May 12, 2026

Introduction to Constraint Hypergraphs

John Morris, PhD
Assistant Professor | Industrial & Systems Engineering
The University of Alabama in Huntsville
john.h.morris@uah.edu

This text was originally prepared in 2025 as part of the author's PhD dissertation. It can be accessed in its original form as part of the full dissertation published by Clemson University, available at https://open.clemson.edu/all_dissertations/4127 Citations of this work should be made to the original dissertation [1].

1. Understanding Information

Reality, as it exists, is unfathomable. Fortunately, as sentient beings we can construct worldviews that allow us to make sense of our surroundings. We do so by identifying patterns in the physical phenomena. Shared patterns of behavior become things: nouns such as bears, airplanes, and rain. Each thing has a unique set of behaviors, such that all bears have some similar behavior that is shared with neither airplanes nor snowflakes. A synonym for a thing is a *system*[1] [2].

The distinctions we draw between things is *information*. This is such a foundational notion that it bears a bit of additional explanation. If we have a set of things, and we can tell the difference between them, then it follows that we are able to treat them all individually. Every thing that is unique becomes a datum–the singular of data. Synonyms for data include labels and values. The last is most appropriate when we want to consider groups of data. All the data that we can distinguish uniquely can be collected into a set. For instance, the things below could be categorized by the set `species`, with values "cow", "tiger", and "pig"; or by `position`: "left", "middle", and "right". We can tell the difference between "left" and "right", but not between "cow" and "middle".

A system that is temporally consistent imparts one more condition: that no two data in a group can be manifest in a single frame of reference. That is, to be temporally consistent, a thing's `species` cannot be both a "cow" and a "tiger", or its `position` both "left" and "right". What a temporal system might, however, change is which value is manifest, the fundamental notion of a dynamic system. Consequently, for dynamic systems we refer to these groups of unique, non-overlapping data as *variables*.

The set of variables that we assign to a system represents all the information that can be gleaned about it. We can associate some loose terms to these concepts: *characterizing* a system is establishing which variables are exhibited by a given system (our symbols above are characterized as having variables of `position` and `species`). *Describing* a system, in contrast, is when we assign values to these variables: to `species`, and "left" to `position`. Most of the goals of science and engineering are to describe

[1] Throughout this paper, text set apart in *italics* generally indicates the term is defined nearby in the text.

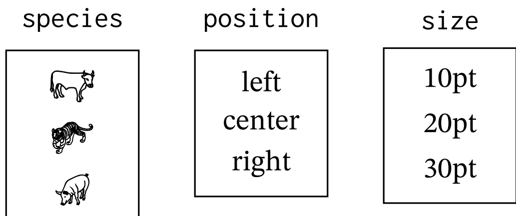
some entity: we want to know when a column will buckle, what virus is infecting a person, or which quantum state will be manifest after a calculation.

There are two mechanisms for describing a system. The first one is *observation*, where a rational agent assigns a datum to some perceived phenomenon. If every variable assigned to a system can be fully observed, then the work of a scientist describing the system is complete. However, it is more often the case that system variables are difficult to observe. For instance, the size of the symbols above is not immediately obvious–at least, not if you don't have a ruler handy. Other examples of unobservable information include future states (such as who will win the next election), inaccessible information (what is the temperature of the earth's core), and lost data (how many species have gone extinct since the earth formed).

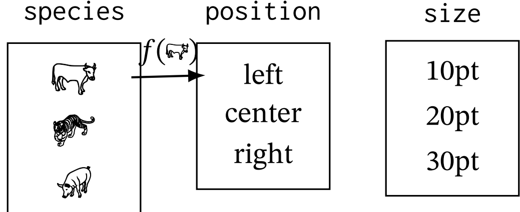
It's clear in these cases that we won't be able to observe everything about a system. The second and final mechanism for describing some entity is *simulation*. A simulation is when we use relationships between variables that we know to inform the variables that we don't. Synonymous terms for this include prediction, inference, and to a lesser extent, estimation and decision-making. The result of a simulation is the provision of a value for some unobserved variable. We refer to the known variables as *inputs* and the artificially assigned variables as *outputs*. Most of this introduction is focused on the mechanisms we use to perform simulation.

1.1. Systems Modeling

The primary component of a simulation is the relationships used to transform inputs to outputs. These relationships are functions, whose domains and codomains lie in the space formed by the input and output variables respectively. The aggregation of functions and variables together forms a model, which describes the *behavior* of the system. Willems, in an influential article [3], gave a mathematical definition for what we mean by behavior. He starts by considering the state space of a system–that is, the vector space formed by the combination of all states that can exist. For our animals, that's every combination of `species`, `position`, and `size` that could be prescribed for the three symbols. Behavior is given by reducing the state space, such that certain state combinations are no longer reachable. This corresponds with the idea of information as distinction–to have unique behavior, a system must not be able to exhibit the same states as another system.

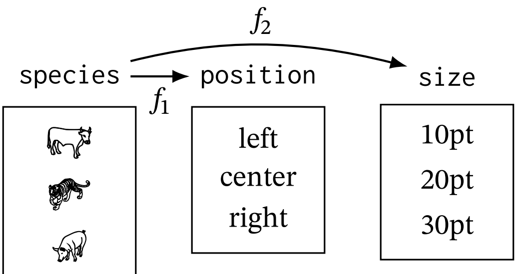


There are two ways to restrict the state space. The first is to eliminate certain states independently. For instance, the position of a symbol will never be "above" or "in front of". These actions are performed when characterizing the system, when the modeler defines what it implied by `position`. As such, they are rarely represented in the model of the system–instead the model of the system is simply redefined to only include valid states. The second method is a dependent restriction, when a state's reachability depends on other values in the system. An example might be stating that must always be in the "left" position, consequently making the state combinations (, center, x_{size}) and (, right, x_{size}). This constraint can be represented as a function, mapping from the domain `cow` to `left`.



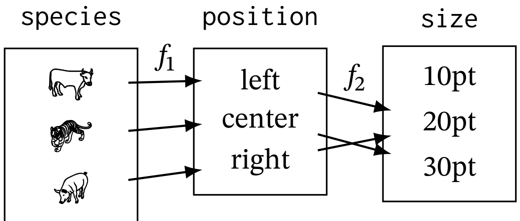
Unconstrained problems can have inequalities as constraints (such as mapping to either "left" or "right"). However, these inequalities cannot be used to describe a real system for two reasons. The first is that they violate temporal consistency. As described above, the animal cannot be both "left" and "right". We need additional information if we want to describe our system using this unconstrained relationship. The second is that these relations are nondeterministic. It's unnecessary in this introduction to explore the philosophy of determinism. Instead, we can merely note that if a behavior cannot be construed to be deterministic, then it becomes impossible to *determine* anything from it. Restricting ourselves to real, temporal, deterministic systems means that we can ignore inequality constraints. The result is that all behaviors for these systems must be represented as algebraic functions.

It bears repeating that our goal with forming models is to fully describe a system–to know everything about it that can be known. To fully constrain a system requires a modeler to form behavioral constraints for every node that is not observed. Such a fully constrained system might look like this:



As shown, if we can observe a symbol's `species`, we can also determine–or simulate–its `position` and `size`. There's another property with functions that makes them useful to use: they compose. Because of this we can form chains of reasoning. To demonstrate this, let's modify the animal set so that the center animal is a little bigger than the rest of them.

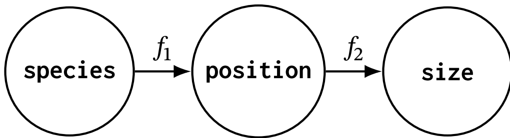
With this new system, let's identify some behaviors. We have a mapping between the `species` of animal and its `position`: on the "left", etc. And we have another one between its `position` and `size`: symbols in the middle are 30pt, otherwise they're 20pt. We can represent these behaviors with the functions $f_1 \coloneqq \texttt{species} \rightarrow \texttt{position}$, and $f_2 \coloneqq \texttt{position} \rightarrow \texttt{size}$ respectively. Our visual model becomes:



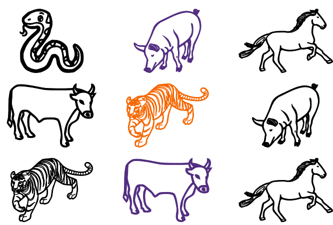
The visual shows the simulations chains in the model. For two functions to compose requires them to overlap in their domain/codomain. When this occurs, it is guaranteed that the output of one function can be used as the input for another function, allowing extended simulation chains. With the model above, we can simulate the size of the tiger symbol by taking $\xrightarrow{f_1}$ “center” $\xrightarrow{f_2}$ “30pt”. This is as if we had a relationship mapping `species` to `size`, even though such a function is not explicitly given in the model. When composed, f_1 and f_2 form a new function with the domain of f_1 and the codomain of f_2. The humble property of composition has powerful implications for building description schemes.

1.2. Graphical Representations

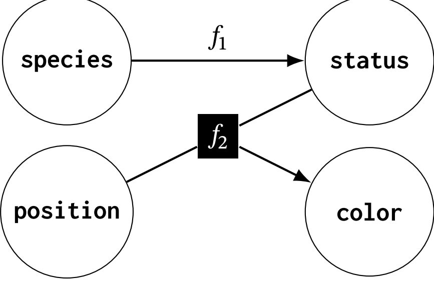
The scale of our small system is quite modest. More typical systems are too complicated for each datum to be represented. Instead, we’ll draw the model graphically: condensing the variables in the system to singular nodes, and draw the relations between them as edges. The result is the following:



So far, we have been working with unary functions–functions taking in only a single parameter. But arity is not restricted when dealing with behaviors. Consider the following expanded group of animals, each of which is described by four different variables: `species`, `position`, `status` (either predator or prey), and `color` (black, purple, or orange). The animals follow one behavior: predators in the center are colored orange, while prey in the center are colored purple.



