Evaluating Design Concepts through Functional Model Simulation

Bryan Wiltgen BRYAN.WILTGEN@GATECH.EDU
Ashok K. Goel GOEL@GATECH.EDU
Design & Intelligence Laboratory, School of Interactive Computing, Georgia Institute of Technology, Atlanta, Georgia 30332, USA

Abstract

Evaluation is a key task in design, and a major goal in computational design is to develop techniques for evaluating designs throughout the design process, starting as early in the process as possible. We describe a computational technique for evaluating design concepts early in conceptual phase of engineering design. Conceptual design in engineering is abstracted as a function to structure mapping and engages the use of functional models of the design concepts. We describe a computational technique called SBFCalc that evaluates design concepts through simulation of their functional models. We demonstrate the capabilities of SBFCalc for evaluating design concepts in biologically inspired engineering design that uses biological analogues to address engineering design problems.

1. Introduction

Design is a fundamentally iterative process of design generation, evaluation and redesign (Dym & Brown, 2012; Pahl et al. 2007). This is because design problems often address complex systems, because designers are often encouraged to be creative, because design concepts often fail, and because the cost of failure of actual designs often can be large. Indeed, evaluation, failure and iteration are so prevalent in design practice that "fail early, fail often" has emerged as a mantra in many a design community. Early and frequent evaluation of design ideas can help expose the structure and the constraints of the design problem space, focus the designer's attention to more productive lines of search and exploration, and help reframe and reformulate the design problem.

Thus, a major goal in research on computational design is to develop techniques for evaluating designs throughout the design process, starting as early in the process as possible. Indeed, the computational design research has built many methods for evaluating designs, ranging from design criticism to geometric modeling to numerical simulation to virtual and physical prototyping, etc. However, most of these evaluation methods are useful only relatively late in the design process, after the conceptual design phase already has been completed. The question thus becomes how can we evaluate engineering designs in the conceptual design phase itself?

In this paper we present a computational technique for evaluating design concepts in engineering design. Conceptual design in engineering is abstracted as a function to structure mapping and engages the use of functional models of the design concepts (Dym & Brown, 2012; Pahl et al. 2007). Thus, one strategy for evaluating design concepts is through functional model simulation. We describe a computational technique called SBFCall for this task.

In the use case for this work, a designer wants to use analogical reasoning to address a given problem in engineering design. After conducting analogical reasoning and coming up with a proposed design, the designer wants to verfy the functional model of that design. To do so, the designer inputs her functional model into SBFCalc, which in turn leverages qualitative and quantitative simulation of the functional model, comparing the results of that simulation to the content of the inputted functional model to detect errors or misconceptions. SBFCalc then outputs its evaluation for the designer to inspect.

2. Related Research

This work builds on several lines of research: conceptual design, functional modeling, qualitative reasoning, analogical design, and biologically inspired design. The process of engineering design consists of several phases; problem formulation and conceptual design are the earliest phases of design (Dym & Brown 2012; Pahl et al. 2007). The task of conceptual design takes a desired function as the input; the goal of the conceptual design task is to output a structure that will deliver the desired function. Thus, the task of conceptual design is abstracted as a *function to structure mapping*. This is why languages for formulating design problems typically specify the functions desired of the design, the operating environment of the system, the performance criteria, and the constraints on the structure of the system (Helms & Goel, 2014; MacLellan et al., 2013).

This abstraction of the conceptual design as a function to structure mapping has led to the development of several functional models of designs (Chandrasekaran, Goel & Iwasaki, 1993; Gero, 1990; Gero & Kannengiesser, 2004; Kitamura et al. 2004; Rasmussen, 1985; Sembugamoorthy & Chandrasekaran, 1986; Umeda et al., 1996; Umeda & Tomiyama, 1997). According to Simon (1996), a functional model of a design provides (i) a functional decomposition of the design, and (ii) a functional explanation of how the structure of the design delivers of the desired functions. Functional models typically use behavior as an intermediate abstraction to explain how the structure of the design achieves the functions desired of it. SBFCalc is based on a specific functional model called the *Structure-Behavior-Function (SBF) model* in which a behavior is a causal process that composes the functions of the subsystems into the functions of the system as a whole (Goel, Rugaber & Vattam, 2009; Goel & Stroulia 1996). SBF models have been used extensively in conceptual design. For example, Goel & Chandrasekaran (1989) used them for diagnosing a design failure during conceptual redesign.

Cognitive systems research on qualitative simulation also has a long history (de Kleer & Brown 1984; Forbus 1984; Kuipers 1986). More recently Bredeweg et al.'s (2009) Garp3 system allows a user to first create qualitative models of ecological systems and then simulate them (see also Qualitative Reasoning & Modeling, 2015). Similarly, the MILA-S system enables the user to first create conceptual models of ecological systems and then simulate them using NetLogo (Joyner, Goel & Papin 2014). Even closer is Klenk et al.'s (2012) work that describes a complementary method for evaluating design concepts: their method combines qualitative simulation with Modelica models of the designs. However, their method does not specifically capture function; SBFCalc combines qualitative simulation with functional modeling. However, because functional models are conceptual representations, qualitative simulation in SBFCalc is constrained and more similar to Rieger & Grinberg's (1976) *procedural simulation*.

Cognitive systems research on analogical reasoning too has a long history (Falkenhainer, Forbus & Gentner 1989; Hofstadter 1996; Holyoak & Thagard 1996). Falkenhainer (1987) evaluated an analogy by simulating a qualitative model of the target concept and comparing the

results of the simulation with the observations. Our work differs from Falkenhainer's because while Falkenhainer leverages observations for verification, our work compares the results of simulation itself to a constructed conceptual model to conduct verification. In addition, we plan to use our computational technique in the future to also validate the source analog of the analogy and maybe the target as well.

Analogical design is a very common method of conceptual design (Goel 1997). Biologically inspired design (Benyus 1997; Baumeister et al., 2012; Hoeller 2013; McKeag 2012) entails analogical design such that while the target design problems come from engineering and other design domains, source analogues come from biology. The rapidly growing movement of biologically inspired design is driven in large part by the need for environmentally sustainable designs. Goel, McAdams & Stone (2014) provide a compilation of recent progress on computational theories, techniques and tools for biologically inspired design.

3. Research Problem

Let us consider the design of the Japanese Shinkansen train described by McKeag (2012) and modeled by Hoeller (2013) as an illustrative example of our research problem. McKeag describes the successful efforts to redesign high-speed Shinkansen trains in the 1990s, which succeeded in part through biological inspiration. The goal was to design a faster train than the then Shinkansen 300 train, but the train produced too much noise at higher speeds because: (1) ground vibrations, (2) aerodynamic noise, and (3) sonic booms when they entered tunnels. Figure 1 visually depicts the biological analogy that aided designers in resolving the third problem. For resolving the problem of sonic booms when entering tunnels, the train designers took inspiration from the shape of beak of the kingfisher bird, which helped them design a new nose for the train. Figure 2 schematically illustrates the biological analogy that formed part of the resolution to the second problem: designers took inspiration from the fimbriae on owl wings and added a small vortex generator to their redesigned pantograph to help reduce the noise it made from turbulence.

