graph transformation rules.*
- Target graph T. *A target graph represents a graph whose matching in a host graph G is to be replaced. If T is a subgraph of a context graph C, then the occurrence of T within the matching of C in G is to be replaced. The graph T is part of the left-hand side of graph transformation rules. In case T consists of a single node, it is called* target node *and denoted by t.*
- Replacement graph R. *A replacement graph represents a graph of which an isomorphic copy is used to replace a matching of the target graph T in the host graph. The graph R is part of the right-hand side of graph transformation rules.*
- *The nodes of the host graph that are connected to the matching of T are called* cut nodes.

Informally, a graph grammar is a collection of graph transformation rules, each of which is equipped with a set of embedding instructions. The application of a graph transformation rule to a host graph G yields a new graph G'. The following definition provides the necessary syntax and semantics.

Definition 5 *(Context-Free Design Graph Grammar)*

A context-free design graph grammar is a tuple $\mathcal{G} = \langle \Sigma, P, s \rangle$ with

- Σ *is the label alphabet used for nodes and edges*[2]*,*
- P *is the finite set of graph transformation rules or productions,*
- *and* s *is the initial symbol.*

The productions of the set P *are graph transformation rules of the form* $T \rightarrow \langle R, I \rangle$ *with*

- $T = \langle V_T, E_T, \sigma_T \rangle$ *is the target graph to be replaced,*
- $R = \langle V_R, E_R, \sigma_R \rangle$ *is the possibly empty replacement graph,*
- I *is the set of embedding instructions for the replacement graph* R*.*

  The semantics of a context-free graph transformation rule $T \rightarrow \langle R, I \rangle$ *is as follows: Firstly, a matching of the target graph* T *is searched within the host graph* G*. Secondly, this occurrence of* T *along with all incident edges is deleted. Thirdly, an isomorphic copy of* R *is connected to the host graph according to the semantics of the embedding instructions.*

  The set I *of embedding instructions consists of tuples* $((h, t, e), (h, r, f))$*, where*

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

  An embedding instruction $((h, t, e), (h, r, f))$ *is interpreted as follows: If there is an edge with label* e *connecting a node labeled* h *with the target node* t*, then the embedding process will create a new edge with label* f *connecting the node labeled* h *with node* r*. See section 3.4 for a detailed discussion about the semantics of embedding instructions.*

The execution of a graph transformation rule p *on a host graph* G *yielding a new graph* G' *is called a* derivation step *and denoted by* $G \Rightarrow_p G'$*. A sequence of such derivation steps is called* derivation*. The set of all graphs that can be generated with* $\mathcal{G}$ *is designated by* $L(\mathcal{G})$*.*

Example. In order to illustrate how a graph transformation rule works, let us view the transformation of a graph G into a graph G', as depicted in Figure 8.

A graph transformation rule $T \rightarrow \langle R, I \rangle$ that performs the transformation shown in Figure 8 has the following components:

- Target graph $T = \langle V_T, E_T, \sigma_T \rangle = \langle \{1, 2\}, \{\{1, 2\}\}, \{(1, b), (2, c)\} \rangle$
- Replacement graph $R = \langle V_R, E_R, \sigma_R \rangle = \langle \{3\}, \emptyset, \{(3, n)\} \rangle$
- Set I of embedding instructions with $I = \{((a, b, f), (a, n, f)), ((e, c, f), (e, n, f))\}$. Alternatively, one can employ variable labels, which yields $I = \{((X, Y, f), (X, n, f))\}$.

[2]Labels are used to specify types and as variables for other labels. To avoid confusion, variable labels will be denoted by capital letters, and all other labels with small letters.

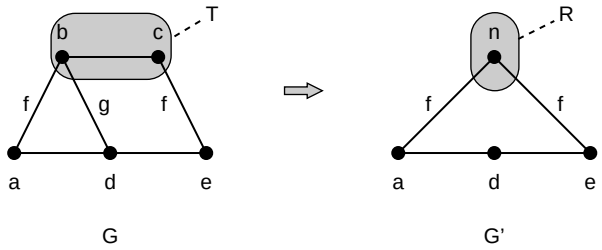


Figure 8: Application of a context-free graph transformation rule on a host graph G showing a target graph T and a replacement graph R.

In many cases one will not need the complete expressive power of the embedding instruction definition. In particular, if the target graph consists of a single node and edge labels are left unchanged, then an embedding instruction may be written as (e, r) instead of $((h, t, e), (h, r, e))$; in the literature such graph grammars are called *neighborhood uniform* graph grammars.

In the following we present the formal representation of a simple design graph grammar for the rules shown in Figures 9 and 10.

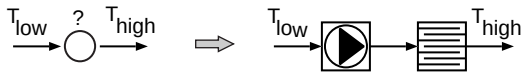


Figure 9: Insertion of a heating chain.

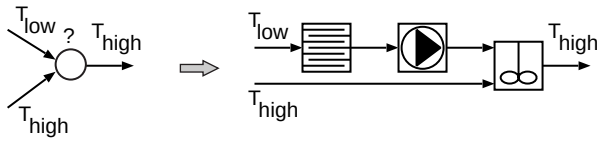


Figure 10: Insertion of a heating chain as a preprocessing step.

Example. Let $\mathcal{G} = \langle \Sigma, P, s \rangle$ be a design graph grammar that specifies some transformations described in section 3.1.

- $\Sigma = \{?, A, B, C, D, low, high, pump, heater, mixer\}$,
- $P = \{r_1, r_2\}$,
- $s = ?$

Rule r_1 is defined as follows:

$$
\begin{aligned}
T &= \langle \{1,2,3\}, \{(1,2),(2,3)\}, \{(1,A),(2,?),(3,B),((1,2),low),((2,3),high)\} \rangle \\
R &= \langle \{4,5,6,7\}, \{(4,5),(5,6),(6,7)\}, \{(4,A),(5,pump),(6,heater),(7,B), \\
&\quad ((4,5),low),((6,7),high)\} \rangle \\
I &= \{((D,A,E),(D,A,E)),((D,B,E),(D,B,E))\}
\end{aligned}
$$

The formal representation of rule r_2 is:

$$\begin{aligned} T &= \langle \{1,2,3,4\}, \{(1,3),(2,3),(3,4)\}, \{(1,A),(2,B),(3,?),(4,C), \\ &\quad ((1,3),low),((2,3),high),((3,4),high)\}\rangle \\ R &= \langle \{5,6,7,8,9,10\}, \{(5,7),(7,8),(8,9),(6,9),(9,10)\}, \\ &\quad \{(5,A),(6,B),(7,heater),(8,pump),(9,mixer),(10,C), \\ &\quad ((5,7),low),((6,9),high),((9,10),high)\}\rangle \\ I &= \{((D,A,E),(D,A,E)),((D,B,E),(D,B,E)),((D,C,E),(D,C,E))\} \end{aligned}$$

3.3 Context-Sensitive Design Graph Grammar

In section 3.2 the notion of context-free design graph grammars was introduced. However, it is conceivable that context-sensitive rules may be necessary, which fact makes context-free graph grammars inadequate. In the literature, the natural extension of context-free graph grammars is called *context-sensitive graph grammars*. The following definition is based on definition 5 of section 3.2.

Definition 6 *(Context-sensitive Design Graph Grammar)*

A context-sensitive design graph grammar is a tuple $\mathcal{G} = \langle \Sigma, P, s\rangle$ as described in Definition 5, whose productions in the set P are graph transformation rules of the form $\langle T, C\rangle \rightarrow \langle R, I\rangle$ with

- *$T = \langle V_T, E_T, \sigma_T\rangle$ 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,*
- *I is the set of embedding instructions for the replacement graph R.*

  The semantics of a graph transformation rule $\langle T, C\rangle \rightarrow \langle R, I\rangle$ is as follows: Firstly, a matching of the context C is searched within the host graph. 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 semantics of the embedding instructions.

  The set I of embedding instructions consists of tuples $((h,t,e),(h,r,f))$, where

  - *$h \in \Sigma$ is a label of a node $v \in G \setminus T$,*
  - *$t \in \Sigma$ is a label of a node $w \in V_T$,*
  - *$e \in \Sigma$ is the label of the edge $\{v,w\}$,*
  - *$f \in \Sigma$ is another edge label not necessarily unequal to e, and*
  - *$r \in V_R \cup V_C$ is a node in R, where V_C is the set of cut nodes.*

  An embedding rule $((h,t,e),(h,r,f))$ is interpreted as in the context-free case.

In the following we will not explicitly distinguish between both graph grammar types, since the used variant is obvious from the context and rule form.

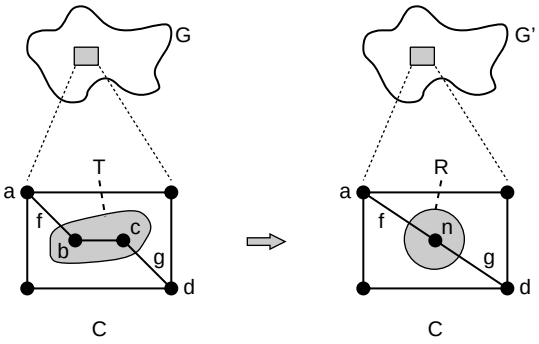


Figure 11: A context-sensitive graph transformation rule showing a host graph G, a target graph T, a context graph C and a replacement graph R.

Example. In order to illustrate how a context-sensitive graph transformation rule works, let us view the transformation of a graph G into a graph G', as depicted in figure 11.

A graph transformation rule $\langle T, C\rangle \rightarrow \langle R, I\rangle$ that performs the transformation shown in figure 11 has the following components:

$$
\begin{aligned}
T &= \langle V_T, E_T, \sigma_T\rangle = \langle \{1,2\}, \{\{1,2\}\}, \{(1,b),(2,c)\}\rangle \\
C &= \langle V_C, E_C, \sigma_C\rangle \\
&= \langle \{3,4,5,6,7,8\}, \{\{3,4\},\{3,7\},\{7,8\},\{4,8\},\{3,5\},\{5,6\},\{6,8\}\}, \\
&\quad \{(3,d),(5,b),(6,c),(8,d),(\{3,5\},f),(\{6,8\},g)\}\rangle \\
R &= \langle V_R, E_R, \sigma_R\rangle = \langle \{9\}, \emptyset, \{(9,n)\}\rangle \\
I &= \{((a,b,f),(a,n,f)),((d,c,g),(d,n,g))\} \text{ or, alternatively, with variable labels} \\
&\quad \{((X,Y,Z),(X,n,Z))\}
\end{aligned}
$$

Remarks. Design graph grammars differ not only in matters of context, but also in the size of the target graph. If all target graphs in the graph transformation rules consist of single nodes, the graph grammar is called *node-based*, otherwise it is called *graph-based*. This distinction is of relevance, since node-based and graph-based graph grammars fall into different complexity classes due to the subgraph matching problem (see 7.3.2) connected to the latter.

Example. The following simple[3] graph transformation rules depicted in Figures 12, 13 and 14 illustrate some cases where node-based graph transformation rules are insufficient. Note that such rules are required for optimization and repair tasks.

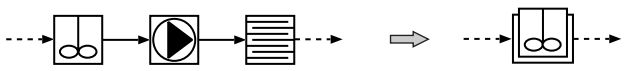


Figure 12: Replacement of a partial chain consisting of a mixer, a pump, and a heat transfer unit with a mixer device with built-in heat transfer.

[3]For the sake of simplicity edge labels have been omitted.

Figure 13: Removal of a superfluous nonterminal node.

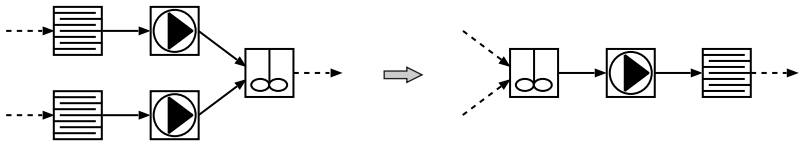


Figure 14: Combination of two identical partial chains through relocation. Depending on the properties of the substances involved, a different mixer device has to be used.

3.4 On the Semantics of Labels

Labels are of paramount importance for the graph transformation process, since all tasks belonging to a transformation step—matching of target and context graphs, embedding of replacement graphs—rely on them. Within this section we address some issues related to labels: terminal and nonterminal labels, variable labels, ambiguities during embedding and conflicting embedding instructions.

3.5 Terminal and Nonterminal Labels

Several approaches distinguish between terminal and nonterminal labels: terminal labels may appear only within the right-hand sides of graph transformation rules; nonterminal labels are used within the left-hand and right-hand sides.

Design graph grammars use the classic concept of graph matching and, therefore, there is no syntactical distinction with respect to terminals and nonterminals in the set Σ. This behavior reflects the modeling structure of the domain.

Furthermore, the above mentioned approaches also distinguish between terminal (or final) and nonterminal graphs—a graph is final if it contains only terminal labels, otherwise it is nonterminal. Design graph grammars do not make this distinction, since this is not always possible or desirable in the technical domains focused.

3.5.1 Variable Labels

Variable labels are introduced for convenience purposes: They allow for the formulation of generic rules, which match situations belonging to identical topologies using different labels; without variable labels one rule for each such situation would have to be devised, leading to a large rule set due to the combinatorial explosion.

The use of variable labels within rules and embedding instructions leads to the question of “variable binding”. Firstly, variables used exclusively within embedding instructions are

used as placeholders for concrete labels of nodes or edges matching the described context; such variables are *unbound* and used within every matching situation. Secondly, variables may be used within rules, i. e., within target, context and replacement graphs, where they represent a specific instance; such variables are *bound* and used uniformly throughout the rule application, i. e., the variable retains its "value" during the replacement and embedding processes.

Note that variable labels prevent a clear distinction between terminal and nonterminal labels: The labels in Σ can no longer be easily classified into terminal or nonterminal by analysis of the graph transformation rules in P.

3.5.2 Ambiguities

The design graph grammar approach, like the classical graph grammars, relies mainly on node and edge labels to describe matchings and embeddings. Since nodes and edges may share identical labels, ambiguities may occur, leading to possibly unwanted embeddings. Ambiguities may stem from identical edge labels of edges connected to a node, from nodes with identical labels in the target or replacement graph, as well as from a combination thereof. Figure 15 shows an example of such a situation. A straightforward solution to this problem is the unique numbering of identical labels.

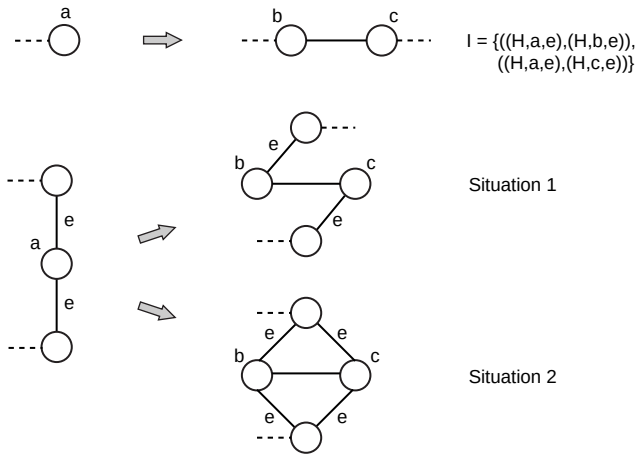


Figure 15: Graph transformation rule and two possible outcomes due to ambiguity.

Furthermore, ambiguous situations may lead to more than two different embeddings. It is conceivable that embeddings resulting from ambiguous situations are incomplete, i. e., not every edge is drawn (in contrast to the situation depicted by Figure 15), making the outcome even more unpredictable. Thus, we require that embeddings resulting from ambiguous situations are complete.

3.5.3 Conflicting Embedding Instructions

The introduction of variable labels leads to a further question pertaining to conflicts within the embedding instructions. It is conceivable that two embedding instructions involving non-variable and variable labels may apply for a given situation. For example, the embedding instructions $((H, t, a), (H, r, b))$ and $((H, T, a), (H, r, b))$ both match the situation where an edge (H, t, a) exists; if both embedding instructions are used, multiple edges may be produced.

A way to prevent this undesired effect is to avoid such embedding rules by making them uniquely applicable. However, this problem may be unavoidable—in this case the *principle of the least commitment* [Russel and Norvig, 1995] shall apply: The most specialized embedding instruction will be used.

4 Analyzing Systems

In section 3 we introduced the concept of graph grammars and explained how a design can be manipulated by means of a graph grammar. In this section we present an approach that works reversely: A given design is analyzed by means of a suitable graph grammar (together with domain knowledge) in order to determine its feasibility.

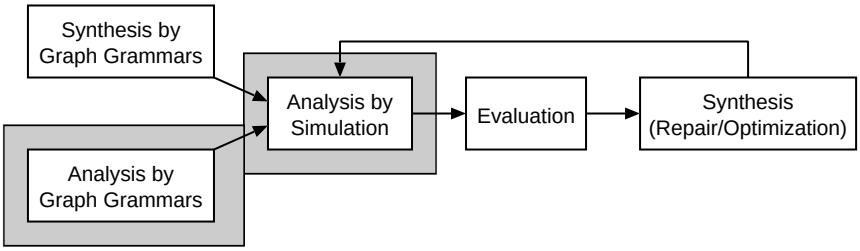


Figure 16: The design cycle.

As depicted by Figure 16, the analysis of a system is part of the whole design process, and it consists of two distinct steps that are dealt with separately. The design process elements can be described as follows:

- *Synthesis by Graph Grammars*. Based on the specified inputs and outputs and any further task requirements, a structure is built in compliance with the design graph grammar used. The resulting structure is feasible by definition, but at an abstract level.
- *Analysis by Graph Grammars*. This step is mainly concerned with the analysis of the structure of the system, for which task graph grammars have proven to be very adequate. A given design is analyzed with respect to its structure and type information, which represents traits from the underlying model at an abstract level. If a structure outstands this analysis step, then it is found to be feasible, and analysis of the underlying model can be started.
- *Analysis by Simulation*. This step pertains to the underlying model, i. e., the behavioral level, and takes the structure for granted. The feasible structure is now brought

to completion by taking the underlying model into consideration. Simulation is not performed at the abstract level, but at a lower level that is closer to the physical representation. A design that withstands this process is considered functional.

- *Evaluation*. A system design that has been found to be functional may or may not fulfill the task demands properly. This step tries to decide, according to some prespecified criteria, if the given design is acceptable or if it should be changed or improved.
- *Synthesis (Repair, Optimization)*. If a design is incorrect or just not fulfills all requirements, then some sort of adjustment must take place. Here, constructive steps are performed to produce an improved version of the original design. After completion, the design is passed to the next stage, analysis by simulation, to check that the resulting design is indeed functional.

4.1 Structure Analysis by Graph Grammars

Structural analysis aims at a preliminary statement concerning the feasibility of a given design of a technical system. This abstract feasibility check involves the design structure and the chosen components, but refrains from delving into the details concerning the underlying model, which are examined within the behavior analysis step. A design's structure and choice of components can be derived by means of design graph grammars that encode engineering knowledge. Thus, it is logical to use a design graph grammar to perform this structural analysis.

In order to use a design graph grammar to check the structural feasibility of a given design, it is necessary to determine if the design can be generated by the grammar—this problem is known as the *membership problem* for graph languages and is addressed in section 7.5.1. Basically, the membership problem is solved by applying grammar rules in a backward fashion in order to derive the initial symbol. The successful derivation of the initial symbol means that the given design belongs to the graph language generated by the design graph grammar, and the used graph transformation rules together with the application order yield a valid inverse derivation.

