In: Proceedings of Qualitative Reasoning (QR2019)

Insane Design Of Lumped Element Models

John T Maxwell III[1] , **Johan de Kleer**[1] and **Matt Klenk**[1]
[1]Palo Alto Research Center
{maxwell, dekleer, klenk}@parc.com

Abstract

The conceptual design process is the task of generating abstract solutions to a design problem. We show that exhaustive enumeration and evaluation of the designs in a design space (e.g., *insane design*) can be made practical for small designs using qualitative reasoning. We also show that it is possible to detect when we have found all of the behaviorally distinct designs in a given design space, and that under certain conditions the time that it takes to find all possible designs grows linearly in the number of components allowed in a design.

1 Introduction

In the conceptual design process, designers explore not only the space of possible solutions but also requirements. This is an ideal problem for qualitative reasoning which enables analysis of potential designs with under-specified parameters and requirements. In previous work, we have shown how search over component topologies combined with qualitative simulation (*envisionment*) enables the identification of conceptual designs that may yield promising solutions [Klenk *et al.*, 2012]. In this work, we address two challenges that must be overcome in next generation conceptual design tools:

- How to represent an exponential set of conceptual designs?
- How to efficiently evaluate these conceptual designs?

The contributions of this work are:

- An automated process for constructing design grammars.
- A formal analysis of the size of this design grammar in terms of the number of components and number of interconnections.
- A new method for identifying behaviorally equivalent designs and terminating search when all behaviorally equivalent designs have been found.
- A case study applied to a design problem of moving water uphill.

We organize the rest of this paper as follows. We begin by outlining our system called the *insane designer*. Next, we introduce design grammars and describe how we automatically construct them from a component model library and design scenario. This is followed by sections describing behavioral equivalence classes and how we identify them. Next, we complement the formal analyses with a case study of designing systems to move water uphill. We close with a discussion of related work and open questions.

2 Design Generation

We call the process of exploring all possible designs *insane design*. Figure 1 provides an overview of our insane designer.

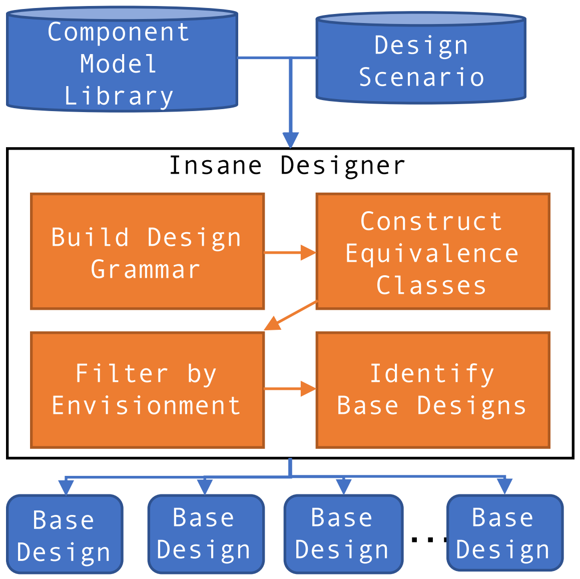

Figure 1: Insane designer takes a component model library and a design scenario as input and returns all minimal conceptual designs that satisfy that scenario.

We are interested in generating all possible conceptual designs up to a given number of components that satisfy a given design scenario. The design scenario is a set of components and connections with an interface to a design along with requirements specified as constraints on qualitative values of the components. (cf Figure 2).

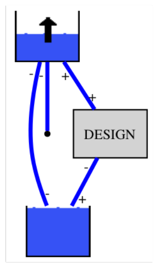

Figure 2: Sample design scenario to move water uphill. Two tanks are connected to an interface. The pressure of one of the tanks is less than the other, and the requirement is that water must flow into the lower pressure tank.

Along with the scenario, we are given a library of components that we can use to create designs. The components in the library can include resistors, capacitors, and batteries (for the electrical domain); pipes and tanks (for the fluid domain); torque sources and inertial discs (for the rotational domain); and pumps and motors (for cross-domain designs) (cf Figure 3).

