UNIVERSAL SYSTEMS SIMULATION VIA CONSTRAINT HYPERGRAPHS WITH APPLICATIONS TO DIGITAL TWINS

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Mechanical Engineering

---

by
John Morris
December 2025

Accepted by
Dr. John Wagner, P.E., Co-Committee Chair
Dr. Gregory Mocko, Co-Committee Chair
Dr. Laura Redmond
Dr. Cameron Turner

Plain Language Abstract

Constraint hypergraphs (CHGs) are a new mathematical method for representing information that captures all the different ways a system can behave. While most methods for information capture focus on describing things, CHGs emphasize simulation. Any model of a system can be turned into a CHG. Additionally, once something is represented in a CHG, all the information that can be known about that entity can be automatically derived from the graph. This allows users to predict what state something will be in without having to manually specify all the inferences leading up to that prediction. There are two major applications of this framework described in the dissertation. The first is model-based engineering platforms, where users can connect all the software and databases used in an organization into a single, executable CHG. The second application is digital twins, which when something physical is represented on a computer. Digital twins are used in manufacturing, healthcare, and science. It is shown how they become far easier to connect and maintain when created as a CHG.

Abstract

The characterization of systems encompasses a variety of modeling frameworks designed to capture specific behaviors and components of various system domains. Whatever the framework, the core elements of a system representation are the information of the system and a description of how that information is related. The relations in deterministic systems are functions, which, when composed to form executable processes, can be used to simulate system data. A declarative modeling framework is one that encodes mechanisms for preparing these simulations within the model structure, allowing an external agent to form the execution processes required for a given context.

To date, no declarative modeling framework has been proposed that allows for these processes to be discovered for any system across any domain. In this dissertation, a new framework called a constraint hypergraph is proposed that provides universal, declarative modeling. System behavior is embedded in this framework as paths through the hypergraph. Simulating a system represented with a constraint hypergraph can be accomplished by an autonomous agent capable of discovering these paths. This, combined with the graphical nature of the framework, allows system information to be interrogated autonomously across domains, enabling multiphysics, multiscale modeling and simulation of complex systems.

Applications of this framework are shown to model-based engineering platforms and digital twins. The former is given by showing how updatable digital threads can be traced throughout a data ecosystem, including across analysis software platforms such as CAD and FEA tools. Constraint hypergraphs are also shown to provide a robust foundation for creating digital twins, leading to twins that are usable, interoperable, maintainable, and verifiable. These applications are demonstrated using custom algorithms published in the open-source package ConstraintHg.

Dedication

To Taylor, Jordan, and Benson

Who are worth far more than any of these words

This research described by this dissertation was supported by the Product Lifecycle Management Center at Clemson University. More information on the center can be found at clemson.edu/centers-institutes/plm.

The views presented are those of the authors and do not necessarily represent the views of Clemson University, the U.S. Navy, U.S. Army, or any other organization.

In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of Clemson University's products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink. If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

Acknowledgements

It would be irresponsible of me not to acknowledge the sacrifices made on my behalf throughout my PhD program. Principle among these efforts are those provided by my advisors, Dr. John Wagner and Dr. Greg Mocko. From the very beginning, they labored to support me both as a scholar and as a human. If I have grown at all over the last few years, it is thanks to their empathy and guidance. I am deeply indebted to them, and appreciative of their generosity towards me. They also provided great opportunities for me through the Product Lifecycle Management Center at Clemson University.

In doing mental battle with some of the more tangled concepts of this dissertation, I've further appreciated conversations with my advisors and my committee, including Dr. Cameron Turner and Dr. Laura Redmond, discussions on digital twins and systems design with Dr. Douglas Van Bossuyt, Dr. Guodong Shao, Dr. Paul Witherwell, Dr. Lisha White, Dr. Duncan Gibbons, Dr. Astrid Layton, Dr. Abheek Chatterjee, and Dr. Joe Gregory, guidance on category theory from Dr. Matthew Macauley, Dr. Spencer Breiner, and Fabrice Razafimahatratra, and research advice from Dr. Christopher Mabey, Dr. Satchit Ramnath, and Dr. Jacob Sorber. This list is insufficient and always will be, but these names represent a portion of all those who have contributed to my own enlightenment, often receiving little benefit in return.

There are several people who have contributed directly to this work. Co-authors include Dr. Wagner, Dr. Mocko, Dr. Van Bossuyt, Dr. Ramnath, and Ed Louis. I also thank Richard Alves for configuring the Spanagel microgrid used to validate the microgrid DT.