The following steps summarize the process of structural analysis by design graph grammars:

1. Invert the design graph grammar, i. e., change each graph transformation rule $\langle T, C\rangle \rightarrow \langle R, I\rangle$ into $\langle R, I\rangle \rightarrow \langle T, C\rangle$.
2. Choose an inverted graph transformation rule for application. If no graph transformation rule is applicable, then the design is not structurally feasible with respect to the design graph grammar used.
3. Apply the chosen graph transformation rule to the design.
4. If the design consists of the initial symbol after application of the graph transformation rule, then the given design is structurally feasible, otherwise continue at step 2.

This abstract algorithm assumes that the order of application of graph transformation rules is irrelevant and does not take backtracking into account, but can be improved to do so.

The following pseudo-code algorithm determines if a specific graph is derivable with a given design graph grammar and corresponds to the above abstract algorithm.

```
boolean DERIVABLE(DESIGN graph, DESIGNGRAPHGRAMMAR 𝒢)
(1)   {
(2)     boolean step_taken;
(3)     g := graph;
(4)     while (V_g ≠ {S}) do
(5)       step_taken := false;
(6)       for (r ∈ 𝒢.rules) do
(7)         if (REVERSEDAPPLICABLE(r, g)) then
(8)           step_taken := true;
(9)           g := REVERSEDAPPLY(r, g);
(10)        fi
(11)      od
(12)      if (not step_taken) then
(13)        return false;
(14)      fi
(15)    od
(16)    return true;
(17)  }
```

In case this feasibility check fails, i. e., the initial symbol cannot be derived from the given design, some adjustments must be made to the faulty context to make the design compliant with the underlying design graph grammar. Note that at this point only adjustments pertaining to node and edge labels are performed; any other changes involve structural transformations, which belong to another step, design repair and optimization.

The requirements listed above lead to the following algorithm:

```
boolean ISFEASIBLE(DESIGN graph, DESIGNGRAPHGRAMMAR 𝒢)
(1)   {
(2)     while (not DERIVABLE(graph, 𝒢)) do
(3)       graph' := ADJUST(graph, 𝒢);
(4)       if graph' ≠ graph then
(5)         // Continue with adjusted graph
(6)         graph := graph';
(7)       else
(8)         // Adjustment is not possible
(9)         return false;
(10)      fi
(11)    od
(12)    return true;
(13)  }
```

The function ADJUST asks the engineer or user to make some corrections to the given design in order to make it compliant with the encoded knowledge. This correction step must be done manually, since an automatic correction would require graph rules for every conceivable error, leading to a hardly manageable rule set and, consequently, to poor performance.

Without applying any restrictions to the design graph grammar and the generated graph language, the membership problem remains NP-complete (see section 7.5.1) and the above algorithm will require exponential time in the size of the design to determine if a given design belongs to the language generated by the design graph grammar. The NP-completeness of this problem is partially due to the *subgraph matching problem* described in section 7.3.2. The theoretical issues concerning the time complexity of this membership test and the possible restrictions that lead to a better performance are discussed in detail in section 7.5.

4.2 Caramel Syrup Example

We shall now simulate the functioning of the above procedures to try to determine if the design is feasible with respect to the used design graph grammar, i. e., with respect to the structure.

The design graph grammar we shall use reflects the transformations depicted in the caramel syrup example of section 2.2. These transformations are performed by rules (**R1**) through (**R6**). Furthermore, we introduce additional rules as illustrated by Figures 23 and 24.

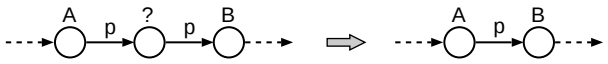


Figure 17: (**R1**) Deletion of nonterminal node.

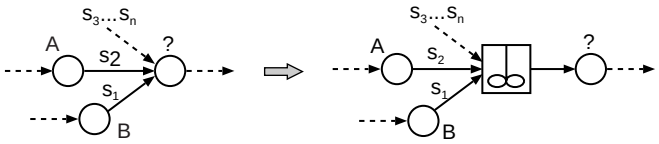


Figure 18: (**R2**) Insertion of a mixing unit-op.

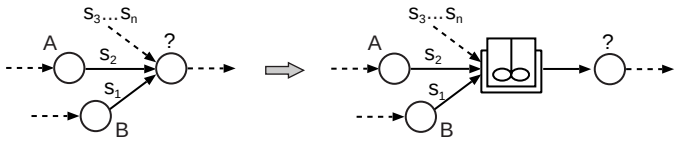


Figure 19: (**R3**) Insertion of mixing unit-op with built-in heat transfer unit.

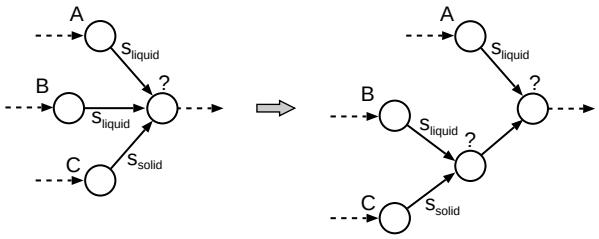


Figure 20: (**R4**) Improvement of mixing properties by handling solid inputs separately.

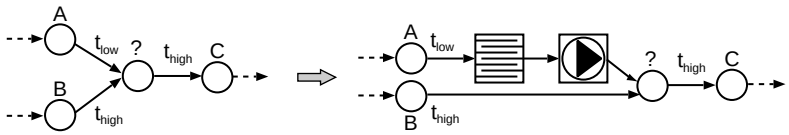


Figure 21: (**R5**) Improvement of dissolving properties by heating an input.

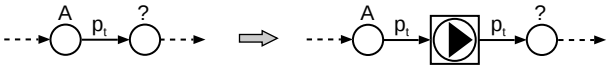


Figure 22: (**R6**) Insertion of a pump unit-op.

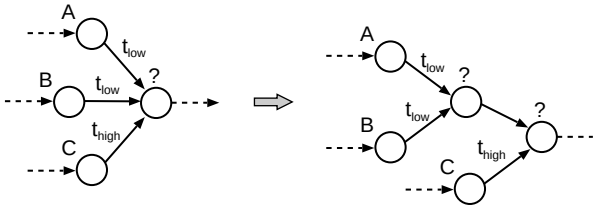


Figure 23: (**R7**) Improvement of mixing properties by handling inputs of different temperatures separately.

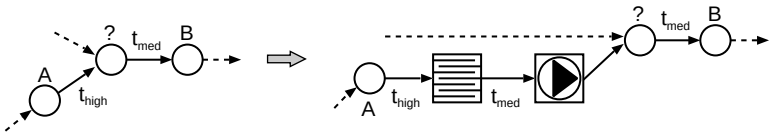


Figure 24: (**R8**) Improvement of dissolving properties by cooling an input.

Now, rules (**R1**) through (**R8**) are used to derive the initial symbol, which process is shown in Figure 25. The initial graph is the first graph in the derivation chain.

Note that at certain points *creative steps*, which correspond to the backward execution of *destructive rules* (such as (**R1**)), have to be taken in order to be able to perform other reduction steps.

4.3 Behavior Analysis by Simulation

In the previous section the feasibility of the design's structure was checked. This fact, however, does not imply a functional design—it only means that the structure is feasible and that

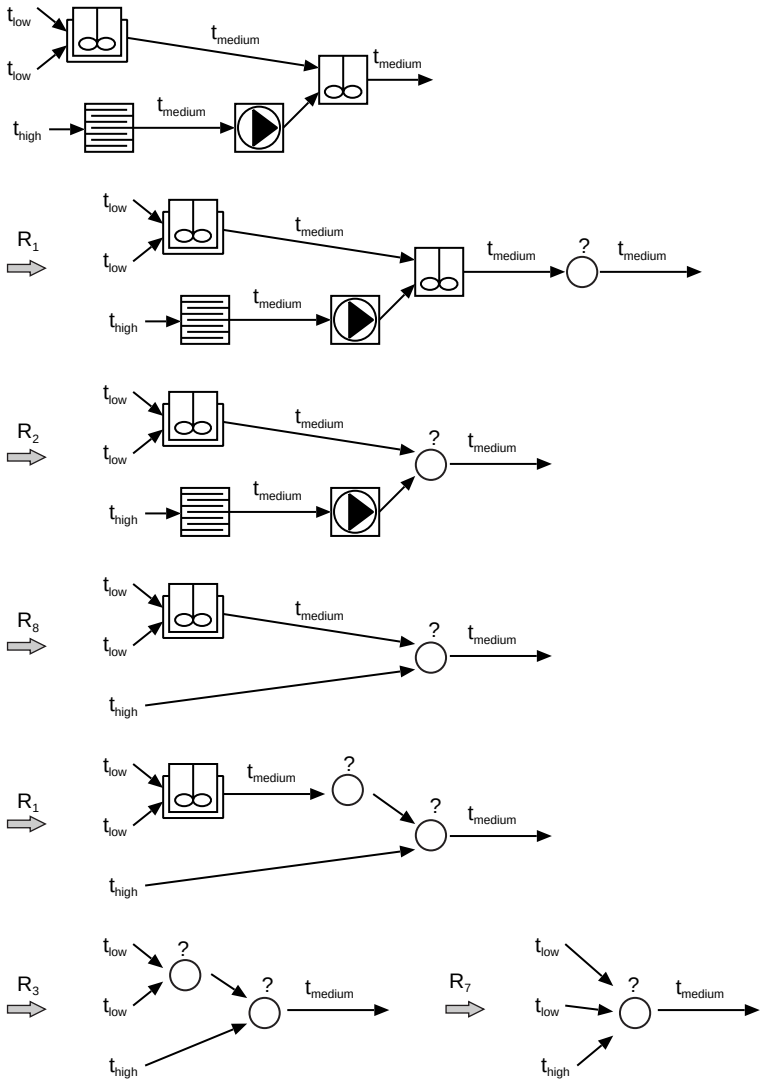


Figure 25: Derivation of the initial symbol.

it *may* belong to a functional design. It is not even guaranteed that this structure is suitable to solve the task at hand.

Thus, another procedure that goes a step further is necessary to decide on a design's functionality: simulation. Now, the feasible structure is enriched with further information pertaining to the chosen devices and involved substances, i. e., we move our focus to the underlying model. At this level, a reliable statement concerning a design's functionality can be derived.

Before our simulation approach is presented, we shortly describe the traditional simulation approaches followed by other systems and compare them to our situation.

4.3.1 Classical Simulation

Existing systems, as described in [Marquardt, 1996], have in common that they somehow produce a mathematical model—the underlying model—of the system to be simulated. Then, this mathematical model is transformed into input for a numerical algorithm, which tries to solve the equations.

The generation of the mathematical model is done either manually, as in equation-oriented systems, or partially automatically, as in block-oriented systems. In either way the plant design is decomposed into its parts, for which mathematical relations are given, and these mathematical relations are connected to each other, providing a model at which level the plant is simulated. Figure 26 illustrates this process.

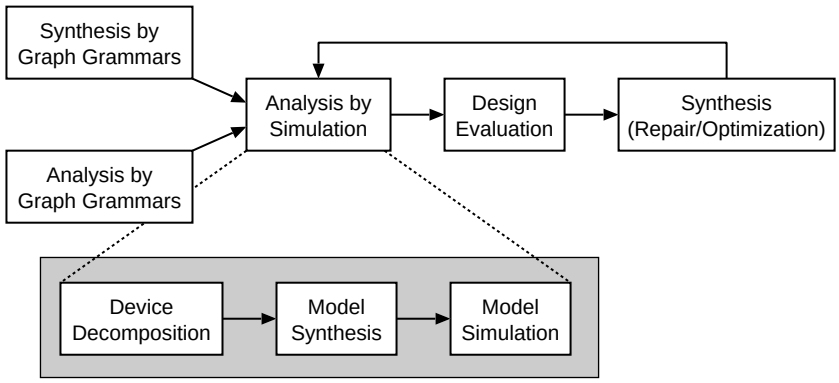


Figure 26: Steps belonging to simulation of a system.

These steps can be described in the following way:

- *Device Decomposition*. This step encompasses the retrieval of the mathematical relations describing the physical properties belonging to the separate devices. At this stage these mathematical relations are collected independently and not coupled in any manner.
- *Model Synthesis*. The mathematical relations collected in the previous step are coupled to form a model describing the system at hand. This step corresponds to a nontrivial process that requires, depending on the modeling depth, a large degree of domain knowledge.
- *Model Simulation*. Within this phase the equation system derived in the last step is solved, yielding results that have to be evaluated in order to decide on the plant's functionality or any necessary corrections.

4.3.2 Qualitative Simulation

The underlying model of our approach, in contrast to the structure generated by the graph grammar, is not abstract, but closer to the physical level. This means that at this level we no longer deal with simplified substance properties or abstract device families, but with crisp substance values and concrete device parameters. However, we still restrain ourselves from

actually working at the physical level, which implies dealing with differential equations, numerical algorithms etc. like the traditional approaches. For our approach the steps belonging to the simulation phase could be described as follows:

- *Device Decomposition*. This step consists of the transition from the abstract level to the concrete parameter level. Every abstract device representation is enriched with the concrete parameters, transforming it into a concrete device. As in the case of traditional systems, this is an information retrieval step.
- *Model Synthesis*. In contrast to the model synthesis of traditional systems, we take the design's structure as a basis to combine the concrete devices.
- *Model Simulation*. Instead of solving complex mathematical equations, model simulation here consists of the execution of functions representing the different transformations performed by the devices and the propagation of these results throughout the structure.

Remarks. In contrast to most traditional approaches, we work solely at the device level. Put in other words, we regard devices as atomic and do not perform any further decomposition, i. e., devices are not broken down into components that do not represent or perform any essential function.

Remarks. Note that within the domain of chemical engineering a mathematical model of a plant design may not exist at all—some aspects still lack a mathematical model and are regarded as *black boxes*. Since our simulation approach works at a higher level, usable results may still be produced.

4.3.3 Complexity of Design Evaluation

As argued in section 5.3, the graphs generated by our approach are topologically restricted—they lack cycles, are directed, and generated by means of mostly context-free rules. Thus, the time complexity to simulate a generated design is linear in the number of nodes, since a topological search is sufficient to visit all nodes in the appropriate order.

Taking the computational effort for the simulation of a device into account, the total computational effort amounts to $O(n \cdot D)$, where D is the maximum effort necessary to simulate a device.

5 Synthesizing Systems

In section 3 we presented a brief introduction to graph grammars, which, depending on their type, provide the necessary mechanisms to solve various design tasks, which are depicted in Figure 27. Now we give an overview of the synthesis approach, which pursues the goal of generating a system from scratch.

Remarks. Note that by synthesis we mean the generation of designs, including structure definition, choice of abstract devices and model synthesis. Due to model simplification, such a

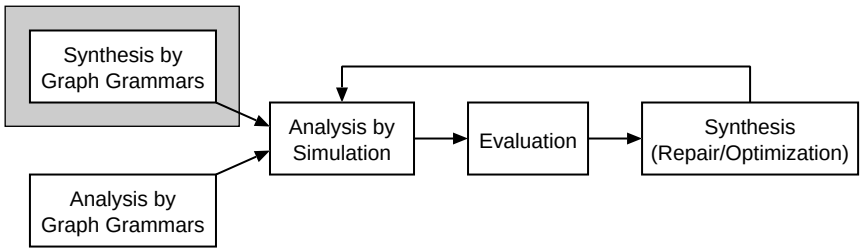


Figure 27: The design cycle.

generated design may not fulfill the demands properly, making some repair or optimization steps necessary. Repair and optimization, although implying synthesis steps, are not dealt with here, but in chapter 6. A more in-depth view on this topic can be found in [Stein, 2001].

5.1 Structure Synthesis by Graph Grammars

In section 2.2 we described the part of the design process that is responsible for structure generation, and in section 2.3 the generation of the underlying model was addressed, together with some other topics such as optimization. Here we concentrate on the structure synthesis, i. e., we care only for the graph grammar related part of the synthesis process.

The synthesis of a chemical plant structure is, as mentioned previously, based on the given input and the desired output. Thus, the generation process begins with an abstract design represented by a single nonterminal node to which edges describing the given inputs and the desired output are connected. The simple algorithm described below reflects this idea.

```
   boolean GENERATE(DESIGN graph, DESIGNGRAPHGRAMMAR 𝒢)
(1)    {
(2)      DESIGN d;
(3)      boolean done := false;
(4)      boolean more_derivations := true;
(5)      while (not done and more_derivations) do
(6)        d := graph;
(7)        more_derivations := APPLYNEWRULESEQUENCE(𝒢, d);
(8)        if (TASKSOLVED(d)) then
(9)          done := true;
(10)       fi
(11)     od
(12)     return done;
(13)   }
```

The procedure APPLYNEWRULESEQUENCE applies graph transformation rules repeatedly to the given design until it reaches a terminal state. The design d then represents a chemical plant proposal that has to be verified by means of the function TASKSOLVED. Besides changing the design, the procedure APPLYNEWRULESEQUENCE returns a boolean value sig-

naling the existence of a further derivation. Each call to APPLYNEWRULESEQUENCE yields a new derivation, if available.

5.2 Caramel Syrup Example Reviewed

In section 2 we described the design procedure for a caramel syrup process and presented a solution to this problem from the point of view of an engineer. Now, we will use a graph grammar to attain the same goal. For this purpose, the graph rules depicted by Figures 17 – 22 in section 4.2 are given. Finally, Figure 28 shows a derivation that produces a feasible design.

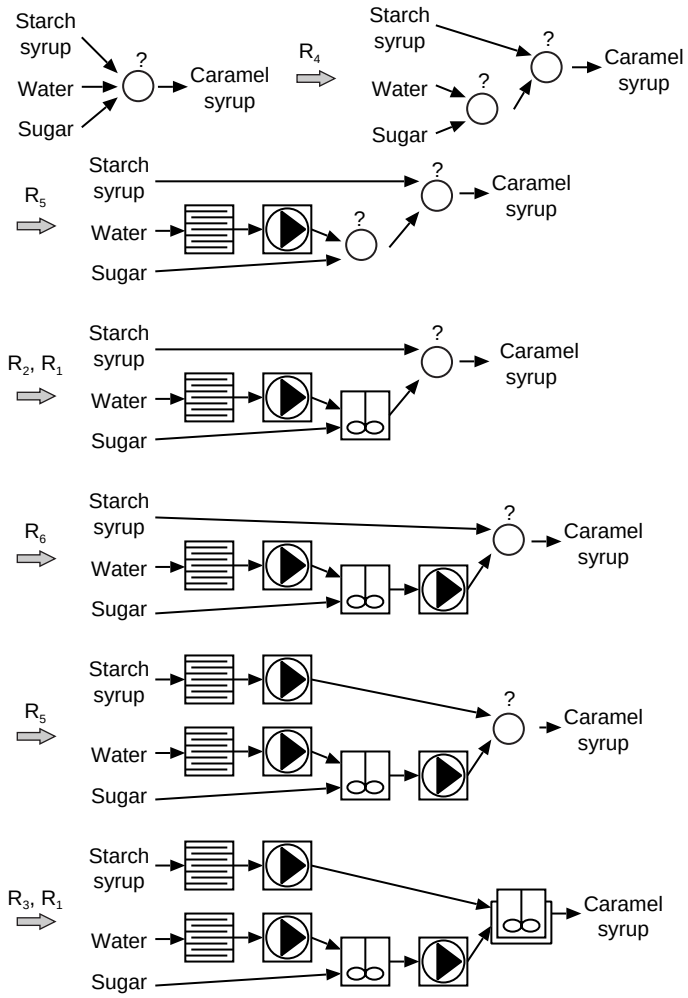


Figure 28: Derivation of the caramel plant design.