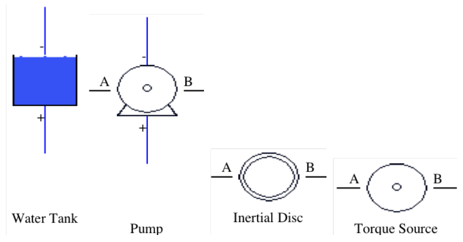

Figure 3: Sample lumped element components

The components have both topological information (e.g., typed connections that determine how components can be connected together) and behavioral information (e.g., differential and algebraic equations that link the connections within a component). The goal is to find designs made up of components from the library that are connected according to their typed connections which satisfy the scenario goals when the designs are substituted for the placeholder in the design scenario.

To make exhaustive enumeration practical, we first create a *design grammar*. The design grammar is a context-free grammar that describes the space of possible designs. Every design that the design grammar describes is a topologically valid design (e.g., connections between components have the right type, and there are no dangling connections not connected to another component or interface).

When we enumerate designs in the design grammar for evaluation, we often encounter designs that are equivalent from a behavioral point of view. For instance, a capacitor and a resistor in series has the same behavior as a resistor and a capacitor in series in a lumped element model. Also, two resistors in series have the same qualitative behavior as one resistor (the two resistors can be replaced with one resistor whose resistance is the sum of the two resistors). Since it is expensive to envision a design to determine whether it satisfies the scenario's goals, it is worthwhile to group equivalent designs into *equivalence classes*.

We perform *envisionment* on the simplest design in each equivalence class in order to determine whether it satisfies the scenario. If the simplest design satisfies the scenario, then all of the designs in the equivalence class also satisfy the scenario. We can enumerate the designs in such equivalence classes to obtain all of the conceptual designs that satisfy the scenario. Some of these designs will be trivial variations of other designs. For instance, one design might add a spurious component to another design that satisfies the scenario. We detect this by deleting components one at a time from a successful design to see whether the design still satisfies the scenario. If it does, then we discard the design as spurious. If there is no deletion that produces a design that satisfies the scenario, then we call that design a *base design*. The output of the insane designer is the set of all base designs that satisfy the scenario.

In this paper, we analyze how the performance of the insane design algorithm scales with the number of components allowed in a design. Doing this analysis shows that exhaustive enumeration is impractical except for trivially small design spaces because the search space becomes very large as the number of components increases. To make exhaustive enumeration practical, we need to limit the search space. Our goal is to limit the search space in such a way that the resulting designs are useful and that the designer understands the implications of the limitations. We limit the design space by letting the user specify an upper bound on how interconnected the design can be and an upper bound on how many state variables are allowed in a design. We explain these limitations in later sections.

We enumerate designs by the number of components they contain. So, first we enumerate all one component designs, then all two component designs, and so on. We build on the designs already done in previous iterations, so that three component designs are built on the two component designs that we have already found. We show that we can detect when there are no more qualitatively distinct designs to be found, allowing us to stop enumerating designs (further designs may be interesting for non-qualitative reasons). If we continue, the amount of work needed is linear in the number of components that we add because we limit the maximum number of state variables in the conceptual designs.

3 Design Grammars

A design grammar has *non-terminals* that are *interfaces* between sets of components and *terminals* that are components from the component library. Each interface is defined by a set of connections and the number of components that its implementation ultimately contains. There are two types of *rules* in our design grammar: (1) interface(N) $\rightarrow$ component interface($N-1$), where N is the number of components of the interface to the left of the arrow, and (2) interface(1) $\rightarrow$ component. We call the interface to the left of the arrow the *primary* interface, and the interface to the right of the arrow the *dependent* interface. The rules also include information about how the interfaces are connected to the component and to each other. We only allow *well-formed rules*, rules where all of the connections have the right type and there are no empty connections.