Figure 1: The bullet shaped nose of the Shinkansen train was inspired in part by the shape of the kingfisher's beak. (Adapted from Biomimicry 3.8's Ask Nature.)

Let us imagine that the designers created a functional model of their proposed solution when they were developing the small vortex generator solution to the aerodynamic noise problem and that they wanted some way to quickly check if this model had any mistakes. With our computational technique, the designers could have given their functional model to it and rapidly received verification of their model. We do not propose in this work that our computational technique provides a complete solution for design concept verification. Instead, we propose it as

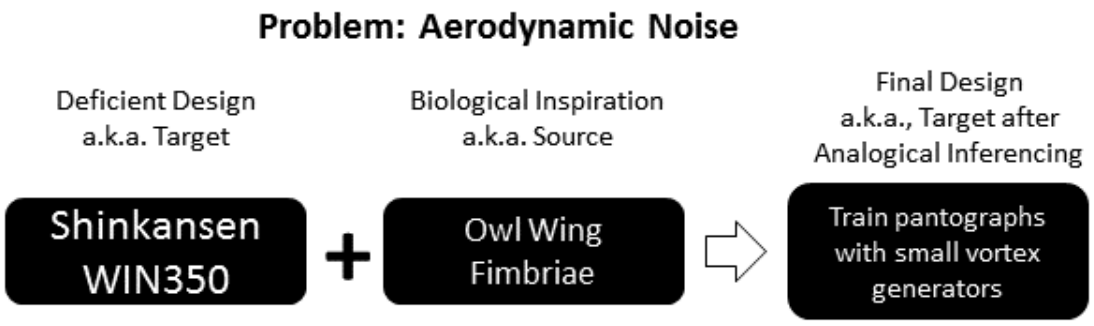

Figure 2. Visual depiction of the owl wing feather biologically inspired design analogy related to the Shinkansen train.

one step a designer or team of designers might take to quickly gather some sense of verification about their design concept.

3.1 A Functional Model of the Shinkansen Train

In this section, we describe a functional model that we developed of the proposed Shinkansen train with a small vortex generator attached to its pantograph and of how the train's interaction with air creates noise. That is, as the train moves, air flows over the pantograph, and specifically, it flows over the small vortex generator on the pantograph. This interaction eventually creates low turbulence, which in turn creates low noise that we attribute as the train making noise. This model is simplified and is intended as a proof-of-concept to demonstrate our computational technique. Although we tried to stay true to the source material, we did develop this model with it being a proof-of-concept for our computational technique in mind. For simplicity, we will hereafter refer to this model as the Shinkansen Train model.

The model we created is an SBF model. An SBF model is divided into three sub-models: a structure model, a behavior model, and a function model. The function model describes the intended or perceived purposes of the system. The behavior model describes the mechanisms by which the functions are achieved. Finally, the structure model describes the physical components, substances, and connections between the components that make up the system.

A function model is composed of one or more functions. Each function has several aspects and here we will focus only on those relevant to our computational technique. They are as follows: (a) a name that uniquely identifies it; (b) a "provides" condition that defines values of properties in the world that must be true at the completion of the function; and (c) a pointer to a behavior that provides an implementation of that function. The boxes in Figure 3[1] depict the functions associated with this train, with the top half of each box being the name of the function, the bottom half being the function's "provides" condition, and the arrow extending from the function to the B inside the circle representing a pointer to a behavior. The function named TrainGeneratesAerodynamicNoise is the top-level function of our model and points to the behavior that we will report on here.

[1] We note that despite whether they had surrounding quotation marks or not, we write all values in this paper as unquoted for simplicity. Surrounding quotation marks for values are ignored in SBFCalc's reasoning.

A behavior model is composed of one or more behaviors. Each behavior is itself composed of states and transitions. A state captures a moment in time and may be annotated as either a start state (from whence the behavior begins) or as a stop state (where the behavior ends). A state may have neither a start nor stop state annotation, in which case it is considered an intermediate state. A state has a condition, which is composed of a collection of component or substance properties and a value for each. A transition describes movement between two states, which we will in this paper call the Before state (where the transition begins) and the After state (where the transition ends). A transition is annotated with zero to many explanations which describe why or how the Before state became the After state. Figure 4 depicts the behavior that implements the TrainGeneratesAerodynamicNoise function. In this figure, boxes represent states with the name of the state in the top part and the condition in the bottom part, and arrows represent transitions. The behavior describes a train accelerating, turbulence forming in the air, and the air creating noise because of that turbulence. The state named StartState is the single start state for this behavior, and transitions between states are depicted as black arrows. For the sake of space, we did not include the transition explanations in this figure. Instead, we list them in Table 1. In that table, we group each set of explanations by its matching transition identifier from Figure 4. We explain the two types of explanations shown in Table 1, equation and function, later in this paper.

4. Technique

Our computational technique is called SBFCalc. Technically, SBFCalc is a client-side Java program that takes an SBF model as input (which we will call the inputted model). During reasoning, it will simulate the behaviors within a clone of the inputted SBF model (which we will call the inferred or simulated model), replacing the state conditions of the behaviors with properties and values that it infers through its simulations. SBFCalc then uses these simulated behaviors along with the inputted model to evaluate the function and behavior models of the inputted model.

4.1 Evaluating the Behavior Model

The behavior model in an SBF model is composed to zero to many behaviors. Each behavior describes the mechanism by which a function in the SBF model gets achieved. It does this by telling a kind of story that describes the states that a system goes through and explaining why the system transitions from one state to the next. When evaluating an SBF model, one must evaluate the behaviors of that model because they represent the low-level descriptions of how the functions are achieved and, thus, errors in behaviors reflect misconceptions (or modeling mistakes) about how the system works.

SBFCalc takes a two-step process to evaluate each behavior in the inputted behavior model. First, it simulates a copy of the behavior in the inferred model, setting the condition for each state in a behavior based on this simulation. Second, afterwards, it compares the matching behavior in the inputted model with the inferred behavior, detecting similarities and differences between each pair of behaviors. These similarities and differences represent the evaluation, with similarities reflecting aspects that SBFCalc agrees with in the inputted behavior and differences reflecting SBFCalc's disagreement and thus potential problems in the inputted behavior.

