Int J Softw Tools Technol Transfer
DOI 10.1007/s10009-008-0068-z

SPECIAL SECTION SFB 614

Coping with large design spaces: design problem solving in fluidic engineering

Benno Stein

Abstract This paper is about tool support for knowledge-intensive engineering tasks. In particular, it introduces software technology to assist the design of complex technical systems. There is a long tradition in automated design problem solving in the field of artificial intelligence, where, especially in the early stages, the search paradigm dictated many approaches. Later, in the so-called modern period, a better problem understanding led to the development of more adequate problem solving techniques. However, search still constitutes an indispensable part in computer-based design problem solving—albeit many human problem solvers get by without (almost). We tried to learn lessons from this observation, and one is presented in this paper. We introduce *design problem solving by functional abstraction* which follows the motto: construct a poor solution with little search, which then must be repaired. For the domain of fluidic engineering we have operationalized the paradigm by the combination of several high-level techniques. The red thread of this paper is design automation, but the presented technology does also contribute in the following respects: (a) productivity enhancement by relieving experts from auxiliary and routine tasks; (b) formulation, exchange, and documentation of knowledge about design; (c) requirements engineering, feasibility analysis, and validation.

Keywords Design automation · Fluidic circuit design · Design graph grammars · Case-based reasoning · Expert critiquing

This research was supported by DFG grants Schw 120/56-3, KL 529/10-3, KL 529/7-3, and KL 529/10-1.

B. Stein (✉)
Faculty of Media, Media Systems, Bauhaus University Weimar, 99421 Weimar, Germany
e-mail: benno.stein@medien.uni-weimar.de

1 Introduction

Before delving into different paradigms for the modeling and the design of a technical system, we recall two extremal points in a range of possible settings:

1. A technical system is specified *explicitly*, i. e., there is a clear understanding of the structure and the behavior of the desired system.
2. A technical system is specified *implicitly* by a set of desired demands or functions, D, and the task is to develop the description of a system that fulfills D.

In the first setting the main concern is modeling. Research on the modeling of technical systems aims at a comprehensive system specification from different viewpoints and at different levels of granularity. Modern CASE tools (based on UML, SDL, or MSC) provide support for the modeling process, which relates to model transformation, coupling of tools, code generation, etc. By contrast, in the second setting no explicit specification of the system is given—we have a configuration or design problem that is characterized by a search space wherein an optimum solution is to be found. Figure 1 illustrates the relation between both settings. It shows the problem of configuration and design as search in a model space atop the technical and software-based modeling paradigms; the lower part of the figure is based on [19].

The contributions of this paper relate to the second setting. Among others, we use graph grammars to describe and to explore a space of models, and we apply case-based reasoning to find adequate behavior models. Moreover, uncausal simulation is employed to analyze a synthesized model, and a rule-based repair language is used to improve suboptimum designs. The paper is organized into two parts. Part one (Sect. 2) contrasts the knowledge-based approach and

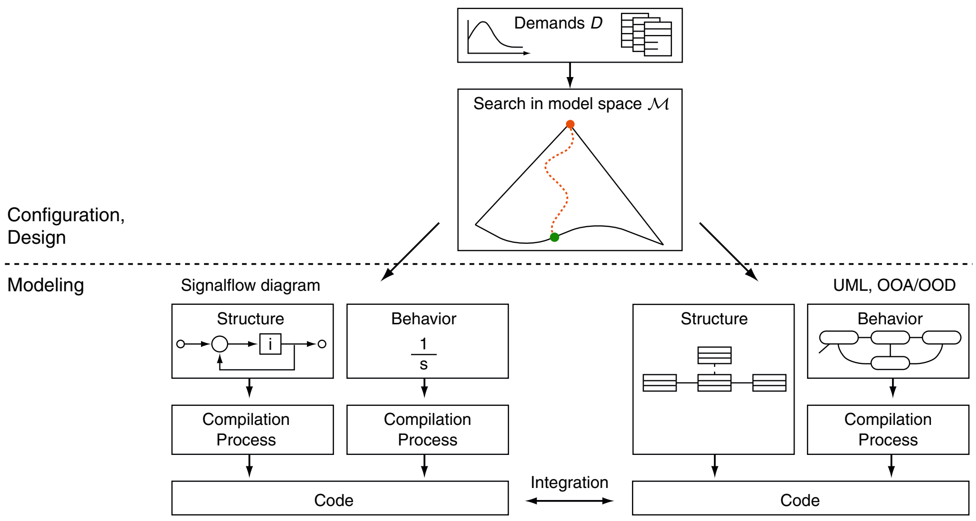


Fig. 1 Solving a design problem means to find a model in the space of possible models (*upper part*). The *lower part* of the figure shows the classical modeling process including model coupling [19]

the search-plus-simulation approach for design problem solving and presents the functional abstraction paradigm as a synthesis of both. Part two (Sect. 3) demonstrates how this paradigm is put to work in the fluidic engineering domain. Aside from motivating the ideas, this section also introduces the employed technologies relating to design graph grammars and cased-based reasoning.

2 Automating design tasks

Design problem solving is the transformation of an implicit description of a non-existing system S, stated in the form of demands D, into an explicit description, say, a model M of the desired system. M may be a construction plan, a drawing, a parts list, or some other document that defines how S is constructed.

The complexity of design tasks varies in an extremely wide range, and one of the first and still useful classification schemes can be found in [2,12,44], illustrated in Fig. 2. They distinguish three classes: (1) The class of creative design, which can be viewed as open-ended; all kinds of inventions belong to this class. Even if the design goals are well-defined, there is no, or only a rough idea of how they can be achieved. (2) The class of innovative designs comprises problems for which powerful decomposition knowledge exists, but design plans for some of the component problems may need a substantial modification. (3) The class of routine designs comprises problems for which knowledge about decomposition and synthesis is completely known. What makes problems of Class 3 tractable is that the structure of the system being designed as well as the parameter ranges are given. Problems of this class are often called configuration problems.

A theory of design that shall cover all kinds of design problems must resort to a generic constraint representation. Reiter [33] has introduced the axiomatic description level of predicate logics as a means to describe generic diagnosis problems. Here we extend his formalism to design problems in order to discuss them in a domain-independent way:

$$\alpha_{CPT} = \underbrace{D \wedge COMPS \wedge SD}_{\textit{configuration}} \quad \underbrace{\wedge\ PARAM}_{+\ \textit{parameterization}} \quad \underbrace{\wedge\ TOP}_{+\ \textit{structure finding}}$$

α_{CPT} is a formula in propositional logics or predicate logics where D is a set of demands, *COMPS* denotes the available components, *SD* is a logic-based formulation of the components' behavior, *PARAMS* defines their parameterization, and *TOP* their topology, i. e., how components are connected to each other. A fulfilling interpretation of α_{CPT} establishes a solution of the design problem.