Even though we only allow one component and one dependent interface per rule, it is possible to create complex design topologies with branches in them (e.g., designs where one component is connected to three or more other components) because there can be connections between the two interfaces that bypass the rule's component[1].

To understand how a design grammar works, imagine one took a design and laid it out as a sequence of components with the external connections at the left (cf Figure 4).

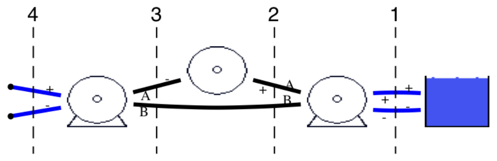

Figure 4: Component sequence with design interfaces shown

Between each pair of components there is a vertical dotted line that represents an interface. The number above the dotted line is the number of components to the right of the interface. The interface also contains a set of connections between components. The area between a pair of interfaces implicitly defines a rule that consists of a component and a set of connections between the component and the interfaces and between the two interfaces.

There may be more than one way to implement an interface using a component. We generate all of the rules for a particular interface by iterating over the components in the library. For each component, we first iterate over the connections and produce a class of rules for each connection by connecting the connection to the first empty connection in the interface of the correct type, if such a connection exists. We then recursively connect the remaining connections in the component. For each connection, we allow two possibilities: (1) Attach the connection to the first empty connection in the interface of the correct type, and (2) to attach the connection to the dependent interface. If there are any empty connections in the primary interface, we add them to the dependent interface. When we are done, we create a dependent interface with a component count of $N-1$, where N is the number of components in the primary interface.

We call the collection of rules that implements a primary interface a *design family*, and we call each rule in a design family a *design alternative*. A design grammar is the collection of design families that are needed to implement the interface specified by a scenario. Figure 5 shows the design families of a two-component design grammar along with their design alternatives and dependent design families (the components and their connections are not shown). The design families are oval, the design alternatives are rectangular.

[1]We could have allowed more than one dependent interface per rule, but it would have made the exposition of the ideas in this paper more complicated.

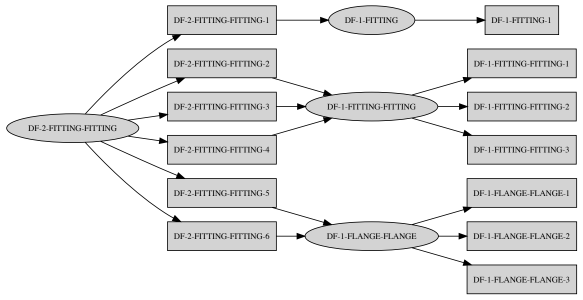

Figure 5: Graph of four design families (ovals) in a design grammar specifying all two component designs. This grammar captures 19 conceptual designs. Each conceptual design is a different path through the tree. There is one path to the top leaf, 3 paths to each of the next three leaves, and 2 paths to the last three leaves $(1 \times 1) + (3 \times 3) + (2 \times 3) = 19$.

4 Analysis of Design Grammar Size

The size of a design grammar is a function of the number of design families and the number of design alternatives per design family. The number of design families in a design grammar depends on the number of interfaces possible. The number of interfaces depends on the maximum number of components that are allowed in our designs and the number of different sets of connections possible. The number of connections is a function of the maximum number of components allowed in a design, which is n. To see this, note that each component in a rule could add at most $m-1$ connections when moving from the primary interface to the dependent interface, where m is the maximum number of connections of any component in the library. This means we could have mn connections in the worst case. Because m is fixed for a given component library, there are at most $O(n)$ connections in an interface.

If there were only one kind of connection, then there would be $O(n)$ sets of connections, one set for each possible number of connections (order does not matter). If there were two kinds of connections (e.g., fluid and rotational mechanics), then we would have to worry about how many there were of each. Order does not matter, but we could have 0, 1, 2, 3, ..., n different connections of each type. So then there are $O(n^2)$ different sets of connections. In general, if there are k different kinds of connections, there can be $O(n^k)$ different sets of connections in the worst case. So, there can be at most $O(n^{k+1})$ different interfaces if we include the different component counts that an interface can have. The number of design alternatives per design family is a function of the number of components in the library and the maximum number of connections per component, but it is not a function of n. Therefore, the size of the design grammar is $O(n^{k+1})$ in the worst case.