I also have benefitted greatly from a supportive lab; concluding my time there is the most painful part of finishing this dissertation. I'm grateful for the critique, the sparring, the bonding, the sharing, and the camaraderie that formed working alongside peers such as Ed, Meredith Sutton, Evan Taylor, Nana Adjei, Nikhil Raj, Abhishek Indupally, Supratik Showdho, Venkat Jayya, Sai Shanoy, Chandima Abeynayake, Katelynn Hughes, Manuel Abadi, and every member of the CEDAR lab.

Most significantly, I have to give thanks to my wife, Kherissa. Together we've gone through a pan-

demic, career changes, three babies, and every twist and turn that comes from raising a family. Her efforts more than matched mine as we've figured out how to keep careers moving, papers being written, and kids getting enough calories. Thank you for reading my worst drafts, listening to my most half-baked ideas, and always believing in us.

Finally, I thank God, from whom comes every truth, blessing, weakness, and strength. My prayer is that good things will come out of this work (D&C 11:12).

Table of Contents

List of Tables

List of Figures

Preface

This dissertation is primarily based on papers published or submitted to publication between 2024 and 2025. These papers are divided by chapter. It is the opinion of the author that the thematic progression of the dissertation is unaffected by the strict inclusion. Note that the text has been adjusted to improve continuity and readability. There is some redundancy between chapters, (such as redefining terms) which was permitted to remain to preserve the context of each chapter. One benefit arising from this is that each chapter can stand on its own, without dependencies on previous chapters.

The two chapters that are not based on a submitted paper are Chapter 5 and the Introduction. The first is a review of digital twins adapted from the dissertation proposal, while the second is styled as a comprehensive guide to constraint hypergraphs, their usage, and applications. The introduction is less formal in tone, and is intended to serve as a general tutorial to the subject. More formal definitions of the terms described in the introduction are given in the remainder of the dissertation. The intent is for a reader to be able to fully grasp the concepts given in the dissertation after reading the introduction. A reader who is unsatisfied by the details given in the introduction can then investigate further by reading the remaining chapters. The thematic flow is the following:

1. Comprehensive introduction

Theory:

2. Definition of constraint hypergraphs
3. Mathematics of modeling and simulation

Applications:

4. Use of constraint hypergraphs in model-based engineering
5. Review of digital twins
6. Use of constraint hypergraphs in representing digital twins

CHAPTER 1

Introduction

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] [1].

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

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

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 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.

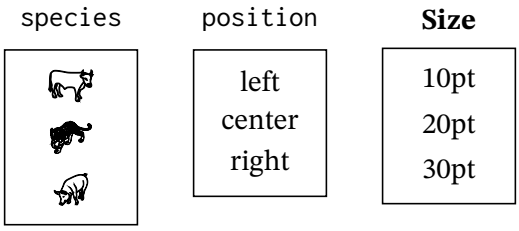
There are two mechanisms for describing a system. The first one is *observation*, where a rational agent assigns a datum to some perceived phenomenon. If every variable assigned to a system can be fully observed, then the work of a scientist describing the system is complete. However, it is more often 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 dissertation is focused on the mechanisms we use to perform simulation.

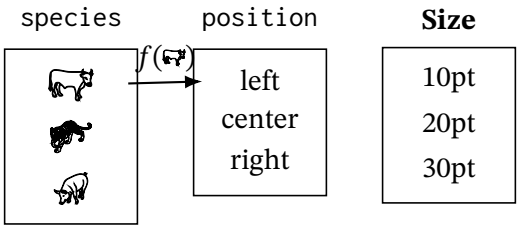
1.1.1. Systems Modeling

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 [2], 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.



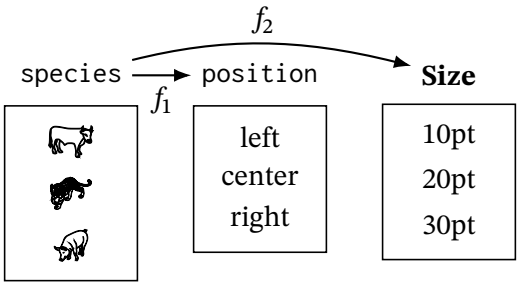
There are two ways to restrict the state space. The first is to eliminate certain states independently. For instance, the position of a symbol will never be "above" or "in front of". These actions are performed when characterizing the system, when the modeler defines what it implied by `position`. As 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