In general there will be a series of rules that apply for a given situation (i. e., sentential form), leading to different solutions of varying quality and cost. Thus, the generation process can be viewed as a tree containing derivations for all possible alternatives, as shown in Figure 29. Note that the graph grammar derivation of Figure 28 corresponds to one branch of this tree.

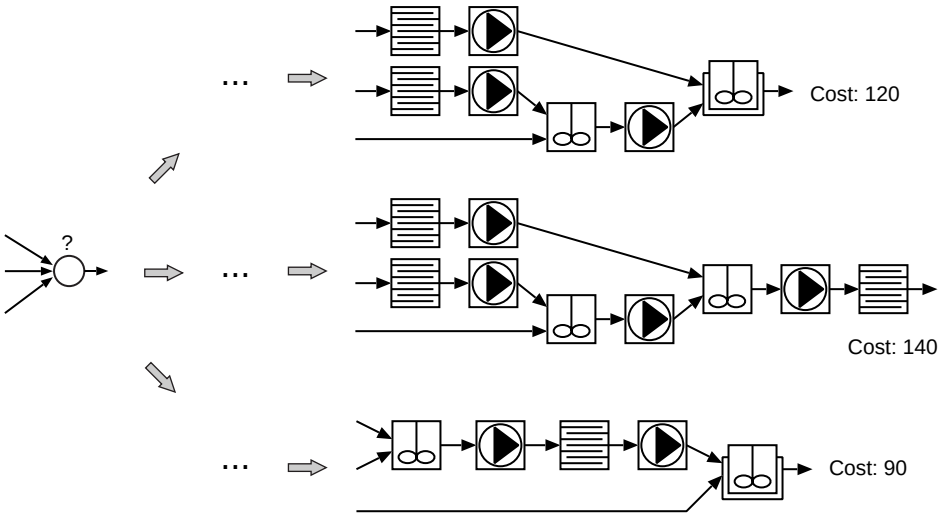


Figure 29: Search tree for the optimization of the design generation process.

Finally, the structure generated is completed into a design by adding the information from the underlying model to the abstract layer. At this point, the synthesis process is finished. Subsequent simulation and evaluation steps decide if the proposed design fulfills the demands adequately or if it requires some adjustments to do so.

5.3 Graph Topology Restrictions

In practice, plants often combine various chemical processes within one single "chain" producing one main product and a series of by-products. This means that the topology represents at least a directed acyclic graph, which can only be generated by a context-dependent graph grammar. However, such grammars imply exponential time complexity, as rules may have more than one nonterminal on the left-hand side (subgraph matching problem, see [Garey and Johnson, 1997]). In order to avoid this drawback, we restrict—as far as possible—the set of graph production rules to context-free rules, which generate a graph in polynomial time [Brandenburg, 1994].

Another restriction that has already been discussed in section 2.1 is enforced by avoiding cycles within a design. Cycles not only hinder an efficient graph grammar processing, but also make simulation more complicated. Through aggregation of cycles the expected time complexity can be substantially reduced.

These restrictions have far-reaching implications for both system synthesis steps. On the one hand they guarantee a better performance due the simplifications imposed by the restrictions, and on the other hand they rule out certain complex structures and more exact simulation results.

6 Design Language

Design repair or optimization is necessary in many cases: Design generation typically produces, due to the simplifications applied to the model, faulty designs requiring some adjustments to become feasible; simulation of manually created designs often reveals deficiencies that must be overcome by means of repair operations; and in some situations designs of technical systems may be optimized to reach higher efficiency and lower costs. A design language in which repair and optimization knowledge is formulated can easily tackle these problems. Figure 30 shows the areas of the design cycle affected by repair and optimization by means of a design language.

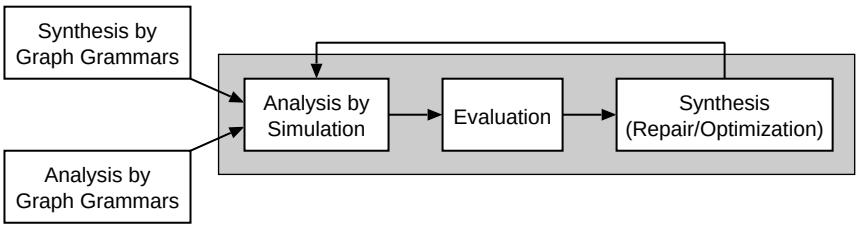


Figure 30: The design cycle.

At this point we only show the requirements that a design language has to fulfill, and refrain from presenting a concrete language specification. Additionally, we address some enhancements to the graphical representation of the design graph grammar.

6.1 Requirements

In this section we address some issues that arise within the context of a design language for repair and optimization of technical systems: the types of possible modifications that are available and the search for appropriate modification application contexts.

6.1.1 Modification Types

As stated above, the design language shall provide means to formulate repair and optimization transformations, which are only needed if some type of fault or insufficiency is detected. Hence, the most important construct of the design language will be of the form *observation* $\rightarrow$ *adjustment*, or, put in other words, *symptom* $\rightarrow$ *remedy*, i. e., rules compose the main part of the design language.

There are different types of modifications for different types of symptoms [Stein and Vier, 1998], which differ in their gravity and range of context:

- *Local modifications*. Local modifications are component-based, i. e., they apply changes to a single component, modifying its behavior, and ignore the component's neighborhood. We differentiate between the following two local modification types:

  - *Parameter modification*. This type of modification corresponds to a simple change of a parameter setting, like increasing the power throughput or changing the dimensions of the vessel of a mixer.
  - *Characteristics modification*. This type of modification is more radical in nature and corresponds to a replacement of a device with another, more fitting device. This modification is only necessary if a parameter modification fails to correct the problem.

- *Global modifications*. Global modifications are related to non-locatable symptoms, i. e., symptoms that are not bound to a specific component but to the system as a whole. These modifications require changes to the system topology by means of addition, deletion or reordering of components. Note that such a modification may correspond to a change of characteristics (local modification due to a non-locatable fault).

6.1.2 Modification Location

The most important issue concerning design repair and optimization is, as observed in [Stein and Vier, 1998], determining *where* a modification is to occur. Obviously, the difficulty to find the modification location depends on the symptom detected—if the fault is component-based, then the location is known; if the fault is non-locatable, then the location for the modification must be searched.

Stein and Vier introduce the notion of *location specifiers* for their design language [Stein and Vier, 1998]. A similar mechanism is also required for our design language, but, instead of adding a new concept to our approach, we resort once again to graph grammars. To this avail, we allow the formulation of repair rules as tuples of the form "(Fault Candidate, Modification, Additional Actions)", where

- *fault candidate* is the location description of a possible modification site and represents the left-hand side of a rule,
- *modification* describes the change to be applied and represents the right-hand side of a rule, or may be empty, i. e., nothing is changed,
- and *additional actions* are low level—or domain specific—actions to be performed together with the graph transformation and are required for parameter settings etc. This mechanism is required to manipulate the underlying model of the design.

Now, instead of performing a search for the fault candidates within the faulty design, the graph grammar's rule-based behavior is exploited. For every symptom a set of remedy rules is built and activated—the search is then performed by the rule processing engine. After one rule has fired, the design has to be simulated once again in order to check if the problem has been solved; if the fault persists or another fault emerges, then the process is reiterated.

6.2 Semantics of Graphical Representation

The graph grammar model introduced in chapter 3 is actually perfectly suited for any conceivable transformation necessary within the scope of design generation, system analysis or design repair and optimization. However, the classical graphical representation of graph transformation rules is not able to reflect the use of some special features of design graph grammars. Additionally, we introduce some graphical features that aim at a better understanding of the rules.

One such aspect pertains to the edges incident to a target node. In many cases only a subset of the incident edges is of interest, the remaining edges are irrelevant. By means of the embedding instructions one can ascertain that these irrelevant edges are restored; however, this is not visible within the graphical representation used so far. Thus, we introduce a new graphical representation for such cases: A dotted edge represents any number of edges that may exist beyond the ones specified explicitly. For readability purposes, the rule designer may also add labels to such edges, such as $0..n$ or $s_1, \ldots, s_m$; these labels do not have any further meaning.

Apart from the dotted edges described above, we also allow the graphical representation of labeled "dangling edges" when appropriate. These edges do not interfere with the matching process and are only relevant for the embedding step. The purpose of this "feature" is to improve readability and stress the importance of these edges for the embedding process. In most cases we will refrain from drawing labeled dangling edges.

Another aspect that has not been addressed yet is the edge label complexity allowed. Within our approach, edge labels correspond to abstract substance properties, such as $v_h t_l$ which means "viscosity high and temperature low". Depending on the number of properties a label has to encompass, a large number of combinations may be the result. Thus, we allow rule edges to match host graph edges whose labels are *subsumed* by the rule edge labels.

Figure 31 illustrates the use of template edges and label subsumption.

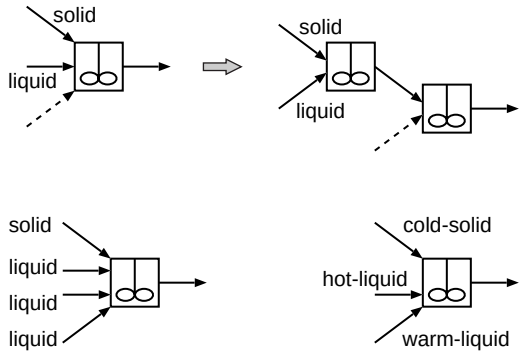


Figure 31: Rule with two possible matches.

Finally, we allow the use of different node representations. In general, nodes are depicted by circles, and their labels are placed outside the circle. Although this representation is sufficient for our needs, the use of specific graphical symbols representing devices of a concrete domain is acceptable; in fact, such graphical symbols correspond to a combined representa-

tion of nodes together with their type defining labels. Note that we also allow the mixed use of graphical representations.

6.3 Caramel Syrup Example Reviewed Again

We now take a further look at the caramel syrup example, whereas emphasis is now laid on the *synthesis-simulation-evaluation* cycle and not on one step alone. For the sake of simplicity, we assume there is only a single fault that has to be corrected, limiting the number of cycle iterations to one.

The design steps undertaken for the solution of the caramel syrup task are[4]:

1. *Demands*. Instead of using relative mass values, as in section 2.2, we now give concrete amounts: 15kg sugar, 45l water and 40l starch syrup.

2. *Synthesis*. The synthesis consists of the generation of a structure, based on the given demands, and the enhancement thereof with concrete device data, which represents the underlying model.

   (a) *Structure generation*.

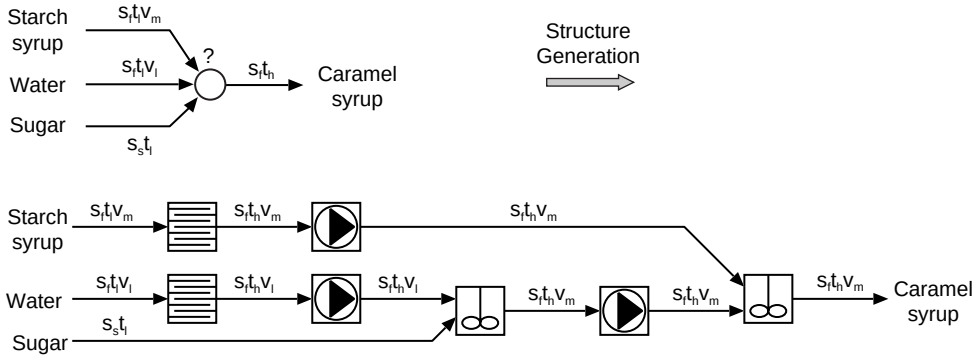


Figure 32: Generation of the caramel syrup design structure.

   Figure 32 shows the structure resulting from the generation process. Note that during the generation process label items may be appended to an existing label, and that the opposite does not occur.

   (b) *Behavior model generation*.

   Figure 33 shows the generated structure together with an excerpt of the underlying model.

3. *Simulation*. Now, substance and mixture values and the results of the functions performed by the devices are propagated throughout the design structure, as shown in Figure 34.

[4]The edge labels are: *t* for "temperature", *v* for "viscosity", and *s* for "state". The subscripts are qualifiers and mean: *s* for "solid", *f* for "fluid", *g* for "gas", *l* for "low", *m* for "medium", and *h* for "high".

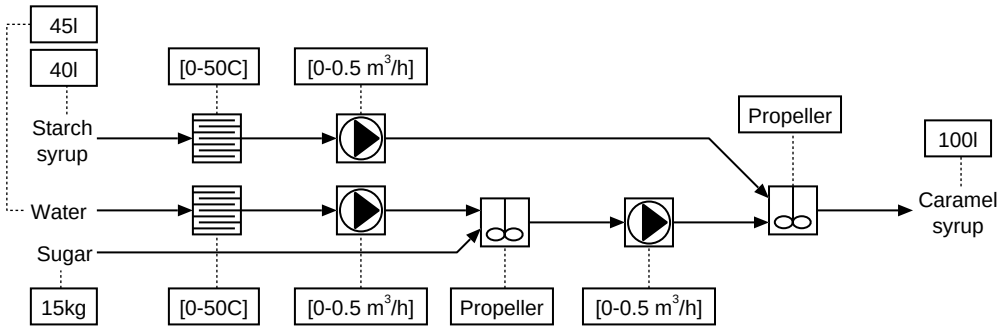


Figure 33: Caramel syrup design structure with underlying model information.

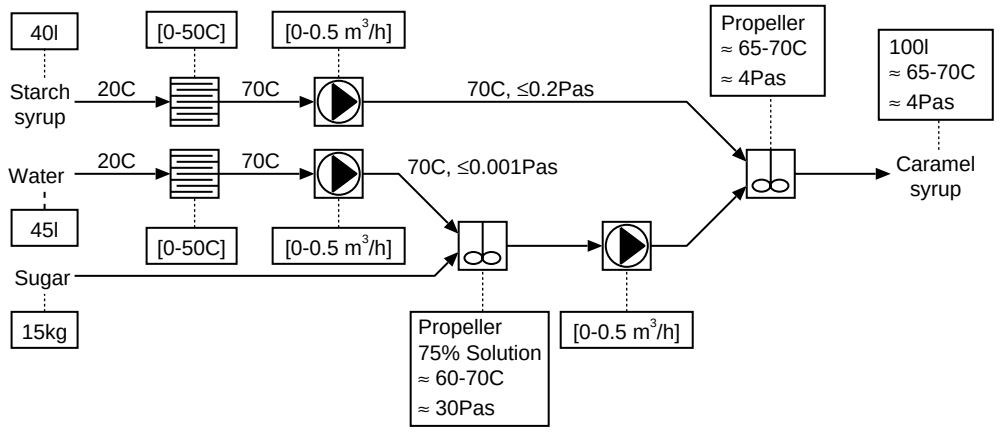


Figure 34: Simulation of the caramel syrup design: propagation of properties and values.

4. *Evaluation*. The results of the simulation and the task requirements are compared to determine if the actual design fulfills the demands adequately. The first (and in our case only) observation is that the output, as produced by our design, is not hot enough—the required output temperate was 110°C, i. e., the output needs to be heated by at least 45°C. The following repair actions compose the actual choice list for this situation:

   - Increase the power of one or more heat transfer units (*parameter modification*).
   - Replace one or more heat transfer units with more powerful devices; alternatively, replace an agitator with one containing a built-in heat transfer device (*characteristics modification*).
   - Insert an additional heat transfer unit to the design (*global modification*).

   In the present case a parameter modification is not possible, since the heat transfer units are already working at maximum power (Δt =50°C). The next simplest change would be the replacement of a device—we choose to use an agitator with a built-in heat transfer unit, as shown in Figure 35.

   If a repair step was necessary, then the process continues with the simulation step, otherwise the design is considered feasible and the design cycle is interrupted here.

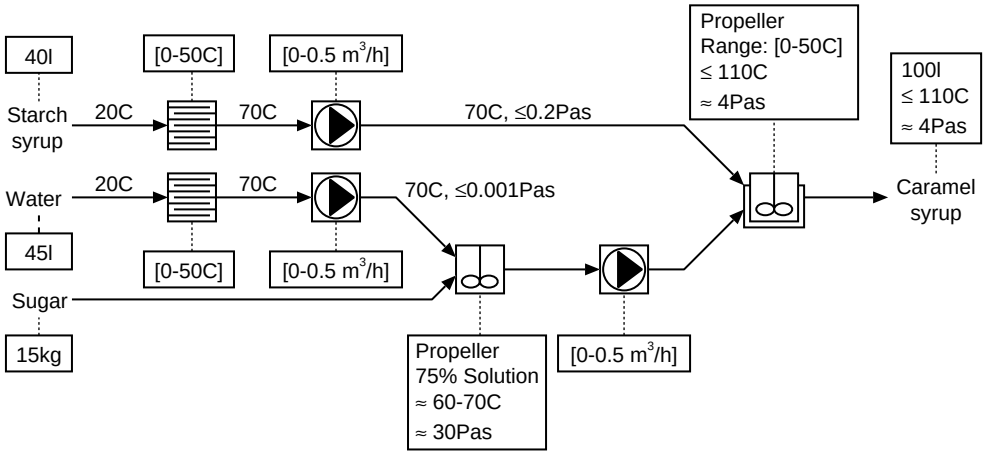


Figure 35: Repaired caramel syrup design showing values after simulation.

7 Theoretical Considerations of Design Graph Grammars

The development and use of design graph grammars (DGGs) is connected to various theoretical issues, which are addressed in this section. Firstly, the relationship of design graph grammars to the classical graph grammars is examined. Secondly, graph-theoretical issues concerning both graph grammars in general and the application to technical systems design are presented.

7.1 Relationship Between Classical Graph Grammars and Design Graph Grammars

An important question pertains to the justification of the development of design graph grammars, which represent a further graph transformation formalism amidst other existing graph grammar concepts. In the following we describe the two general approaches to graph transformation along with their most prominent graph grammar representants and point out their advantages and disadvantages. Some attention is also paid to hybrid concepts, which try to combine the aforementioned approaches. We then compare design graph grammars with the classical graph grammars and establish their relationship.

7.1.1 The Connecting Approach

The connecting approach is a node-centered concept that aims at the replacement of nodes or subgraphs by graphs. The item to be replaced is deleted along with all incident edges, and the replacement graph is embedded into the host graph by connecting both with new edges. These new edges are constructed by means of some mechanism that specifies the embedding.

In the literature graph grammars are often distinguished by the size of the left-hand sides of rules, leading to two approaches of inherently different complexity: node replacement

and graph replacement graph grammars, called node-based and graph-based, respectively. Additionally, each of these two approaches is divided into context-free and context-sensitive subclasses.

Several graph grammars follow the connecting approach. According to [Engelfriet and Rozenberg, 1997], the most well-known node replacement graph grammar families are the *node label controlled* (NLC) and the *neighborhood controlled embedding* (NCE) graph grammars, whose node-based versions we describe in the following.