The complexity to determine a fulfilling interpretation for α_{CPT} depends on the degrees of freedom in the design process. Within a configuration problem (Class 3) merely the truth values in *COMPS* are to be determined, whereas in a behavior-based design problem (Class 2) the components need to be parameterized. If yet the system structure is to be found, the design problem entails creative aspects. Though an axiomatic formulation of a design problem is always possible, it is uncommon to do so: the encoding of a design problem as a satisfiability problem disguises

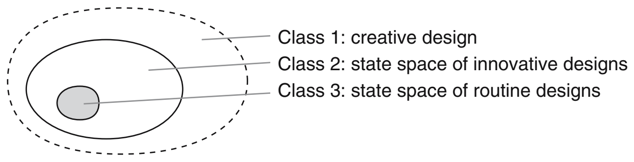

Class 1: creative design
Class 2: state space of innovative designs
Class 3: state space of routine designs

Fig. 2 A coarse taxonomy of design problems, oriented at the size of the underlying state space [12]. Elements of Class 3 are also called configuration problems

valuable domain knowledge that is necessary for an efficient operationalization. Later on, when dealing with a concrete problem in a particular domain, we will resort to problem-specific formulations.

How can the design of complex technical systems be automated? A commonly accepted answer is: *"By operationalizing expert knowledge."*—Experts, say, engineers need little search during problem solving, and computer programs that operationalize engineering knowledge have been proven successful in various complex design tasks. A second, also commonly accepted answer to the above question is: *"By means of search."* This answer reflects the way of thinking of an AI pragmatist, who believes in deep models and the coupling of search and simulation. Deep models, or, models that rely on first principles have been considered the worthy successor of the simple associative models [4,7]; they opened the age of the so-called Second Generation Expert Systems [6,41]. In this sense, Sect. 2.2 formulates design problems as instances of particular search-plus-simulation problems.

Though the search-plus-simulation paradigm can be identified behind state-of-the-art problem solving methodologies [1,11], many systems deployed in the real world are realized according to simpler associative paradigms [2,3,22,30,36,38], to mention only a few. As a source for this discrepancy we discover the following connection: the coupling of search and simulation is willingly used to make up for missing problem solving knowledge but, as a "side effect", often leads to intractable problems.

Drawing the conclusion "knowledge over search" is obvious on the one hand, but too simple on the other: what can be done if the resource "design knowledge" is not available or cannot be elicited, or is too expensive, or must tediously be experienced? we can learn from human problem solvers where to spend search effort deliberately in order to gain the maximum impact for automated problem solving. The paper in hand gives such an example: in Sect. 2.3 we introduce the paradigm of functional abstraction to address behavior-based design problems. It develops from the search-plus-simulation paradigm by untwining the roles of search and simulation; in this way it forms a synthesis of the aforementioned approaches.

2.1 Thesis: knowledge is power[1]

Human problem solving expertise is highly effective but of heuristic nature; moreover, it is hard to elicit but pretty easy to process [17]. Successful implementations of knowledge-based design algorithms do not search in a gigantic space of behavior models but operate in a well defined structure space instead, which is spanned by compositional (left) and taxonomic relations (right):

$$c \rightarrow c_1 \wedge \cdots \wedge c_k \qquad c \rightarrow c_1 \vee \cdots \vee c_k$$

The c_i denote components, and together both types of rules prescribe a decomposition hierarchy in the form of an And-Or-graph. Another class of design algorithms employ the case-based reasoning paradigm retrieve-and-adapt, an advancement of the classical AI paradigm generate-and-test [20, 34,35]:

$$SIM(D_1, D_2) \rightarrow USABLE(M_1, D_2),$$

which claims that the known solution M_1 for a demand set D_1 can be used (adapted) to satisfy a demand set D_2, if D_1 and D_2 are similar.

2.2 Antithesis: search does all the job

A search problem is characterized by a search space consisting of states and operators. The states are possible complete or partial solutions of the search problem, the operators define the transformations from one state into another. Here, in connection with behavior-based design problems, the search space is a set $\mathcal{S}$ of possible systems. Solving a design problem means to find a system $S^* \in \mathcal{S}$ that fulfills the given set D of demands. Typically, S^* is not found by experimenting in the real world but by operationalizing a search process after having mapped the system space, $\mathcal{S}$, onto a model space, $\mathcal{M}$. The set $\mathcal{M}$ comprises all models that could be visited during the search.[2] It is the job of a design algorithm to efficiently find a model $M^* \in \mathcal{M}$ whose simulation produces a behavior that complies with D and that optimizes a possible goal criterion. Figure 3 illustrates the connections.

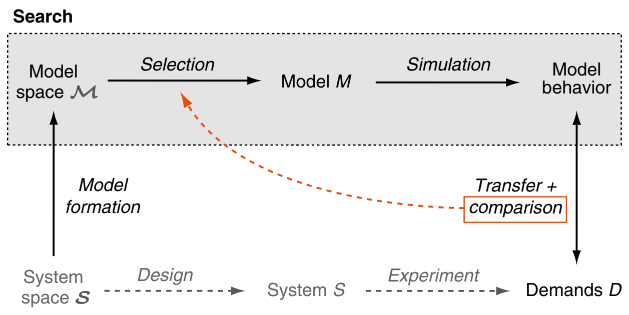


Fig. 3 Generic scheme of design problem solving: given is a space $\mathcal{S}$ of possible design solutions and a set of demands D. On a computer, $\mathcal{S}$ is represented as a model space, $\mathcal{M}$, wherein a model M^* is searched whose behavior fulfills D

[1] This famous phrase is often attributed to Edward A. Feigenbaum, though he did not originate the saying.

[2] In accordance with Minsky [24] we call M a model of a system S, if M can be used to answer questions about S. M may establish a structural, a functional, an associative, or a behavioral model.

This design scheme is inviting: giving a mapping from systems $\mathcal{S}$ to models $\mathcal{M}$—which can be stated straightforwardly in engineering domains—the related synthesis problem can be solved by the search-plus-simulation paradigm.[3] As already mentioned, several implementations of successful design systems do not follow this paradigm. They contain an explicit representation of an engineer's problem solving knowledge instead, say, his or her model of expertise. A problem solver that has such knowledge-based models at its disposal spends little effort in search—a fact which makes these models appearing superior to the deep models used in the search-plus-simulation paradigm. On the other hand, several arguments speak for the latter; a compelling one has to do with knowledge acquisition: in many situations it is not feasible for technical or economical reasons to acquire the necessary problem solving knowledge for tailored models of expertise.[4]

2.3 A synthesis: untwine search and simulation

Applying just search-plus-simulation renders most real-world design tasks intractable because of the mere size of the related model space $\mathcal{M}$. If search cannot be avoided, search effort must be spent deliberately. In this situation we can learn from the problem solving behavior of engineers:

1. Engineers solve a design problem at the level of function rather than at the level of behavior, accepting to miss the optimum.
2. Engineers adapt a suboptimum solution rather than trying to develop a solution from scratch, accepting to miss the optimum.
3. Engineers can formulate repair and adaptation knowledge easier than a synthesis theory.

When we combine these observations we obtain the paradigm *design problem solving by functional abstraction*, which is illustrated in Fig. 4. Put it overstated, the paradigm says: "At first, we construct a poor solution of a design problem, which then must be repaired." Note that the first three steps of this method resemble syntax and semantics (the horseshoe principle) of the problem solving method "Heuristic Classification", which became popular as the diagnosis approach underlying Mycin [5].

[3] Gero [12], for example, proposes a cycle that consists of the steps synthesis, analysis, and evaluation. Sinha et al. [40] present a framework to implement simulation-based design processes for mechatronic systems [28].