We can model this system as the following, where f_1 is the function identifying whether a `species` is predator or prey, and f_2 determines what the symbol’s `color` should be:



Here we see the first hyperedge: a multiple-arity edge corresponding to the multiple-arity function $f_2 \coloneqq (\texttt{status}, \texttt{position}) \rightarrow \texttt{color}$. The domain becomes the Cartesian product of its arguments, so that every combination of (`status`, `position`) is mapped to `color`. The presence of a hyperedge means that this graph is now a *hyper*graph. Representing a system with a hypergraph will make things a little

more complex, but will enable some remarkable capabilities in modeling and simulation. Note that we can get information about our system using composition in the same way as the previous graph: knowing an animal's species lets us simulate its status via f_1, and we can combine the output of f_1 with knowledge of an animal's position to simulate its color via f_2.

This style of representation is called a constraint hypergraph (CHG): the hypergraph formed of all the behavioral constraints imposed upon a system. Representing a system as a CHG enables two important capabilities for modeling: *universality*, such that all behaviors can be represented in a CHG; and *declarativity*, where all simulations are encoded into the model structure. The rest of this introduction will cover what these properties are, why they are important, and how they are provided by CHGs, starting with a better understanding of a CHG.

2. Background

The task of representing behaviors is timeless, but specific notions have played more explicit roles in the evolution of CHGs. The first significant contribution was the establishment of foundation for mathematics based exclusively on functions by Church in 1933 [4] (and revised in 1941 [5]). The notation of Church's lambda calculus was borrowed by McCarthy in 1960 in developing the first functional programming language, Lisp [6]. Lisp was the first of many developments in functional programming, giving rise to languages such as Haskell and Miranda [7]. This provided useful platforms for decomposing systems, especially when Willems gave his behavioral interpretation of a system in 1989 [8].

Separately, the notions of set and information theory paved the way for the study of cybernetics, founded by Ashby in 1956 [9]. Ashby used the approach outlined above to decompose heterogeneous systems into a series of sets related by mappings, including references to arrow-based "kinematic graphs" whose traces showed the effects of system elements on each on other [10].

While Ashby was mostly concerned about analyzing biological systems, his work sparked efforts by Friedman in his doctoral dissertation in 1969. Friedman specifically showed how deconstructing a system into its functional constraints provides a basis for ascribing properties of computability to it [11]. To do so, Friedman used four different representations: set-based mappings, algebraic constraints, a constraint matrix showing connections between variables, and a bipartite graph (where one set of nodes represented variables and the other one represents the relations mapping between them, as shown in Figure 1).

Friedman's framework provided ways to analyze a complex, heterogeneous system, [12], however, one activity that he did not investigate was how these frameworks could be used to describe systems–the task that we attempted to do with the sets of animals above. We can call this task *interrogation*, and informally define it as the action of discovering a value for a state in our system. In other words, interrogation is the work of describing (or knowing) some bit of information.

This paper takes Friedman's model graphs and expands them into a tool for interrogation. It was shown in [13] that interrogation requires mechanisms for dealing with cycles, multiple edges, and pathfinding. Providing all of these is what transforms the model graph into a CHG–a framework for system modeling and simulation.

3. Constraint Hypergraph Properties

But first, we need a more rigorous definition of what a CHG is. Under the parlance of category theory [14], a CHG is a partial subcategory on **Set**, whose objects are sets and whose morphisms are partial functions. For a general reader, we are fortunately not required to understand anything additional of category theory to appreciate how CHGs work. Without considering categories, we can show CHGs as a hypergraph. Each node in the graph represents a variable, and can be assigned a value taken from

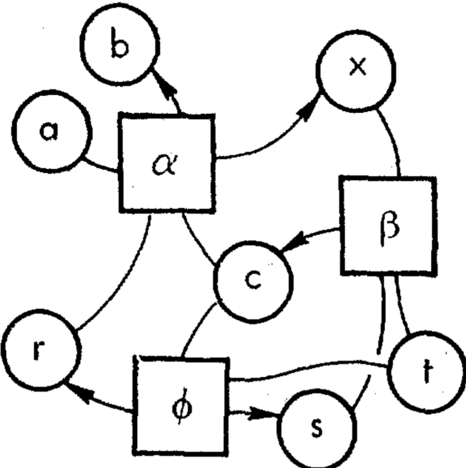


Figure 1: Bipartite "model graph" proposed by Friedman where circles represent variables and rectangles represent relations. Extracted with permission from George J. Friedman and Cornelius T. Leondes. "Constraint Theory, Part I: Fundamentals". *IEEE Transactions on Systems Science and Cybernetics* 5.1 (Jan. 1969), pp. 48–56. ISSN: 2168-2887. DOI: 10.1109/TSSC.1969.300244. *Copyright © 1969, IEEE*

the variable's range (which is allowed to be infinite). The nodes are connected by directed hyperedges. Each hyperedge leads from a set of nodes known as sources to a single node known as the target. For example, the simple CHG shown in Figure 2 has three edges. Variables A and B are the sources for edge f_1, whose target is D.

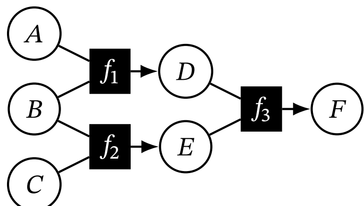


Figure 2: A basic CHG with six nodes and three hyperedges.

Each hyperedge provides a relation, mapping each value in its source set to a value in its target set. The source set for a hyperedge is formed by taking the Cartesian product of each of the edge's sources. For the CHG in Figure 2, the mappings given by the three edges are $\{A \times B \xrightarrow{f_1} D; B \times C \xrightarrow{f_2} E; D \times E \xrightarrow{f_3} F\}$.

3.1. Universality

On the surface, all we've done is provided a way to show function mappings graphically. One might reasonably wonder about the framework's significance. Though the underlying notions of variables and functions might seem unsophisticated, their simplicity belies the power of a CHG to represent complicated systems. There are only two elements in a CHG: the sets of values represented as nodes, and the constraints represented as edges. We have already discussed how every system behavior can be represented as a functional constraint. And it was established by Shannon back in the early years of the twentieth century that set theory served as a foundation for information–as Ashby summarized, "a system ...means, not a thing, but a list of variables" [9]. We can take from these two notions all sets can

be represented as by sets while functions can represent every system behavior.

We can take this declaration quite literally. By breaking down a system into its primitive elements, a CHG can be used to represent any system. This makes CHGs a sort of universal language, that is, they are able to capture the meaning given by any modeling framework. Whether the system is given as a circuit diagram, a Petri net, a Markov chain, a set of algebraic equations, a flowchart, or an entity-relationship diagram, all models can be transformed into a CHG [13].

Furthermore, CHGs can be used to capture behavior across system domains. Most frameworks, such as bond graphs or Gantt charts, express information for a specific domain (energy flows and task scheduling, respectively, in this case). Yet a stakeholder often needs information across these domains–for instance, how would increasing the power output of a hydraulic pump impact manufacturing throughput? A traditional systems engineering approach requires information to be manually passed from the bond graph to the Gantt chart. But using a CHG we can represent the entire manufacturing system in a single framework, allowing the system to be interrogated across the two domains.

3.2. Declarativity

In addition to edges, we can describe the paths through the hypergraph. A path is a series of functions whose composition connects a set of nodes to a single output. For instance, the edges f_1, f_2, and f_3 in Figure 2 compose together to map the set $A \times B \times C$ to F. In a normal graph, a path is a chain, where a series of edges connect the source node to the terminal. In a hypergraph, a path is a tree, where each edge leads to a node that is either the terminal node or a node that is in the domain of another edge–the difference being that there is no condition on the edges forming a series. Although there are only three edges in the CHG in Figure 2, there are actually six paths, as shown in Figure 3.

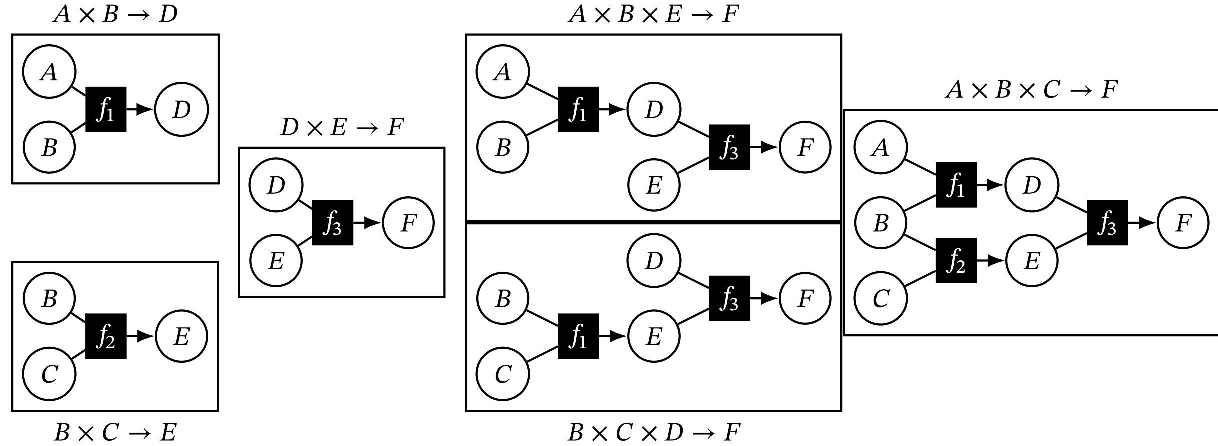


Figure 3: A breakout of all six paths of the CHG shown in Figure 2, with the composed mappings from the input (source) nodes to the output (target) node.

As shown in Figure 3, each path represents a function composed from the edges in the graph that transforms a set of input nodes to an output node. This means that each path represents a potential simulation that can be run on the system. This bears repeating: each path in the CHG represents a way that some node can be simulated based on known values of another set of nodes. The combined set of all paths represents every simulation that can be run on a system.

This is a significant result: in essence we have found a way to encode simulations into the structure of a model. Most, if not all, other model frameworks are not capable of this. Block diagrams, for instance, encode a single simulation, but cannot describe alternate ways of understanding a system. For all intents and purposes, a block diagram provides a single path from Figure 3, while the CHG

provides all of them.