4.1.1 Simulating a Behavior

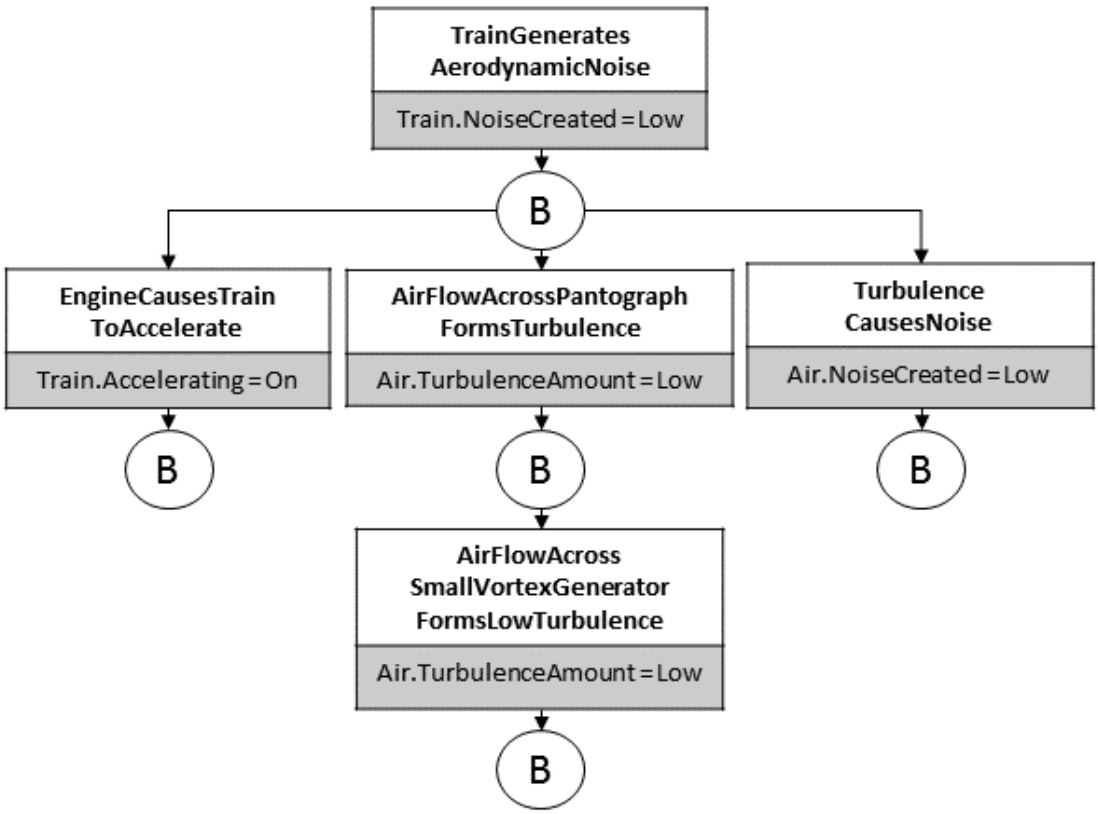

Figure 3. Function decomposition for the Shinkansen Train model. Each box represents a function and gives its name (top half) and "provides" condition (bottom half). A B in a circle represents a behavior. An arrow from a function to a behavior represents the behavior that the function points to, and an arrow from a behavior to a function represents a function that is in a function explanation for that behavior.

Here we describe how SBFCalc simulates a behavior. SBFCalc currently reasons about each behavior in a behavior model independently. For each behavior, it begins at a start state in that behavior, and it will traverse each outgoing transition and infer the values of the subsequent state using the equation and function explanations and using implicit value forwarding. To connect with our earlier terminology, the start state is the Before state and a subsequent state is an After state. SBFCalc will then recursively repeat this process for each subsequent state until it runs out of states to traverse. It will then move on to the next start state in the behavior and so on until there are no more start states to reason about.

Below, we describe how SBFCalc reasons about equation and function explanations, we describe the implicit value forwarding technique, and we conclude by discussing how all these techniques combine to infer an After state's condition.

4.1.1.1 Equation Explanations

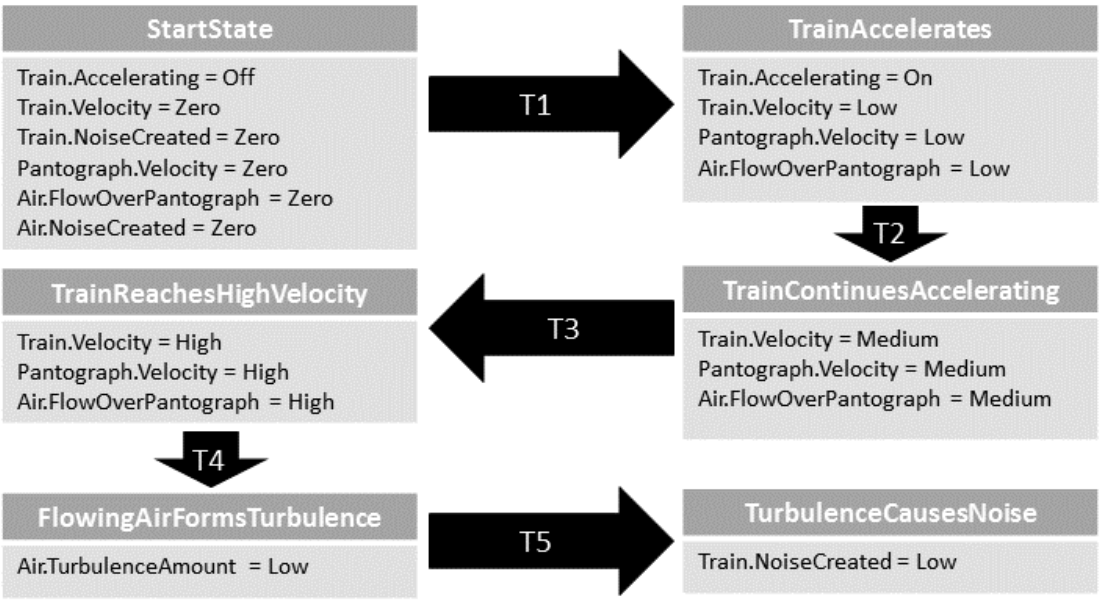

Figure 4. The behavior that that implements the TrainGeneratesAerodynamicNoise function in the Shinkansen Train model. The explanations on the transitions are listed in Table 1.

Recall that in a behavior a transition describes the movement of the modeled system from one state (the Before state) to another state (the After state), and a transition is annotated with zero to many explanations. Each explanation explains why or how some or all of the system moves from the Before state to the After state. Equation explanations are a kind of explanation. Each equation explanation depicts, for our purposes, either a qualitative or quantitative equation that essentially express specific relationships between properties of the world. We go into detail about the syntax of each equation type below.

We focus on equation explanations because they enable us to predict the values of properties without needing to do things like parse natural language or use external knowledge. Instead, one can just work out the value of an expression. That said, equation explanations do not reflect the broad range of explanation types available in SBF, and in fact, we do leverage function explanations too (see below). Still, reasoning about even a subset of the explanation types nevertheless allows us to make progress towards behavioral simulation.