[4] Other advantages bound up with this paradigm are: the possibility to explain, to verify, or to document a reasoning process, the possibility to reuse the same models in different contexts, the extendibility to new device topologies, or the independence of human experts.

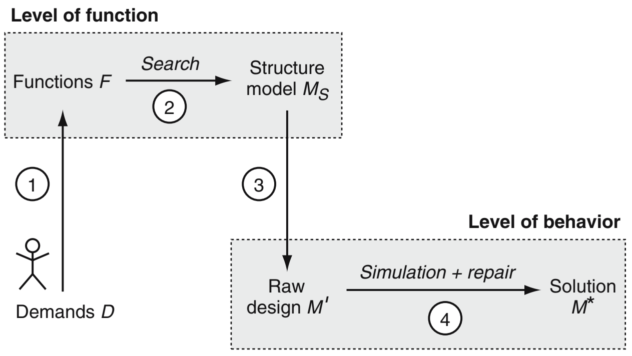


Fig. 4 The paradigm of functional abstraction in design problem solving. Observe that discrete reasoning (search) has been decoupled from approximation-based reasoning (simulation + repair): The former is used to find a structure model M_S, the latter is used to repair a suboptimum raw design

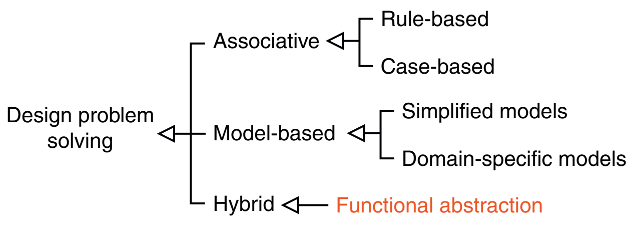


Fig. 5 A general taxonomy of principles and techniques for design problem solving. Model-based design problem solving is successful if it based on simplified models or if it is restricted to a narrow part of the domain in question

Key idea of design by functional abstraction is to construct candidate solutions within a simplified design space, which typically is some structure model space. A candidate solution, M_S, is transformed into a preliminary raw design, M', by *locally* attaching behavior model parts to M_S. The hope is that M' can be repaired with reasonable effort, yielding an acceptable design M^*. Design by functional abstraction makes heuristic simplifications at two places: The original demand set, D, is abstracted toward a functional specification F (Step 1 in Fig. 4), and, M_S is transformed locally into M' (Step 3).

A characteristic of the functional abstraction paradigm is the combination of associative techniques, which restrict the search space toward a tractable size, with model-based techniques, which are responsible for the modeling fidelity. Figure 5 organizes principles and techniques for design problem solving. The next section presents an application of this paradigm in the fluidic engineering domain.

3 Design problem solving in fluidic engineering

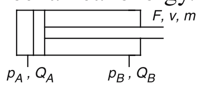
Even for an experienced engineer the design of a hydraulic (fluidic) system is a complex and time-consuming task, that, at the moment, cannot be automated completely.[5] The effort for acquiring the necessary design knowledge exceeds by far the expected payback. Moreover, the synthesis search space is extremely large and hardly to control—despite the use of knowledge-based techniques.

Two possibilities to counter this situation are "competence partitioning" and "expert critiquing". The idea of competence partitioning is to separate the creative parts of a design process from the routine jobs, and to provide a high level of automation regarding the latter [42]. Expert critiquing, on the other hand, employs expert system technology to assist the human expert rather than to automate a design problem in its entirety [10,16]. In this respect, design by functional abstraction can be understood as a particular expert critiquing technique.

3.1 The hydraulic domain

Hydrostatic drives provide advantageous dynamic properties and are an important driving concept for industrial applications. A hydrostatic drive (a hydraulic circuit) consists of various components that provide and distribute pressure p and flow Q. The components can be distinguished with respect to their basic function:

(a) *Working elements* This class contains all kinds of cylinders and motors; they transform hydraulic energy into mechanical energy. Example:

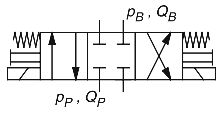


$F = p_A A_K - p_B A_R - m\dot{v} - d_z v$

F, m, v *: force, mass, velocity*
A_k, A_R, d_z*: piston areas, friction*

The symbol shows a differential cylinder, the equation models the balance of forces.

(b) *Control elements* This class contains directional valves used for the control of a working element. Example:



$p_P = p_B + sign(Q_p) R_h Q_P^2$
R_h*: hydraulic resistance*

The symbol shows an electrically actuated proportional valve, the equation defines the pressure drop between two of the connections, assuming a turbulent flow.

(c) *Supply elements* Pumps (left symbol) and tanks (right symbol) are the important elements in this class.

(d) *Auxiliary elements* All elements which do not fall in one of the above classes belong to this class. Examples include pipes, t-connections, pressure relief valves, and throttles.

[5] The presented concepts have been applied and evaluated in the hydraulic domain; however, they can be used in the pneumatic domain in a similar way, suggesting us to use the terms "hydraulic" and "fluidic" interchangeably.

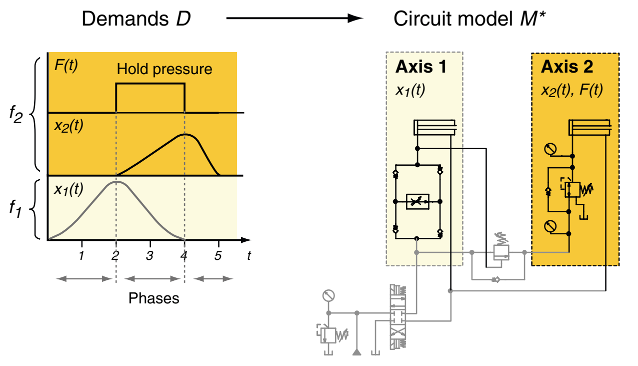
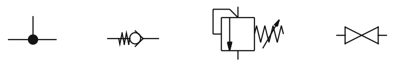


Fig. 6 Hydraulic design is the translation of a demand specification (*left*) to a circuit model (*right*). The *light area* in the demand specification is operationalized by the left Axis 1; the *dark area* is operationalized by the right Axis 2. The coordinated interplay is realized by the coupling of the axes



The above classification does not follow engineering conventions in every respect since it disregards a component's usage within a circuit. For example, a directional valve that does not control a working element is an auxiliary element. Note that the behavior models are more complex than shown, defining state-dependent, stationary and dynamic behavior. Similarly, the specification of the demands D can be manifold and pretty complex: D may prescribe driving profiles, force profiles, key numbers of extreme values of physical quantities, parameter constraints, and the like. In this connection the term "fluidic axis" comes into play: a fluidic axis is a dedicated subcircuit that fulfills a single function f of a fluidic system; it defines the connections and the interplay among the components that realize f. A fluidic function f is determined by its driving profile or force profile, whereas a profile is composed of 2–5 phases that define certain operation modes such as accelerating, holding, or constant drive, for example.

The left-hand side of Fig. 6 shows a demand specification D with two position profiles, $x_1(t)$, $x_2(t)$, and a force profile, $F(t)$. The right-hand side shows the design solution M^*, comprised of two hydraulic axes that are coupled by a sequential coupling. The coloring of D and M^* associates the profiles with the respective axes operationalizing them.