NLC Graph Grammars

Node label controlled graph grammars represent a very simple node replacement mechanism used to perform graph transformations on undirected node-labeled graphs. A graph transformation step is based merely on node labels, i. e., there are no application conditions or contexts to be matched. The embedding is determined by a set of embedding instructions shared by all graph transformation rules. The following definition resembling the one of [Engelfriet and Rozenberg, 1997] introduces NLC grammars formally[5].

Definition 7 *(NLC Graph Grammar)*

A NLC graph grammar is a tuple $\mathcal{G} = \langle \Sigma, P, I, s \rangle$ *with*

- Σ *is the set of nonterminal and terminal labels,*
- P *is the finite set of graph transformation rules or productions of the form* $t \rightarrow R$*, where* $t \in \Sigma$ *and* R *is a labeled graph,*
- I *is an embedding relation, and*
- s *is the initial symbol.*

The embedding relation is a binary relation over Σ*. Each embedding instruction* (h, r) *states that the embedding process should create an edge connecting each node of the replacement graph labeled* r *with each node of the host graph labeled* h *that is a neighbor of the target node.*

Remarks. The domain of technical systems imposes a series of requirements, of which some cannot be met by NLC grammars due to weaknesses of this mechanism:

1. There is no way to specify a context. Therefore, it is not possible to distinguish between different situations related to one single item.
2. There is no way to distinguish between individual nodes in the replacement graph, since the embedding mechanism relies solely on labels.

[5] In the literature it is usually distinguished between different label alphabets. For the sake of simplicity, we use one single alphabet including all necessary label types.

NCE Graph Grammars

Neighborhood controlled embedding graph grammars represent a more sophisticated node replacement mechanism[6] used to perform graph transformations on directed or undirected labeled graphs[7]. A graph transformation step is based on node labels and edge labels, thus providing further discerning power. The embedding is determined by a set of embedding instructions belonging to each graph transformation rule. The following definition [Engelfriet and Rozenberg, 1997] introduces NCE grammars formally.

Definition 8 *(NCE Graph Grammar)*

An NCE graph grammar is a tuple $\mathcal{G} = \langle \Sigma, P, s \rangle$ *with*

- Σ *is the alphabet of node and edge labels, and includes terminal and nonterminal labels,*
- P *is the finite set of productions, and*
- s *is the initial symbol.*

The productions of the set P *are tuples of the form* $t \rightarrow \langle R, I \rangle$ *with*

- $t \in \Sigma$ *is the label belonging to a node* v *in the host graph,*
- $R = \langle V_R, E_R, \sigma_R \rangle$ *is the non-empty replacement graph,*
- I *is the set of embedding instructions for the replacement graph* R *and consists of tuples* $(h, e/f, r)$*, where*
  - $h \in \Sigma$ *is a node label and* $e \in \Sigma$ *is an edge label in the host graph,*
  - $f \in \Sigma$ *is another edge label,*
  - *and* $r \in V_R$ *is a node of the replacement graph.*

An embedding rule $(h, e/f, r)$ *has the same meaning as in definition 5, where it is written as* $((h, t, e), (h, r, f))$.

Despite their superiority over NLC graph grammars, NCE graph grammars do not cope either with the requirements of the tasks of the domain of technical systems:

1. The mechanism for context specification is weak, since a context is restricted to incident edges.
2. Contexts cannot be specified explicitly; they are formulated within the embedding instructions. Such contexts cannot provide any means of rule application control.

Again, these drawbacks of the NCE graph grammar mechanism lead to the conclusion that this concept is not powerful enough to tackle design tasks in the domain of technical systems. Compared to the NLC approach, though, the NCE approach is more powerful.

[6] NCE grammars are an extension of NLC grammars.

[7] In the literature on the subject, NCE grammars with and without edge labels and edge directions are distinguished by the prefixes "*e*" (for edge labels) and "*d*"(for directed edges) added to the NCE acronym. Thus, there are NCE, eNCE, dNCE and edNCE graph grammars. For the sake of simplicity, we omit these prefixes.

7.1.2 The Gluing Approach

The gluing approach is an edge-centered concept that aims at the replacement of hyperedges or hypergraphs by hypergraphs. Each hyperedge or hypergraph possesses a series of attachment nodes which represent the interfaces to the outer world. Within a replacement step, the item to be replaced is deleted from the host hypergraph with exception of the attachment nodes, which are at the same time external nodes of the host hypergraph, and the new hypergraph is embedded in its place by unifying (gluing) its attachment nodes with the external nodes.

As in the connecting approach case, there also exist several hypergraph grammar types following the gluing approach. The most well-known family is called *hyperedge replacement* (HR) grammar.

HR Grammars

Similarly to the connecting approach case, where node-based and graph-based grammars are distinguished, we distinguish between hyperedge-based and hypergraph-based HR grammars. Hyperedge-based HR grammars are defined as in [Drewes et al., 1997].

Definition 9 *(Hyperedge Replacement Grammar)*

A hyperedge replacement grammar is a tuple $\mathcal{G} = \langle \Sigma, P, s \rangle$ *where*

- Σ *is the set of terminal and nonterminal labels*[8]*,*
- P *is a finite set of productions over* Σ*, each of which has the form* $T \rightarrow R$*, where* T *is a hyperedge label and* R *is the replacement hypergraph,*
- *and* $s \in \Sigma$ *is the initial symbol.*

Remarks. HR grammars are intrinsically context-free, since the item of the left-hand side of a rule is completely deleted and replaced by the hypergraph of the right-hand side. In order to introduce matching-level context—a context that serves as an application condition—into this concept one would have either to extend HR grammars or to integrate the context into the target hyperedge or hypergraph. This means that the context would have to be deleted and restored by means of the replacement hypergraph.

Hyperedge replacement grammars are not powerful enough for the design tasks envisioned. The following weaknesses hinder the use of this concept:

- HR grammars lack the concept of matching-level context. This fact implies that a required context has to be integrated within the target and replacement hypergraphs.
- HR grammars are weaker than the NCE grammars of the connecting approach in their expressiveness (see [Engelfriet and Rozenberg, 1997], page 4).

[8] Only the hyperedges are labeled in HR grammars.

7.1.3 Hybrid Approaches

Apart from design graph grammars there exist other hybrid approaches in the literature. In [Courcelle et al., 1993] the authors present another hybrid graph grammar, the *handle hypergraph grammar*. This hybrid graph grammar, as the name already implies, is based on the hyperedge replacement approach and has some node replacement features. A similar approach that has a simpler rewriting mechanism is the *HR grammar with eNCE rewriting*, presented in [Kim and Jeong, 1996]. Another hybrid approach, the *hypergraph NCE graph grammar*, is introduced in [Klempien-Hinrichs, 1996]. This concept is based on the node replacement approach. In the following we briefly describe these approaches and address their usability for design purposes.

Handle Hypergraph Grammars

Handle hypergraph (HH) grammars rewrite handles. A handle is an edge (or hyperedge) together with all of its nodes. Within a hypergraph transformation step a handle is deleted, including all incident edges; the replacement hypergraph is then embedded into the host hypergraph by means of embedding instructions based on the connecting approach [Courcelle et al., 1993, Engelfriet and Rozenberg, 1997].

HH grammars have a simpler embedding mechanism than traditional node replacement graph grammars, since the deleted edges incident to the handle do not have to be distinguished but only restored; therefore, there is no edge relabeling. On the other hand, a handle is the smallest item that can be replaced; this means that rules have to match and delete at least one edge and two nodes (excluding incident edges from the host hypergraph), whereas in the design graph grammar approach the smallest item is a single node, which seems more flexible. Since the domain of technical systems requires a node centered mechanism and due to the above disadvantages, we conclude that this concept does not meet our requirements.

Hypergraph Replacement with eNCE Rewriting

Hypergraph replacement grammars with an eNCE way of rewriting (HRNCE) are handle-rewriting grammars like the HH grammars described above. HRNCE grammars possess a simple structure and are as easy to use as NLC grammars, but are still powerful in terms of expressiveness [Kim and Jeong, 1996].

Again, the design tasks imposed by the domain of technical systems require a node centered concept, and, although HRNCE grammars represent a powerful manipulation mechanism, they share the same disadvantages with the HH grammars. Thus, HRNCE grammars are not fitting for our purposes.

Hyperedge Neighborhood Controlled Embedding Graph Grammars

Hyperedge NCE (hNCE) graph grammars generalize classical NCE grammars by extending the traditional approach to handle hyperedges instead of ordinary edges. The necessity for

this enhanced NCE grammar arises from the need to perform special hypergraph transformations, which cannot be expressed by hyperedge or handle rewriting in hypergraphs or by node replacement in bipartite graphs, on Petri nets [Klempien-Hinrichs, 1996].

Basically, an hNCE grammar works exactly like an NCE grammar: A nonterminal target node, together with all incident hyperedges, is deleted. Then, the replacement hypergraph is embedded into the host graph by adding hyperedges that are created in compliance with the embedding instructions. hNCE grammars can generate the same graph languages generated by HR grammars, and, according to the author of [Klempien-Hinrichs, 1996], hNCE grammars are assumed to have at least the same generative power as S-HH grammars[9].

hNCE grammars, although apparently versatile and powerful in terms of expressiveness, only extend the NCE concept to hyperedges and hypergraphs. This additional functionality is not required for the tasks of the domain of technical systems, and only adds further overhead, since the use of hyperedges and hypergraphs make the formulation of graph transformation rules cumbersome. Thus, this concept is not adequate for our needs.

7.1.4 Design Graph Grammars

As seen in the previous sections, neither classical node replacement graph grammars nor hyperedge replacement grammars provide sufficient expressive power for design tasks in the domain of technical systems; even the powerful hybrid approaches proved to be inadequate for our needs. Design graph grammars, on the other hand, encompass the benefits of the connecting and gluing approaches:

- *Context/Matching*. Precise matching through explicit context (HR)
- *Replacement paradigm*. Concise formulation of node-based graph transformation rules (NLC/NCE)
- *Embedding*. Access to individual nodes of the replacement graph (NCE)

Furthermore, design graph grammars add some features of their own:

- *Embedding*.
  - Extended replacement graph formulation
  - Enhanced embedding instructions
  - Flexible rule formulation by means of variable labels
- *Matching*. Distinction between classical and strict degree matchings

Remarks. Design graph grammars represent a hybrid approach combining the strengths of node replacement and hyperedge replacement grammars while overcoming their weaknesses. The core of the design graph grammar approach is node replacement based, though.

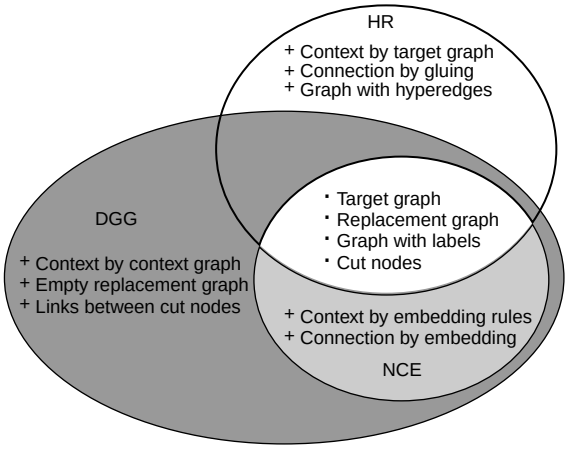


Figure 36: Features of the different graph grammar concepts.

Figure 36 shows the relationship of design graph grammars to the classical grammars with respect to their features.

In the following we present some formal results that establish the relationship between design graph grammars and the classical graph and hypergraph grammars. Let $\mathcal{L}_{Class}$ denote the set of graph languages $L(\mathcal{G})$ that can be generated by some graph grammar $\mathcal{G} \in Class$.

Theorem 1 *($\mathcal{L}_{NLC} \subseteq \mathcal{L}_{DGG}$)*

Every NLC graph language generated by a node-based NLC graph grammar can be generated by a node-based, context-free design graph grammar.

Proof. Let an arbitrary NLC grammar $\mathcal{G} = \langle \Sigma, P, I, s \rangle$ for an NLC graph language L be given. We construct a node-based, context-free DGG $\mathcal{G}' = \langle \Sigma', P', s' \rangle$ based on $\mathcal{G}$ such that $L(\mathcal{G}') = L$.

Obviously, $s' = s$ and $\Sigma' = \Sigma$. The set of graph transformation rules P' is defined as follows. P' contains a graph transformation rule $r' : t \to \langle R, I' \rangle$ for each $r \in P$ with $r : t \to R$. The embedding instruction set I' of each $r' \in P'$ contains the same embedding instructions as the set I; therefore, the embedding instruction set of each rule in P' is identical (in order to simulate the global set I of $\mathcal{G}$). For each embedding instruction $i \in I$ with $i = (h, r)$ there is an embedding instruction $i' \in I'$ with $i' = ((h, t, \perp), (h, v, \perp))$, where $\sigma_R(v) = r$. It is clear that $L(\mathcal{G}') = L$. ◇

Theorem 2 *($\mathcal{L}_{NCE} \subseteq \mathcal{L}_{DGG}$)*

Every NCE graph language generated by a node-based NCE graph grammar can be generated by a node-based, context-free design graph grammar.

Proof.

Taking an arbitrary NCE grammar $\mathcal{G} = \langle \Sigma, P, s \rangle$ for an NCE graph language L as a starting point, we construct a node-based, context-free DGG $\mathcal{G}' = \langle \Sigma', P', s' \rangle$ whose generated language $L(\mathcal{G}') = L$.

[9] S-HH grammars are *separated HH* grammars, i. e., no two nonterminal hyperedges are adjacent in the right-hand side of hypergraph transformation rules or in the initial symbol or hypergraph.

Due to the strong similarity between both concepts, the construction is straightforward. We set $\Sigma' = \Sigma$, $P' = P$ *and* $s' = s$. *The graph transformation rules are identical in syntax and semantics for both concepts; only the syntax of the embedding instructions differ: For each NCE embedding instruction* $i \in I$ *with* $i = (h, e/f, r)$ *there is a DGG embedding instruction* $i' \in I'$ *with* $i' = ((h, t, e), (h, r, f))$. *Obviously,* $L(\mathcal{G}') = L$ $\diamond$.

Theorem 3 $(\mathcal{L}_{HR} \subseteq \mathcal{L}_{DGG})$

Every HR language generated by a hyperedge-based HR grammar can be generated by a node-based, context-free design graph grammar, if hypergraphs are interpreted as bipartite graphs.

Proof. According to [Engelfriet and Rozenberg, 1990] and [Engelfriet and Rozenberg, 1997], page 57pp., $\mathcal{L}_{B_{nd}-edNCE} = \mathcal{L}_{HR}$. *Hence, HR languages generated by HR grammars can be generated by* nonterminal neighbor deterministic *boundary edNCE grammars, which in turn can be simulated by DGGs, since* $\mathcal{L}_{B_{nd}-edNCE} \subseteq \mathcal{L}_{B-edNCE} \subseteq \mathcal{L}_{edNCE}$. *Thus, DGGs can generate HR languages and it follows that* $\mathcal{L}_{HR} \subseteq \mathcal{L}_{DGG}$. $\diamond$

Figure 37 summarizes the above statements by illustrating the expressive power of design graph grammars.

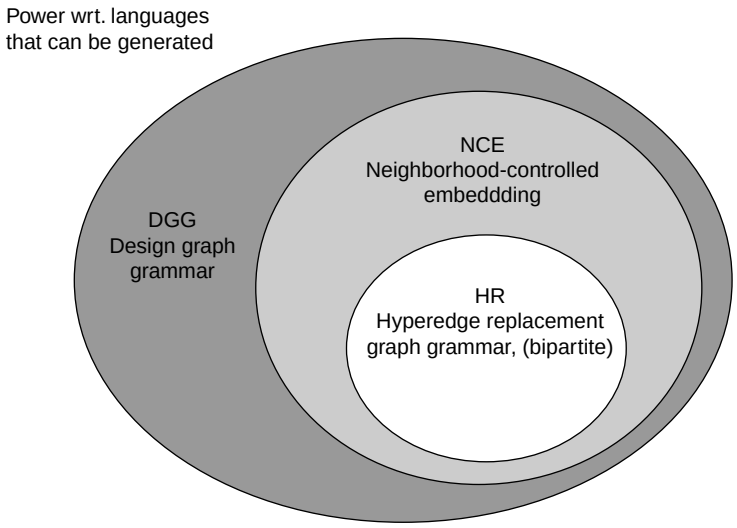


Figure 37: The expressive power of design graph grammmars.

7.2 Relationship to Programmed Graph Replacement Systems

Design graph grammars as proposed here shall enable domain experts to formulate design expertise for various design tasks. Design graph grammars result from the combination of different features of the classical graph grammar approaches, while special effort has been spent to keep the underlying formalism as simple as possible.

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[10], for instance, offers declarative and procedural elements [Schürr, 1989, 1991] for data flow oriented, object oriented, rule-based, and imperative programming styles. A direct comparison between PROGRES to 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—say: one possibility—to realize a design graph grammar by reproducing its concepts. In this connection PROGRES fulfills the requirements of design graph grammars for the most part. However, PROGRES lacks the design graph grammar facilities for the formulation of context, deletion operations, and matching control, which have to be simulated by means of complex rules. Such a kind of emulation may be useful as a prototypic implementation, but basically, it misses a major concern of design graph grammars: Their intended compactness, simplicity, and adaptivity with respect to a concrete domain or task.

7.3 The Problem of Matching

Matching is a vital part of any rule-based concept. In order for a graph transformation rule to fire it is necessary that a matching of the left-hand side be found within the host graph. Additionally, the embedding process requires that individual nodes be matched so that edges can be drawn between them.

Matching is already a nontrivial issue in the context-free, graph-based case, as implied by the subgraph matching problem described in section 7.3.2. The inclusion of context adds to the complexity of matching, because node-based matchings with context are then comparable to graph-based matchings. With context even the simplest case becomes nontrivial.

The specification of context as a means to restrict the application of a graph transformation rule to a certain situation is an essential requirement for design purposes. In the following the different types of context are examined and the resulting consequences identified. Then, the subgraph matching problem, a problem also related with context, is described as well as a measure to diminish its effect. Finally, we address the problem of context within backward execution of graph transformation rules.

7.3.1 Context and Its Consequences

As stated above, we differentiate between matchings with context and without context. This leads to the following classification:

1. *Node-based matching without context*. This type of matching pertains to a single node and disregards its context completely. Within a graph, a matching of a certain node takes at the most linear time in the size of the graph.

[10] We chose PROGRES for illustration purposes only; the line of argument applies to other tools such as PAGG (see [Schürr, 1997b] for a brief description and further pointers) or Fujaba [Nickel et al., 2000] as well.

2. *Node-based matching with incident edges*. This type of matching yields a single node together with incident edges, which represent a very small and restricted context. The search for such a node requires linear time in the size of the graph, if it can be assumed that node degree is bounded by a constant, which is usually the case.

3. *Node-based matching with context graph*. A matching of this type includes a node and a nontrivial context, which size is only bounded by the host graph itself. Thus, the most expensive matching can be achieved in the node-based case.

4. *Graph-based matching without context*. Graph-based matchings without context share the same worst-case complexity as the previous case. In average, though, one can expect this type of matching to be of a larger scope, and therefore more expensive.

5. *Graph-based matching with context graph*. This type of matching represents the most difficult case and shares the same worst-case complexity as the previous matching type. However, since context has to be matched as well, it is to be expected that this type of matching behaves worse than the graph-based matching without context in the average case.