Embedding these simulations into the model structure is essential to creating a *declarative* framework. A declarative framework is one where an agent can autonomously simulate the system without having to be explicitly told how to do so. Most frameworks are imperative, which means the modeler provides exact instructions on how a simulation should be executed. These instructions form a procedure, and generally start with a set of inputs at the start that get transformed to a set of outputs at the end.

An analogy of an imperative framework is giving a friend a set of driving instructions: start here, then turn North onto Main Street, turn right after the gas station, then left at the second stop sign, etc. If followed correctly, such instructions will get your friend to their destination. But your friend has no ability to redirect, because they don't have any understanding of the geography–only the ordered steps you prescribed. Contrast this with a declarative framework, which would be analogous to giving your friend a map. Then, instead of blindly following driving instructions, your friend can use the map to take whatever route they prefer. Similarly, with a declarative framework, an executing agent has the ability to simulate the system without requiring an explicitly prescribed procedure. This is because the agent has some mechanism for understanding the system's behavior.

If a modeler wants to simulate a different set of outputs using an imperative framework, then they have to create a new model. For the very simple system shown in Figure 2 that would mean six different models to give all the ways of simulating the system! As you might guess, the amount of models required to imperatively define all simulations grows considerably as the complexity increases. In fact, it grows exponentially, on the order of $\mathcal{O}(2^n)$ for n variables in a system, quickly becoming prohibitively expensive. On the other hand, using a CHG means that every simulation is discoverable from the model itself. As long as an agent can "read the map," we can provide the model and have the correct simulation autonomously discovered.

3.3. Interoperability

Declarativity is key for doing multiscale simulations. In a multiscale model, effects are prescribed at both the micro and macro levels (and often more gradients between). For example, we might want to simulate the turbulent airflow around an airplane's wing (micro), as well as all the flight paths in a nation's airspace during the day (macro). The different scales lead to competing definitions: to capture the fluid behavior of the first model requires time steps in the thousandths of seconds, while flight positions can be calculated every minute. Going back and forth between these time steps leads to long, convoluted processes in an imperative framework.

In a declarative CHG, on the other hand, any coupling between the time steps is described by paths that go between the two levels. More specifically, the relationship between the micro seconds and the macro minutes only has to be encoded in one place in the hypergraph, and then any simulation that requires that relationship will automatically make use of it. Furthermore, the mathematics of function composition enforce order, so that the effects will only be calculated at the proper moment in the simulation–ensuring that inter-system coupling will not invalidate the simulation [15].

These integrations are enabled by the graphical nature of a CHG. In a hypergraph, every node becomes a port by which information can be communicated. The corollary of this is that any signal can be exchanged between systems along the ports in the hypergraph. This is not true for typed frameworks, such as in object-oriented modeling, where ports must be selectively prescribed [16]. Integrating two systems represented by CHGs occurs by merging shared properties. This is accomplished by performing a union operation on two graphs, such that all nodes in one CHG are combined with the overlapping nodes in the other.

The new paths formed by merging the CHGs represent the emergent behavior arising from the

system integration.This emergence is shown in Figure 4, where two CHGs exist: one with four nodes $\mathcal{C}_1 := \{A, B, C, D\}$ and the other with three $\mathcal{C}_2 := \{C, D, E\}$. Both CHGs have two relations. $\mathcal{C}_1$ describes the effect of A on D, and $\mathcal{C}_2$ describes how D affects E, but only when the two are merged together (across the shared nodes C and D) can the effect of A on E be discovered. By combining the two CHGs, we have created four more relations through function composition, which have emerged out of the system aggregation.

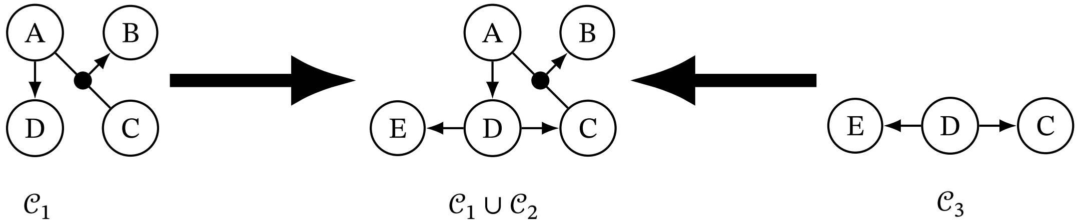


Figure 4: Example of merging two CHGs (left and right) into a single CHG (center) via a union operation.

4. Structure of a CHG

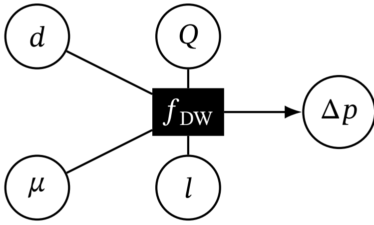
4.1. Pathfinding

Declarative simulation requires an agent who can interpret the model structure and synthesize the executable processes. To perform declarative simulation, we should be able to describe the information we know (our inputs) and have the agent automatically interrogate the system to provide the information we want to know (our outputs). We have already seen how a CHG captures an executable process as a path through the hypergraph. This redefines the work of a declarative agent as finding paths in the CHG. While pathfinding in a hypergraph is a little more difficult than a normal graph, the real wrinkles in pathfinding come from additional structures that have to be considered in a CHG. Let's build these wrinkles out one by one, starting with the easiest case.

In a simple, directed hypergraph, pathfinding is a linear extension of traditional searching on a graph. One can use methods like A* or DFS to find an optimal route from the source set S to a target node T. Ausiello, who did significant work on the theory of directed hypergraphs, showed how you could modify Djikstra's algorithm to optimally solve a hypergraph [17]. By moving from node to node, and keeping track of which branches you've explored, you can establish the shortest path between any S and T.

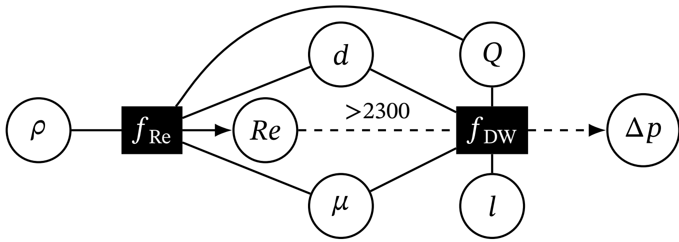
Djikstra's algorithm assumes that every edge originating from some node is always viable to traverse. The modified algorithm extends this condition to say that a hyperedge is viable to traverse if the agent has explored every source node for the hyperedge. However, this requirement is too strong for a CHG, which, if you recall from Section 3, is only required to have partial functions. A partial function is a function that does not provide a mapping for every element in its domain, meaning that an edge leading from a source node may not be viable for every value that node can take on.

Partiality is essential to representing reality, were we often do not have mappings for every variable value. Take for instance, an example from fluid dynamics. You can use the Darcy-Weisbach equation to calculate the pressure loss Δp across a pipe as $\Delta p = \frac{128}{\pi}\frac{Q\mu l}{d}$, where Q is the volumetric flow rate, μ is the dynamic viscosity, and d and l are the pipe's diameter and length, respectively. The hypergraph of this particular relation might look something like this:



However, this particular form of the relationship is only valid if the flow is laminar, namely if the Reynold's number Re is less than 2300 [18]. You can calculate Re as $4\frac{\rho Q}{\pi d \mu}$, where ρ is the fluid's density. If our CHG didn't permit partiality, then the hypergraph above would imply that f_{DW} maps every combination of Q, d, l, and μ. How do we describe the limited validity of our relation?

The trick is to map out the explicit range of values for which f_{DW} is valid as a subset of the full domain. This turns f_{DW} into a partial function. To do this, we add Re to f_{DW}'s source set. Recall that the domain of a function is the Cartesian product of its arguments. By adding Re as input to we are implying that the mapping for f_{DW} actually goes from $Q \times \mu \times d \times l \times Re$, allowing us to specify which values of Re are unknown for f_{DW}. We show this in the following CHG:



The important part of this addition is the dashed line connecting Re to f_{DW}, which we'll use to indicate that only a subset of the values of Re are mapped by the connected function. Now any pathfinding agent attempting to traverse f_{DW} will need to check whether $Re > 2300$, if not, then the edge should not be considered valid. Note that this check happens at run time, because the agent has to check the specific value given during the simulation. The agent cannot know this until Re is known–either provided as an input or calculated using f_{Re}. Although partial functions are essential for modeling reality, they greatly increase the difficulty of simulating the CHG–not in the least because we're prohibited from presolving the graph using the modified Djikstra's algorithm. With a partial hypergraph, the optimal route from A to B is not universally given because it depends on the specific value of A being simulated; not all edges are available for all values.

On the other hand, weakening the CHG definition to include partial functions greatly expands our functionality. We can now turn on and off edges based on the values the agent encounters. We can also do path switching, where the value of the source nodes influences which target a path leads to. This is helpful for marking certain states as reachable only under certain conditions. For instance, we might only calculate a diffusion coefficient if the flow is turbulent ($Re > 2300$). The concept of labeling subsets of viable values is most akin to establishing a validity frame. In practice, we can express a validity frame as a function $f_{\texttt{via}}$ that has the same domain as a hyperedge (made of the product of each source node), and maps to the set of Booleans ("true", "false") values. Passing an input value s to $f_{\texttt{via}}$ gives us the information about whether s is a viable value for the partial edge. If $f_{\texttt{via}}(s) =$ "true", then we can use the edge to map s to the target.

4.2. Cycles

In addition to path selection, partiality is essential to one more structural component of a CHG: cycles. A cycle is a path that ends on one of the nodes in its source set. There's a case to be made that constraint networks should not permit cycles, as a cycle indicates that a variable's value is dependent upon itself. This is an illogical proposition for a causal system–how could a function calculating *Re* logically use *Re* as an argument?

However, there are many cases where we use variables to represent a series of data, such as a state that varies in time. Often, in time-varying models, the behavior of a system does not change between time steps. The representative model establishes a pattern that is repeatedly solved for each time step. In this case each calculated output is dependent upon its own values from a previous time step.