We can see that as the number of components in a design grows, the size of the design grammar grows very quickly, especially with cross-domain designs. To make design generation practical, we impose a bound on the number of connections that an interface can have. This corresponds to a limitation on how connected a design is overall. Limiting the number of connections makes the designs more modular. Consider if the maximum number of connections allowed is c. Then the number of different sets of connections is a function of c, but it is no longer a function of n. Therefore, the size of the design grammar is $O(n)$ in the worst case.

5 Design Equivalence

Two designs are equivalent if they exhibit the same behavior at their interfaces. We only consider linear systems and therefore there are only four types of equations. The first type arises primarily from definitions and conservation laws (v_i are the variables of the system):

$$\Sigma_i v_i = 0.$$

Examples of such equations arise from Kirchoff's voltage and current laws. The second type arises from constitutive laws:

$$v_i = K_{ij} v_j.$$

An example of such an equation is Ohm's law: $v = iR$. The third type arises from state variables:

$$v_i = D_{ij} \dot{v}_j.$$

An example of such an equation describes the behavior of a capacitor: $i = C\dot{v}$. The fourth type arises from inputs:

$$v = 1.5.$$

For example the battery supplies 1.5 volts.

From systems theory we know that every system can be described by:

$$\dot{x} = Ax + Bu(t) \tag{1}$$

x are the state variables and the matrices A and B will be unique up to permutation. Unfortunately, in our a approach a design is not a complete system because of the external connections and cannot be described in this way. For example, a design consisting only of a resistor has three external variables i_1, i_2 and v. The current through the resistor will be determined by how it is connected to other designs. A design may have fewer equations than unknowns. It is only a fragment of a system; only a complete system with many composed design can be characterized by Equation 1. In order to reduce the search space we want to detect duplicate designs as soon as possible. These designs may have different component connection topologies and may have different numbers of components. Although there are extensive types analyses from systems theory, here we apply a simplified approach which provides a significant computational advantage. It is computationally too expensive to detect all equivalent designs according to systems theory.

We attempt to rewrite the equations into an equivalent set referring only to interface and state variables. From the original set of equations we use Guassian elimination to remove all the internal non-state variables.

If (2) is a set of equations with external variables P_{11} and P_{13}, positive constants c_2 and c_4, and state variable P_{15}, then (3) is its simplified equivalent in standard form.

$$\begin{gathered} Q_2 = d/dt(c_4 P_{15}) \\ P_9 = -c_2 Q_2 \\ P_{15} = P_1 - P_{13} \\ P_9 + P_{11} = P_1 \end{gathered} \tag{2}$$

$$P_{15} = P_{13} - P_{11} - d/dt(c_{18} P_{15}) \tag{3}$$

The analysis can sometimes determine a set of equations is inconsistent. We discard designs which have inconsistent equations. For instance, the equation $c_i = -c_j$ is inconsistent if c_i and c_j are constant parameters that are known to be positive.

In most cases, the equations can be simplified to the form shown in Equation 4 (V_i are only the interface and state variables):

$$C_i + \Sigma_j D_{ij} V_j + \Sigma_j E_{ij} \dot{V}_j = 0. \tag{4}$$

Where all the coefficients C_i, D_{ij}, E_{ij} can be algebraic functions of the parameters of the design. In matrix form,

$$C + DV + E\dot{V} = 0.$$

(This is a generalization of Generalized Thevenin form.) Given a lexographic order we reduce these equations to a canonical form. A design with a set of equations equivalent to those of another modulo permutation of the ordering is equivalent. This approach greatly reduces the space explored.