For obvious reasons one should avoid formulating graph transformation rules more complex than case 2. However, in many cases, especially with respect to repair and optimization, graph transformation rules with nontrivial matchings are required.

7.3.2 The Subgraph Matching Problem

Subgraph matching[11] is a widely known NP-complete problem [Garey and Johnson, 1997, Köbler et al., 1993], and therefore no algorithm implementing a solution to this problem will run in polynomial time, assuming that $NP \neq P$.

One can make use of additional information to accelerate the subgraph matching step, whereas the problem and its complexity remain unchanged. In [Bunke et al., 1991a,b] the authors describe an efficient graph grammar implementation based on the Rete algorithm [Forgy, 1982, Forgy and Shepard, 1987] that achieves considerable speedups in the best case. This same approach can be adapted for our design graph grammar.

Example. Let the following graph rules (actually only the left-hand sides) be given[12] as in Figure 38.

The corresponding Rete network including the activations is depicted by Figure 39.

Remarks. The above example uses context-sensitive rules to illustrate how the Rete network is compiled. Although the rules of our design graph grammars are primarily context-free, this approach remains fully applicable within the design analysis context, since graph rules are to be executed in a backward fashion.

[11] In the field of graph theory this problem is known as the *subgraph isomorphism problem*. It should not be mistaken with the *graph isomorphism problem*, which lies in NP, but for which it is still open if it is NP-complete [Garey and Johnson, 1997, Arvind et al., 1998, Köbler et al., 1993, Mehlhorn, 1984].

[12] This example is a slightly modified version of the example found in [Bunke et al., 1991a].

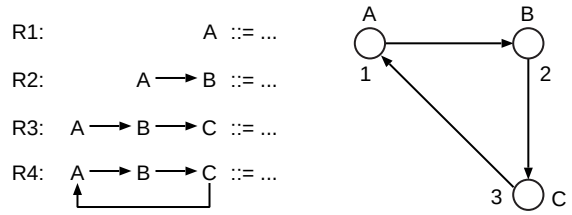


Figure 38: Graph rule left-hand sides and a sample graph.

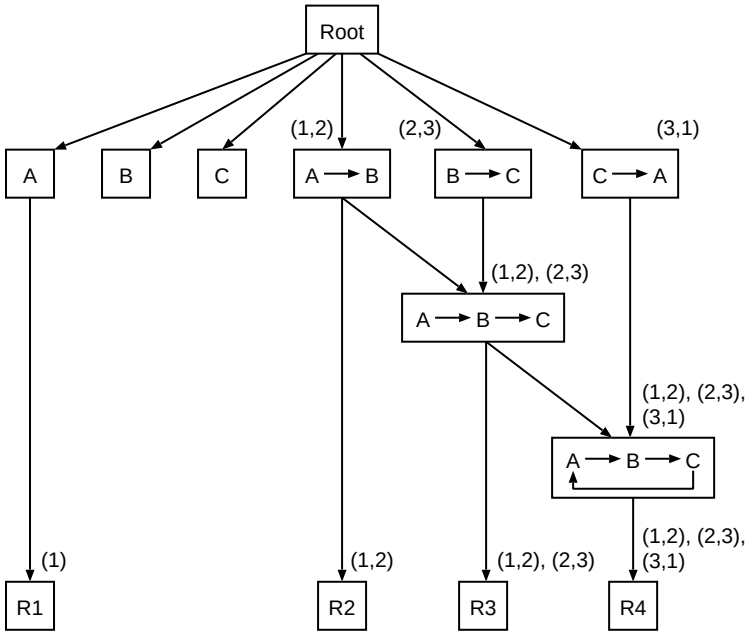


Figure 39: The compiled Rete network for the example of Figure 38.

Remarks. Figure 39 clearly shows that the Rete network is a compact structure considering all partial and total matches. The ability to combine partial matches to handle multiple rule activations is the decisive factor here and accounts for the performance gain reached through the use of this concept, which trades space for time.

7.3.3 Context within Backward Execution

As seen in section 4, the structural analysis of designs is done by means of graph transformation rules executed in a backward fashion. A graph transformation rule $\langle T, C\rangle \rightarrow \langle R, I\rangle$ is interpreted as $\langle R, I\rangle \rightarrow \langle T, C\rangle$, i. e., the replacement graph R represents the new target graph, the target graph T represents the new replacement graph, and the context graph C specifies the embedding explicitly. The embedding instructions I play the role of the new context—this fact gives rise to some issues that are addressed here.

Firstly, a context may be omitted, i. e., the graph transformation rule is context-free. A backward execution of such a graph transformation rule leads to the question of how to deal with the embedding instructions: Either they are used solely for connection purposes

and not as a context specification (context-free backward execution), or they are used for connection and context purposes (context-sensitive backward execution).

Secondly, embedding instructions lack the exactness of a true context graph, since they only represent rules that specify embeddings—if there is no applicable situation, then an embedding instruction is ignored. Thus, different "contexts" may be matched by the embedding instructions.

7.4 Foundations of Derivations and Membership

This section is dedicated to the basic properties of design graph grammars, which allow a classification of design graph grammars into different subclasses with certain properties. Furthermore, special restrictions that lead to interesting and promising results related to the membership problem are addressed.

7.4.1 Basic Properties

There are a series of basic properties of graph grammars that can be examined, but the probably most important property is *confluence*. Confluence has far-reaching consequences, since many NP- or PSPACE-complete problems related to graph grammars that have this property, such as the membership problem, can be solved in (nondeterministic) polynomial time (depending on other properties as well). However, before we proceed with the definition of confluence, we provide some other basic notions.

Lemma 1 *(Associativity of Design Graph Grammars)*

Let $\mathcal{G} = \langle \Sigma, P, s \rangle$ be a design graph grammar, with $T_1 \rightarrow \langle R_1, I_1 \rangle$ and $T_2 \rightarrow \langle R_2, I_2 \rangle$ graph transformation rules thereof, and H a host graph. Let R_1 have a matching of T_2. Then

$$H[T_1|R_1][T_2|R_2] = H[T_1|R_1[T_2|R_2]]$$

.

The following definition of confluence is based on [Engelfriet and Rozenberg, 1997].

Definition 10 *(Confluence)*

A context-free design graph grammar $\mathcal{G} = \langle \Sigma, P, s \rangle$ is confluent, *if for every pair of rules $T_1 \rightarrow \langle R_1, I_1 \rangle$ and $T_2 \rightarrow \langle R_2, I_2 \rangle$ with R_i contains a matching of T_{3-i} for $i \in \{1, 2\}$, and for any arbitrary host graph H containing matchings of T_1 and T_2, the following equality holds:*

$$H[T_1|R_1][T_2|R_2] = H[T_2|R_2][T_1|R_1]$$

Put in other words, a design graph grammar is confluent if the order of rule application is irrelevant.

The following definition of confluence (based on the definition of confluence for edNCE grammars in [Engelfriet and Rozenberg, 1997]) is more detailed and makes an a-priori statement possible.

Definition 11 *(Confluence 2)*

A context-free design graph grammar $\mathcal{G} = \langle \Sigma, P, s \rangle$ *is* confluent*, if for all graph transformation rules* $T_1 \rightarrow \langle R_1, I_1 \rangle$ *and* $T_2 \rightarrow \langle R_2, I_2 \rangle$ *in* P*, all nodes* $x_1 \in V_{R_1}, x_2 \in V_{R_2}$*, and all edges labels* $\alpha, \delta \in \Sigma$*, the following equivalence holds:*

$$\exists \beta \in \Sigma : ((t_2, t_1, \alpha), (t_2, x_1, \beta)) \in I_1 \text{ and } ((\sigma(x_1), t_2, \beta), (\sigma(x_1), x_2, \delta)) \in I_2$$
$$\Leftrightarrow$$
$$\exists \gamma \in \Sigma : ((t_1, t_2, \alpha), (t_1, x_2, \gamma)) \in I_2 \text{ and } ((\sigma(x_2), t_1, \gamma), (\sigma(x_2), x_1, \delta)) \in I_1$$

Remarks. The above definition allows for an algorithmic confluence test of context-free design graph grammars. Furthermore, confluence can only be guaranteed for constructive transformations; thus, the presence of destructive graph transformation rules makes a confluence statement improbable.

Theorem 4 *(Context-free Design Graph Grammars and Confluence)*

Context-free design graph grammars are not inherently confluent.

Proof. Let $\mathcal{G} = \langle \Sigma, P, s \rangle$ *be a context-free design graph grammar and* $H = \langle \{v\}, \emptyset, \{(v, t_1)\} \rangle$ *a host graph. Let* $r_1, r_2 \in P$ *be two graph transformation rules as follows:*

r_1: $t_1 \rightarrow \langle R_1, I_1 \rangle$ *with* $R_1 = \emptyset$ *and* $I_1 = \emptyset$

r_2: $t_1 \rightarrow \langle R_2, I_2 \rangle$ *with* $R_2 = \langle \{v_1, v_2\}, \{(v_1, v_2)\}, \{(v_1, t_1), (v_2, t_2), (\{v_1, v_2\}, e)\} \rangle$ *and* $I_2 = \emptyset$

With these two rules the following derivations are possible:

1. $H \Rightarrow_{r_1} H_1 = \emptyset$
2. $H \Rightarrow_{r_2} H_2 \Rightarrow_{r_1} H_{21} = \langle \{v_2\}, \emptyset, \{(v_2, t_2)\} \rangle$

Since $H_1 \neq H_{21}$*,* $\mathcal{G}$ *is not confluent.* ◇

7.4.2 Special Restrictions for Membership Test

As hinted previously, the confluence property leads to positive results and is therefore desirable. In this section two possible restrictions to general design graph grammars are presented, each of which implies confluence or even stronger properties.

Leftmost Derivation

Leftmost derivations of design graph grammars are achieved by imposing a linear order on the nodes of the right-hand sides of the graph rules—this is necessary since there is no natural linear order as in the case of string grammars. The following definitions and results are based on [Engelfriet and Rozenberg, 1997].

Definition 12 *(Ordered Graph)*

A graph $G = \langle V_G, E_G, \sigma_G \rangle$ is an ordered graph, *if there is a linear order $(v_1, \ldots, v_n)$ with $v_i \in V_G$ for $1 \leq i \leq n$ and $|V_G| = n$.*

The embedding of graphs preserves the imposed order: Let G and H be disjoint ordered graphs and linear orders $(v_1, \ldots, v_G)$ and $(w_1, \ldots, w_H)$, respectively. Let $v = v_i$ be the node of the host graph G that is to be substituted; then, the order of the substitution $G[v/H]$ is obtained by replacing v_i in the order of G with the order of H, i. e., the order of the substitution is $(v_1, \ldots, v_{i-1}, w_1, \ldots, w_H, v_{i+1}, \ldots, v_G)$.

Definition 13 *(Leftmost Derivation)*

Let $\mathcal{G}$ be a design graph grammar. For an ordered graph H, a derivation step $H \Rightarrow_{v,p} H'$ of $\mathcal{G}$ is a leftmost derivation step *if v is the first nonterminal node in the order of H (p represents here the graph transformation rule used). A derivation is leftmost if all its steps are leftmost.*

The graph language leftmost generated by G *is denoted by $L_{lm}(\mathcal{G})$.*

Remarks. The ordering of the sentential forms has no influence on the language $L(\mathcal{G})$ generated by a graph grammar $\mathcal{G}$.

Lemma 2 *(Expressiveness of Leftmost Generated Languages)*

For every confluent design graph grammar $\mathcal{G}$ it holds that $L_{lm}(\mathcal{G}) = L(\mathcal{G})$.

Proof. See [Engelfriet and Rozenberg, 1997], page 39, where a proof for confluent NCE grammars is given. Due to the strong similarity between design graph grammars and NCE grammars, the proof for design graph grammars is analogous. ◇

Theorem 5 *(Characterization of Leftmost Generated Languages)*

The class of languages leftmost generated by design graph grammars is equal to the class of languages generated by confluent design graph grammars.

Proof. See [Engelfriet and Rozenberg, 1997], page 41pp., where a proof for confluent NCE grammars is given. Due to the strong similarity between design graph grammars and NCE grammars, the proof for design graph grammars is analogous. ◇

Boundary Restriction

Since design graph grammars are NCE graph grammars, various properties valid for NCE grammars also hold for design graph grammars. However, important properties such as confluence, decidability of the membership problem etc. do not necessarily hold for the whole class. For certain subclasses, on the other hand, it can be shown that these properties hold. In the following we introduce one such class, whose definition stems from [Engelfriet and Rozenberg, 1997].

Definition 14 *(Boundary Design Graph Grammar)*

A design graph grammar $\mathcal{G} = \langle \Sigma, P, s \rangle$ with directed edges and edge labels is boundary, *or a boundary design graph grammar, if for every production $T \rightarrow \langle R, I \rangle$,*

(B1) R does not contain edges between nonterminal nodes, and

(B2) I does not contain embedding instructions $((\sigma, t, \beta), (\sigma, x, \gamma))$ where σ is nonterminal.

In [Engelfriet and Rozenberg, 1997] it is also shown that only one of the conditions (B1) or (B2) is actually necessary, since each condition implies the other one.

Design graph grammars are not boundary by definition, since both (B1) and (B2) do not hold. However, design graph grammars can be easily adjusted to have property (B1), by introducing additional terminal nodes between adjacent nonterminal nodes, or property (B2), by restricting the embedding instructions to only take into account terminal node labels in the host graph, making design graph grammars boundary.

Lemma 3 *(Expressiveness of Boundary Design Graph Grammars)*

Every boundary design graph grammar is confluent.

Proof. Boundary graph grammars are confluent by definition [Engelfriet and Rozenberg, 1997], page 56. This follows from condition (B2) of definition 14.

Consequences of the Restrictions

In the previous sections we introduced two possible restrictions to design graph grammars that are easy to perform. The application of these restrictions has various interesting consequences [Engelfriet and Rozenberg, 1997]:

- Design graph grammars do need not to be structurally restricted in order to have confluence. The use of leftmost derivations suffices in order to produce confluent graph languages. Thus, the restriction of design graph grammars to some confluent subclass, such as boundary design graph grammars, is not necessary.
- Confluent context-free (design) graph grammars are associative.
- The membership problem for confluent design graph grammars is in NPTIME ([Engelfriet and Rozenberg, 1997], page 82).
- The membership problem for boundary design graph grammars is in PTIME, if, due to labeling restrictions, the subgraph matching problem can be solved in polynomial time ([Slisenko, 1982, Rozenberg and Welzl, 1986, Schuster, 1987]).

Boundary Design Graph Grammars

We presented two restrictions which yield statements concerning the time complexity of the membership problem. We now apply the boundary restriction, as described in definition 14, to design graph grammars to guarantee that the membership problem is in PTIME.

In order to make design graph grammars boundary, we choose to fulfill condition (B1) of Definition 14 and introduce additional terminal nodes called *junctions* (also called "T-connections"). The nodes are inserted into rules having at least two adjacent nonterminals on the right-hand side. As an example, we apply this restriction to a design graph rule of section 5.2, of which only rule (**R4**) is actually changed, as depicted by Figure 40.

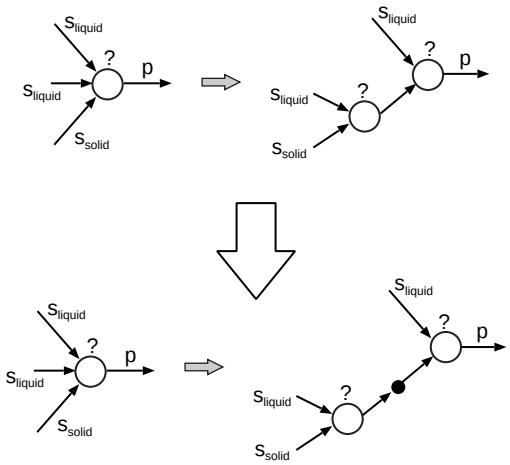


Figure 40: Change of rule (**R4**) resulting from the boundary restriction.

Remarks. The additional nodes (and edges) resulting from the boundary restriction can be easily removed by means of a post-processing routine. Since this post-processing step only consists of removing additional nodes of degree 2, the required effort amounts to linear time in the size of the graph. Thus, the complete process remains polynomial.

7.5 Membership and Derivation in Design

This section is dedicated to the problems of membership and derivation as applied to design tasks. In particular, the membership problem is examined and statements about its complexity made, whereas special attention is given to the polynomial case and the graph class restrictions needed to make this possible. Furthermore, the problems of shortest derivations and distance between graphs, which are closely related to the synthesis task, are addressed.

7.5.1 The Membership Problem for Graph Languages

To solve the membership problem for graph languages a method is required with which a graph can be parsed and a derivation tree based on the design graph grammar can be constructed. It suffices, however, to know that the given graph was generated from the initial symbol of the design graph grammar, i. e., a derivation tree is not absolutely necessary.

In the area of string languages there are some algorithms that were devised to solve exactly the same problem. One of these is the Cocke-Younger-Kasami algorithm (CYK algorithm) described in [Hopcroft, 1979]. The basic idea is to start from the given sentential form and apply the grammar productions backwards, taking all possible combinations in

consideration. If the word belongs to the language generated by the string grammar, then the initial symbol will be derived. The CYK algorithm is a dynamic programming procedure taking $O(n^3)$ time in the length of the input word. In order to guarantee this runtime complexity, the grammar must be in Chomsky normal form, i. e., rules may only have either one terminal or two nonterminal symbols on the right-hand side.

Example. In the following we illustrate how the CYK algorithm works. For this purpose, let the following simple string grammar in Chomsky normal form be given:

$$
\begin{aligned}
S &\rightarrow C_{11}X \mid C_{12}X && (1)\\
C_{11} &\rightarrow T_1X && (2)\\
C_{12} &\rightarrow T_1X && (3)\\
T_1 &\rightarrow C_{21}X \mid C_{22}X && (4)\\
C_{21} &\rightarrow T_2X && (5)\\
C_{22} &\rightarrow T_2X && (6)\\
T_2 &\rightarrow t && (7)\\
X &\rightarrow x && (8)
\end{aligned}
$$

Figure 41 shows how the CYK algorithm works on the input $txxxx$; Figure 42 shows a parse tree for the word $txxxx$. Note that the parsing tree is a binary tree (due to the Chomsky normal form), and that the table generated by the CYK algorithm has the same structure.

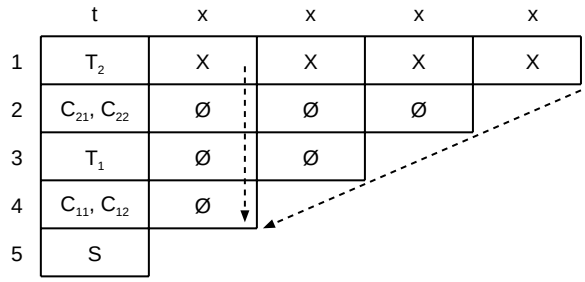


Figure 41: Recognition of the word $txxxx$ by the CYK algorithm.

In contrast to graph grammars, string grammars possess a linear ordering that specifies the context—or relevant neighborhood—of a symbol, namely the symbols located to the left and right. The CYK algorithm (and most parsing algorithm for string languages) makes use of this trivial property, as well as of the normal form used for the string grammar, as can be seen in Figures 41 and 42, and of the intrinsic freedom of rule application order. Graphs, unlike string words, do not have this linear ordering property, which fact makes the search for a rule with matching right-hand side a toilsome job due to the subgraph matching problem (see section 7.3.2 for further details). Figure 43 shows a graph and a string graph together with their relevant contexts.