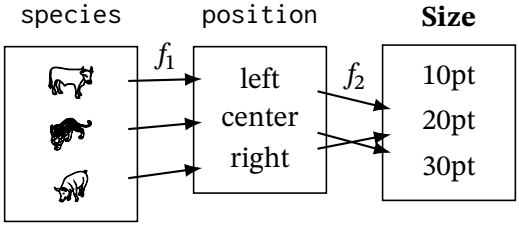
relationship. The second is that these relations are nondeterministic. It's unnecessary for this dissertation to explore the philosophy of determinism. Instead, we can merely note that if a behavior cannot be construed to be deterministic, then it becomes impossible to *determine* anything from it. Restricting 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.

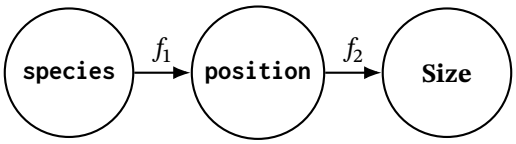
With this new system, let's identify some behaviors. We have a mapping between the `species` of animal and its `position`:  on the "left", etc. And we have another one between its `position` and `size`: symbols in the middle are 30pt, otherwise they're 20pt. We can represent these behaviors with the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



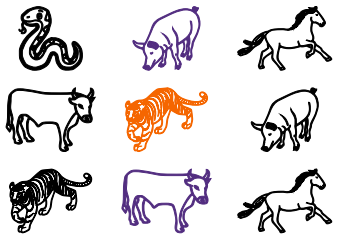
The visual shows the simulations chains in the model. For two functions to compose requires them to overlap in their domain/codomain. When this occurs, it is guaranteed that the output of one function can be used as the input for another function, allowing extended simulation chains. With the model above, 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.

1.1.2. Graphical Representations

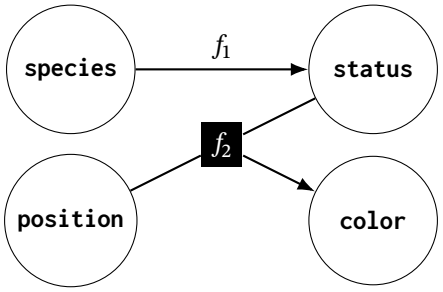
The scale of our small system is quite modest. More typical systems are too complicated for each datum to be represented. Instead, we'll draw the model graphically: condensing the variables in the system to singular nodes, and draw the relations between them as edges. The result is the following:



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 $f_2 \coloneqq (\texttt{status}, \texttt{position}) \to \texttt{color}$. 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.

1.2. 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 [3] (and revised in 1941 [4]). The notation of Church's lambda calculus was borrowed by McCarthy in 1960 in developing the first functional programming language, Lisp [5]. Lisp was the first of many developments in functional programming, giving rise to languages such as Haskell and Miranda [6]. This provided useful platforms for decomposing systems, especially when Willems gave his behavioral interpretation of a system in 1989 [7].

Separately, the notions of set and information theory paved the way for the study of cybernetics, founded by Ashby in 1956 [8]. 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 [9].

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 [10]. 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 1.1).

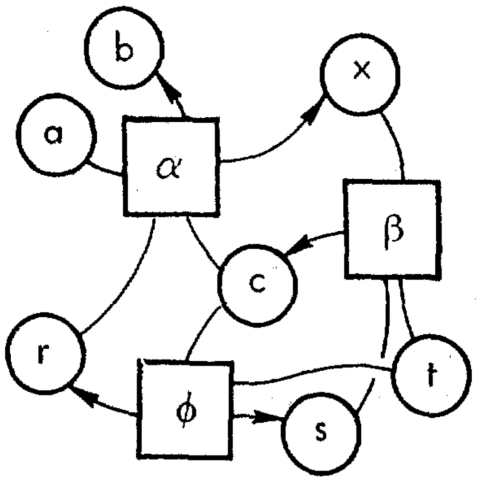


Figure 1.1: Bipartite "model graph" proposed by Friedman where circles represent variables and rectangles represent relations. Extracted with permission from George J. Friedman and Cornelius T. Leondes. "Constraint Theory, Part I: Fundamentals". *IEEE Transactions on Systems Science and Cybernetics* 5.1 (Jan. 1969), pp. 48–56. ISSN: 2168-2887. DOI: 10.1109/TSSC.1969.300244. *Copyright © 1969, IEEE*

Friedman's framework provided ways to analyze a complex, heterogeneous system, [11], 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 dissertation takes Friedman's model graphs and expands them into a tool for interrogation. It will be shown in Chapter 2 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.