6 Analysis of the Number of Distinct Designs

The number of distinct designs is a function of the number of equations per design and the number of different equations that are possible. The number of equations per design is a function of the number of distinct variables used by the design, which is the number of external variables plus the number of state variables. The number of external variables is not a function of n since the number of connections is bounded, but the number of state variables can be a function of n since each component could introduce one or more state variables. Therefore, we let the user specify a maximum number of state variables allowed in a design. This means that the number of distinct variables is no longer a function of n.

Since the total number of variables allowed is bounded in the insane designer, the number of equations in the standard form is bounded (there is at most one equation per variable). So the number of equations per design is not a function of n, and the number of variables per equation is not a function of n.

Unfortunately, the number of different coefficients for each variable is a function of n since each coefficient can be an arbitrary algebraic expression of the $O(n)$ component parameters in the design. If it were possible to always simplify the coefficients to a single parameter, then the number of distinct equations possible would be bounded and there would only be a finite number of distinct conceptual designs, independent of the number of components that a design could have. This would only be true because we had set a limit on the number of state variables and the number of connections per interface.

7 Finding Equivalence Classes

The design scenario doesn't impose any conditions on the form of the design other than its external connections and the maximum number of components that it can have. Therefore, we do not need the design itself to do evaluation, just the equations that describe its behavior. We can use the design grammar to efficiently enumerate all of the distinct sets of equations if we take design equivalence into account.

We create equivalence classes for the design families bottom up. We begin with design families that only have one component. For each design family, we enumerate each design alternative associated with the family, extract its equations, put the equations in standard form and simplify their coefficients. When we have done this for all of the design alternatives in a design family we group the equations into equivalence classes, and associate the equivalence classes with the design family.

Next we process design families that have two components. For each design family, we enumerate each design alternative associated with the family. For each design alternative, we enumerate the equivalence classes of its dependent design family. For each equivalence class, we combine the equations for the equivalence class with the equations for the design alternative and standardize and simplify the result. When we have done this for all of the design alternatives we group the sets of equations into equivalence classes, and associate the equivalence classes with the design family.

We continue this process for each number of components until we reach the maximum number of components specified by the user. We then run the envisioner on the smallest design in each equivalence class and determine whether the scenario goals are satisfied. We discard any equivalence classes that do not satisfy the scenario goals.

We can now enumerate all of the designs that satisfy the scenario by starting with an equivalence class that satisfies the scenario goals and recursively choosing one design alternative from each dependent equivalence class. Each resulting design is syntactically well-formed and satisfies the scenario goals.

8 Early Termination of Search

As described above, we stop creating equivalence classes when we reach design families that have the maximum number of components specified by the user. Sometimes, though, we can detect that no more distinct designs are possible, and we can stop creating equivalence classes early.

We detect that no more distinct designs are possible when the number of equivalence classes for each design family at a particular number of components are the same as the number of equivalence classes for the corresponding design family at the previous number of components. For instance, Table 1 shows the number of equivalence classes for a very simple design problem that only has four interfaces: Int1, Int2, Int3, and Int4. Each cell corresponds to a design family where the row indicates the interface and the column the number of components to the right of the interface. Notice that starting at 12 components, the number of equivalence classes for each design family is the same as the number of equivalence classes for the corresponding design family at the previous number of components.

	7	8	9	10	11	12	13	14
Int1	4	4	4	4	4	**4**	**4**	**4**
Int2	2	2	2	2	2	**2**	**2**	**2**
Int3	43	49	51	51	51	**51**	**51**	**51**
Int4	22	24	26	28	30	**30**	**30**	**30**

Table 1: Number of equivalence classes per design family

If we are only interested in behaviorally distinct designs, then there is no point in continuing to create equivalence classes after 12 components. One reason for continuing to create equivalence classes is that there might be new designs that are behaviorally the same as the designs found so far, but which have other interesting properties based on the topology of their components. If we continue creating equivalence classes, the amount of additional work done will be linear in the number of components because it only depends on the equivalence classes at the previous step.