Below, we describe the syntax of both equation types and how SBFCalc reasons over them. We note that it is possible for equations not following this syntax to exist within an SBF model. SBFCalc will only reason over an equation that has the prefix "quant:" or "qual:" in its description, signifying that they will follow the syntaxes we are about to describe.

4.1.1.1.1 Quantitative Equation Explanations

A quantitative equation says that a property's value in the After state will be equal to a mathematical expression whose variables, which are component or substance properties, all resolve to numerical values. The syntax of a quantitative equation is as follows:

```
quant: <Property> = <Expression>
```

Table 1. Explanations for the transitions in Figure 4. These are also true for Figure 8.

Transition Identifier	Explanations on that Transition
T1	• Function: EngineCausesTrainToAccelerate • Equation E1 "qual: Pantograph.Velocity is directly proportional to the qualitative expression Train.Velocity:After - Train.Velocity:Before" • Equation E2 "qual: Air.FlowOverPantograph is directly proportional to the qualitative expression Pantograph.Velocity:After - Pantograph.Velocity:Before"
T2	• Equation: E1 "qual: Train.Velocity is directly proportional to the qualitative expression Train.Accelerating:After " • Equation E2 "qual: Pantograph.Velocity is directly proportional to the qualitative expression Train.Velocity:After - Train.Velocity:Before" Equation E3 "qual: Air.FlowOverPantograph is directly proportional to the qualitative expression Pantograph.Velocity:After - Pantograph.Velocity:Before"
T3	The same explanations as for T2
T4	• Function AirFlowAcrossPantographFormsTurbulence
T5	• Function TurbulenceCausesNoise • Equation E1 "qual: Train.NoiseCreated is directly proportional to the qualitative expression Air.NoiseCreated:After - Air.NoiseCreated:Before"

Here, "quant:" signifies that this is a quantitative equation that SBFCalc should reason over. <Property> is some property of a component or substance for which we are assigning a value (e.g., Box.Weight, where Box is a component and Weight is one of its properties). <Expression> refers to a mathematical expression, which may contain properties as variables. Each property in <Expression> has an additional :Before or :After tag, signifying if the value should be taken from the property's value in the Before state or the After state, respectively. For example, an <Expression> could be `Box.NumberOfOranges:After * Orange.Weight:Before`.

Actual solving of an <Expression> is handled in essentially two steps. First, SBFCalc attempts to replace with its value any property for which SBFCalc knows the value. Second, SBFCalc uses the built-in JavaScript engine in Java to automatically process the expression String. Finally, if an exception was not thrown (signifying a successful solving of the expression), SBFCalc will store the resulting value as the After value for <Property>.

To solve an <Expression>, all properties within the expression must be resolvable to numerical values. SBFCalc checks to see if it has a value for all the properties within an <Expression> (note: at this time, SBFCalc does not ensure that these values are numerical or, in the case of qualitative expressions that we will describe later, values in known quantity spaces). If there are any After properties within the expression for which SBFCalc does not have an associated value, SBFCalc may need to solve another equation explanation in the transition before it can resolve the After variable in that expression. For example, consider the following hypothetical situation: there are two equation explanations on the same transition:

```
(1) quant: Box.Weight = Box.NumberOfOranges:After *
                        Orange.Weight:Before
(2) quant: Box.NumberOfOranges = Box.NumberOfOranges:Before + 1
```

To solve the <Expression> in equation (1), SBFCalc must know the value for Box.NumberOfOranges:After, which requires solving equation (2). SBFCalc tackles situations like these in two ways. First, it will reason about function explanations and conduct implicit value forwarding (both described below) before reasoning about equations so that it can know as many After values as possible before reasoning about equations. Second, it takes an iterative approac h to solving equations by solving at most one equation at a time where that equation has no unresolved properties. If there are ever equations remaining to solve but none of them are solvable, SBFCalc will fail an assertion and exit. An <Expression> might also be unsolvable because it contains Before properties for which SBFCalc does not know a value. This is also covered by the aforementioned iterative approach because all properties in an <Expression> must be resolvable for it to be solvable, including Before properties.

In Figure 5, we depict a hypothetical example of reasoning with quantitative equation explanations. We note that this example also uses implicit value forwarding, which we describe later in this paper. In this example, the Before and After state are describing the change in weight of a box due to an increasing number of bricks in that box. To do this, we annotate the transition between these two states with two quantitative equation explanations. The first equation, E1, describes how to calculate the weight of the box. We note here that choosing whether to use the brick's weight in the After state (which we chose to use) versus the brick's weight in the Before state is arbitrary because it does not change. The second equation, E2, describes how an additional brick is being added from the Before state to the After state.

4.1.1.1.2 Qualitative Equation Explanations

In our sense of the term, a qualitative equation says that a property's value, which is defined as a quantity in a predefined quantity space, in the After state is either directly or inversely proportional to a qualitative or quantitative expression. Currently, there are two predefined quantity spaces implemented in SBFCalc, one with the quantities Zero, Low, Medium, High, and Maximum and the other with the quantities Off and On. We note that it should be straightforward within SBFCalc to change these quantity spaces and to add additional quantity spaces. A qualitative expression is one in which all properties are expected to resolve to values in the aforementioned two quantity spaces, and a quantitative expression is exactly the same as that

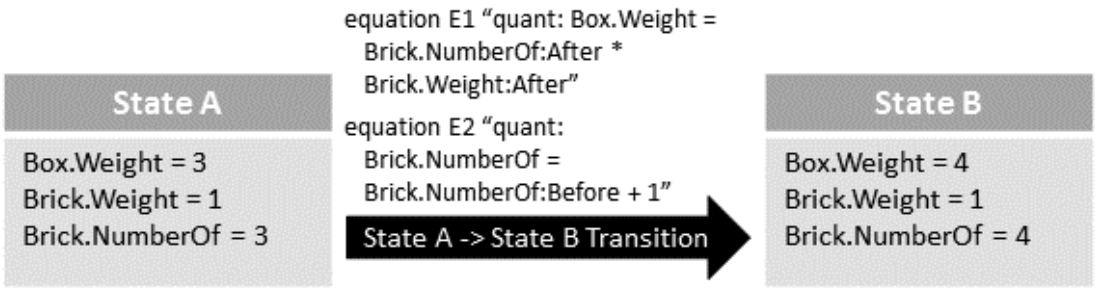

Figure 5. Hypothetical example of reasoning using quantitative equations.

described in the prior section. The syntax of a qualitative equation is as follows:

```
qual: <Property> is (directly | inversely) proportional to the
      (quantitative | qualitative) expression <Expression>
```

Here, "qual:" signifies that what follows is a qualitative equation that SBFCalc can process. <Property> means the same as it did when discussing quantitative equations except that its value will be a quantity in a quantity space rather than a numerical value. The modeler must decide whether to use the keyword "directly" or "inversely", which we will describe momentarily. The modeler must also decide whether to use the keyword "qualitative" or "quantitative", which defines whether SBFCalc expects the following <Expression> to be a qualitative expression or a quantitative expression, respectively.