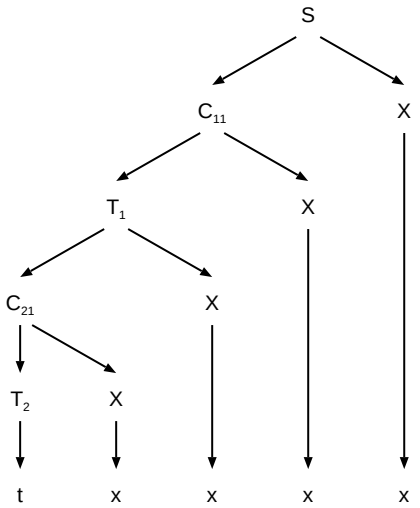


Figure 42: A parse tree derived by the CYK algorithm for the word $txxxx$.

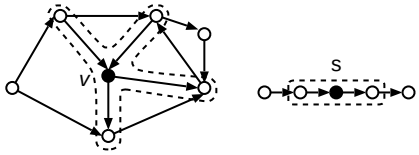


Figure 43: Relevant contexts in graphs and strings.

Indeed, in [Brandenburg, 1983] it is shown that the membership problem is NP- or PSPACE-complete for a variety of graph languages, including restricted context-free languages such as the ones generated by NLC grammars, which are a special case of NCE grammars[13]. Furthermore, it is argued in [Brandenburg, 1983] that the *finite Church Rosser* property[14] is of crucial importance for the existence of a polynomial time recognition algorithm.

7.5.2 Solving the Membership Problem in Polynomial Time

As seen in the previous section, the membership problem for graph languages imposes exponential time complexity on any algorithm attempting to solve it. This statement applies to the general case of arbitrary graph languages, which is far more than is required for our purposes. Indeed, by restricting the graph language class under consideration and using additional information, polynomial time complexity can be achieved.

In the literature one can find some approaches that solve the membership problem in polynomial time, two of which are *precendence graph grammars* [Kaul, 1986] and *rooted context-free flowgraph languages* [Lichtblau, 1991]. In the following we give a brief description of these two approaches and reflect on the consequences for our problem.

[13]NLC grammars do not have edge directions or labels. See section 7.1 or [Engelfriet and Rozenberg, 1997] for details on NLC and NCE grammars.

[14]This property states that non-overlapping rewriting steps can be performed in any order. This property is also known as *confluence*.

Rooted Context-Free Flowgraph Languages

Flowgraph languages are context-free graph languages that supply a suitable mechanism to represent the control flow of source programs[15]; they have a strong resemblance to series-parallel graphs, to which the graphs generated by our design graph grammars for the domain of chemical engineering are also similar.

Rooted flowgraphs are graphs containing nodes (roots) that are connected to all other nodes by means of paths. They are of vital importance for the polynomial time recognition algorithm, since the nodes have to be ordered somehow and these roots provide the ideal starting points.

In order to test if a given graph belongs to the language of rooted context-free flowgraphs, the given graph and the flowgraph grammar have to be ordered. This is done by imposing ordered spanning trees on the graph as well as on the graph grammar. This spanning tree is then used to guide the reduction process, which acts in accordance with the given order. The recognition algorithm based on these prerequisites requires polynomial time in the size of the input graph—further details can be found in [Lichtblau, 1991] and [Lichtblau, 1990].

Whether a similar algorithm for the membership problem for non-rooted flowgraph grammars exists still remains an open problem [Lichtblau, 1991].

Precedence Graph Grammars

Precedence graph grammars are context-free graph grammars with additional precedence relations[16]. In the general case, the membership problem is PSPACE-complete; however, if certain conditions are met, the membership problem is decidable in $O(n^2)$ time, where n is the number of nodes of the input graph.

In the field of string languages, only fast parsing algorithms taking linear time in the length of the input have become widespread. There, linear complexity can be attained by means of the introduction of additional precedence relations, the requirement of the LL(k) or LR(k) property [Hopcroft, 1979] etc. Apart from the use of precedence relations, all other approaches rely on the linear order of strings—the efficiency of LR(k) methods, for example, is based on the fact that the set of all valid prefixes can be formulated as a regular language. Precedence relations, on the other hand, allow for processing in any order or even in parallel.

Every pair of adjacent symbols is assigned a precedence determining which symbol is to be processed first. Precedences always refer to a node pair (v, w), and each precedence may be of one of the following types:

1. Node v is to be processed before node w.
2. Node v is to be processed after node w.
3. Nodes v and w are to be processed simultaneously.

[15] The information and results presented in this section stem primarily from [Lichtblau, 1991].
[16] The information and results presented in this section stem primarily from [Kaul, 1986].

4. Nodes v and w can be processed in any order.

In order to achieve the time complexity of $O(n^2)$ claimed above, the following conditions must hold:

- *The graph grammar is confluent*. Confluence is essential here, since there are cases where the order of reduction steps is arbitrary or not specified. If the graph grammar is not confluent, then the reduction process may not be able to reach the initial symbol, although it is derivable.
- *The precedence relations are disjoint*. This feature ensures that every node pair is assigned a unique precedence relation, thus preventing any ambiguity in the reduction process.
- *The graph productions are uniquely reversible*. This requirement arises from the fact that every reduction step, i. e., backward execution of a rule, must be deterministic and achievable without backups. Again, ambiguity is to be avoided.

In [Kaul, 1986], precedence graph grammars do not have edge labels in the usual sense; the precedences are edge attributes. The edge label alphabet of design graph grammars can be enhanced to include "precedence labels", increasing the size of the edge label alphabet by a factor of at most four[17].

Remarks. Please note that the precedence graph grammar approach is not only applicable for special context-free graph classes such as outerplanar or series-parallel graphs, but for any context-free graph language for which a precedence graph grammar with the above properties can be given.

For more details concerning precedence graph grammars, refer to [Kaul, 1986] and [Kaul, 1987], where practical applications for precedence graph grammars are presented.

Remarks. The approaches presented above require additional information or mechanisms in order to reach polynomial time complexity. Thus, design graph grammars have to be extended to encompass and make use of these approaches. As mentioned above, this can be done by adding special symbols to the edge label alphabet or by imposing some order on the nodes of the graph to be tested and on the nodes of the graph grammar rules.

In section 7.4.2 we present some theoretical results found in the literature, among which is the statement that the membership problem for boundary NCE grammars is in PTIME. This result is, as stated earlier, theoretical in nature and does supply neither any concrete parsing or recognition algorithm nor any statement regarding a precise time complexity.

7.5.3 Shortest Derivation

The length of a derivation is an adequate measure for the complexity of a generated design. Depending on the design graph grammar used and on the order of graph transformation rules applied, a derivation will take at least linear time with respect to the size of the graph,

[17] Actually, every edge label should only be assigned one precendence, therefore only making the existing labels longer and not increasing their number at all.

assuming that each rule application will generate a finite number of terminal nodes only; on the other hand, the worst case runtime complexity for a derivation is unbounded if cyclic partial derivations exist and destructive graph transformation rules are applied. Thus, only a statement concerning the lower bound for the time required for the derivation process is possible, i. e., a prediction can only be ventured for the shortest derivation.

The design graph grammar concept, as presented in section 3, does not impose any restriction upon the transformational behavior of rules. In fact, the example of section 5.2 contains three different types of rules: rules that fire only once, e. g. *R1*, rules that fire linearly in the number of inputs, e. g. *R4*, and rules that can fire arbitrarily often, e. g. *R6*. The existence of the last rule type implies that the graph rule system may not terminate. In fact, within a concrete technical domain such as the domain of chemical engineering one can distinguish between the following types of rules:

- *Chain rules.* Rules may only produce a single output, and rules may not split nonterminal nodes into further nonterminal nodes, such as in rule *R4*. Furthermore, we forbid cycles within the generated design, thus avoiding the repeated execution of rule sequences.

  Due to these restrictions, the size of chemical plant designs generated by these rules is linearly related to the number of inputs available. Therefore, the computational effort—in terms of the number of rules applied—to produce a feasible design using rules of this type is of the order $O(n)$.

- *Splitting rules.* Now we drop the splitting restriction on rules, i. e., splitting rules such as *R4* are allowed. Depending on the number of inputs, further nonterminal nodes may be produced. This results in $O(n^2)$ rule applications.

- *Unrestricted rules.* Lastly, rules that multiply the number of outputs are also allowed. Since with these rules arbitrarily many new "inputs" can be generated, the computational effort is unbounded.

Accordingly, the overall complexity is unbounded, if all rule types are allowed. However, the design graph grammar for chemical plants presented in section 5.1 does not contain rules of the last type, thereby limiting the overall computational effort for the generation of one chemical plant design to $O(n^2)$ rule applications.

The termination drawback mentioned above can be avoided by forbidding the repeated use of the same rule within the same context—in fact, graph grammar implementations do include facilities to specify forbidden and allowed rules (see *programmed graph replacement systems* in [Schürr, 1997b]).

As hinted above, the presence of cycles containing destructive transformations within a derivation may lead to an unbounded complexity. However, by means of restrictions on the rule structures that prevent such cycles, one may avoid this drawback. Before we address this issue we shall introduce some necessary notions.

Definition 15 *(Derivation)*

A derivation is a sequence of graphs $\pi = (G_1, \ldots, G_n)$ *for which the simple derivation* $G_i \Rightarrow G_{i+1}, i \in \{1, \ldots, n-1\}$, *has been achieved by applying a graph transformation rule.* π_G *denotes a*

derivation based on graph transformation rules of a design graph grammar $\mathcal{G} = \langle \Sigma, P, s \rangle$, and $\pi_{\mathcal{G}}(G)$ denotes a derivation $(s, \ldots, G)$.

The shortest derivation is denoted by π^*.

Remarks. A derivation $\pi = (G_1, \ldots, G_n)$ may also be written as $G_1 \Rightarrow^* G_n$. Although the latter form is widely used, we choose to use the first form for convenience, since statements such as "$G \in (G_1, \ldots, G_n)$" are more intuitive than "$G \in G_1 \Rightarrow^* G_n$".

Definition 16 *(Derivation Rule Sequence)*

Let $\pi = (G_1, \ldots, G_n)$ be a derivation. We define the derivation rule sequence belonging to π as $\rho_\pi = (r_1, \ldots, r_{n-1})$, where $G_i \Rightarrow G_{i+1}$ is done by means of a graph transformation rule r_i, $1 \leq i \leq n-1$.

Typically, the human understanding of the design of technical systems bears a monotonic character. This means that the design process is constructive, deletion operations are avoided where possible, leading to a system with the smallest number of steps possible. The following definitions shed some light on this matter.

Definition 17 *(Deletion Operation)*

A deletion operation is a graph transformation step $G \Rightarrow G'$ such that

- $|V_T| > |V_R|$ *or*
- $|E_T| > |E_R|$

whereby $\langle V_T, E_T, \sigma_T \rangle$ and $\langle V_R, E_R, \sigma_R \rangle$ represent the target and replacement graphs, respectively.

Remarks. Definition 17 implies that constructive graph transformation steps may perform partial deletions, as long as there are more insertions.

Figure 44 illustrates the consequence of the presence of deletion operations within a derivation.

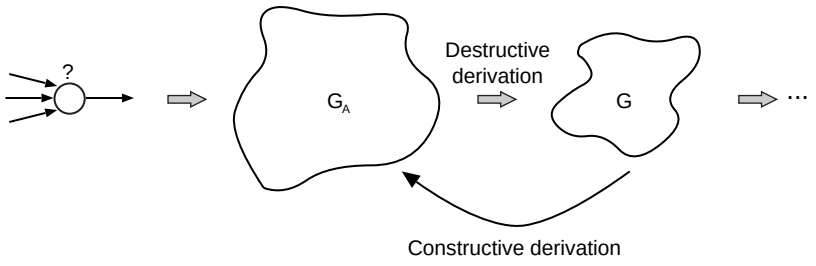


Figure 44: A derivation containing deletion operations. Due to the cycle the derivation length is unbounded.

With the aid of the above notions the aforementioned restriction to rule structures, which represents a special property, can be introduced formally.

Definition 18 *(Monotonicity, Shortcut-Free)*

Let G, G' be graphs and $\mathcal{G}$ a design graph grammar. A derivation $\pi = (G, \ldots, G')$ is called monotonic, if and only if ρ_π does not involve deletion operations.

$\mathcal{G}$ is monotonic, if and only if for every $G \in L(\mathcal{G})$ there exists a monotonic derivation $\pi_{\mathcal{G}}(G)$.

$\mathcal{G}$ is called shortcut-free, if for every $G \in L(\mathcal{G})$ the shortest derivation is a monotonic derivation.

Remarks. Shortcut-freedom means that there is no shortest derivation containing deletion operations.

Figure 45 shows a monotonic derivation.

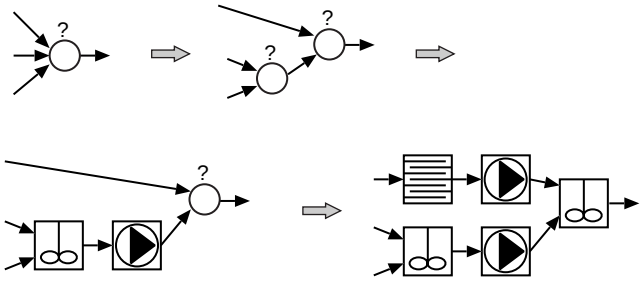


Figure 45: A monotonic derivation of a chemical plant.

7.5.4 Distance between Graphs

Another issue that is closely related to the shortest derivation problem addressed in the previous section is the distance between graphs. But, instead of providing some means to predict the effort necessary to generate a design fulfilling the given constraints, the focus now lies in supplying a statement concerning the quality of the design.

The quality of a design G is measured by the distance to the ideal design G^*, as provided by an expert. In practice, this is done by determining the necessary graph transformation steps required for the derivation $G \Rightarrow^* G^*$ and calculating the involved effort.

Since our approach is bound to a concrete design graph grammar within a given domain, we have to determine the distance between a design G and the ideal design G^* by means of the graph transformations supplied by the design graph grammar. Hereby we assume that the ideal design G^* is also derivable with the given design graph grammar. Hence, we distinguish between the *direct* distance between two graphs as well as the *derivational* distance between two graphs. Figure 46 depicts both situations.

Talking about Figure 46, it is clear that the derivational distance between the two designs is equivalent to the effort necessary for the "derivation" $G \Rightarrow^* G_A \Rightarrow^* G^*$. Put in other words, the distance between G and G^* is bounded by the effort required to transform G back into an ancestor G_A and the effort required to derive G^* from this common ancestor G_A (in the following let G_A denote an ancestor sentential form).

Remarks. In some favorable cases it may happen that a design G is an ancestor of the ideal

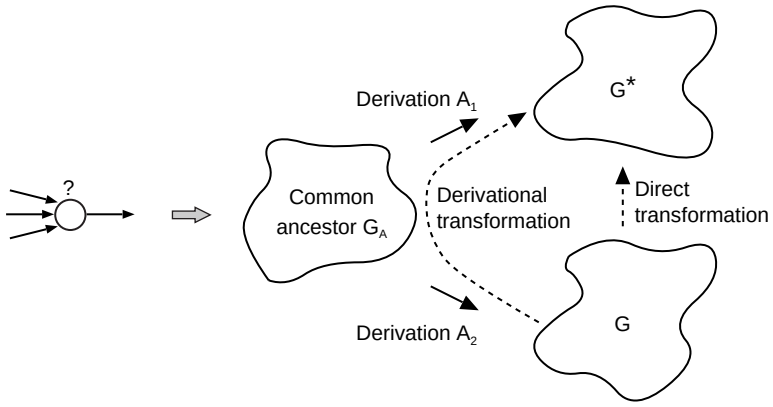


Figure 46: Distance between a design G and the ideal design G^* with respect to the design graph grammar derivation.

design G^*.

Determining the Graph Transformation Sequence

The effort required to solve the task of determining the graph transformation sequence necessary for the derivation $G \Rightarrow^* G^*$ depends on two factors: the degree of information given and the desired granularity of the distance statement.

In order to determine the derivational distance between a design G and the ideal design G^* one needs the derivations belonging to these two graphs. Four different cases can be distinguished, representing the degree of information supplied:

1. Derivations of G and G^* are unknown.
2. Derivation of G is unknown, derivation of G^* is known.
3. Derivation of G is known, derivation of G^* is unknown.
4. Derivations of G and G^* are known.

The first two cases can be ruled out, since the design G has been automatically generated—its derivation is therefore known. Thus, only the last two cases remain as possible starting points. A derivation of G^*, if not available, can be determined by means of the methodology presented in section 4.

As far as the desired granularity of the distance statement is concerned, one has to decide how much effort to invest in calculating the derivational distance described above. On the one hand, a naive approach consisting of a simple comparison of derivations is conceivable. This approach implies comparing $\pi(G)$ and $\pi(G^*)$ element-wise, i. e., searching for a graph $G_A \in \pi(G) \cap \pi(G^*)$. This approach leads to a gross upper bound for the derivational distance between G and G^*. On the other hand, a more elaborate approach involving finding the "largest" common ancestor results in a lower upper bound for the derivational distance.

The search for a common ancestor is a nontrivial task involving solving the graph matching problem mentioned in section 7.3.2. The search for the "largest" common ancestor is even more toilsome, since there may exist more than one derivation for a given graph. This means that the comparison of alternative derivations may be necessary. Please note that this problem does not correspond to the NP-hard *maximal common subgraph problem* [Koch, 2001], although the algorithms described there could be used to find a maximal common subgraph, which in turn represents at least an approximation of the largest common ancestor.

Again, the monotonicity property proves to be a valuable feature of a design graph grammar because it makes the search for a common ancestor much easier. In fact, the absence of deletion operations reduces the search space considerably, since monotonicity implies there is an upper bound for the derivation length, whereas with deletion operations a derivation may be arbitrarily long. Thus, some way to determine if a design graph grammar is monotonic is mandatory.

Lemma 4 *(Monotonicity Requirements)*

Let a design graph grammar $\mathcal{G} = \langle \Sigma, P, s \rangle$ be given. $\mathcal{G}$ is monotonic, if the following holds for every graph transformation rule $r = \langle T, C \rangle \rightarrow \langle R, I \rangle$ of P: R encompasses a matching of T.

Put in other words, the target graph is a subgraph of the replacement graph.

Determining the Effort of the Transformation

After determining the graph transformation rules required for the derivation $G \Rightarrow^* G^*$, the effort necessary for this transformation can be calculated from both the domain and the graph-theoretical point of view.

In order to take the domain into account, we introduce a function $c_{dom} : P \rightarrow \mathbf{R}_0^+$ that yields for a graph grammar $\mathcal{G} = \langle \Sigma, P, s \rangle$ the effort for the application of a given rule $r \in P$ within the domain *dom*. Now, the overall domain effort when transforming a design according to a derivation π can be computed as follows:

$$effort(\pi) = \sum_{r \in \rho_\pi} c_{dom}(r)$$

If a function c_{dom} cannot be stated, a function $c_{gg} : P \rightarrow \mathbf{R}_0^+$ that computes the graph-theoretical effort, which includes aspects such as context and matching, must be used instead:

$$effort(\pi) = \sum_{r \in \rho_\pi} c_{gg}(r)$$

8 Summary

The paper in hand introduced a methodology to solve structure related design tasks, providing the foundations for an automation of the design process as a whole at the level of parameterized building blocks.