However, we don't need a cycle to model this kind of relationship, we can just make a different node for every instance of a variable. Let's say, for example, that we wanted to solve for the position x of a car moving at constant velocity v. The first step would be to set the starting position x_0 and add it to a hypergraph. We can then add another node for the position after Δt seconds had gone by and call it x_1. The relationship between x_1 and x_0 is $x_1 = x_0 + v\Delta t$, which we can easily make into a hyperedge. We could repeat this process for the position after $2\Delta t$ seconds had gone by, noting that $x_2 = x_1 + v\Delta t$. This results in the drawn out hypergraph shown in Figure 5, where the dots indicate that the pattern repeats for every node x_i up to x_n.

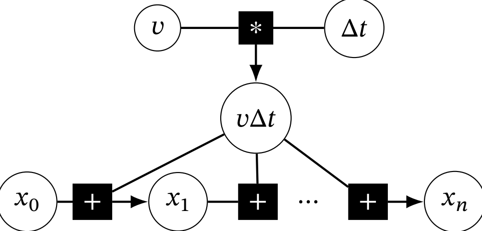


Figure 5: A simple hypergraph explicitly mapping out the relationships between variables across time steps.

Such a modeling process is not only tedious, it lacks expressability. What we really want to model is the relationship $x_{i+1} = x_i + v\Delta t$, rather than every incremental relation. The trick for adding the variables x_i and x_{i+1} to the hypergraph is to use cycles. Cycles enable arbitrary indexing of a variable, allowing us to express these recursive type expressions without have to explicitly map out every single instance of a variable, as shown in Figure 6.

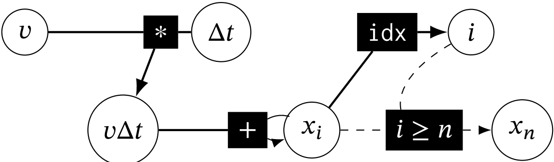


Figure 6: A non-simple hypergraph with a cycle.

By creating a cycle, we've indicated that x_i can take on multiple values, as many as the solver can find. This gives us flexibility in expressing our system; rather than make a new hypergraph for every max value n (with a longer and longer chain), we can now just trace a path around our cycle as many times as we need to calculate x_n.

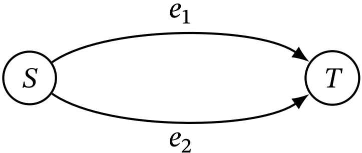
Note though that the path from x_i to x_n is not a full path–it has partiality. This is critical for a cycle. Cycles, by themselves, do not tell the solver when they should be solved. Instead, the solver needs some

kind of exit path to take–an edge that only becomes viable after some condition has been met. In this case, the condition is that the index i of x is greater than or equal to n. Until that condition is reached, a solver is forced to continually trace around the cycle, increasing the index count each time. Only after cycling n times will the exit edge become viable, allowing the solver to calculate x_n. Without this exit condition, our pathfinding agent is liable to become stuck traversing the cycle infinitely.

4.3. Model Selection

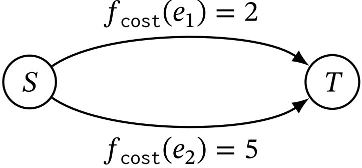
The final wrinkle for a pathfinder to address is dealing with competing models, a process known as model selection. In model selection, a modeler considers two or more models that can be used to simulate the same variable. While both are considered valid, the modeler must have some criteria for determining which is preferred for interrogation. This criteria might be how quickly the models can be executed, the accuracy of the models, the level of trust the modeler has, or even tool availability. We can look at how this takes place in a CHG by considering the most elemental case: picking between two edges.

Our motivation for doing so comes from understanding the word *model* as a set of relations describing a system's information–a hypergraph, in other words. That means that every CHG is a model, and every subgraph is a model. The smallest hypergraph you could conceivably identify as a model would consequently be a single edge mapping between two nodes. Model selection requires picking between at least two models, so the minimal representative case is two edges mapping between the same two nodes, such as:



We cannot have both edges in a simulation. At best, they return the same value, making their calculation redundant. But usually two different models will return different values–otherwise there wouldn't be competing models in the first place. In this case the solver has to establish which calculation is better, since a causal system can't exist in two different states at once.

To determine which model is better requires additional information about the models' suitability. Mathematically, grading various entities results in an order, where each entity can be described as greater than or less than another. We can represent this order using positive, real numbers–another series with a defined order. To do so we have to create yet another function f_{cost} whose domain is the competing edges $\{e_1, e_2\}$ and codomain as the positive, real numbers $\mathbb{R}^+$. Once we can find f_{cost} we can use it to assign weights to each edge, so that the edge with the lowest weight will be preferred in the simulation path. This might look like:



Note that a higher score doesn't prohibit an edge from being used in a simulation path–it may be the case that e_1 is not valid for the found value of S. But having f_{cost} provides a mechanism for the pathfinding agent to prefer e_1 to e_2, assuming both are viable options. With this, we've provided the primary structures of a universal systems representation framework: variables, relations, partiality, cycles, and edge weights. Now we can put all these together to represent a more realistic system.

5. Demonstration of Simulation

A good system to demonstrate building a CHG is a planar pendulum. The primary states for this system are the angular position θ, velocity ω, and acceleration α, with units of rad, rad/s, and rad/s^2 respectively. We'll also assign variables for the gravitational acceleration g (m/s^2), length of the pendulum tether r (m), and the time t (s). The main equation of motion for the system is

$$\alpha = -\frac{g}{r}\sin\theta \tag{1}$$

The other relations are the integration relationships between α, ω, and θ. To keep things simple, we can write these using a simple, first-order Eulerian integration:

$$\omega_i = \omega_{i-1} + \alpha_i \Delta t \tag{2}$$
$$\theta_i = \theta_{i-1} + \omega_i \Delta t \tag{3}$$

where Δt is the time step $t_i - t_{i-1}$. This system is shown in Figure 7, while the CHG is shown in Figure 8.

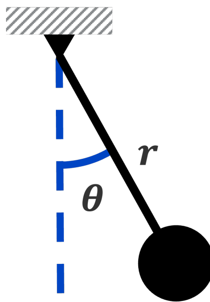


Figure 7: Schematic for a simple planar pendulum.

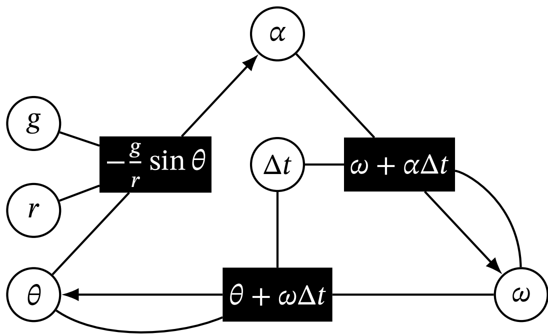


Figure 8: CHG of a simple planar pendulum.

The cycle from θ to α to ω and back to θ is clear from Figure 8. The model right has no partial edges described, and there are no competing edges (a simple hypergraph), so we don't need to add edge weights. We'll add both of these later in this example. For now, let's just show how a basic simulation works.

5.1. ConstraintHg

As described in Section 4.1, to simulate a system declaratively we need an agent that can perform pathfinding. We'll use ConstraintHg,[2] an open-source algorithm I wrote that's implemented in Python, and whose source code is in Appendix **??**. ConstraintHg uses a breadth-first search, exhaustively exploring every edge leading from every source node it encounters. That means that while it's not particularly fast, it is able to parse cycles in a CHG.

Before we can simulate our pendulum, we have to pass the CHG to ConstraintHg. The file will look like the one in Block 1. First we'll import the library as `from constrainthg import Hypergraph`. Then we define the methods (functions) used by the edges as `def Rtheta_to_alpha()` and `def Rintegrate()`. Finally, we'll create each edge in the hypergraph using the `add_edge()` method. This method takes in a list of source nodes, the target node for the edge, and the function (`rel`, for relation) used to calculate the edge. An additional parameter, `index_offset`, tells the solver to increment the index of the target node by one when solving the edge. This is crucial to iterating through the cycle.

Listing 1: CHG of simple pendulum expressed in Python using the ConstraintHg library.

```
from constrainthg import Hypergraph
from numpy import sin

def Rtheta_to_alpha(theta, g, r):
   return -g / r * sin(theta)

def Rintegrate(initial, slope, step):
   return initial + slope * step

hg = Hypergraph()

hg.add_edge(
   sources=dict(theta='theta', g='g', r='r'),
   target='alpha',
   rel=Rtheta_to_alpha,
   index_offset=1,
)

hg.add_edge(
   sources=dict(initial='omega', slope='alpha', step='del_t'),
   target='omega',
   rel=Rintegrate,
)

hg.add_edge(
   sources=dict(initial='theta', slope='omega', step='del_t'),
   target='theta',
   rel=Rintegrate,
)
```

A simulation requires a set of input nodes and a desired output. Here our inputs are an initial angular position of $\frac{\pi}{4}$ radians, gravitational acceleration of 9.81 m/s^2, a length of 0.25 m, and an initial velocity of 0 m/s (starting from rest). Let's simulate the angular position through two time steps. The simulation call looks like this:

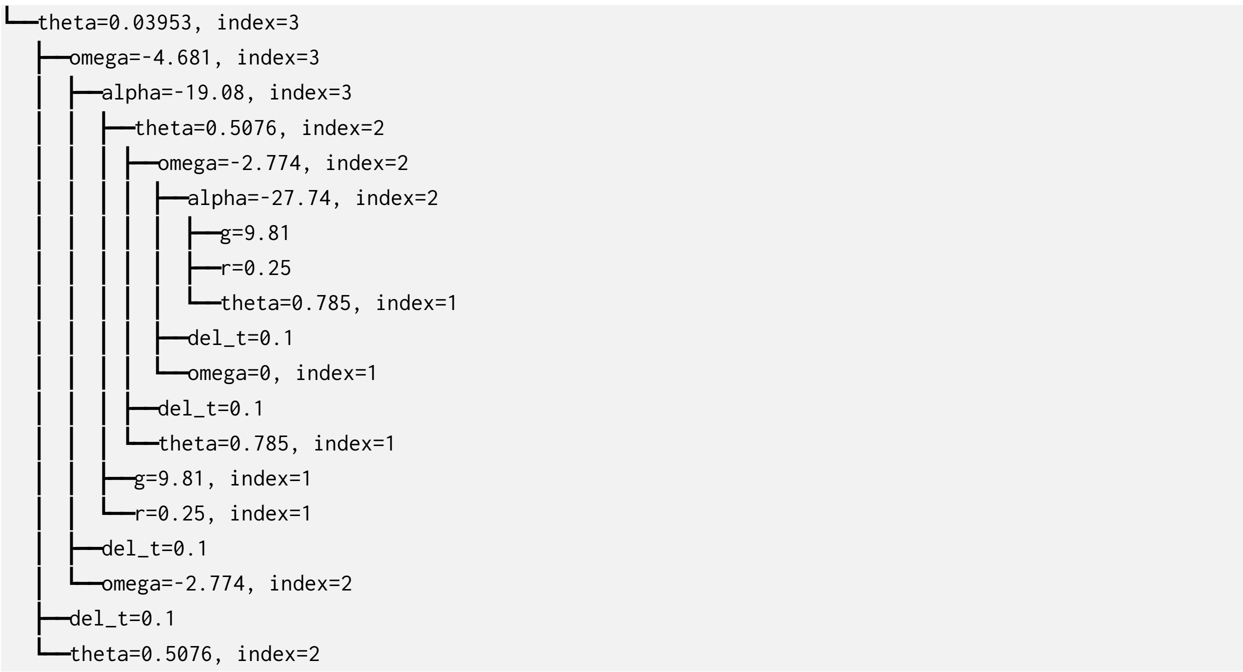
```
hg.solve(
    target='theta',
```

[2]ConstraintHg is available on the Python Package Index (PyPI). The repository is hosted on GitHub at github.com/jmorris335/constrainthg, and the documentation is linked here.

```
    inputs={'theta': 0.785, 'omega': 0.0, 'g': 9.81, 'r': 0.25, 'del_t': 0.1},
    min_index=3,
)
```

Notice that nothing about the simulation path is being defined, only the inputs and outputs. This is all that's needed for our declarative solver to autonomously find the path. If we print the path taken by the solver, we get the following output:

Listing 2: Simulation path found and printed by ConstraintHg of simple pendulum simulation.



```
└──theta=0.03953, index=3
   ├──omega=-4.681, index=3
   │  ├──alpha=-19.08, index=3
   │  │  ├──theta=0.5076, index=2
   │  │  │  ├──omega=-2.774, index=2
   │  │  │  │  ├──alpha=-27.74, index=2
   │  │  │  │  │  ├──g=9.81
   │  │  │  │  │  ├──r=0.25
   │  │  │  │  │  └──theta=0.785, index=1
   │  │  │  │  ├──del_t=0.1
   │  │  │  │  └──omega=0, index=1
   │  │  │  ├──del_t=0.1
   │  │  │  └──theta=0.785, index=1
   │  │  ├──g=9.81, index=1
   │  │  └──r=0.25, index=1
   │  ├──del_t=0.1
   │  └──omega=-2.774, index=2
   ├──del_t=0.1
   └──theta=0.5076, index=2
```

The output is a little verbose, but not unreadable. The set of paths is a tree, whose leaf nodes (the inputs) are located mostly in the center of the output. The first step of the path is in the center, where the inputs of g, r, θ are used to calculate α. Because the agent is solving a cycle, the index of α is incremented, resulting in `alpha=-27.74, index=2`. The output α_2 along with Δt and ω_1, become the inputs for the next step calculating ω_2. This repeats until we reach θ_3 at the top of the printout. Without being explicitly told how to simulate the system, it was able to figure out how to chain together a full scale simulation of the pendulum.

5.2. Increasingly Complicated Examples

We can continue to build our model to demonstrate more features of ConstraintHg. Let's increase our model's fidelity by considering damping on the pendulum. We model damping as a force proportional to the system's velocity. This updates our equation of motion to be:

$$\alpha = -\frac{g}{r}\sin\theta - c\omega \tag{4}$$

Where c is a coefficient for damping with units of 1/s. There are a few ways to add this function into the hypergraph, but the most basic is to make an entirely new edge for Equation 4. This is shown in Figure 9, where we should note the competing edges that both point to α. These edges represent the damping equation (Eq. 4) and the initial equation of motion Eq. 1.

To tell the ConstraintHg which edge to select we need to assign edge weights. We'll choose to make the damping one more preferable due to its better accuracy. In the software, we do this by writing

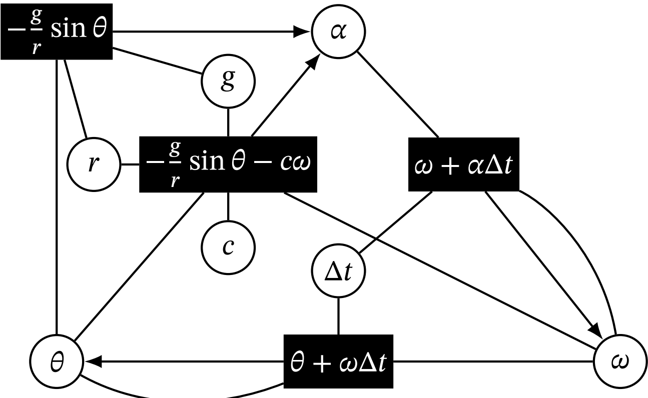


Figure 9: CHG of a simple planar pendulum with damping.

`weight=10` in the `add_edge` method.

Even with these weights, running the simulation again still returns the same result. That's because we haven't supplied an input value for c, making the edge for Equation 4 incalculable. However, this illustrates a notion of intelligence: with the edge weights provided, the solver adapts to what information is known, in this case reverting to the less accurate (but viable) model. But if we supply a value for c, the smart solver chooses the more accurate model, providing the following output shown in Block 3.

Listing 3: Simulation path found by and printed by ConstraintHg of damped pendulum simulation.

```
└──theta=0.06727, index=3
   ├──omega=-4.404, index=3
   │  ├──alpha=-16.3, index=3
   │  │  ├──theta=0.5076, index=2
   │  │  │  ├──omega=-2.774, index=2
   │  │  │  │  ├──alpha=-27.74, index=2
   │  │  │  │  │  ├──theta=0.785, index=1
   │  │  │  │  │  ├──omega=0, index=1
   │  │  │  │  │  ├──c=1
   │  │  │  │  │  ├──r=0.25
   │  │  │  │  │  └──g=9.81
   │  │  │  │  ├──omega=0, index=1
   │  │  │  │  └──del_t=0.1
   │  │  │  ├──theta=0.785, index=1
   │  │  │  └──del_t=0.1
   │  │  ├──omega=-2.774, index=2
   │  │  ├──c=1
   │  │  ├──r=0.25
   │  │  └──g=9.81
   │  ├──omega=-2.774, index=2
   │  └──del_t=0.1
   ├──theta=0.5076, index=2
   └──del_t=0.1
```

We can use conditional viability to do make modeling more convenient. Let's say we wanted to know how long it takes for the pendulum to settle after being disturbed. First, we define a variable `is_settled` whose values "True" and "False" represent whether the pendulum is settling. We can also

define the criteria for settling as the magnitudes of velocity and position being less than some threshold. These criteria can be expressed as a function and wrapped into a new edge, as shown in the first part of Block 4.

We can also define a `settling_time` as a variable. One way to calculate `settling_time` is by multiplying the time step with the index of θ. However, this only gives us the current time in the simulation, not the final settling time. To get the latter, we'll need a partial mapping that is only valid if the system is settled. We show this by passing another function $f_{\texttt{via}}$ to the edge that checks if the inputs to the edge are viable. In this case, the method will check whether `is_settled` is "True". You can see this edge in the bottom part of Block 4, passed to the `via` key of `add_edge`.

Listing 4: Additional edges for calculating settling time for the damped pendulum.

```
hg.add_edge(
   sources=dict(vel='omega', pos='theta'),
   target='is_settled',
   rel=lambda vel, pos : abs(vel) < 0.01 and abs(pos) < 0.01,
)

hg.add_edge(
   sources=dict(del_t='del_t', is_settled='is_settled', idx=('is_settled', 'index')),
   target='settling_time',
   rel=lambda del_t, idx : idx * del_t,
   via=lambda is_settled : is_settled == True,
)
```

Now we only need to ask for solver to find the settling time, and the simulation will tell us that the time the pendulum comes to rest is 3.06 seconds. We can then plot the states found by the solver against these time stamps to get the visualization in Figure 10.

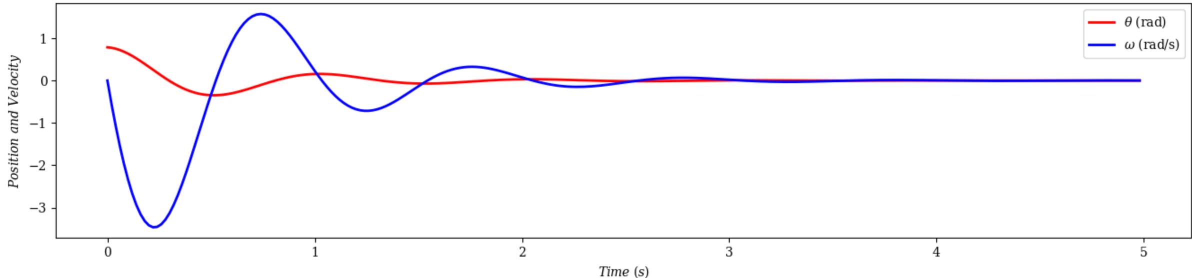


Figure 10: Plot of the states simulated in the hypergraph over 150 values.