To solve a quantitative expression, SBFCalc performs the same procedure as described in the prior section. To solve a qualitative expression, SBFCalc first replaces all the properties with their respective values. These values should be quantities in the predefined quantity spaces, but currently, SBFCalc does not ensure this. The same procedure for handling whether the <Expression> is solvable described in the context of quantitative expressions is also used here. Next, SBFCalc replaces the qualitative values with their numerical equivalents, which in the implementation relate to their array indices, but conceptually, could be any numerical value. Finally, the system solves the <Expression> as if it were a quantitative expression.

After solving the <Expression>, SBFCalc inspect its result to see if it is either less than zero, equal to zero, or greater than zero. If the modeler chose the keyword "directly", meaning directly proportional, the value of <Property> will increase if the result of <Expression> was greater than zero, stay the same if the result was equal to zero, or decrease if the result was less than zero. If the modeler chose the keyword "inversely", meaning inversely proportional, the increase and decrease conditions are reversed. The change in <Property> is however limited in that a property's value can never increase beyond the maximum quantity in the quantity space, nor can it ever decrease below the minimum quantity in the quantity space. Currently, minimum and maximum values correspond to array indices, but this is just an implementation decision.

In Figure 6, we depict a hypothetical example of reasoning using both a qualitative equation explanation and a function explanation. The Before and After state pair of this example describes how the room remaining in a cup decreased because the amount of soda poured into that cup increased. To do this, we annotated the transition between the two states with both a qualitative equation explanation and a function explanation. We will focus here on the qualitative equation explanation and describe function explanation reasoning in the following section, but suffice it to say that reasoning about the function explanation enables us to determine that the After value of Soda's AmountPoured property is Medium. The qualitative equation E1 describes how the RoomRemaining property of Cup is inversely proportional to the change in value of Soda's AmountPoured property. Thus, if AmountPoured increases between the Before and After states, then RoomRemaining will decrease, and vice versa (with the limitation that they cannot increase or decrease beyond the bounds of their quantity space). Note here how we get at the change in AmountPoured by calculating the difference between AmountPoured in the After state and AmountPoured in the Before state.

4.1.1.2 Function Explanations

A function explanation is another type of explanation that might exist on a transition. The existence of a function explanation indicates that some other function, which we call a sub-

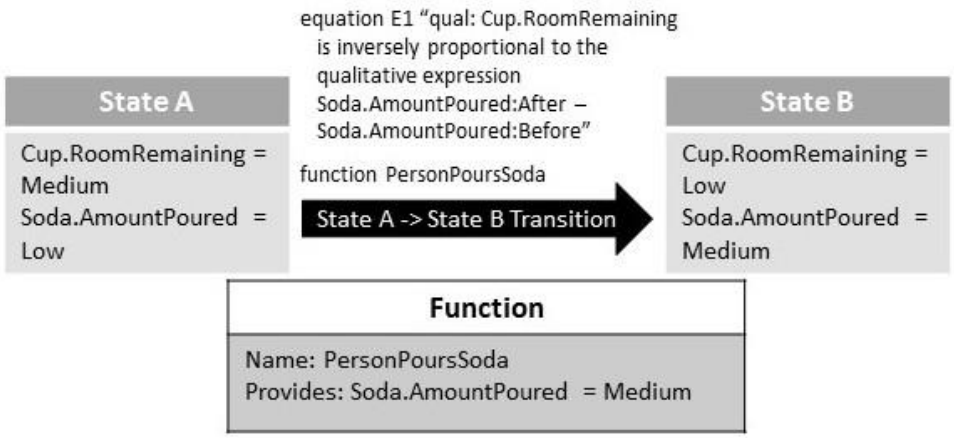

Figure 6. Hypothetical example for qualitative equation reasoning and function explanation reasoning. The behavior pointed to by the function is omitted in this figure.

function, is responsible for or explains why some or all of the properties' values change from the Before state to the After state. Functions and sub-functions relate to an essential concept in SBF modeling: the functional decomposition. That is, a given function may have sub-functions that enable its behavior; those sub-functions might themselves have sub-functions; and so on. The relationships between the functions in Figure 3, from our Shinkansen Train model, is an example of a functional decomposition. The TrainGeneratesAerodynamicNoise function depends on the EngineCausesTrainToAccelerate and other functions to enable its behavior. Functional decompositions allow us to compartmentalize and abstract parts of a system, enabling individual functions (and their behaviors) to be relatively small and easy to reason about while still allowing for complex systems to be described. Sub-functions can also enable convenient re-use, for a single sub-function (via function explanations) could be re-used across the behavior without needing to duplicate the sub-function's implementing behavior in multiple places across the behavior.

Because functional decomposition is an important and valuable aspect of SBF modeling, SBFCalc utilizes function explanations as part of its behavior simulation. Conceptually, a function explanation says that a function happens in the transition between two states and that the results of that function should exist in the After state. The "provides" condition of a function, which describes the values of relevant properties at the completion of the function, is one place to look for the results of the function, but the "provides" condition is inputted by the modeler and may not reflect the actual output of the behavior that implements that function.

Instead of using the "provides" condition, SBFCalc will do the following when it reasons about a function explanation. It will run a simulation (separate from the current inferred model so as not to interfere with anything) of the behavior pointed to by the function in the function explanation. Then, it will set the property and value pairs from that behavior's output, which we define as the union of all the stop state conditions, as property and value pairs for the After state in the original behavior that we were simulating. Any behavior could have function explanations as part of it, so simulating a behavior pointed to by a function explanation (let us call this a sub-behavior) could lead to simulating yet another behavior, and so on.

Figure 6 depicts a hypothetical example of reasoning by a function explanation. When SBFCalc confronts this example, it would simulate the behavior pointed to by the function PersonPoursSoda and use its output to infer After state values of any properties in that output. Assume in this case that the behavior pointed to by the function PersonPoursSoda has the same output as its provides condition. Thus, SBFCalc infers that the AmountPoured property of Soda in After state is equal to Medium.

4.1.1.3 Implicit Value Forwarding

In addition to reasoning about equation and function explanations, SBFCalc implements what we call implicit value forwarding. This technique is implicit because the model creator need not do anything to enable it and because it's not explicitly annotated in the model. In a given Before and After state pair, SBFCalc may not be able to infer the After state values for all the properties in the Before state. For each property for which this is the case, SBFCalc assumes that the value of that property remains the same. Thus, it will set the After state's value for that property to be the same as the Before state's value.