Through simplification and by means of graph grammars, the tasks associated with a design problem—structure generation, behavioral model synthesis, structural and behavioral analysis, design evaluation, design repair and design optimization—become tractable. The concepts introduced are uniformly applicable throughout the design cycle and allow for automation in areas that have been as yet left untouched by traditional approaches, of which structural synthesis and analysis benefitted the most.

The presented concepts were applied to the domain of chemical engineering, where a chemical process is modeled as a graph whose nodes describe the unit-operations and whose edges specify the properties of the processed substance at a simplified level. Modifications of a chemical process are defined as node-insertion and node-deletion operations, which in turn are formalized by means of graph grammars.

Thus, design solutions for a chemical processing problem can be produced and verified automatically by applying graph production rules that encode an engineer's design knowledge. However, drawbacks to this approach do exist: design generation has a theoretically unbounded runtime behavior, and the verification of a design solution does not work for arbitrary structures—it must comply with the encoded design knowledge and the structural restrictions imposed by the graph grammar model. Given a properly encoding of the design knowledge, a large set of feasible designs can be generated or verified at an acceptable computational effort.

All in all, our approach provides insights and evaluation of the methodologies necessary for an entire automation of the design process within technical domains. In particular, the use of model simplification and structural manipulation by graph grammars proves to be advantageous within this context.

A Graph Grammar Applications Within Design

Within this paper we have concentrated on important tasks related to the design of technical systems: analysis, synthesis and optimization of structures. These are tasks located at a global level with respect to the overall design of a system. However, there are a series of other tasks that play a minor role within the design process but that are nonetheless necessary within certain contexts. Some of these special tasks can be tackled by means of design graph grammars.

As hinted in section 3.1, there are various conceivable operations on structures, and some of the examples presented there belong to special tasks as mentioned above. The following examples stem from work on projects dealing with design aspects in different domains; more details and further examples can be found in [Stein, 2001].

A.1 Structural Simplification: Hydraulic Plants

The maintenance of hydraulic plants is a challenging job for present day engineers. The size and complexity of hydraulic plants exceed the human capacity to manage them efficiently, thus making additional support a necessity. In special, the design task "analysis" for diagnosis purposes is of importance.

In [Schulz, 1997, Stein and Schulz, 1998] the concept of *hydraulic axes* plays a major role within the analysis of a hydraulic plant. Hydraulic axes represent substructures within a hydraulic plant that perform a function; the recognition of all hydraulic axes of a hydraulic plant yields the set of all functions present within the plant—in a certain sense one could say the recognition of hydraulic axes is the recognition of the building blocks that compose the global plant. Figure 47 shows a hydraulic plant and its hydraulic axes.

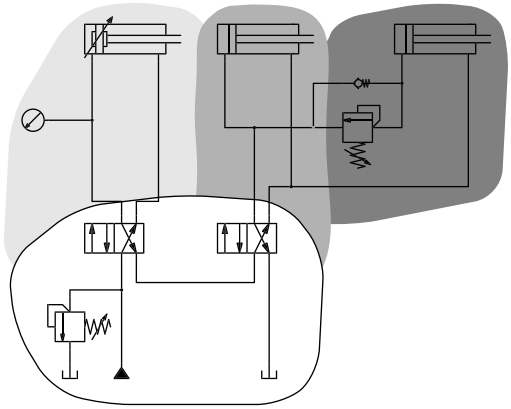


Figure 47: A hydraulic circuit containing three hydraulic axes. The unshaded area is shared by all three axes.

The recognition of the hydraulic axes of a plant does not suffice to fully analyze a hydraulic plant. Within the diagnosis context the knowledge about the relationships between the individual axes is essential for a precise statement concerning a faulty component, since a defect within a hydraulic axis often influences the behavior of other axes, spreading the

faulty behavior throughout the hydraulic plant. Thus, the relationship between each pair of hydraulic axes within a hydraulic plant must be considered. Figure 48 shows the couplings between the hydraulic axes depicted in Figure 47.

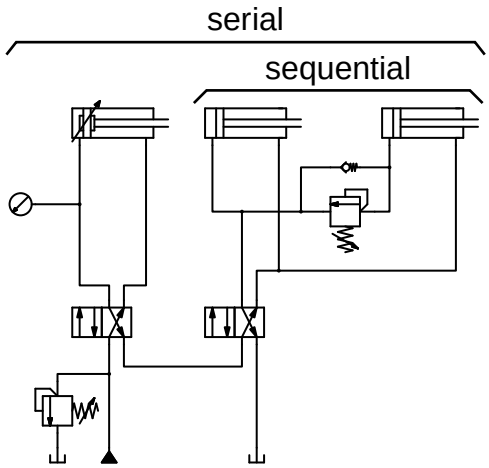


Figure 48: Coupling of hydraulic axes.

The tasks described above—the recognition of hydraulic axes and of their relationship to each other—are efficiently solved by means of path search algorithms. This is due to the inherent structure of hydraulic axes; each hydraulic axis possesses a pump, representing a pressure source, some valves for control together with additional auxiliary components, and cylinders and motors, representing the working devices responsible for the output. However, hydraulic axes often possess substructures that hinder a full recognition: circuit loops, dead branches etc. Thus, a hydraulic circuit has to be simplified prior to applying path searching methods; Figure 49 illustrates the simplification process.

The following simple design graph grammar suffices to perform the transformation described by Figure 49.

First, let the following assumptions be made:

- Supply elements, i. e., pumps and tanks, are designated by the label “p”,
- Working elements, i. e., cylinders and motors, are represented by the label “w”,
- Control elements, i. e., valves, are designated by the label “v”,
- Junction nodes, also called *Tri-Connections*, are represented by the label “j”,
- and all other auxiliary elements are designated by the label “a”.

Let $\mathcal{G} = \langle \Sigma, P, s \rangle$ be a design graph grammar for the structural simplification of hydraulic circuits where $\Sigma = \{p, w, v, j, a, s, H, I, J, K, L, M, N\}$, s is the initial symbol (unused here) and $P = P_{compression} \cup P_{merging}$ is the set of graph transformation rules.

Now, we first provide the compression related graph transformation rules $P_{compression}$:

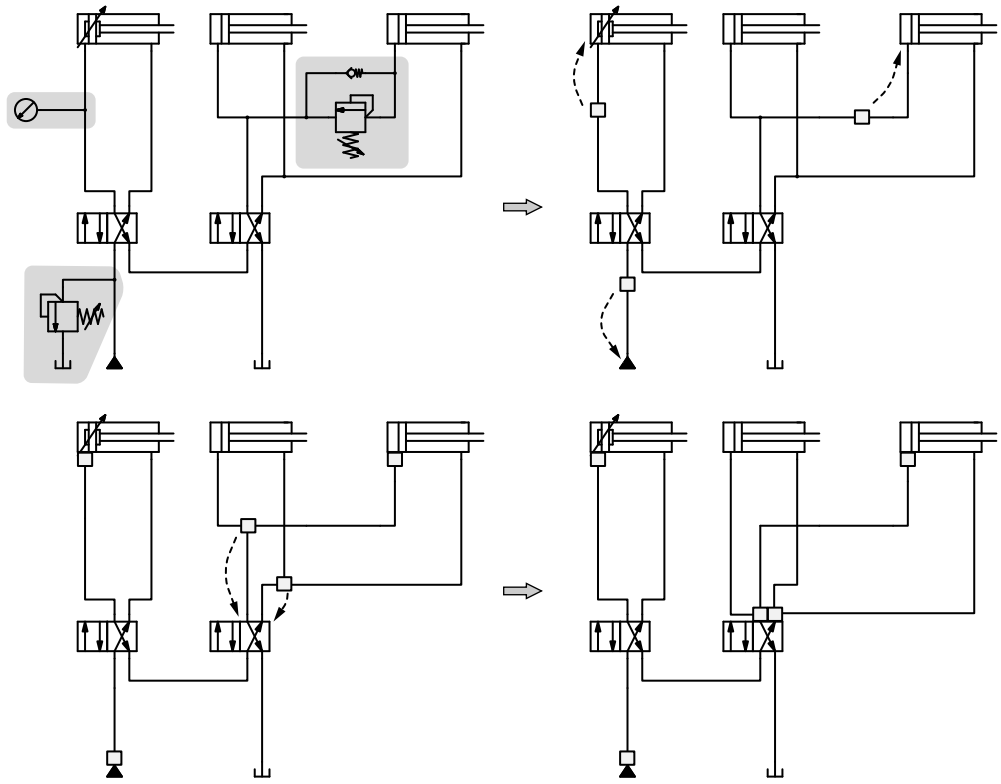
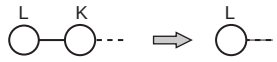


Figure 49: Simplification of a hydraulic circuit by structural compression and merging.

1. Compression of dead branches

$$\begin{aligned} T &= \langle V_T, E_T, \sigma_T \rangle = \langle \{1,2\}, \{\{1,2\}\}, \{(1,L),(2,K)\} \rangle \\ R &= \langle V_R, E_R, \sigma_R \rangle = \langle \{3\}, \emptyset, \{(3,L)\} \rangle \\ I &= \{((J,K,H),(J,L,H))\} \end{aligned}$$

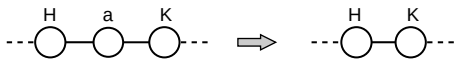
The graphical representation of this rule is as shown below.



2. Compression of chains

$$\begin{aligned} T &= \langle V_T, E_T, \sigma_T \rangle = \langle \{1,2,3\}, \{\{1,2\},\{2,3\}\}, \{(1,H),(2,a),(3,K)\} \rangle \\ R &= \langle V_R, E_R, \sigma_R \rangle = \langle \{4,5\}, \{\{4,5\}\}, \{(4,H),(5,K)\} \rangle \\ I &= \{((I,H,M),(I,H,M)),((J,K,M),(J,K,M))\} \end{aligned}$$

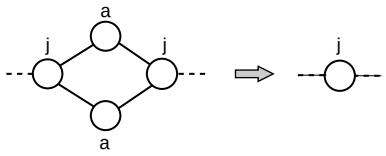
The graphical representation of this rule is depicted below.



3. Compression of loops

$$\begin{aligned} T &= \langle V_T, E_T, \sigma_T \rangle = \langle \{1,2,3,4\}, \{\{1,2\},\{1,3\},\{2,4\},\{3,4\}\}, \\ &\quad \{(1,j),(2,a),(3,a),(4,j)\}\rangle \\ R &= \langle V_R, E_R, \sigma_R \rangle = \langle \{5\}, \emptyset, \{(5,j)\}\rangle \\ I &= \{((H,j,I),(H,j,I))\} \end{aligned}$$

The graphical representation of this rule is illustrated below.



The set of graph transformation rules $P_{merging}$ is defined as follows:

1. Merging with working elements

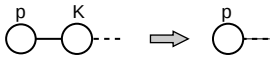
$$\begin{aligned} T &= \langle V_T, E_T, \sigma_T \rangle = \langle \{1,2\}, \{\{1,2\}\}, \{(1,w),(2,K)\}\rangle \\ R &= \langle V_R, E_R, \sigma_R \rangle = \langle \{3\}, \emptyset, \{(3,w)\}\rangle \\ I &= \{((H,w,L),(H,w,L)),((J,K,M),(J,w,M))\} \end{aligned}$$

The above formal representation is equivalent to the following graphical notation:

2. Merging with supply elements

$$\begin{aligned} T &= \langle V_T, E_T, \sigma_T \rangle = \langle \{1,2\}, \{\{1,2\}\}, \{(1,p),(2,K)\}\rangle \\ R &= \langle V_R, E_R, \sigma_R \rangle = \langle \{3\}, \emptyset, \{(3,p)\}\rangle \\ I &= \{((J,K,M),(J,p,M))\} \end{aligned}$$

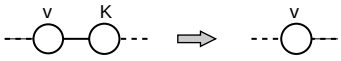
The graphical representation of the above graph transformation rule is as follows:



3. Merging with control elements

$$\begin{aligned} T &= \langle V_T, E_T, \sigma_T \rangle = \langle \{1,2\}, \{\{1,2\}\}, \{(1,v),(2,K)\}\rangle \\ R &= \langle V_R, E_R, \sigma_R \rangle = \langle \{3\}, \emptyset, \{(3,v)\}\rangle \\ I &= \{((H,v,M),(H,v,M)),((J,K,N),(J,v,N))\} \end{aligned}$$

Again, the corresponding graphical representation is depicted below:



Remarks. Some of the above graph transformation rules possess identical structures and differ only with respect to node and edge labels. In such cases one could argue that *label classes* would reduce the number of graph transformation rules noticeably. For example, all merging rules could be written as a single rule using label classes; however, the embedding instructions would have to be adapted to match the most general case.

A.2 Model Reformulation: Wave Digital Structures

Wave digital structures (WDS) form a particular class of signal flow graphs where the signals are linear combinations of the electric current and flow. WDS represent a concept to translate electrical circuits from the electrical u/i-domain into the a/b-wave-domain; this translation establishes a paradigm shift and is called, in terms of models, *model reformulation*. With respect to this concrete example, this reformulation is bound up with several advantages, which are addressed in [Fettweis, 1986].

When migrating from an electrical circuit towards a WDS, the underlying model is completely changed: The structure model of the electrical circuit, $M_S^{u/i}$, is interpreted as a series-parallel graph with closely connected components and transformed into an adaptor structure, $M_S^{a/b}$.

Figure 50 shows the reformulation of a series-parallel structure tree of an electrical circuit into a corresponding adaptor structure. The nodes labeled by "s" and "p" indicate series and parallel connections in the circuit.

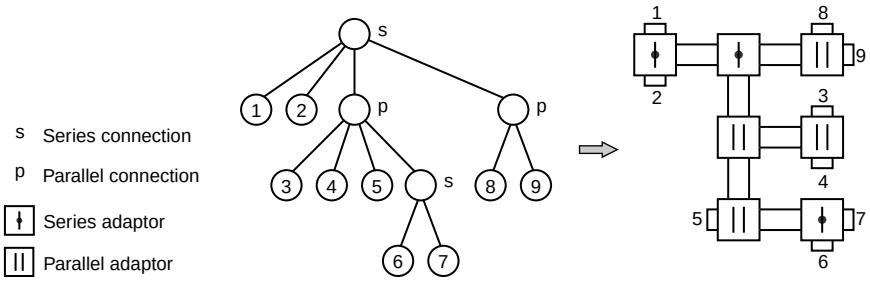


Figure 50: Overview of the mapping $M_S^{u/i} \longrightarrow M_S^{a/b}$.

The following design graph grammar[18] performs the model reformulation depicted in Figure 50 for arbitrary structure models $M_S^{u/i}$.

$\mathcal{G} = \langle \Sigma, P, z \rangle$ with $\Sigma = \{z, p, s, i, X, Y, A, B, C, D, E, F, G, H, I, J\}$, z is the initial symbol (can be neglected), and P is the set of graph transformation rules, which are presented in the following.

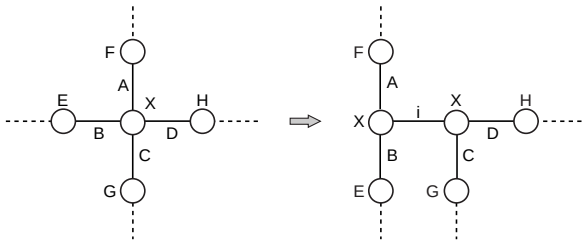
1. Splitting rule for nodes with more than three edges.

$$T = \langle V_T, E_T, \sigma_T \rangle = \langle \{1,2,3,4,5\}, \{\{1,5\}, \{2,5\}, \{3,5\},$$

[18] Since the involved transformation is a translation rather than generation, it would be better to speak of graph transformation systems instead of graph grammars.

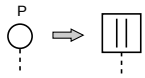
$$
\begin{aligned}
& \{4,5\}\},\{(1,E),(2,F),(3,G),(4,H),(5,X),\\
& (\{1,5\},B),(\{2,5\},A),(\{3,5\},C),(\{4,5\},D)\}\rangle\\
R \;=\; & \langle V_R, E_R, \sigma_R\rangle = \langle\{6,7,8,9,10,11\},\{\{6,7\},\{7,8\},\{7,9\},\{9,10\},\{9,11\}\},\\
& \{(6,F),(7,X),(8,E),(9,X),(10,G),(11,H),\\
& (\{6,7\},A),(\{7,8\},B),(\{7,9\},i),(\{9,10\},C),(\{9,11\},D)\}\rangle\\
I \;=\; & \{((F,X,A),(F,7,A)),((E,X,B),(E,7,B)),((G,X,C),(G,9,C)),\\
& ((H,X,D),(H,9,D)),((I,E,J),(I,E,J)),((I,F,J),(I,F,J)),\\
& ((I,G,J),(I,G,J)),((I,H,J),(I,H,J))\}
\end{aligned}
$$

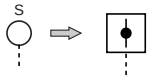
For illustrative reasons we resort to the graphical representation from now on and refrain from using the formal version if appropriate.

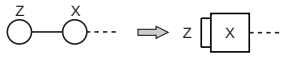


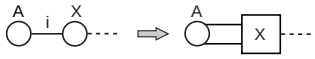
2. Marking rule for edges connecting inner nodes.

The above rules are sufficient to perform the structural transformation required. The following rules belonging to an additional design graph grammar are necessary to change the appearance of the final structure into an adaptor structure as depicted in Figure 50.

1. Display of a parallel node. 

2. Display of a serial node. 

3. Display of a port node. 

4. Display of node connector. 

A.3 Model Reformulation: Parallel-Series Graphs

Model reformulation, as motivated in section A.2, occurs in various forms and within different contexts; however, many of them share the same basic prerequisites and goals. The model

reformulation task of translating a structure description tree, such as the one depicted by Figure 50, into a parallel-series graph represents an adequate abstraction of the corresponding tasks within concrete technical domains.

In a certain sense the goal of this model reformulation task is to translate the *structural view* of the system of interest into its *topological view*. The initial structure is a structure description tree: Inner nodes represent either parallel or serial parts of the described structure, leaves represent edge labels. Figure 51 shows a structure description tree and the corresponding parallel-series graph.

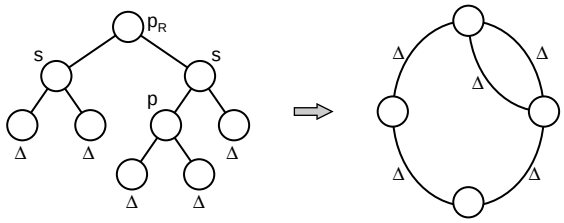


Figure 51: A structure description tree and its corresponding parallel-series graph.

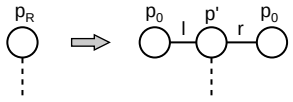
The following design graph grammar performs the transformation required by the model reformulation task.

$\mathcal{G} = \langle \Sigma, P, z \rangle$ with $\Sigma = \{z, p_R, s_R, p_0, s_0, p', s', e, l, r, \Delta, A, B, H, I, J, K, L, M\}$, z is the initial symbol (can be neglected), and P is the set of graph transformation rules, which are presented in the following.