1.3. Constraint Hypergraph Properties

But first, we need a more rigorous definition of what a CHG is. Under the parlance of category theory [12], 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 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 1.2 has three edges. Variables A and B are the sources for edge f_1, whose target is D.

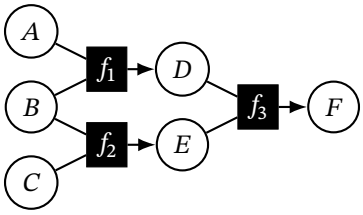


Figure 1.2: A basic CHG with six nodes and three hyperedges.

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 1.2, the mappings given by the three edges are $\{A\times B \xrightarrow{f_1} D; B\times C \xrightarrow{f_2} E; D\times E \xrightarrow{f_3} F\}$.

1.3.1. 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" [8]. 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 (as shown in Chapter 2).

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.

1.3.2. 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 1.2 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 1.2, there are actually six paths, as shown in Figure 1.3.

As shown in Figure 1.3, 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

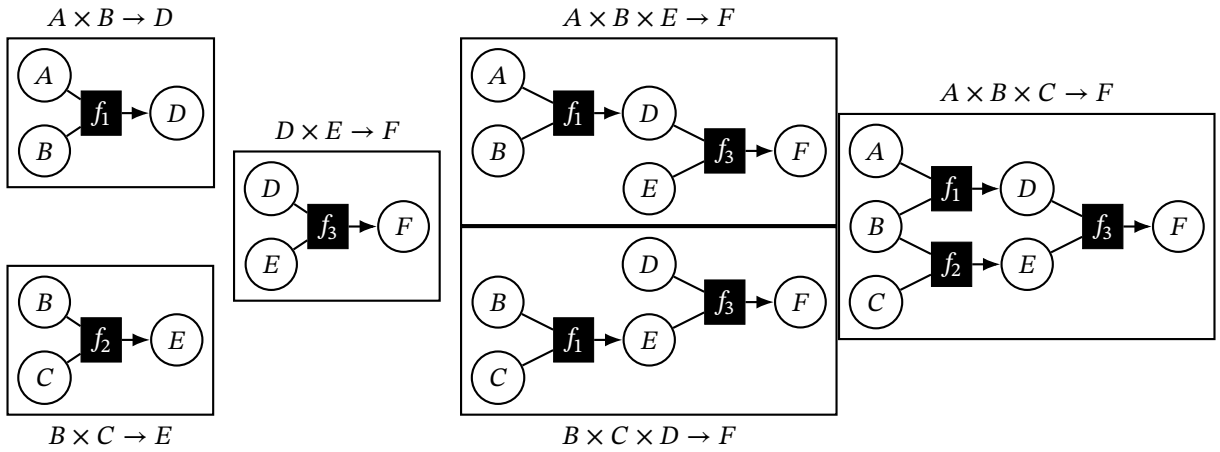


Figure 1.3: A breakout of all six paths of the CHG shown in Figure 1.2, with the composed mappings from the input (source) nodes to the output (target) node.

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 1.3, 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 1.2 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.

1.3.3. 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. More information on declarative versus imperative modeling is given in Chapter 3.

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 [13]. 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 1.4, where two CHGs exist: one with four nodes $\mathcal{C}_1 \coloneqq \{A, B, C, D\}$ and the other with three $\mathcal{C}_2 \coloneqq \{C, D, E\}$. 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.

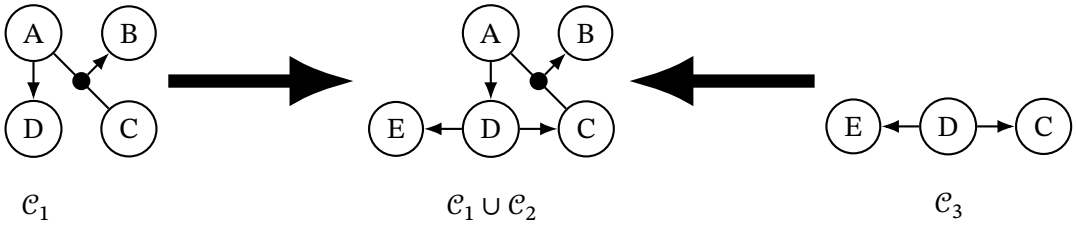


Figure 1.4: Example of merging two CHGs (left and right) into a single CHG (center) via a union operation.