3.2 Applying the functional abstraction paradigm

The starting point for a design process is a task that shall be operationalized by hydraulics: a lifting problem, the actuation of a press, or the realization of a robot's kinematics. The result of the design process is a system consisting of hydraulic components. Taken the view of configuration, the designer of a fluidic system selects, parameterizes, and connects

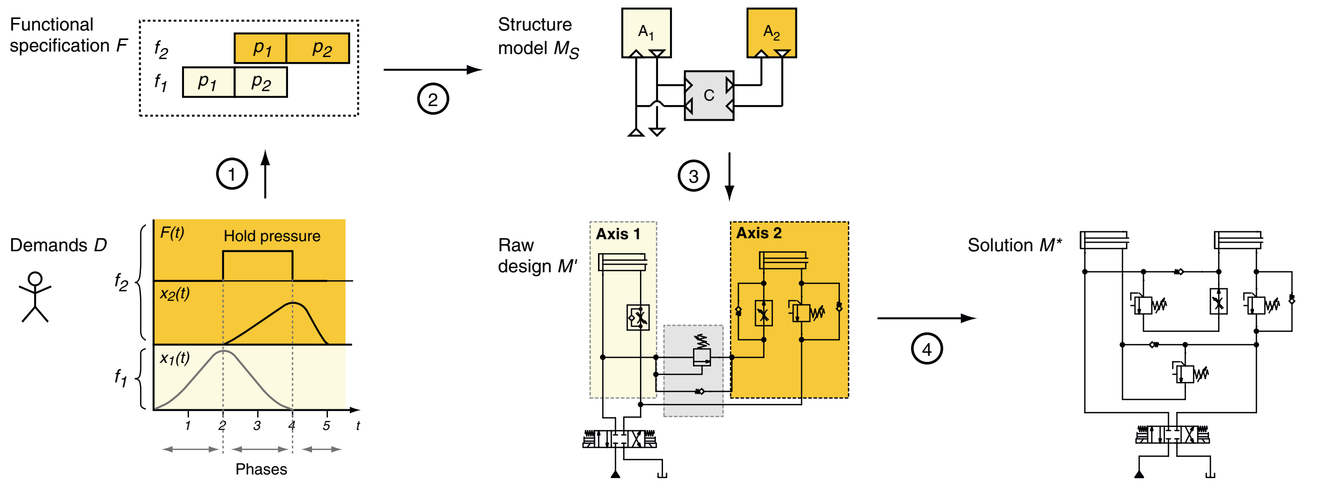


Fig. 7 The functional abstraction paradigm applied to fluidic circuit design: demand abstraction *(1)*, search of a solution M_S in the structure model space *(2)*, transformation of M_S into a preliminary raw design M' *(3)*, simulation of the raw design M' and repair of M' towards the final design solution M^* *(4)* [43]

components like pumps, valves, and cylinders such that the demands D are fulfilled by the emerging circuit. However, solving a fluidic design problem at the component level is pretty hopeless. Hence the idea is to perform a configuration process at the level of functions instead, which in turn requires that fluidic functions possess constructional equivalents that can be treated in a building-block-manner. In the fluidic engineering domain this requirement is fairly good fulfilled; the respective building blocks are the fluidic axes.

To automate design processes of the kind shown in Figure 6, so to speak, to automate the mapping $D \longrightarrow M^*$, we apply the paradigm of functional abstraction (see Fig. 7 and recall Fig. 4):

1. The demand specification, D, is abstracted towards a functional specification $F = \{f_1, \ldots, f_k\}$. A function $f \in F$ in turn is decomposed into its phases, and all phases implied by F are analyzed with respect to their dependencies and arranged in a schedule.
2. At this functional level a structure model M_S according to the coupling of the fluidic functions in F is generated.
3. M_S is completed towards a tentative behavior model M', by plugging together locally optimized fluidic axes. Here, this step is realized by a case-based reasoning approach.
4. The tentative behavior model M' is repaired, adapted, and optimized globally.

Design by functional abstraction rigorously simplifies the underlying domain theory. The tacit assumptions are as follows: (a) each set of demands, D, can be translated into a set of fluidic functions, F, (b) each function $f \in F$ can be mapped one to one onto a fluidic axis A that operationalizes f, (c) D can be realized by coupling the respective axes for the functions in F, whereas the necessary coupling information can be derived from D, and (d) fluidic axes are only coupled in a standardized manner.

While the first point goes in accordance with reality, the Points (b) and (c) imply that a function f is neither realized by a combination of several axes nor by constructional side effects. This assumption, as well as the assumption made in Point (d) represent considerable simplifications: actually the structure of a fluidic systems adds to its overall function. Nevertheless, at the fluidic axes level a human designer treats structure and function also more independent of each other than at the component level.

Note that a human designer is capable of working at the component level, *implicitly* creating and combining fluidic axes towards an entire system. His ability to derive function from structure—and vice versa: structure *for* function—allows him to construct a fluidic system without the idea of high-level building blocks.

The following subsections describe the basic elements of our design approach, i. e., Steps 2, 3, and 4.

3.3 Step 2: topology generation

If the synthesis of fluidic systems is performed at the level of function, the size of the synthesis space is drastically reduced. To be specific, we allow only structure models that can be realized by a recursive application of the three coupling rules shown in Fig. 8. The search within this synthesis space is operationalized by means of a design graph grammar, which generates reasonable topologies with respect to the functional specification F. The result of this step is a structure model M_S; M_S defines a graph whose nodes correspond to fluidic

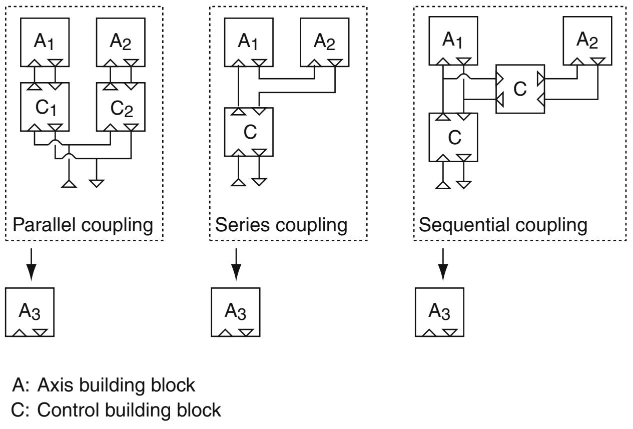


Fig. 8 The three coupling types that are allowed to realize a circuit's topology. The axis building block A_3 shown below each rule shall indicate the rules' recursive applicability

functions and coupling types.

Design graph grammars The systematics of design graph grammars was introduced in Schulz et al. [39]. An important issue was to simplify and to enhance the use of grammars as a tool to define design knowledge in technical domains. Design graph grammars are closely related to node replacement graph grammars. Among others, they allow for a straightforward specification of contexts, which is indispensable to describe the various kinds of manipulations at technical systems.

A graph grammar is a collection of graph transformation rules each of which is equipped with a set of embedding instructions. What happens during a graph transformation is that a node t or a subgraph T in the original graph G, called the host graph, is replaced by a graph R. Say, R is embedded into G. The subsequent definition stems from Schulz et al. [39].