9 Case Study: Moving Water Uphill

To demonstrate the importance of equivalence classes, we present a case study of applying the insane designer on the problem of moving water uphill. We are given a scenario (Figure 2) and a small library of components: water tank, pump, inertial disc, and torque source (cf Figure 3) plus a pipe, a fluid ground, and a torque ground. We generate all possible designs from 4 components up to 7 components. We limit the designs to a single state variable and three connections per interface.

components	designs	classes	time (s)
4	462	29	1
5	2255	60	2
6	11601	87	6
7	64576	144	21

Table 2: Performance of insane design with increasing components

Table 2 shows how the insane designer performs as the number of components increases (envisionment time is excluded). Note that while the number of designs is clearly exponential, the number of equivalence classes is almost linear.

In the 4 component design, the design grammar has 16 design families and 54 design alternatives. This design grammar is a packed representation of 462 different possible designs with only 29 equivalence classes of designs. The process so far takes less than a second. The envisioner then evaluates the simplest design from each of the 29 equivalence classes and determines that 16 of them satisfy the scenario. This takes 55 seconds. It then enumerates 292 designs that satisfy the scenario from the 16 equivalence classes. Of the 292 designs, there are only 9 distinct base designs, a few of which are shown in Figure 6.

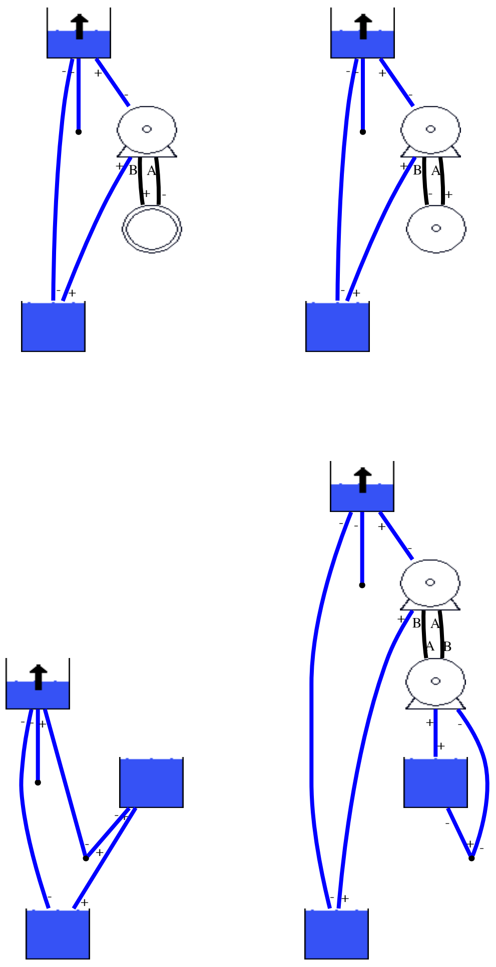
Figure 6: Sample designs for moving water uphill. The top two designs are an spinning inertial disc (flywheel) and a torque source powering a pump. The lower left design increases the water in the upper tank by adding another tank with a greater pressure. The lower right design is a water wheel where falling water is used to spin another wheel that raises water.

10 Related Work

Automated conceptual design approaches can be categorized by the types of domain knowledge utilized. The first set of approaches include those that use knowledge in addition to behavioral models to construct conceptual designs. These techniques can be divided into three groups [Chakrabarti *et al.*, 2011]: (1) Function-based synthesis employ functional decompositions to describe design alternatives [Stone and Wood, 2000] and hierarchically constrain the search space [Neema *et al.*, 2003]. (2) Analogical design creates new conceptual designs by adapting existing designs organized with semantic knowledge in a library [Goel *et al.*, 2015]. (3) Graph-grammar methods formulate the conceptual design problem as a set of grammar rules that specify how to change a graph representing the design [Kurtoglu *et al.*, 2010]. Of the knowledge-based approaches, our approach is most closely related with the Graph-grammar methods, but our grammar is generated automatically from the types of connections as opposed to being hand-authored by a designer.