1.4. Structure of a CHG

1.4.1. Pathfinding

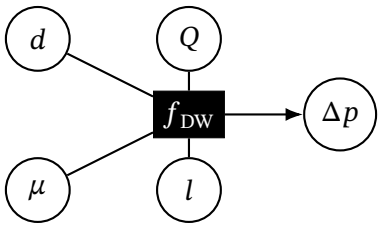
Declarative simulation requires an agent who can interpret the model structure and synthesize the executable processes. To perform declarative simulation, we should be able to describe the information we know (our inputs) and have the agent automatically interrogate the system to provide the information 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 [14]. 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.

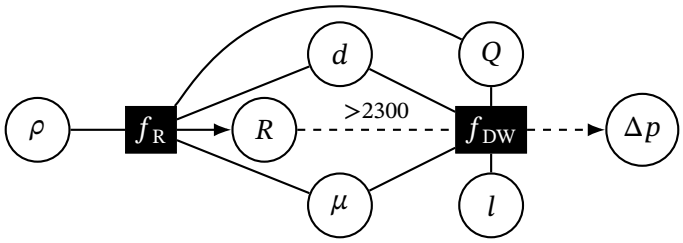
Djikstra's algorithm assumes that every edge originating from some node is always viable to traverse. The modified algorithm extends this condition to say that a hyperedge is viable to traverse if the agent has explored every source node for the hyperedge. However, this requirement is too strong for a CHG, which, if you recall from Section 1.3, 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 R is less than 2300 [15]. You can calculate R 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 R to f_{DW}'s source set. Recall that the domain of a function is the Cartesian product of its arguments. By adding R as input to we are implying that the mapping for f_{DW} actually goes from $Q \times \mu \times d \times l \times R$, allowing us to specify which values of R are unknown for f_{DW}. We show this in the following CHG:



The important part of this addition is the dashed line connecting R to f_{DW}, which we'll use to indicate that only a subset of the values of R are mapped by the connected function. Now any pathfinding agent attempting to traverse f_{DW} will need to check whether $R > 2300$, 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 R is known–either provided as an input or calculated using f_R. 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 ($R > 2300$). 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.

1.4.2. 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 R logically use R 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 1.5, where the dots indicate that the pattern repeats for every node x_i up to x_n.

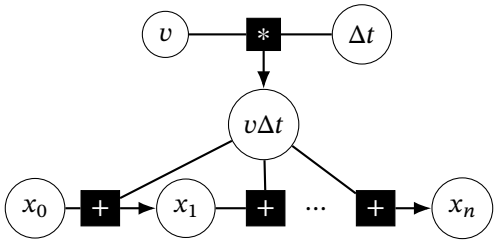


Figure 1.5: A simple hypergraph explicitly mapping out the relationships between variables across time steps.

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 1.6.

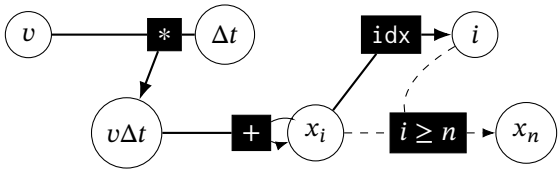


Figure 1.6: A non-simple hypergraph with a cycle.

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.

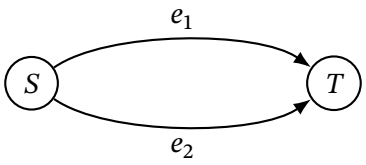
Note though that the path from x_i to x_n is not a full path–it has partiality. This is critical for a cycle. Cycles, by themselves, do not tell the solver when they should be solved. Instead, the solver needs some kind of exit path to take–an edge that only becomes viable after some condition has been 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.

1.4.3. 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.

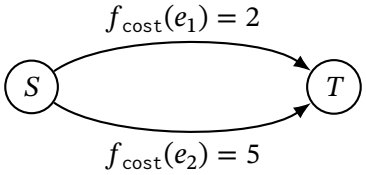
Our motivation for doing so comes from understanding the word *model* as a set of relations describ-

ing 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.

1.5. 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

$$\alpha = -\frac{g}{r}\sin\theta \tag{1.1}$$

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:

$$\omega_i = \omega_{i-1} + \alpha_i \Delta t \tag{1.2}$$

$$\theta_i = \theta_{i-1} + \omega_i \Delta t \tag{1.3}$$

where Δt is the time step $t_i - t_{i-1}$. This system is shown in Figure 1.7, while the CHG is shown in Figure 1.8.