Definition 1 (design graph grammar, DGG). A context-sensitive design graph grammar is a tuple $\mathcal{G} = \langle \Sigma, P, s \rangle$ where

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

The graph transformation rules in P are of the form $\langle T, C \rangle \to \langle R, I \rangle$ where

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

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

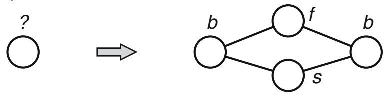
- $h \in \Sigma$ is a label of a node v in $G \backslash T$,
- $t \in \Sigma$ is a label of a node w in T,
- $e \in \Sigma$ is the edge label of $\{v, w\}$,
- $f \in \Sigma$ is another edge label not necessarily different from e, and
- $r \in V_R$ is a node in R.

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

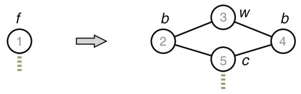
Remarks (1) Each graph is defined as a triple consisting of a node set V, an edge set E, and a labeling function σ that maps from the alphabet Σ onto $V \cup E$. (2) Labels in Σ can be used to specify node types, edge types, and variables. By convention the type labels are noted in lower case, while variable labels are noted in upper case. (3) The syntax of the rules in P and the instructions in I facilitate a uniform specification of different grammars types, such as node-based, graph-based, context-free, or context-dependent. In particular, a context-free rule is written as $T \to \langle R, I \rangle$, and embedding instruction without edge labels can be abbreviated as $((h, t), (h, r))$.

A grammar for circuit topologies Below we give a design graph grammar that generates the class of circuit topologies motivated in Fig. 8. The rigor of structural simplifications that are implied by the functional abstraction paradigm is reflected by the small size of the design graph grammar $\mathcal{G} = \langle \Sigma, P, ? \rangle$ that defines the synthesis space $L(\mathcal{G})$. The label alphabet Σ is $\{b, t, f, s, c, w, A, B\}$, designating biconnections, triconnections, fluidic functions, supply elements, control elements, working elements, and two variables. P contains seven rules of the form $T \to \langle R, I \rangle$ presented below; apart from rule (1b) only the graphical form is shown.

(1a) Initial rule:



(1b) Function-to-axis conversion:



$$T = \langle V_T, E_T, \sigma_T \rangle = \langle \{1\}, \{\}, \{(1, f)\} \rangle$$
$$R = \langle V_R, E_R, \sigma_R \rangle = \langle \{2, 3, 4, 5\}, \{\{2, 3\}, \{2, 5\}, \{3, 4\}, \{4, 5\}\}, \{(2, b), (3, w), (4, b), (5, c)\} \rangle$$
$$I = \{((A, f), (A, c))\}$$

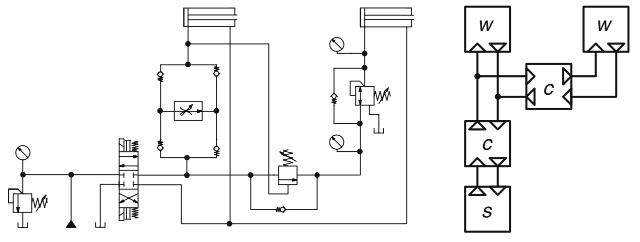


Fig. 9 The circuit diagram of Fig. 6 and its related building block structure (*right*)

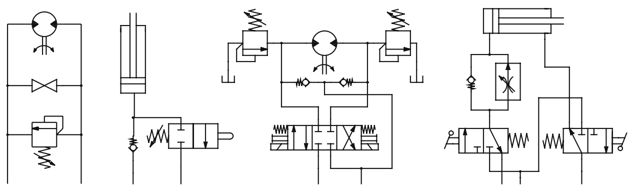
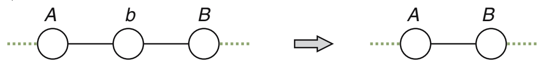


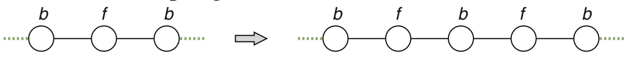
Fig. 10 Four fluidic (hydraulic) axes for different functions and of different complexity. Each axis represents a highly configurable case; the case base $\mathcal{C}$ contains about seventy of such prototypes

(1c) Biconnection deletion:

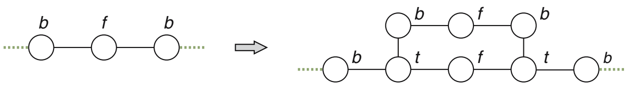


The following rules (2a)–(2c) encode the introduction of a particular coupling type.

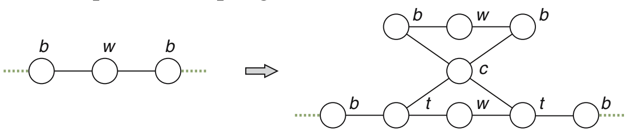
(2a) Series coupling:



(2b) Parallel coupling:



(2c) Sequential coupling:



The following four display rules change the appearance of a labeled graph according to the syntax of the building block structure in Fig. 8:

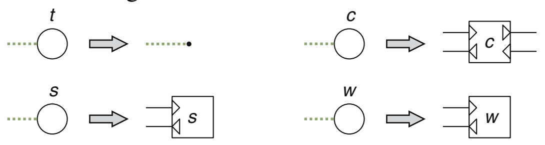


Figure 9 shows the circuit from Fig. 6 and its building block structure. The building block structure corresponds to the derivation (1a) $\longrightarrow$ (1b) $\longrightarrow$ (2c) $\longrightarrow$ (1c), followed by the application of matching display rules.

3.4 Step 3: case-based design of fluidic axes

A structure model M_S is completed towards a behavior model by individually mapping its nodes onto appropriate subcircuits that represent fluidic axes or coupling networks. Figure 10 shows four subcircuits each of which representing a particular fluidic axis. The mapping of a fluidic function f onto an axis can be accomplished ideally with case-based reasoning: domain experts were able to compile a case base $\mathcal{C}$ of about seventy axes that can be used to cover a wide spectrum of fluidic functions.

Diagrams as shown in Fig. 10 can be considered as case representations at the mental model level. A case is a container for three types of objects: a specification of the fluidic function in form of operation modes or so-called phases, a list of parameters and their ranges, and rules that encode adaptation knowledge:

Phase 1		**Parameters**	
Type	Constant drive	Max. pressure	60 Bar
Precision	0.3	Max. flow	200 l/min
Force	–	Operation range	Medium pressure
Distance	2,000 mm	Piston area	120 cm^2
Duration	2.5 s	Ring area	100 cm^2
…		…	

Scaling rules	
Name	Force-scaling-rule-1
Actions	(SETQ A_K (/ (* A_K (- F))
	(* 0.9 10 A_R P_MAX@Q)))
Qualifiers	((<= F 0) (< (* 0.9 10 P_MAX@Q A_K) (- F)))
…	