6. Applications of Constraint Hypergraphs

The pendulum example is very simple–pendulum dynamics have been worked out for hundreds of years. Other concepts such as declarative modeling, functional programming, and general systems modeling have also been established for decades. The novelty of this work is not in executing the simulation or modeling the system, rather it is the combination of all these concepts into a single framework. The CHG represents, for the first time, a mathematical solution for performing declarative simulation on a multi-domain systems model. While it may not be interesting to simulate a pendulum, the above example demonstrates how CHGs can be used to form intelligent information retrievers: agents that can synthesize the behavior of a system without having to be directed by a human expert. The applications of this are wide-ranging, especially in multi-domain, multiscale systems found in large engineer-

ing projects. For instance, consider the breadth of product requirements a system such as an airplane. Such requirements might include:

- Maximum manufacturer's cost.
- Material selection restricted to materials passing certain performance tests.
- Maximum turnaround time to refit plane after flight.
- Maximum rates for energy consumption and fuel use.
- Minimum passenger and cargo capacities.
- Maximum rate of released pollutants.
- Minimum wing loading before buckling.
- Maximum vibrations for passengers.
- Minimum interior pressure and temperature constraints.
- Minimum visibility requirements for pilots.

This is only a mild list, full requirement documents might list constraints in the thousands. Yet, despite its brevity, this list still illustrates the diverse considerations that go into designing a complex system. Calculating metrics for these ten requirements might require models in domains such as economics, fluid dynamics, combustion engines, geometry, power and energy systems, chemistry, ergonomics, structural mechanics, thermodynamics, telecommunications, acoustics, environmental sciences, compound materials, and jet propulsion, as illustrated in Figure 11. Each model in each domain will likely be expressed in a different framework such as a circuit diagram, an energy bond graph, a stress-strain plot, a database of material properties. This greatly complicates the work of design. While each model is capable of expressing intra-system behavior, few if any can capture the inter-system relations. How, for instance, could an engineer calculate what effect lowering the temperature of the cargo hold might have on the time required to service the airplane?

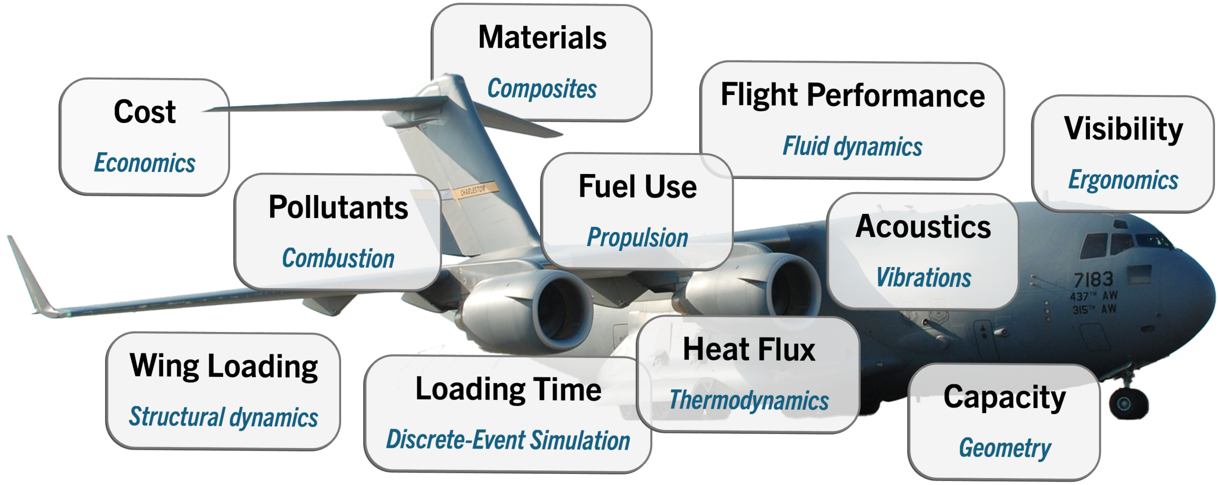


Figure 11: Illustration of information and relevant subsystem domain for a complex system. *Image by 1st Lt. Jen Richard [19].*

To capture these interactions requires updatable *digital threads*, referring to the concept of information being ported throughout an informatics system. Say an engineer establishes the ideal cargo hold temperature to be 45º F. A digital thread traces out each place where `cargo_temp = 45` is referenced. An updatable digital thread adds to this notion by requiring each port making use of `cargo_temp` to update in response to changes elsewhere on the thread. Updatable digital threads allow models to be connected to each other along prescribed ports. A model that calculates `cargo_temp` can update the thread, and all other models using `cargo_temp` will automatically update.

6.1. Cross Platform Digital Threads

One of the most significant use cases of a CHG is the tracing of digital threads, perhaps the core action required for robust model-based engineering. All relations can be represented and connected through the universal language of a CHG. This is juxtaposed against many traditional frameworks, where specialized analysis formats leads to sequestered information that cannot be ported between subsystem models. Perhaps nowhere is this more stark than with geometric models composed in Computer-Aided Design (CAD) software. Integrations with CAD systems are notoriously difficult, with complex APIs, competing, non-universal standards, and hidden dependencies that make robust solutions difficult to find.

But, by focusing on system behavior, we can turn this paradigm on its head. We won't muck about with trying to transform one software into another–creating a universal interface is an intractable problem. The key rather is to discover the constraints calculated by each software and represent them as edges in a CHG. If we can do this, then each path in a CHG will tell us a valid way to pass information between software tools. Let's do a quick example to see what this looks like in practice.

Say we wanted to use our pendulum in a grandfather clock. We'll describe the period of the pendulum T as:

$$T = 2\pi\sqrt{\frac{r}{g}} \tag{5}$$

Usually, when designing, we want to configure the pendulum to achieve a desired period T^*. To do this we can rearrange Eq. 5 to solve for the pendulum's length. For instance, if we want T^* to be 2 seconds, the ideal pendulum length r^* would be calculated as:

$$r^* = \frac{gT^{*2}}{4\pi^2} = \frac{(9.81)(2^2)}{4\pi^2} = 0.993 \text{ m} \tag{6}$$

Our assumption in calculating r^* is that the pendulum's center of mass is in the center of its bob. However, that is not true of a manufactured pendulum, whose center of mass is affected by the mass of the tether. We'll need to design the pendulum so that the distance from the point of rotation to its center of mass r is equal to r^*, balancing the mass of the tether and the mass of the bob. We'll design our pendulum to have a cylindrical tether and bob made out of aluminum and brass respectively. The mass m of a cylinder is $\rho\pi R^2 h$, where R and h are the radius and height of the cylinder and ρ is its density. The center of mass for a cylinder lies at half of its height, giving the following equation for the center of mass of the full pendulum:

$$r = \frac{\frac{h_t}{2}m_t + \frac{h_b}{2}m_b}{m_t + m_b} \tag{7}$$

where the indices t and b stand for parameters for the *t*ether and *b*ob respectively. This results in the CHG in Figure 12.

Notice there is no need to tie this into a distinct software package, as all the relationships are algebraic. To do a simulation on this model only requires a platform that can do arithmetic, like Python or C. However, sometimes we get relational rules that require more advanced tools to calculate. For instance, if our bob or tether have additional features such as tapped connections, decorative trim, or fastening hardware, we might prefer to use a CAD system to calculate center of mass rather than an idealized algebraic model.

CHGs are function-based, so to wrap the CAD system we have to express its functionalities in terms of functions. We can write a script that generates a solid body from our geometric parameters,

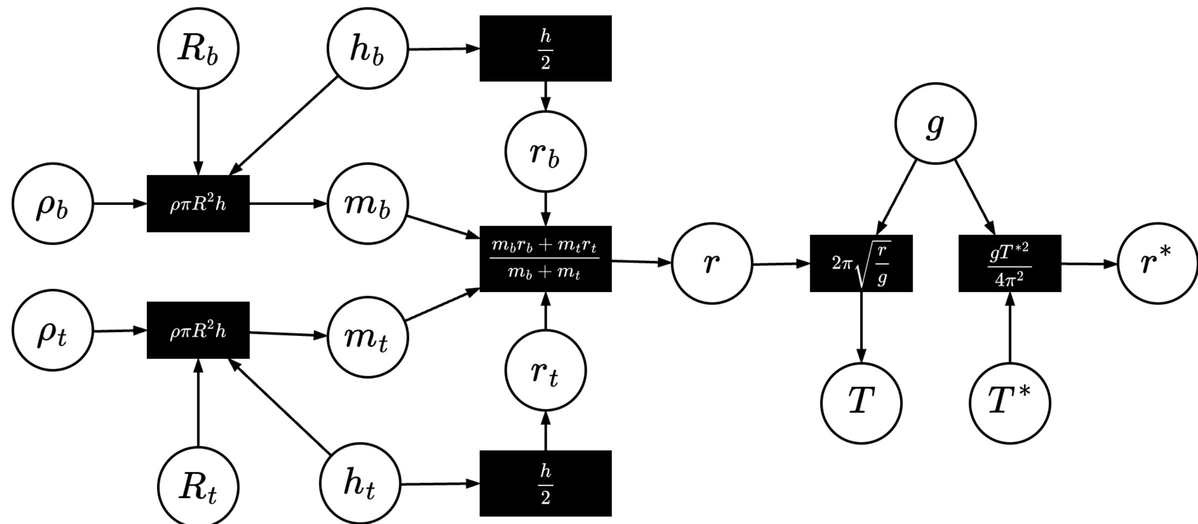


Figure 12: A CHG for a pendulum with a cylindrical tether and bob, where all the relationships are algebraic.

and then save this solid body as a node `B`. Then we can use the SOLIDWORKS API to calculate the body's mass properties. The specific function for calculating the center of mass in SOLIDWORKS is `IMassProperty.CenterOfMass`[3]. If we create an IMassProperty `IMP` from `B`, then we can calculate the center of mass using this function.

Our CHG now looks like Figure 13, where the red "SW" cube indicates a relation that requires SOLIDWORKS to compute. The "Create Body" is our script for constructing a cylinder, while the other SOLIDWORKS calls access specific functions of the SOLIDWORKS API. With the CHG as configured, we can create the solid geometry of our clock pendulum, and use it to calculate our actual period compared to our desired one.

Figure 13 represents the integration of SOLIDWORKS with our dynamic models using updatable digital threads. You'll notice the graph in Figure 13 has gotten far messier. That's because working with CAD APIs is intrinsically complicated–there's a lot of work that goes under the hood to make CAD usable. A CHG, it's worth noting, does not reduce a system's complexity–all the same effort that would go into using SOLIDWORKS's API must still be conducted to create the CHG. However, we will see now how using the CHG introduces two new possibilities in doing model-based engineering.