Figure 7 depicts a hypothetical example without (the top half of the figure) and with (the bottom half of the figure) implicit model forwarding. Without implicit value forwarding, the value for the WaterFlowing property of Hose is missing in State B—the After state—because it could not be inferred from any explanation on the transition. With implicit value forwarding, SBFCalc still could not infer the value from any explanation, but this time it set Hose.WaterFlow = On for State B, forwarding it from State A—the Before state.

4.1.1.4 Inferring the Next State's Contents

In the previous sections, we have looked at how SBFCalc reasons about quantitative and qualitative equations, how it reasons about function explanations, and how it uses implicit value forwarding to infer values in the After state. Here, we will describe at a high level how, given a Before state, an After state, and a set of explanations, SBFCalc combines all of the aforementioned techniques to infer the values of the After state. We will ignore low-level details such as when our system gathers the list of equation explanations from the transition.

When conducting its reasoning, SBFCalc builds up a map that connects Before and After state properties to their values in those states. This map is only applied to the After state at the end of reasoning about this Before and After state pair. SBFCalc will first store in the map as Before property and values pairs all the property and value pairs of the Before state. Next, it reasons over any function explanations (if any exist) on the transition, storing in the map as After state values of properties the property and value pairs that it infers. Next, our system will store in the map as After state values of properties any property and value pairs from the Before state that (a) have not already been set by the function explanation reasoning and (b) will not be set by the equation explanation reasoning (determined by inspecting equations that SBFCalc can reason over). This is implicit value forwarding. After that, SBFCalc will process the equation explanations (if any exist), reasoning over those equations with the "qual:" and "quant:" prefixes. When reasoning over the equation explanations, our system will use the iterative method we described earlier to check properties in <Expression>'s. After an equation is successfully reasoned over, its result is stored in the map as the After state value for the appropriate property. Finally, SBFCalc will then use the map to set the After state to have those property and value pairs stored within the map as After state property and value pairs.

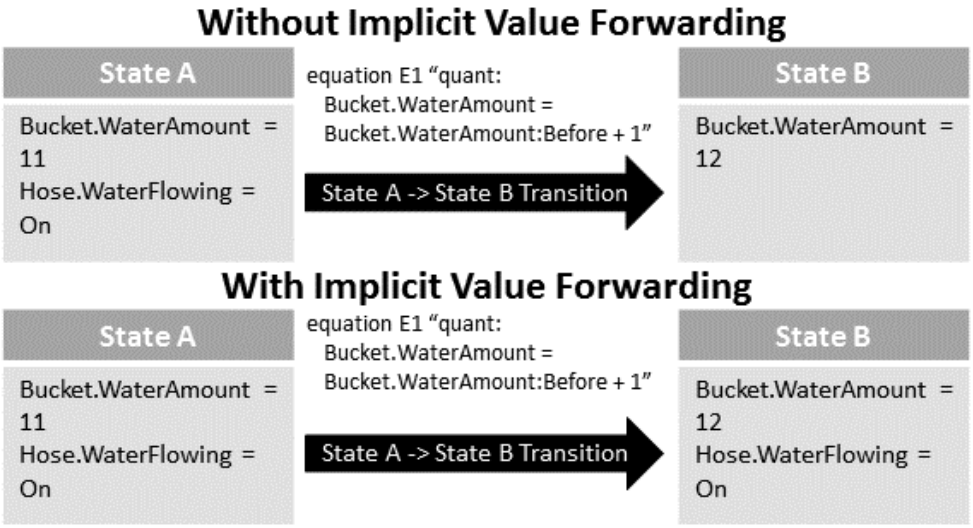

Figure 7. Hypothetical example without (top half) and with (bottom half) implicit value forwarding.

4.2 Evaluating the Function Model

In SBF, the function model is comprised of one or more functions. Function provides a powerful organization and compartmentalization tool in SBF modeling, and it is critical to verify the function model so as to determine if the design concept being modeled actually achieves the functions that the modeler intends for it to achieve. Our approach for function model verification is to determine the extent to which the behaviors that are associated with the functions (and thus are intended to implement or achieve the functions) are compatible with those functions. If so, we infer that the functions are achieved as planned by the modeler.

We will now describe this verification process in more detail. After it has completed its behavior simulations and evaluations, SBFCalc evaluates all the functions in the inputted model. Conceptually, the idea is as follows: the "provides" condition of a function states what about the world must be true at the resolution of the function. The behavior pointed to by the function should implement the function. Therefore, the stop state(s) of a behavior should reflect a world state that is compatible with the "provides" condition of its function for the behavior and function to be in agreement.

More specifically, SBFCalc will, for each inputted function, compare the property and value pairs from the "provides" condition of that inputted function with the property and value pairs comprising the output of the inferred behavior that matches the inputted behavior it points to. SBFCalc's primary output from this reasoning is two lists: a list of property and value pairs that are compatible between the function and behavior and a list of pairs that are incompatible.

We define the output of a behavior as the union of all the final state conditions for that behavior, resulting in a map of property and value pairs. SBFCalc assumes that there is only one value per property in this map and will fail an assertion if multiple values for a single property are detected.

To determine compatibility, SBFCalc does the following. For each property and value pair in the output of the inferred behavior, SBFCalc will check if there is a matching property in the set of property and value pairs from the "provides" condition of the inputted function. If there is and if both are assigned the same value, SBFCalc declares these to be compatible. If there is and if they are assigned different values, SBFCalc declares these to be incompatible. If there is no matching property, SBFCalc declares this property and value pair to be compatible because it doesn't conflict with anything in the function's "provides" condition. After doing this, SBFCalc will loop through all the property and value pairs from the inputted function's "provides" condition. If it finds any property that is not in the map of property and value pairs in the inferred behavior's output, it will flag this property and value pair as incompatible because the inferred behavior's output should have it (since the function "provides" it) and does not.

In the results below, we show an example of function model evaluation through analysis of our Shinkansen Train model.

4.3 Evaluating the Structure Model

The third and final aspect of an SBF model is its structure model, which describes the physical components and substances of the model and the connections between the components. SBFCalc does not itself address structure model evaluation. However, we are currently developing a complementary computational technique involving model comparison that should, as part of it, address this topic.

5. Results

In this section, we present a preliminary evaluation of SBFCalc. To do so, we investigated SBFCalc's performance on the Shinkansen Train model that we described above. We inputted the Shinkansen Train model into SBFCalc, which produced multiple results. Here, we will report on the evaluation of the TrainGeneratesAerodynamicNoise function and its associated behavior.

First, let us discuss the behavior. Figure 8 visualizes the behavior results. Note that, since SBFCalc only changes state conditions, the explanations on transitions depicted in Table 1 have not changed. In this Figure, only the differences are shown. The set of similar property and value pairs is identical to the state conditions show in Figure 4. We have prefixed property and value pairs in states with a + if they are new in the inferred behavior compared to the inputted behavior, and all property and value pairs are prefixed with this. There were no properties in a state that existed in both the inferred and inputted behavior and had different values.