The heart of the case-based reasoning approach is the measure $\varphi(f, g)$, which evaluates two fluidic functions, f, g, respecting their similarity. φ maps from the Cartesian product of the domain of fluidic functions onto the interval [0; 1]. The basic characteristic of a fluidic function is defined by both the sequence and type of its phases. Valuating two fluidic functions respecting their similarity thus means to compare their phases. Phases in turn are characterized by their type, precision, force, distance, and duration, and for each characteristic a special phase similarity measure is defined. Table 1 gives the definition for ϕ_{pType}, the similarity for two single phase types. Given two fluidic functions, f, g, the similarities of their phase type sequences, $\varphi_{pType}(f, g)$, computes as follows:

$$\varphi_{pType}(f, g) = \frac{1}{\max\{m, n\}} \sum_{i=1}^{\min\{m,n\}} \phi_{pType}(p_i^{(f)}, p_i^{(g)}),$$

where $p_i^{(f)}$ and $p_i^{(g)}$ denote the type of phase i for the fluidic function f and g respectively. Four measures of this kind are combined within $\varphi(f, g)$ and used to rank the cases in $\mathcal{C}$ with respect to the fluidic functions in F. This comparison considers also case adaptability, which depends on the qualifiers and actions encoded in the scaling rules. The scaling rules operationalize engineering know-how according to dif-

Table 1 The similarity measure ϕ_{pType}, which quantifies the similarity between two phase types

Phase type	Position	Constant	Accelerate	Hold-Pos	Hold-Press	Press	Fast
Position	1	0.3	0.5	0	0	0.3	0.7
Constant		1	0	0	0	0.7	0.7
Accelerate			1	0	0	0.2	0.4
Hold-Pos				1	0.6	0.3	0
Hold-Press					1	0.8	0
Press						1	0
Fast							1

ferent adaptation schemes and have been developed in close cooperation with domain experts [15].

The result of Step 3, i. e., the composition of adapted fluidic axes according to M_S, yields a preliminary design solution, the raw design M'. It is important to note that both case retrieval and case adaption do not employ simulation; they must be understood as a heuristic assessment whether a certain combination of fluidic axes is able to fulfill the demands. Simulation comes into play in the next step, which is Step 4 under the functional abstraction paradigm.

3.5 Step 4: a design language for repair

This step starts with a simulation of the raw design M'.[6] The gained insights point to—presumably existing—deficits in the raw design. There is a good chance that M' has the potential to fulfill D, say, that a sequence of repair steps can be found to transform an unsatisfactory raw design M' into a solution M^*. An example for such a repair measure is the following piece of design knowledge:

"An insufficient damping can be improved by installing a by-pass throttle."

The measure contains several forms of implicit engineering know-how, among others: (a) a by-pass throttle is connected in parallel, (b) the component to which it is connected is a cylinder, (c) if there are several cylinders in the system, an engineer knows the best-suited one, and, (d) a by-pass throttle is a valve. Figure 11 illustrates the repair rule.

What is more, engineers use design knowledge in a flexible way; i. e., a particular piece of knowledge can be applied within different contexts in a variety of circuits. Flexibility is a major reason which makes it difficult to encode the expressiveness of the above example on a computer. If we were confronted only with systems of the same topological set-up, then measures like the above (*"Install a by-pass throttle."*) could be hard-wired. To formulate modification knowledge

[6] Simulation as well as the evaluation of simulation results poses a bunch of challenges which are not discussed in this place.

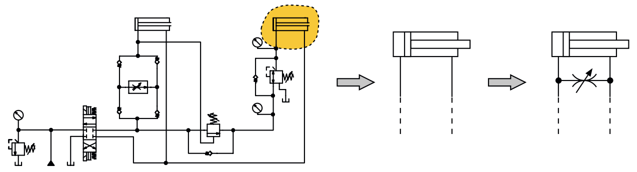


Fig. 11 Illustration of the repair rule *"An insufficient damping can be improved by installing a by-pass throttle."*: interpret context (cylinder), choose the best context among several candidates (left cylinder), apply repair action to the context (connect throttle in parallel)

like above we have developed a scripting language tailored to fluidic circuit design. Its key concepts are outlined below.

- *Design graph grammar* The descriptive means of design graph grammars is used to precisely formulate the semantics of location and action that is encoded in an engineer's repair knowledge. The following graph transformation rule $\langle T, C\rangle \rightarrow \langle R, I\rangle$ operationalizes the repair knowledge of the preceding example, the insertion of a by-pass throttle tv (cf. Fig. 12).

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

- *Modification schemes* Actions can be organized within so-called modification schemes, which provide certain macro facilities with local variables, loops, and branching.
- *Meta knowledge* To control the application of different adaptation measures, they are characterized by values from [0; 1], indicating the effectiveness of a measure,

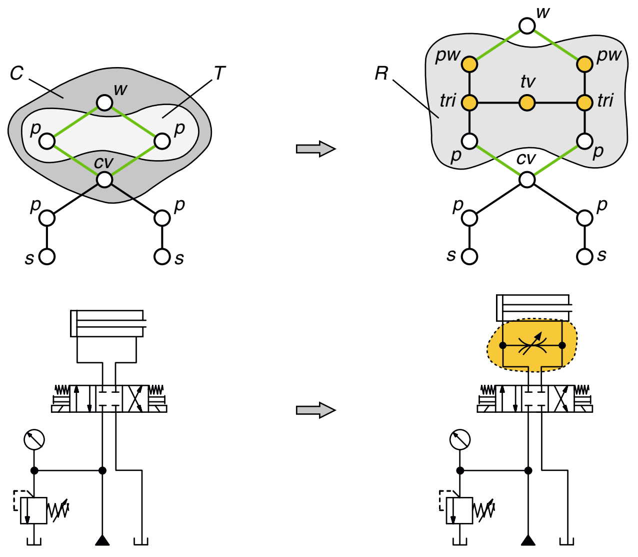
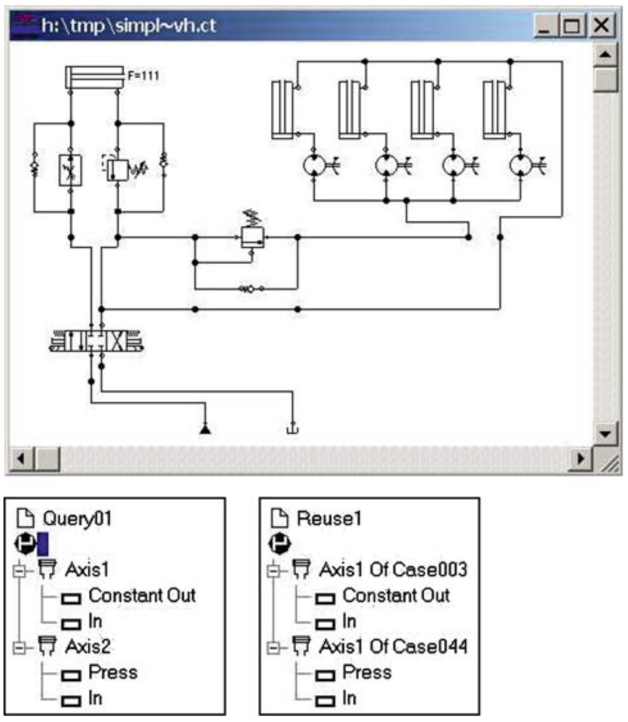


Fig. 12 Application of the graph grammar rule from above, which defines the insertion of a by-pass throttle in parallel to a cylinder

undesired side effects (called repercussion), and realization cost. From these values an overall confidence κ is computed that relies on application requirements and the designer's preferences κ_{eff}, κ_{rep}, and κ_{cost}. Table 2 shows an expert's evaluation of measurements to increase a cylinder's damping factor.