1. Initial rule for parallel rooted description tree

$$\begin{aligned} T &= \langle V_T, E_T, \sigma_T \rangle = \langle \{1\}, \emptyset, \{(1, p_R)\} \rangle \\ R &= \langle V_R, E_R, \sigma_R \rangle \\ &= \langle \{2,3,4\}, \{\{2,3\}, \{3,4\}\}, \{(2, p_0), (3, p'), (4, p_0), (\{2,3\}, l), (\{3,4\}, r)\} \rangle \\ I &= \{((H, p_R, I), (H, p', I))\} \end{aligned}$$

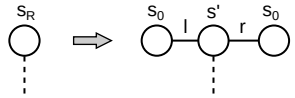
   The rule formally described above corresponds to the following graphical representation:



2. Initial rule for serial rooted description tree

$$\begin{aligned} T &= \langle V_T, E_T, \sigma_T \rangle = \langle \{1\}, \emptyset, \{(1, s_R)\} \rangle \\ R &= \langle V_R, E_R, \sigma_R \rangle \\ &= \langle \{2,3,4\}, \{\{2,3\{, \{3,4\}\}, \{(2, s_0), (3, s'), (4, s_0), (\{2,3\}, l), (\{3,4\}, r)\} \rangle \\ I &= \{((H, s_R, I), (H, s', I))\} \end{aligned}$$

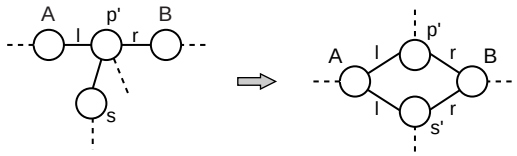
   The graphical representation of the above rule is as follows:



3. Creation of parallel threads

$$
\begin{aligned}
T &= \langle V_T, E_T, \sigma_T \rangle = \langle \{1,2\}, \{\{1,2\}\}, \{(1,p'),(2,s)\} \rangle \\
R &= \langle V_R, E_R, \sigma_R \rangle = \langle \{3,4\}, \emptyset, \{(3,p'),(4,s')\} \rangle \\
I &= \{((H,p',l),(H,p',l)),((H,p',r),(H,p',r)),((H,p',l),(H,s',l)), \\
&\quad ((H,p',r),(H,s',r)),((H,p',I),(H,p',I)),((H,s,I),(H,s',I))\}
\end{aligned}
$$

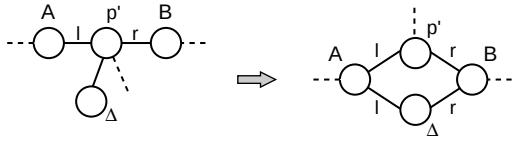
The above rule corresponds to the following graphically:



Creation of parallel threads containing a leaf node

$$
\begin{aligned}
T &= \langle V_T, E_T, \sigma_T \rangle = \langle \{1,2\}, \{\{1,2\}\}, \{(1,p'),(2,\Delta)\} \rangle \\
R &= \langle V_R, E_R, \sigma_R \rangle = \langle \{3,4\}, \emptyset, \{(3,p'),(4,\Delta)\} \rangle \\
I &= \{((H,p',l),(H,p',l)),((H,p',r),(H,p',r)),((H,p',l),(H,\Delta,l)), \\
&\quad ((H,p',r),(H,\Delta,r)),((H,p',I),(H,p',I))\}
\end{aligned}
$$

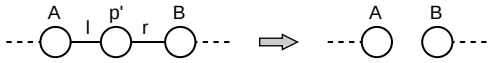
The graphical representation is:



4. Removal of empty parallel node

$$
\begin{aligned}
T &= \langle V_T, E_T, \sigma_T \rangle = \langle \{1,2,3\}, \{\{1,2\},\{2,3\}\}, \{(1,A),(2,p'),(3,B), \\
&\quad (\{1,2\},l),(\{2,3\},r)\} \rangle \\
R &= \langle V_R, E_R, \sigma_R \rangle = \langle \{4,5\}, \emptyset, \{(4,A),(5,B)\} \rangle \\
I &= \{((H,A,L),(H,A,L)),((H,B,L),(H,B,L))\}
\end{aligned}
$$

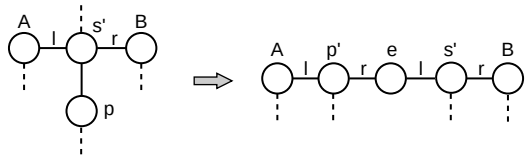
The formal definition of the above rule conforms with the following graphical representation:



5. Creation of serial thread

$$
\begin{aligned}
T &= \langle V_T, E_T, \sigma_T \rangle = \langle \{1,2,3,4\}, \{\{1,2\},\{2,3\},\{2,4\}\}, \\
&\quad \{(1,A),(2,s'),(3,B),(4,p),(\{1,2\},l),(\{2,3\},r)\}\rangle \\
R &= \langle V_R, E_R, \sigma_R \rangle = \langle \{5,6,7,8,9\}, \{\{5,6\},\{6,7\},\{7,8\},\{8,9\}\}, \\
&\quad \{(5,A),(6,p'),(7,e),(8,s'),(9,B),(\{5,6\},l),(\{6,7\},r),(\{7,8\},l),(\{8,9\},r)\}\rangle \\
I &= \{((H,s',I),(H,s',I)),((H,p,I),(H,p',I)),((H,A,I),(H,A,I)), \\
&\quad ((H,B,I),(H,B,I))\}
\end{aligned}
$$

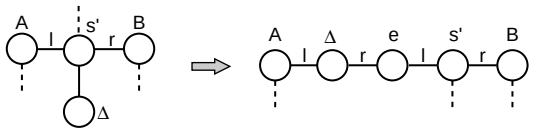
The graphical rule depicted below reflects the above formal definition:



Creation of a serial thread containing a leaf node

$$
\begin{aligned}
T &= \langle V_T, E_T, \sigma_T \rangle = \langle \{1,2,3,4\}, \{\{1,2\},\{2,3\},\{2,4\}\}, \\
&\quad \{(1,A),(2,s'),(3,B),(4,\Delta),(\{1,2\},l),(\{2,3\},r)\}\rangle \\
R &= \langle V_R, E_R, \sigma_R \rangle = \langle \{5,6,7,8,9\}, \{\{5,6\},\{6,7\},\{7,8\},\{8,9\}\}, \\
&\quad \{(5,A),(6,\Delta),(7,e),(8,s'),(9,B),(\{5,6\},l),(\{6,7\},r),(\{7,8\},l),(\{8,9\},r)\}\rangle \\
I &= \{((H,s',I),(H,s',I)),((H,A,I),(H,A,I)),((H,B,I),(H,B,I))\}
\end{aligned}
$$

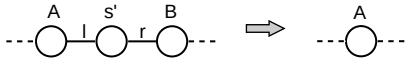
Again, the rule described above corresponds to the following graphical representation:



6. Removal of empty serial node

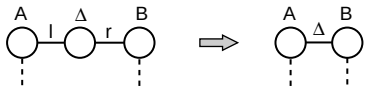
$$
\begin{aligned}
T &= \langle V_T, E_T, \sigma_T \rangle = \langle \{1,2^*,3\}, \{\{1,2\},\{2,3\}\}, \\
&\quad \{(1,A),(2,s'),(3,B),(\{1,2\},l),(\{2,3\},r)\}\rangle \\
R &= \langle V_R, E_R, \sigma_R \rangle = \langle \{4\}, \emptyset, \{(4,A)\}\rangle \\
I &= \{((H,A,I),(H,A,I)),((H,B,I),(H,A,I))\}
\end{aligned}
$$

Once again, the above formal description corresponds to the following graphical representation:



7. Creation of a labeled edge

$$
\begin{aligned}
T &= \langle V_T, E_T, \sigma_T \rangle = \langle \{1,2,3\}, \{\{1,2\},\{2,3\}\}, \{(1,A),(2,\Delta),(3,B), \\
&\quad (\{1,2\},l),(\{2,3\},r)\}\rangle \\
R &= \langle V_R, E_R, \sigma_R \rangle = \langle \{4,5\}, \{\{4,5\}\}, \{(4,A),(5,B),(\{4,5\},\Delta)\}\rangle \\
I &= \{((H,A,I),(H,A,I)),((H,B,I),(H,B,I))\}
\end{aligned}
$$



Remarks. As with the rules presented in section A.1, the use of *label classes* would result in a smaller rule set by combining all graph transformation rules with identical oder nearly identical structures. Again, the embedding instructions would have to be adapted to match the most general case.

Example. Figure 52 illustrates the usage of the design graph grammar described above. This design graph grammar allows for parallelism, which was implicitly used in the derivation shown in Figure 52.

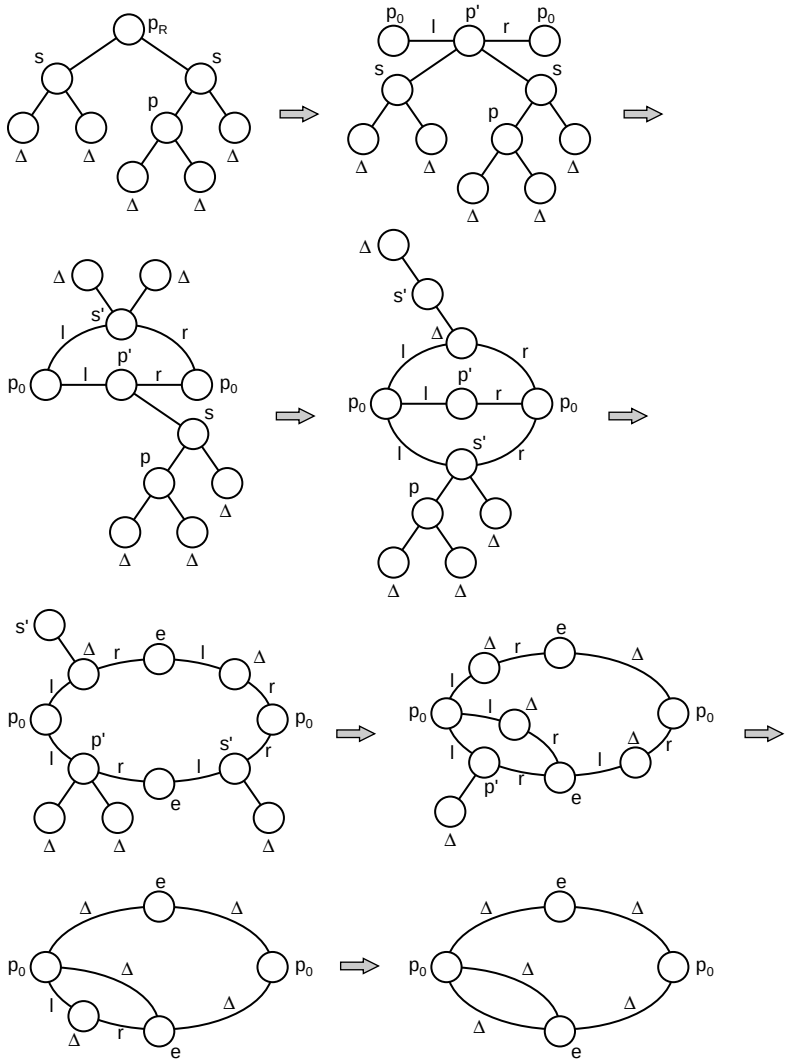


Figure 52: Transformation of a structure description tree into a parallel-series graph.

References

V. Arvind, R. Beigel, and A. Lozano. The Complexity of Modular Graph Automorphism. In *Proc. 15th Annual Symp. on Theoretical Aspects of Computer Science*, volume 1373 of *LNCS*, pages 172–182, 1998.

F. Brandenburg. On the Complexity of the Membership Problem of Graph Grammars. In M. Nagl and J. Perl, editors, *Graphtheoretic Concepts in Computer Science*, pages 40–49, Linz, 1983. Trauner Verlag.

F. Brandenburg. Designing Graph Drawings by Layout Graph Grammars. In R. Tamassia and I. G. Tollis, editors, *Proc. DIMACS Int. Work. Graph Drawing, GD*, number 894 in Lecture Notes in Computer Science, LNCS, pages 416–427, Berlin, Germany, 1994. Springer-Verlag.

A. Brinkop and N. Laudwein. Konfigurieren von industriellen Rührwerken. *KI*, 2:54–59, 1993.

H. Bunke, T. Glauser, and T.-H. Tran. An Efficient Implementation of Graph Grammars based on the RETE Matching Algorithm. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proc. 4th. Int. Workshop on Graph Grammars and their Application to Computer Science*, volume 532 of *Lecture Notes in Computer Science*, pages 174–189. Springer-Verlag, 1991a.

H. Bunke, T. Glauser, and T.-H. Tran. Efficient Matching of Dynamically Changing Graphs. In *Selected Papers from 7th Scandinavian Conf. Theory and Applications of Image Analysis*, pages 110–124. World Scientific, 1991b.

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

F. Drewes, H.-J. Kreowski, and A. Habel. Hyperedge Replacement Graph Grammars. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation*, pages 95–162. World Scientific, Singapore, 1997.

H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2 Applications, Languages and Tools. World Scientific, 1999a.

H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 3 Concurrency, Parallelism and Distribution. World Scientific, 1999b.

J. Engelfriet and G. Rozenberg. A Comparison of Boundary Graph Grammars and Context-Free Hypergraph Grammars. *Inform. and Comput.*, 84:163–206, 1990.

J. Engelfriet and G. Rozenberg. Node Replacement Graph Grammars. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation*, pages 1–94. World Scientific, Singapore, 1997.

A. Fettweis. Wave Digital Filters: Theory and Practice. *Proceedings of the IEEE*, 74(2): 270–327, Feb. 1986.

C. L. Forgy. Rete: A Fast Algorithm for the Many Pattern / Many Object Pattern Match Problem. *Artificial Intelligence*, 19:17–37, 1982.

C. L. Forgy and S. J. Shepard. Rete: A Fast Match Algorithm. *AI Expert*, 2(1):34–40, Jan. 1987.

M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1997.

J. E. Hopcroft. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

D. Jungnickel. *Graphs, Networks and Algorithms*, volume 5 of *Algorithms and Computation in Mathematics*. Springer, 1999.

M. Kaul. *Syntaxanalyse von Graphen bei Präzedenz-Graph-Grammatiken*. PhD thesis, Fakultät für Mathematik und Informatik, Universität Passau, Passau, Germany, 1986.

M. Kaul. Practical Applications of Precedence Graph Grammars. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Graph Grammars and Their Application to Computer Science*, number 291 in Lecture Notes in Computer Science, pages 326–342, Berlin, 1987. Springer-Verlag.

C. Kim and T. 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.

R. 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.

A. Knoch and M. Bottlinger. Expertensysteme in der Verfahrenstechnik – Konfiguration von Rührapparaten. *Chem.-Ing.-Tech.*, 65(7):802–809, 1993.

J. Köbler, U. Schöning, and J. Torán. *The Graph Isomorphism Problem: Its Structural Complexity*. Birkhäuser, 1993.

I. Koch. Enumerating All Connected Maximal Common Subgraphs in Two Graphs. *Theoretical Computer Science*, 250(1–2):1–30, 2001.

M. Korff. Application of Graph Grammars to Rule-based Systems. In H. Ehrig, editor, *Graph Grammars and Their Application to Computer Science*, number 532 in Lecture Notes in Computer Science, pages 505–519, Berlin, 1991. Springer-Verlag.

U. Lichtblau. *Flußgrapggrammatiken*. PhD thesis, Universität Oldenburg, Oldenburg, Germany, 1990.

U. Lichtblau. Recognizing Rooted Context-Free Flowgraph Languages in Polynomial Time. In H. Ehrig, editor, *Graph Grammars and Their Application to Computer Science*, number 532 in Lecture Notes in Computer Science, pages 538–548, Berlin, Germany, 1991. Springer-Verlag.

W. Marquardt. Rechnergestützte Erstellung verfahrenstechnischer Prozeßmodelle. *Chem.-Ing.-Tech.*, 64(1):25–40, 1992.

W. Marquardt. Trends in Computer-Aided Process Modeling. *Computers chem. Engng.*, 20 (6/7):591–609, 1996.

K. Mehlhorn. *Data Structures and Algorithms*, volume 2 Graph Algorithms and NP-Completeness. Springer, Berlin, 1984.

U. Nickel, J. Niere, and A. Zündorf. Tool demonstration: The Fujaba Environment. In *Proc. 22nd Intl. Conference on Software Engineering*, pages 742–745. ACM Press, 2000.

C. C. Pantelides. Speedup – Recent Advances in Process Simulation. *Comput. chem. Engng.*, 12(7):745–755, 1988.

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 chem. Engng.*, 15(1):53–72, 1991.

S. Räumschüssel, A. Gerstlauer, E. D. Gilles, B. Raichle, M. Zeitz, and W. Marquardt. An Architecture of a Knowledge-Based Process Modeling and Simulation Tool. In *Proc. IMACS/IFAC 2nd Intl. Symposium on Mathematical and Intelligent Models in System Simulation*, volume 2, pages 242–247, 1993.

J. Rekers and A. Schürr. A Graph Grammar Approach to Graphical Parsing. Technical Report 95-15, Department of Computer Science, Leiden University, 1995.

G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1 Foundations. World Scientific, 1997.

G. Rozenberg and E. Welzl. Boundary NLC Graph Grammars—Basic Definitions, Normal Forms, and Complexity. *Information and Control*, 69:136–167, 1986.

S. Russel and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, N.J., 1995.

A. Schulz. Graphenanalyse hydraulischer Schaltkreise zur Erkennung von hydraulischen Achsen und deren Kopplung. Master's thesis, University of Paderborn, Department of Mathematics and Computer Science, 1997.

A. 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.

A. Schürr. PROGRES: A VHL-Language Based on Graph Grammars. In H. Ehrig, H. 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.

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

A. Schürr. Programmed Graph Replacement Systems. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation*, pages 479–546. World Scientific, Singapore, 1997b.

A. Schürr, A. Winter, and A. Zündorf. Visual Programming with Graph Rewriting Systems. In *Proc. 11th Intl. IEEE Symposium on Visual Languages*. IEEE Computer Society Press, 1995.

R. Schuster. *Graphgrammatiken und Grapheinbettungen: Algorithmen und Komplexität*. PhD thesis, Universität Passau, 1987.

A. Slisenko. Context-Free Grammars as a Tool for Describing Polynomial-Time Subclasses of Hard Problems. *Inf. Proc. Letters*, 14:52–56, 1982.

B. Stein. *Model Construction in Analysis and Synthesis Tasks*. Professorial dissertation (to appear), University of Paderborn, Department of Mathematics and Computer Science, 2001.

B. Stein and A. Schulz. Topological Analysis of Hydraulic Systems. Technical Report tr-ri-98-197, University of Paderborn, 1998.

B. Stein and E. Vier. An Approach to Formulate and to Process Design Knowledge in Fluidics. In N. E. Mastorakis, editor, *Recent Advances in Information Science and Technology*, pages 237–242. World Scientific Publishing, 1998.

G. Stephanopoulos, G. Henning, and H. Leone. Model.la. A Modeling Language for Process Engineering - I. The Formal Framework. *Computers chem. Engng.*, 14(1):813–846, 1990.

M. van Eekelen, S. Smetsers, and R. Plasmeijer. Graph Rewriting Systems for Functional Programming Languages. Technical report, Computing Science Institute, University of Nijmegen, 1998.