The first paradigm shift is going from workflows to functions. A workflow is a series of steps for completing some task, such as the steps necessary to calculate the pendulum's period. The traditional way of integrating software involves constructing the workflow and establishing at every point what messages should be exchanged between involved software platforms. This workflow is executed by a calling agent (such as Teamcenter, shown in Figure 14) and details how model relations should be executed. One possible workflow for calculating the pendulum's desired length could be:

1. Pass the geometric parameters ρ, h, and R into SOLIDWORKS.
2. Use the geometric parameters for the bob and the tether to create a solid body `B` in SOLIDWORKS.
3. Use `B` to create an IMassProperty `IMP` in SOLIDWORKS.
4. Use `IMP` to calculate the mass m and length r for both the bob and tether in SOLIDWORKS.
5. Pass m and r for the bob and tether back to an algebraic calculator.

[3]An IMassProperty is a construct from the SOLIDWORKS API that allows users to query the mass properties of solid bodies. More information available at help.solidworks.com/2025/english/api/sldworksapiprogguide/Overview/Mass_Properties.htm

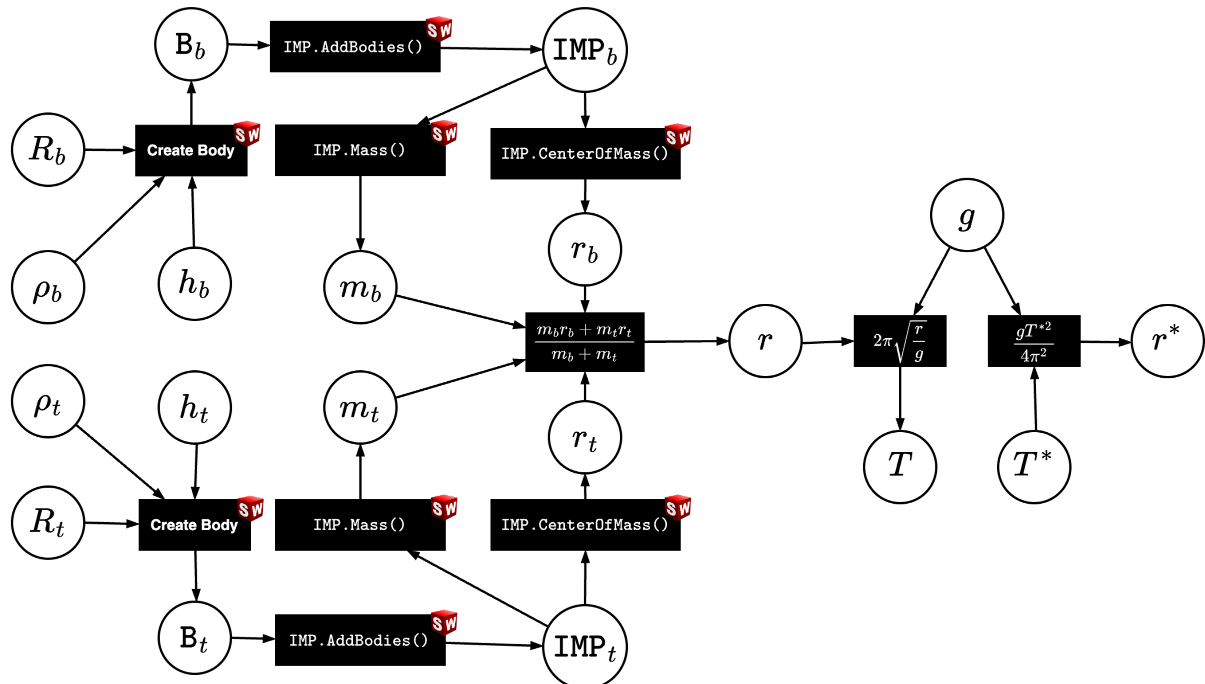


Figure 13: A CHG for a pendulum with a cylindrical tether and bob, where some relationships are calculated in SOLIDWORKS.

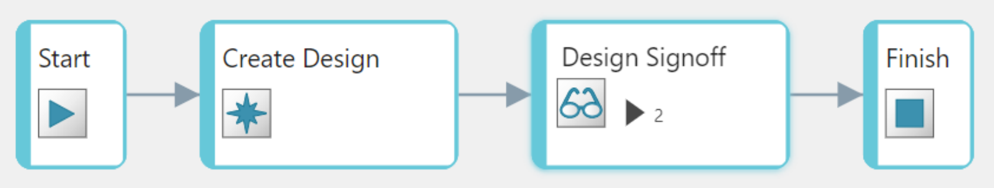


Figure 14: An example design workflow in Teamcenter, a product data management application owned by Siemens [20].

6. Calculate the pendulum radius r as $\frac{m_b r_b + m_t r_t}{m_b + m_t}$.

Though this workflow is technically valid, it is highly inflexible. If we wanted to change our model to calculate the mass of a sphere versus a cylinder, we'd have to create an entirely new model. This is because the workflow specifies a single starting point and ending point–it cannot be adapted to new information.

Contrast this with the CHG, which describes the steps of a workflow rather than the workflow itself. Mathematically this is no different from finding simulations in the CHG. The steps are the function relations, and the order is given by paths. Because these paths can be composed with each other, a process that started from a spherical bob could just as readily make use of the `IMassProperty.CenterOfMass` function.

The second paradigm shift is from software as processor to software as calculator. Notice how instead of passing a suite of information to SOLIDWORKS, the CHG picks out the various functionalities that SOLIDWORKS can perform. A pathfinding agent can pick and choose which functionalities should be employed at which stage in the simulation, with the order and validity enforced by the graph structure. A path might bounce back and forth between several applications repeatedly; e.g. simu-

lating information in one package and then passing the outputs to another. The result is that all the types of inter-software communication possible for an organization can be captured in a single model. This model can express the behavior of the system's geometry, kinematics, dynamics, and every other domain, with the CHG showing how information flows through the system and across analytical platforms.

The result of embedding application functionalities as edges in a hypergraph is a much more robust platform for model-based engineering. All models, regardless of domain, can be expressed and integrated into a single CHG. Each node represents an artifact that can be used by an agent such as a document, variable, requirement, or specification. The CHG forms an authoritative source of truth, which users can interrogate to learn information about the products they are designing [21].

6.2. Digital Twins

In addition to model-based engineering, CHGs provide a powerful framework for understanding *digital twins*. A digital twin (DT) is way of virtually representing some real entity. It is often described as an "atoms-to-bits" representation, because it replicates something physical as a digital construct. The point of using a DT is to interrogate the real system, giving us all the information we would want to know about how the system is defined and what state it is in. This idea of a DT is not far from the notion of a system that we've been working with, with the exception that a DT must be linked to something real. While we can make models of any concept we like–say pendulums of grandfather clocks–our models won't be DTs unless we can point to the actual grandfather clock pendulum they represent. The distinction we're making is can be seen between definite versus indefinite articles; you can make a model of *a* pendulum, but a DT can only represent *the* pendulum.

How do we represent something real? The key is to include observations of the real system in the virtual domain. While it is usually impossible to observe every state in a system, the more observations we can make the closer our representation is to the real entity we're representing. If all we do is observe our system, then our DT becomes nothing more than list of measured values that could be represented with a database. But any state that can't be observed explicitly must be simulated using a model. To make the full system interrogatable, the DT must provide all the observations and simulations made of the system. Consequently, DTs are often portrayed as the integration of data with models [22].

This description of a DT (described more fully in [23]) is specifically worded to show how it is fully a system representation, just with the added specification that the system it represents be real and specifiable. Because we have already shown how CHGs can be used to represent all systems, this means that CHGs can represent all DTs as well. Though several frameworks have been proposed for framing DTs, such as knowledge graphs or Bayesian networks, CHGs are the first framework that fully encode the behavior of a system into the representation framework, making DTs more independent and reconfigurable. This addresses a major issue for science and manufacturing. For instance, the National Academies of Sciences, Engineering, and Medicine [24] recently conducted a large-scale consensus study report investigating the needs for future research on DTs, writing:

> Model management is key for supporting the digital twin evolution. For a digital twin to faithfully reflect temporal and spatial changes where applicable in the physical counterpart, the resulting predictions must be reproducible, incorporate improvements in the virtual representation, and be reusable in scenarios not originally envisioned. This, in turn, requires a design approach to digital twin development and evolution that is holistic, robust, and enduring, yet flexible, composable, and adaptable. **Digital twins require a foundational backbone** that, in whole or in part, is reusable across multiple domains, supports multiple diverse activities, and serves the needs of multiple users. ... Sustaining a robust, flexible, dynamic, accessible, and secure digital twin is a key consideration for creators,

funders, and the diverse community of stakeholders. *[emphasis added]*

The motivation behind this grand challenge, which has been echoed by numerous other researchers [22, 25–30], is that DTs provide methods for understanding the physical world. Whether the thing being represented is the world's climate [31], a river [32], a person [33, 34], or a machine [35, 36], the way that we perceive and make decisions about reality is captured by a DT–or at least should be, if we are to understand the decisions that are made. By providing CHGs as a usable tool for creating DTs, we can address many of the problems that have come from representing these multi-domain, multiscale, complex systems [23].

7. Summary

In this intro we've fully introduced CHGs, what they are, and what they're used for. We started by defining a system as a set of data, and how our goal when working with systems is to understand something about them. There are two ways we are able to do this: observation, when we make a measurement of a physical phenomenon; and simulation, when we predict data through execution of a model. Together, we called these methods interrogation.

Of the two mechanisms, simulation is the more difficult one to accomplish virtually as it requires describing a system's behavior. We reported an important definition of behavior as being a set of constraints on the states a system can exhibit. These constraints can be represented as functions that determine the information expressible by a system given certain conditions. We took a system's information and represented it as nodes in a graph, and then represented the functions as edges mapping between them. Because these functions are multiple-arity this forms a hypergraph. The specific construct of functions mapping between state variables was termed a CHG.