The second set of approaches use only behavioral models of the components to identify conceptual designs that match a particular behavior. These approaches typically take a optimization or other search methods. For example, genetic algorithms have been used to design analogue circuits that meet a behavioral specification [Grimbleby, 2000]. Another approach uses quantified satisfaction to design digital circuits [Feldman *et al.*, 2019]. Our approach is similar to the above in that we use only behavioral models of the components. Our approach differs in that our behavioral models capture the qualitative behavior of the continuous quantities in the design, as opposed to quantitative or discrete models.

11 Conclusion

In this paper, we have introduced an insane designer that enumerates conceptual designs using a library of component models. Each conceptual design is qualitatively verified in that its envisionment in the design scenario indicates that it may meet the requirements. The core contribution of this paper is a procedure to automatically generate a scalable design grammar that captures all possible conceptual designs, an efficient way of identifying feasible designs, systematic analyses of these methods, and an empirical evaluation with a case study of a design scenario for moving water uphill.

In addition to evaluating on additional design scenarios, we plan to connect the insane designer to quantitative simulators (e.g., Modelica [Fritzson, 2004]) and use the qualitative model to guide quantitative parameter search for a set of components. Because qualitative simulation is exponential and we use it to evaluate conceptual designs, we will explore new envisionment techniques that only expand trajectories that may reach the desired behavior. This include expanding the design scenario with temporal logic [Vardi, 2008].

Our goal is to develop conceptual design techniques that can be adapted to augment human designers by searching and summarizing large search spaces.

Acknowledgements

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA) under contract number HR001118C0041. The views, opinions, and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

References

[Chakrabarti *et al.*, 2011] Amaresh Chakrabarti, Kristina Shea, Robert Stone, Jonathan Cagan, Matthew Campbell, Noe Vargas Hernandez, and Kristin L Wood. Computer-based design synthesis research: an overview. *Journal of Computing and Information Science in Engineering*, 11(2):021003, 2011.

[Feldman *et al.*, 2019] Alexander Feldman, Johan de Kleer, and Ion Matei. Design space exploration as quantified satisfaction. *arXiv preprint arXiv:1905.02303*, 2019.

[Fritzson, 2004] P. Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Press, Piscataway, NJ, 2004.

[Goel *et al.*, 2015] Ashok K Goel, Gongbo Zhang, Bryan Wiltgen, Yuqi Zhang, Swaroop Vattam, and Jeannette Yen. On the benefits of digital libraries of case studies of analogical design: Documentation, access, analysis, and learning. *AI EDAM*, 29(2):215–227, 2015.

[Grimbleby, 2000] James B Grimbleby. Automatic analogue circuit synthesis using genetic algorithms. *IEE Proceedings-Circuits, Devices and Systems*, 147(6):319–323, 2000.

[Klenk *et al.*, 2012] Matthew Klenk, Johan de Kleer, Daniel G Bobrow, Sungwook Yoon, John Hanley, and Bill Janssen. Guiding and verifying early design using qualitative simulation. In *ASME 2012 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, pages 1097–1103. American Society of Mechanical Engineers, 2012.

[Kurtoglu *et al.*, 2010] Tolga Kurtoglu, Albert Swantner, and Matthew I Campbell. Automating the conceptual design process:"from black box to component selection". *AI EDAM*, 24(1):49–62, 2010.

[Neema *et al.*, 2003] Sandeep Neema, Janos Sztipanovits, Gabor Karsai, and Ken Butts. Constraint-based design-space exploration and model synthesis. In *EMSOFT*, pages 290–305, 2003.

[Stone and Wood, 2000] Robert B Stone and Kristin L Wood. Development of a functional basis for design. *Journal of Mechanical design*, 122(4):359–370, 2000.

[Vardi, 2008] Moshe Y Vardi. From church and prior to psl. In *25 years of model checking*, pages 150–171. Springer, 2008.