In the example the installation of a throttle in by-pass to the cylinder is ranked first option; the resulting drain flow through the by-pass moves the eigenvalues of the related transfer function to a higher damping. An extensive theory on the analysis, the assessment, and the adaptation of hydrostatic drives can be found in Vier [45].

Connections to CBR research Case-based reasoning has been a very active research field, and many systems and technologies have been developed to apply the CBR paradigm to knowledge-intensive tasks. In the field of design only few systems have been developed that are concerned with deep, behavior-based models. The most relevant work related to our research is SCHEMEBUILDER [27], a system that also deals with fluid engineering design, but which neither applies a simulation-based analysis nor a sophisticated repair technology. Case combination and adaptation for engineering design has been specifically treated in [14,18, 21,31,32]. However, these approaches stop short of generating models that can directly be analyzed at the behavior level and understood at the mental model level—both of which can be realized with the functional abstraction paradigm and has been implemented within ARTDECO (see the next subsection).



Query01
Axis1
Constant Out
In
Axis2
Press
In

Reuse1
Axis1 Of Case003
Constant Out
In
Axis1 Of Case044
Press
In

Fig. 13 A design query (*bottom left*), the functional description of a solution (*bottom right*), and the automatically generated drawing of the design solution

3.6 Realization: design automation with ARTDECO

The outlined concepts have been implemented within the design assistant ARTDECO, which is linked to the FLUIDSIM drawing and simulation environment for fluidic systems [15]. The design assistant enables a user to formulate his design requirements as a set of fluidic functions F. For an $f \in F$ a sequence of phases can be defined, and several characteristic parameters such as duration, precision, or maximum values can be stated for each.

Given some F, the retrieval mechanism of the design assistant searches the case base for axes fitting best the specified functions, according to the measure φ. Afterward, these building blocks are scaled and composed towards a new system which then is simulated and further improved. Finally, a drawing is generated which can directly be analyzed and evaluated by a human designer. Figure 13 shows a query (top left), the functional description of the generated design (below), and the related drawing.

The key question is *"How good are the designs of* ARTDECO*?"* A direct evaluation of the generated models is restricted: an absolute measure that captures the design quality does not exist, and the number of properties that characterizes a design is large and hardly to quantify. On the other hand, the quality of a generated design can be rated *indirectly*, by measuring its "distance" to a design solution created by a human

Table 2 Possible remedies to increase a cylinder's damping factor that has been judged being too low

Modification measure	Effectiveness	Repercussion	Cost	κ
Throttle in mainstream	0.1	0.4	0.8	0.39
Throttle in side stream	0.4	0.4	0.5	0.44
Throttle in by-pass	0.8	0.4	0.5	0.64
Damping network	0.9	0.8	0.1	0.61
Velocity feedback	0.6	0.8	0.3	0.52
Acceleration feedback	0.8	0.8	0.3	0.63

The computation of κ relies on the preferences $\kappa_{eff} = 0.5$, $\kappa_{rep} = 0.15$, and $\kappa_{cost} = 0.35$, which have been proposed by experts

Table 3 Runtime and quality results of automatically generated designs

Number of axes	Time for retrieval (s)	Time for reuse (s)	OK (at $\bar{\varphi} > 0.9$) (%)	Quality %		
1	$\ll$1	0.10	80	60(+)	35(o)	5(–)
2	$\ll$1	0.63	75	50(+)	45(o)	5(–)
3	$\ll$1	0.91	70	40(+)	50(o)	10(–)
4	$\ll$1	1.43	60	20(+)	65(o)	15(–)
5	$\ll$1	2.00	20	5(+)	80(o)	15(–)

The column "OK" shows the portion of design solution whose simulation fulfills D with A high similarity value ($\bar{\varphi} > 0.9$). The column "Quality" shows the expert evaluation: (+), (o), and (–) indicate a small, an acceptable, and a large modification effort to transform the computer solution into a solution accepted by the human expert. The underlying computing platform was a standard PC

expert. The experimental results presented in Table 3 report on such a competition. $\bar{\varphi}$ defines the averaged similarities of φ with respect to F. An automatically generated design is computer-accepted if $\bar{\varphi} \in [0.9; 1]$. The computer-accepted designs where analyzed by human experts with respect to their "quality", expressed as the real modification effort that is necessary to transform them into a human-acceptable solution. As expected, the more complex F becomes, the weaker develops the quality of the computer solutions: for small and medium-sized circuits the computer solution needs marginal rework. On the other hand, the computer solution for complex demand sets can still be repaired with acceptable effort by a human expert.

Simulation The automation of design problem solving in the described manner, including nearly arbitrary axes composition, circuit topology modification, and component parameterization, requires excellent simulation capabilities. From the simulation standpoint the key challenge pertains to model formation, i. e., the automatic transformation of a high-level circuit description in the form of a diagram down to an algorithmic model [42]. The simulation engine in FLUIDSIM comes along with the necessary technology: (a) an uncausal simulation language,[7] (b) recent algorithms for the analysis of so-called stiff systems [8,13], and (c) a knowledge-based interplay between the compilation of model equations and the application of an integrator's solution equations. This way it can resemble the famous DASSL algorithm [29], but also apply an inline integration strategy to several integration procedures [9].

Another strong point is the tight integration of computer algebra at simulation runtime, which provides a high level of flexibility for behavior analysis. Nonetheless simulation performance is not compromised since the simulation engine has a just-in-time compiler built in. Meyer zu Eißen and Stein [23] discuss how this simulation engine is provided as a Web service.

[7] FLUIDSIM implements the Modelica™ language for the modeling of physical systems (http://www.modelica.org). Modelica is an open and standardized specification, following the state of the art modeling paradigms [25,26].

Conclusion

The success of a design approach depends on the underlying model space—say, its size, and the way it is explored. A tractable model space is in first place the result of adequate models, which in turn are the result of a skillful selection, combination, and customization of existing construction principles. Human designers need little search because they use adequate models.

This paper introduces the functional abstraction paradigm as such a modeling principle and shows how it is applied to the field of fluidic engineering. Its key idea is to construct suboptimum candidate solutions within a simplified synthesis space, which then must be repaired. But even with such a simplification the automated design of fluidic circuits remains a challenge. This is reflected by the combined use of several powerful design technologies like design graph grammars, case-based reasoning, or uncausal simulation.

Our operationalization of the design approach is promising: the experiments show for medium-sized circuit acceptable design results. Note that even a design of medium quality can be used by a human expert as starting point; moreover, the presented design approach is not a dead end: it can be improved by enlarging the case base or by extending the repair language.