One of the major benefits of using a CHG is that simulation is reduced to tracing paths through the hypergraph, where every path represents a potential simulation that can be performed on the system. This forms a functional, declarative framework for representing systems. We discussed some of the more complicated structures within a CHG–cycles and multi-edges–and then gave an example of using a CHG to simulate a system based on the ConstraintHg software. The final part of this paper focused on the applications of CHGs to creating foundations for model-based engineering and digital twins.

References

[1] John Morris. "Universal Systems Simulation via Constraint Hypergraphs with Applications to Digital Twins". PhD thesis. Clemson, SC, USA: Clemson University, Dec. 12, 2025. https://open.clemson.edu/all_dissertations/4127 (visited on 01/20/2026).

[2] François E. Cellier. *Continuous System Modeling*. New York, NY: Springer, 1991. 755 pp. ISBN: 978-1-4757-3922-0. DOI: 10.1007/978-1-4757-3922-0.

[3] Jan C. Willems. "The Behavioral Approach to Open and Interconnected Systems". *IEEE Control Systems Magazine* 27.6 (Dec. 2007), pp. 46–99. ISSN: 1941-000X. DOI: 10.1109/MCS.2007.906923.

[4] Alonzo Church. "A Set of Postulates For the Foundation of Logic". *Annals of Mathematics* 34.4 (1933), pp. 839–864. ISSN: 0003-486X. DOI: 10.2307/1968702. JSTOR: 1968702.

[5] Alonzo Church. *The Calculi of Lambda Conversion*. Annals of Mathematics Studies 6. Princeton, NJ: Princeton University Press, 1941. 1 p. ISBN: 978-0-691-08394-0. DOI: 10.1515/9781400881932.

[6] Paul Hudak. "Conception, Evolution, and Application of Functional Programming Languages". *ACM Comput. Surv.* 21.3 (Sept. 1, 1989), pp. 359–411. ISSN: 0360-0300. DOI: 10.1145/72551.72554.

[7] Bruce J. MacLennan. *Functional Programming: Practice and Theory*. Reading, Mass: Addison-Wesley, 1990. 596 pp. ISBN: 978-0-201-13744-6.

[8] Jan C. Willems. "Models for Dynamics". *Dynamics Reported*. Ed. by U. Krichgraber and H. O. Walther. Vol. 2. Wiesbaden: Vieweg+Teubner Verlag, 1989, pp. 171–266. ISBN: 978-3-519-02151-3. DOI: 10.1007/978-3-322-96657-5_5.

[9] W. Ross Ashby. *An Introduction to Cybernetics*. Internet. London: Chapman & Hall, 1956. http://pcp.vub.ac.be/books/IntroCyb.pdf (visited on 02/26/2025).

[10] W Ross Ashby. *The Set Theory of Mechanism and Homeostasis*. Technical Report 4.7. Urbana, IL, USA: University of Illinois, Sept. 1962. https://digital.library.illinois.edu/items/3c7af690-29ac-0136-4d81-0050569601ca-4 (visited on 10/14/2025).

[11] George J. Friedman and Cornelius T. Leondes. "Constraint Theory, Part I: Fundamentals". *IEEE Transactions on Systems Science and Cybernetics* 5.1 (Jan. 1969), pp. 48–56. ISSN: 2168-2887. DOI: 10.1109/TSSC.1969.300244.

[12] George J. Friedman and Phan Phan. *Constraint Theory*. Vol. 23. IFSR International Series on Systems Science and Engineering. Cham: Springer International Publishing, 2017. ISBN: 978-3-319-54791-6. DOI: 10.1007/978-3-319-54792-3.

[13] John Morris, Gregory Mocko, and John Wagner. "Unified System Modeling and Simulation via Constraint Hypergraphs". *J. Comput. Inf. Sci. Eng.* 25.6 (Apr. 4, 2025), p. 061005. DOI: 10.1115/1.4068375.

[14] Saunders Mac Lane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics 5. New York: Springer-Verlag, 1971. 262 pp. ISBN: 978-0-387-90035-3.

[15] John Morris, Gregory Mocko, and John Wagner. "Effects of Functional and Declarative Modeling Frameworks on System Simulation". *J. Dyn. Sys., Meas., Control* (Jan. 2026). DOI: 10.1115/1.4070883.

[16] Peter Wegner. "Concepts and Paradigms of Object-Oriented Programming". *SIGPLAN OOPS Mess.* 1.1 (Aug. 1, 1990), pp. 7–87. ISSN: 1055-6400. DOI: 10.1145/382192.383004.

[17] Giorgio Ausiello et al. *Optimal Traversal of Directed Hypergraphs*. ICSI Technical Report ICSI TR-92-073. Berkeley, CA: International Computer Science Institute, Sept. 1992. https://www.icsi.berkeley.edu/icsi/publication_details?n=778 (visited on 08/09/2024).

[18] Hermann Schlichting and Klaus Gersten. *Boundary-Layer Theory*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017. ISBN: 978-3-662-52917-1. DOI: 10.1007/978-3-662-52919-5.

[19] 1st Lt. Jen Richard. *American Servicemembers, Helicopters Deploy to Haiti*. Mar. 21, 2010. https://www.af.mil/News/Photos/igphoto/2000379980/ (visited on 10/17/2025).

[20] Siemens. *Workflow Example*. https://docs.sw.siemens.com/documentation/external/PL20201019171517939/en-US/aw_collection_sc/aw/5.2/aw_collection_sc/common/en_US/graphics/graphicLibrary/awc/workflow/wf-example-WFD.png (visited on 10/21/2025).

[21] John Morris et al. "Declarative, Multi-physics Simulation Between Applications via Constraint Hypergraphs". *Under review with J. Comput. Inf. Sci. Eng.* (Oct. 2025).

[22] Douglas L. Van Bossuyt et al. "The Future of Digital Twin Research and Development". *J. Comput. Inf. Sci. Eng.* 25.8 (Apr. 16, 2025), p. 080801. DOI: 10.1115/1.4068082.

[23] John Morris et al. "Constraint Hypergraphs as a Unifying Framework for Digital Twins". *Under review with Systems Journal* (July 2025). DOI: 10.48550/arXiv.2507.05494.

[24] National Academies of Sciences, Engineering, and Medicine. *Foundational Research Gaps and Future Directions for Digital Twins*. Consensus Study Report. Washington, D.C.: The National Academies Press, Mar. 28, 2024. DOI: 10.17226/26894.

[25] Stefan Boschert, Christoph Heinrich, and Roland Rosen. "Next Generation Digital Twin". *Proceedings of TMCE 2018*. Twelfth International Symposium on Tools and Methods of Competitive Engineering. Las Palmas de Gran Canaria, Spain: University of Technology, Delft, May 7, 2018. ISBN: 978-94-6186-910-4. https://www.researchgate.net/publication/325119950_Next_Generation_Digital_Twin (visited on 01/18/2025).

[26] Xiaochen Zheng, Jinzhi Lu, and Dimitris Kiritsis. "The Emergence of Cognitive Digital Twin: Vision, Challenges and Opportunities". *International Journal of Production Research* 60.24 (Dec. 17, 2022), pp. 7610–7632. ISSN: 0020-7543. DOI: 10.1080/00207543.2021.2014591.

[27] Alessandro Ricci et al. "Web of Digital Twins". *ACM Trans. Internet Technol.* 22.4 (Nov. 14, 2022), 101:1–101:30. ISSN: 1533-5399. DOI: 10.1145/3507909.

[28] Anto Budiardjo and Doug Migliori. *Digital Twin System Interoperability Framework*. Digital Twin Consortium, Dec. 7, 2021. https://www.digitaltwinconsortium.org/wp-content/uploads/sites/3/2022/06/Digital-Twin-System-Interoperability-Framework-12072021.pdf (visited on 12/07/2021).

[29] Xiangdong Wang et al. "Knowledge-Graph-Based Multi-Domain Model Integration Method for Digital-Twin Workshops". *Int J Adv Manuf Technol* 128.1–2 (Sept. 2023), pp. 405–421. ISSN: 0268-3768, 1433-3015. DOI: 10.1007/s00170-023-11874-4.

[30] Roberto Minerva and Noël Crespi. "Digital Twins: Properties, Software Frameworks, and Application Scenarios". *IT Professional* 23.1 (Jan. 2021), pp. 51–55. ISSN: 1941-045X. DOI: 10.1109/MITP.2020.2982896.

[31] Jörn Hoffmann et al. “Destination Earth – A Digital Twin in Support of Climate Services”. *Climate Services* 30 (Apr. 1, 2023), p. 100394. ISSN: 2405-8807. DOI: 10.1016/j.cliser.2023.100394.

[32] Xiaopeng Wang et al. “How a Vast Digital Twin of the Yangtze River Could Prevent Flooding in China”. *Nature* 639.8054 (Mar. 2025), pp. 303–305. ISSN: 1476-4687. DOI: 10.1038/d41586-025-00720-0.

[33] R. Laubenbacher et al. “Building Digital Twins of the Human Immune System: Toward a Roadmap”. *NPJ Digit. Med.* 5.1 (1 May 20, 2022), pp. 1–5. ISSN: 2398-6352. DOI: 10.1038/s41746-022-00610-z.

[34] Dassault Systèmes. *Meet “Emma Twin,” Dassault Systèmes’ Avatar Showcasing How Virtual Twins Drive Healthcare Innovation*. Newsroom. Sept. 18, 2023. https://www.3ds.com/newsroom/press-releases/meet-emma-twin-dassault-systemes-avatar-showcasing-how-virtual-twins-drive-healthcare-innovation (visited on 05/22/2024).

[35] Duncan W. Gibbons and Paul Witherell. “Model-Based Production Operational Control for Metal Additive Manufacturing”. *Proceedings of the Thirty-Fifth Annual International Solid Freeform Fabrication Symposium*. 35th Annual International Solid Freeform Fabrication Symposium. Austin, TX, US, Aug. 11, 2024. https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=958335 (visited on 06/10/2025).

[36] Zijue Chen et al. “Service Oriented Digital Twin for Additive Manufacturing Process”. *Journal of Manufacturing Systems* 74 (June 1, 2024), pp. 762–776. ISSN: 0278-6125. DOI: 10.1016/j.jmsy.2024.04.015.