We interpret our results as meaning two things. First, SBFCalc agreed with all the state conditions that we proposed in our inputted behavior (since there were no properties with different values upon comparison), which is a positive result because we tried to build the behavior correctly. Second, SBFCalc identified a large number of new property and value pairs as differences, which likely came from a combination of implicit value forwarding and function explanation reasoning. Although these differences are legitimate differences, we are currently considering whether to position new property and value pairs as something conceptually different from differences because they don't represent contradictions in the model and may in fact have been implied by the modeler because of a modeling assumption.

Figure 9 visualizes the function results. Property and value pairs with a + prefix were in the behavior output but not in the function's "provides" condition. Only one property and value pair, Air.NoiseCreated = Low, is missing this prefix; it was in both the behavior's output and the

function's "provides" condition. As can be seen, SBFCalc deems the output of the inferred behavior to be completely compatible with the inputted function. It is interesting to note the vast number of extra property and value pairs outputted by the behavior, which suggests a more complex world state than just looking at the function's "provides" condition alone would suggest.

6. Conclusion

Computational design seeks to evaluate designs as early in the design process as possible. Given that the task of conceptual design is a function to structure mapping, the task typically produces not only a design concept but also a functional model of the design. The functional model provides both a functional decomposition of the design and a functional explanation of how the structure of the design delivers its functions. In this paper, we described an automated computational technique called SBFCalc for evaluating design concepts. SBFCalc evaluates a design concept through a simulation of the functional model of the design concept. We showed how SBFCalc can validate a proposed design concept in biologically inspired design. In addition, although we do not explore it here, we also propose that a designer could also use this same technique to validate the source analog, which corresponds to a biological system in our context, and perhaps also to validate the target, which corresponds to a deficient design in our context.

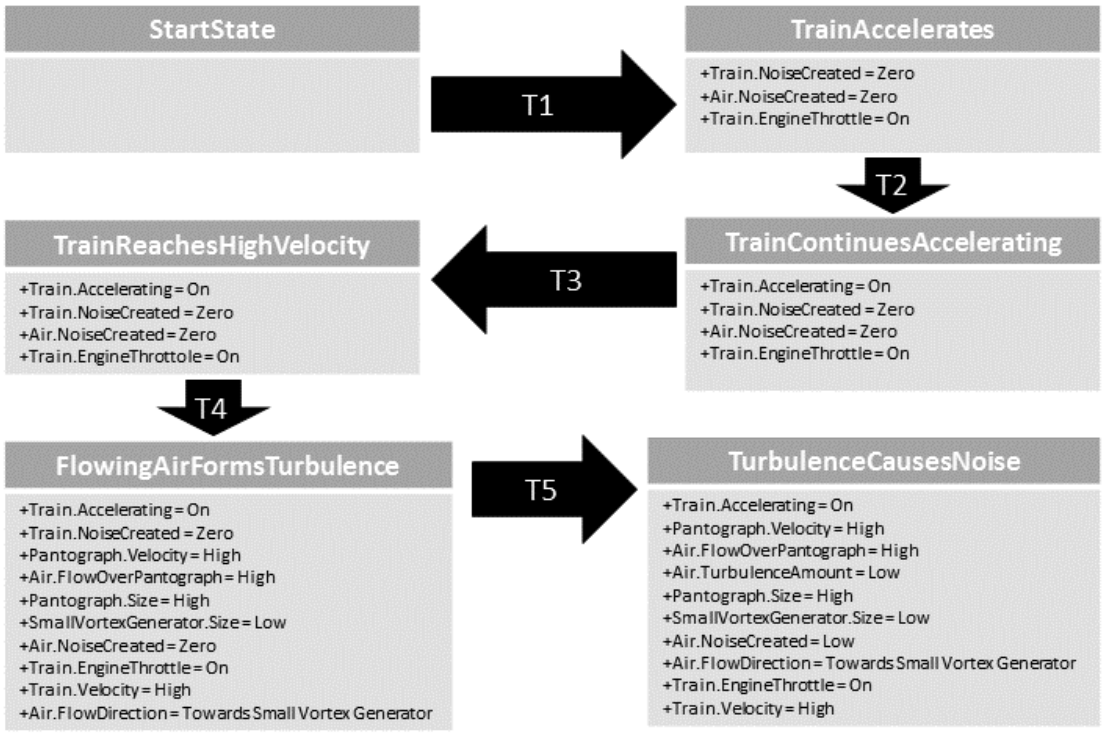

Figure 8. Visualization of the results of evaluating the behavior that implements the TrainGeneratesAerodynamicNoise function of the Shinkansen Train model. This Figure only depicts the differences identified in the results.

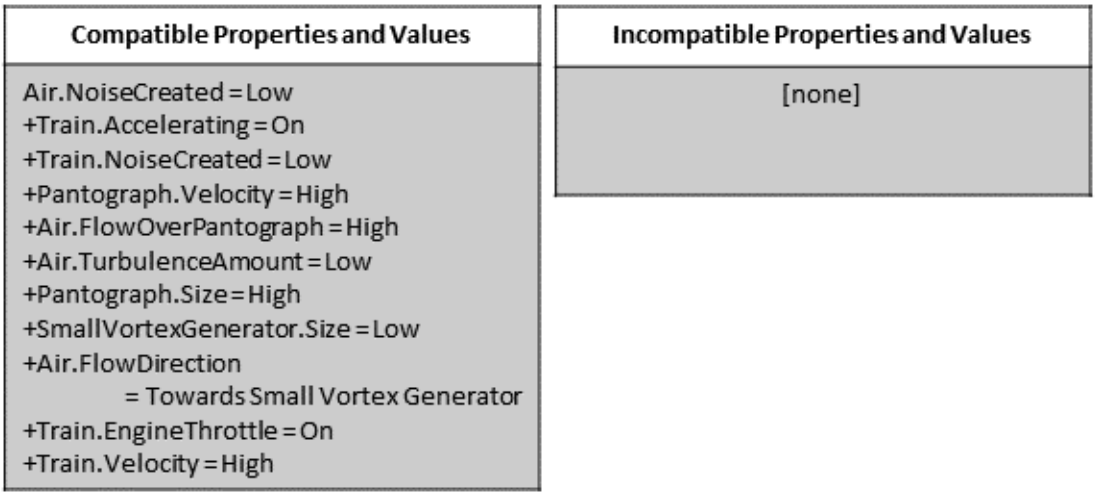

Figure 9. Visualization of the results of evaluating the TrainGeneratesAerodynamicNoise function of the Shinkansen Train model.