References

1. Antonsson, E., Cagan, J.: Formal Engineering Design Synthesis. Cambridge University Press, Cambridge (2001) ISBN 0-521-79247-9
2. Brown, D., Chandrasekaran, B.: Design Problem Solving. Morgan Kaufmann, CA (1989)
3. Buchanan, B., Shortliffe, E.: Rule-Based Expert Systems. The MYCIN Experiments of the Stanford Heuristic Programming Project. Addison-Wesley, Massachusetts (1984)
4. Chandrasekaran, B., Mittal, S.: Deep versus compiled knowledge approaches to diagnostic problem-solving. In: Waltz, D. (ed.) Proceedings of the National Conference on Artificial Intelligence, Pittsburgh, PA, pp. 349–354. AAAI Press, Cambridge (1982). ISBN 0-86576-043-8
5. Clancey, W.: Heuristic classification. Artif. Intell. **27**, 289–350 (1985)
6. David, J., Krivine, J., Simmons, R. (eds.) Second Generation Expert Systems. Springer, New York (1992) ISBN 0-387-56192-7
7. Davis, R.: Expert systems: where are we? And where do we go from here? AI Mag. **3**(2), 3–22 (1982)
8. Dormand, J.: Numerical Methods for Differential Equations. CRC Press, New York: London, Tokyo (1996)
9. Elmqvist, H., Otter, M., Cellier, F.: Inline integration: a new mixed symbolic/numeric approach for solving differential-algebraic equation systems. In: Proceedings of the European Simulation Multiconference, ESM'95, Prague, Czech Republic, June 1995, pp. xxiii–xxxiv
10. Fischer, G., Nakakoji, K., Ostwald, J., Stahl, G.: Embedding critics in design environments. Knowl. Eng. Rev. **8**(4), 285–307 (1993)
11. Forbus, K., de Kleer, J.: Building Problem Solvers. MIT Press, Cambridge (1993). ISBN 0-262-06157-0
12. Gero, J.: Design prototypes: a knowledge representation scheme for design.. AI Mag. **11**, 26–36 (1990)
13. Hairer, E., Wanner, G.: Solving Ordinary Differential Equations II. Stiff and Differential-Algebraic Problems, 2nd edn. Springer, New York (1996)
14. Hinrichs, T., Kolodner, J.: The roles of adaptation in case-based design. In Proceedings of Cambridge AAAI Press / MIT Press, AAAI (1991)
15. Hoffmann, M.: Zur Automatisierung des Designprozesses fluidischer Systeme. Dissertation, University of Paderborn, Department of Mathematics and Computer Science (2000)
16. Hägglund, S.: Introducing expert critiquing systems. Knowl. Eng. Rev. **8**(4), 281–284 (1993)
17. Karbach, W., Linster, M.: Wissensakquisition für Expertensysteme. Carl Hanser Verlag Munich. (1990) ISBN 3-446-15979-7
18. Kumar, H., Krishnamoorthy, C.: A framework for case-based reasoning in engineering design. Artif. Intell. Eng. Des. Anal. Manuf. **9**(3), 161–182 (1995)
19. Kühl, M., Reichmann, C., Spitzer, B., Müller-Glaser, K.: Eine durchgehende Entwurfsmethodik für das Rapid Prototyping von eingebetteten Systemen. In: Workshop Modelltransformation und Werkzeugkopplung (2001)
20. Leake, D.: Case-Based Reasoning: Issues, Methods, and Technology (1995)
21. Maher, M., de Silva Garza, A.: Case-based reasoning in design. IEEE Expert **12**(2) (1997)
22. McDermott, J.: R1: a rule-based configurer of computer systems. Artif. Intell. **19**, 39–88 (1982)
23. Meyer zu Eißen, S., Stein, B.: Realization of web-based simulation services. Comput. Ind. Spec. Issue Adv. Comput. Support. Eng. Serv. Processes Virtual Enterp. **57**(3), 261–271 (2006)
24. Minsky, M.: Models, minds, machines. In: Proceedings of the IFIP Congress, pp. 45–49 (1965)
25. Modelica Association: Modelica—A Unified Object-Oriented Language for Physical Systems Modeling: Tutorial. Modelica Association, Linköping, Sweden (2000a)
26. Modelica Association: The Modelica Specification, version 2.0. Modelica Association, Linköping, Sweden (2000b)
27. Oh, V., Langdon, P., Sharpe, J.: Schemebuilder: an integrated computer environment for product design. In: Computer Aided Conceptual Design. Lancaster International Workshop on Engineering Design (1994)
28. Paredis, C., Diaz-Calderon, A., Sinha, R., Khosla, P.: Composable models for simulation-based design. Eng. Comput. **17**(2), 112–128 (2001)
29. Petzold, L.: A description of DASSL, a differential-algebraic system solver. In: Scientific Computing, pp. 65–68 (1983)
30. Puppe, F.: Systematic Introduction to Expert Systems, Knowledge Representations and Problem-Solving Methods. Springer, New York (1993)
31. Purvis, L., Pu, P.: An approach to case combination. In: Proceedings of the Workshop on Adaptation in Case Based Reasoning, European Conference on Artificial Intelligence (ECAI 96). Budapest, Hungary (1996)
32. Raphael, B., Kumar, B.: Indexing and retrieval of cases in a case-based design system. Artif. Intell. Eng. Des. Anal. Manuf. **10**, 47–63 (1996)
33. Reiter, R.: A theory of diagnosis from first principles. Artif. Intell. **32**(1), 57–95 (1987)
34. Richter, M.: The knowledge contained in similarity measures, October 1995. Some remarks on the invited talk given at ICCBR'95 in Sesimbra, Portugal (1995)
35. Richter, M.: Introduction to CBR. In: Lenz, M., Bartsch-Spörl, B., Burkhard, H.-D., Weß, S. (eds.) Case-Based Reasoning Technology. From Foundations to Applications. Lecture Notes in Artificial Intelligence 1400, pp. 1–15. Springer, Berlin (1998)
36. Rychener, M.: Expert Systems for Engineering Design. Academic Press, Dublin (1988) ISBN 0-12-605110-0
37. Schlotmann, T.: Formulierung und Verarbeitung von Ingenieurwissen zur Verbesserung hydraulischer Systeme. Diploma thesis, University of Paderborn, Institute of Computer Science (1998)
38. Schmidt, L., Cagan, J.: Configuration design: an integrated approach using grammars. ASME J. Mech. Des. **120**(1), 2–9 (1998)
39. Schulz, A., Stein, B., Kurzok, A.: On the automated design of technical systems. Technical Report tr-ri-00-218, University of Paderborn, Department of Mathematics and Computer Science (2001)
40. Sinha, R., Paredis, C., Khosla, P.: Behavioral model composition in simulation-based design. In: Proceedings of the 35th Annual Simulation Symposium, pp. 309–315. San Diego, California, (2002)
41. Steels, L.: Components of expertise. AI Mag. **11**(2), 28–49 (1990)
42. Stein, B.: Functional Models in Configuration Systems. Dissertation, University of Paderborn, Institute of Computer Science (1995)

43. Stein, B.: Model Construction in Analysis and Synthesis Tasks. Habilitation, Department of Computer Science, University of Paderborn, Germany (2001)
44. Tong, C.: Towards an engineering science of knowledge-based design. Artif. Intell. Eng. **2**(3), 133–166 (1987)
45. Vier, E.: Automatisierter Entwurf geregelter Hydrostatischer Systeme, vol. 795 of Fortschritt-Berichte VDI. Reihe 8. VDI, Düsseldorf (1999)