SBFCalc builds on two lines of research: functional modeling and qualitative simulation. While qualitative simulation provides techniques for deriving behavior from structure of systems, functional modeling provides schemas for explaining how the structure of the design delivers its functions, using behavior as an intermediate level of abstraction between structure and function. When SBFCalc combines functional modeling and qualitative simulation, functions (1) help decompose the simulation of the system as a whole into simulations of its subsystems, and (2) help organize and abstract the simulations of the subsystems into simulation of the system as a whole.

Acknowledgements

We thank Arvind Jagannathan and Spencer Rugaber for their contributions to SBF modeling and the SBF editor.

References

Baumeister, D., Tocke, R., Dwyer, J., Ritter, S., & Benyus, J. (2012) *Biomimicry Resource Handbook*. Biomimicry 3.8, Missoula, MT, USA.

Benyus, J. (1997) *Biomimicry: Innovation In- spired by Nature*, William Morrow, 1997.

Bredeweg, B., Linnebank, F., Bouwer, A., Liem, Jochem. (2009) Garp3 – Workbench for Qualitative Modeling and Simulation. *Ecological Informatics* 4: 263-281.

Chandrasekaran, B., Goel, A., & Iwasaki, Y. (1993) Functional Representation as a Basis for Design Rationale. *IEEE Computer* 26(1):48-56, 1993.

de Kleer, J., & Brown, J.S. (1984) A Qualitative Physics Based on Confluences. *Artificial Intelligence* 24 (1-3): 7-83.

Dym, C., & Brown, D. (2012) *Engineering Design: Representation and Reasoning.* Cambridge University Press.

Falkenhainer, B. (1987). An Examination of the Third Stage in the Analogy Process: Verification-Based Analogical Learning. In *Procs. Tenth International Joint Conference on Artificial Intelligence* (IJCAI-87), volume 1, pp. 260-263.

Falkenhainer, B., Forbus, K., Gentner, D. (1989) Structure-Mapping Engine: Algorithm and Examples. *Artificial Intelligence,* 41(1): 1-63.

Forbus, K. (1984) Qualitative Process Theory. *Artificial Intelligence,* 24(1-3): 85-168.

Gero, J. (1990). Design prototypes: a knowledge representation schema for design. *AI Magazine*, 11(4), 26.

Gero, J., & Kannengiesser, U. (2004). The situated function–behavior–struc- ture framework. *Design Studies,* 25, 373–391.

Goel, A. (1997) Design, Analogy and Creativity. *IEEE Intelligent Systems*, 12(3):62-70, May/June 1997.

Goel, A., & Chandrasekaran, B. (1989) Functional Representation of Designs and Redesign Problem Solving. In *Proc. Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89),* Detroit, August, 1989, pp. 1388-1394.

Goel, A., McAdams, D., & Stone, R. (editors, 2014) *Biologically Inspired Design: Computational Methods and Tools.* London, UK: Springer-Verlag.

Goel, A., Rugaber, S., & Vattam, S. (2009). Structure, behavior & function of complex systems: The structure–behavior–function modeling language. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* 23, 23–35.

Goel, A., & Stroulia, E. (1996). Functional device models and model-based diagnosis in adaptive design. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* 10, 355–370.

Goel, A., Vattam, S., Wiltgen, B., & Helms, M. (2012) Cognitive, Collaborative, Conceptual and Creative - Four Characteristics of the Next Generation of Knowledge-Based CAD Systems: A Study in Biologically Inspired Design. *Computer-Aided Design,* 44(10): 879-900.

Helms, M., Goel, A. (2014) The Four-Box Method: Relating Problem Formulation and Analogy Evaluation in Biologically Inspired Design. *ASME Journal of Mechanical Design, 136(11).*

Hoeller, N. (2013) Structure-Behavior-Function and Functional Modeling. *Zygote Quarterly* 5:152-170.

Hofstadter, D., (editor, 1996) *Fluid Concepts & Creative Analogies: Computer Models of the Fundamental Mechanisms of Thought.* Harvester Wheatsheaf.

Holyoak, K., & Thagard, P. (1996) *Mental Leaps: Analogy in Creative Thought.* MIT Press.

Joyner, D., Goel, A., & Papin, N. (2014). Intelligent Generation of Agent-Based Simulations from Conceptual Models. In *Procs. Eighteenth International Conference on Intelligent User Interfaces (IUI2014)*, pp. 289-298.

Kitamura, Y., Kashiwase, M., Fuse, M., & Mizoguchi, R. (2004). Deploy- ment of an ontological framework for functional design knowledge. Ad- vanced Engineering Informatics 18(2), 115–127.

Klenk, M., de Kleer, J., Bobrow, D., Yoon, S., Hanley, J., & Janssen, B. (2012) Guiding and Verifying Early Design Using Qualitative Simulation. In *Proceedings of the ASME 2012 IDETC*, Chicago, IL, 2012.

Kuipers, B. (1986) Qualitative Simulation. *Artificial Intelligence*, 29(3): 289-338.

MacLellan, C., Langley, P., Shah, J., & Dinar, M. (2013) A Conceptual Aid for Problem Formulation in Early Conceptual Design. *ASME Journal of Computing and Information Science in Engineering* 13(3).

McKeag, T. (2012). Special Feature: Auspicious Forms. *Zygote Quarterly*, Summer 2012, pp. 14-36.

Pahl, G., Beitz, W., Feldhusen, J., & Grote, K. (2007). *Engineering Design: A Systematic Approach*, 3rd Ed. (Wallace, K., & Blessing, L., Translators and Editors), New York: Springer–Verlag.

Rasmussen, J. (1985). The role of hierarchical knowledge representation in decision making and system management. *IEEE Transactions on Systems, Man, and Cybernetics* 15(2), 234–243.

Qualitative Reasoning & Modeling (2015). Garp3 Software. Last retrieved on March 1, 2015 from https://ivi.fnwi.uva.nl/tcs/QRgroup/QRM/software/.

Rieger, C., & Grinberg, M. (1978). A system of cause–effect representation and simulation for computer-aided design. In *Artificial Intelligence and Pattern Recognition in Computer-Aided Design*, J. Latombe (editor), pp. 299–333. Amsterdam: North-Holland.

Sembugamoorthy, V., & Chandrasekaran, B. (1986). Functional representa- tion of devices and compilation of diagnostic problem-solving systems. In *Experience, Memory, and Learning* (J. Kolodner & C. Riesbeck, Eds.), pp. 47–73. Mahwah, NJ: Erlbaum.

Simon, H. (1996). *Sciences of the Artificial*, 3rd ed., Cambridge, MA: MIT Press.

Umeda, Y., Ishii, M., Yoshioka, M., Shimomura, Y., & Tomiyama, T. (1996) Supporting conceptual design based on the function–behavior–state modeler. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* 10, 275–288.

Umeda, Y., & Tomiyama, T. (1997). Functional reasoning in design. *IEEE Expert* 12(2), 42–48.