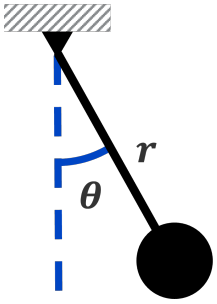


Figure 1.7: Schematic for a simple planar pendulum.

The cycle from θ to α to ω and back to θ is clear from Figure 1.8. 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.

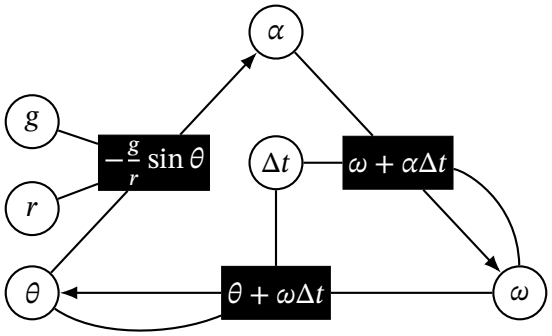


Figure 1.8: CHG of a simple planar pendulum.

1.5.1. ConstraintHg

As described in Section 1.4.1, 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 F. 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 1.1. 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.

Block 1.1: 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()
```

[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.

```
hg.add_edge(
   sources={'theta': 'theta', 'g': 'g', 'r': 'r'},
   target='alpha',
   rel=Rtheta_to_alpha,
   index_offset=1,
)

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

hg.add_edge(
   sources={'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',
    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:

Block 1.2: 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.

1.5.2. 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:

$$\alpha = -\frac{g}{r}\sin\theta - c\omega \tag{1.4}$$

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 1.4. This is shown in Figure 1.9, where we should note the competing edges that both point to α. These edges represent the damping equation (Eq. 1.4) and the initial equation of motion Eq. 1.1.

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 `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 1.4 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

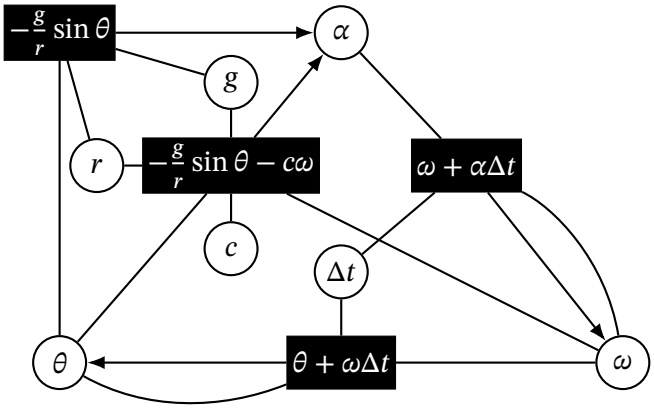


Figure 1.9: CHG of a simple planar pendulum with damping.

c, the smart solver chooses the more accurate model, providing the following output shown in Block 1.3.

Block 1.3: 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 1.4.

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 1.4, passed to the `via` key of `add_edge`.

Block 1.4: 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 1.10.

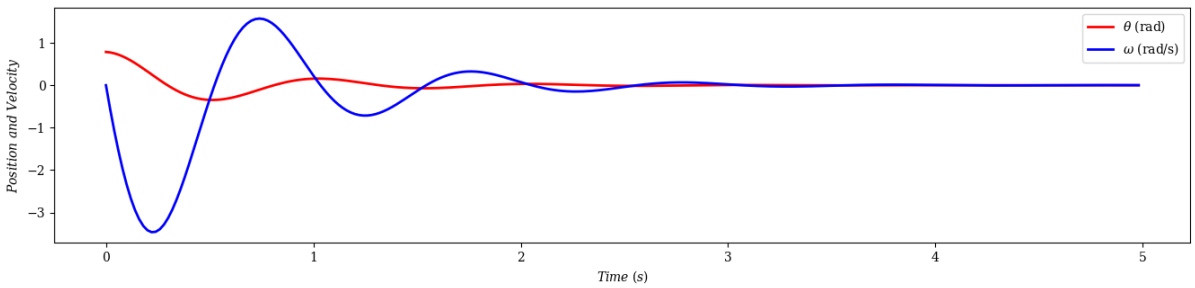


Figure 1.10: Plot of the states simulated in the hypergraph over 150 values.

1.6. 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 engineering 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 1.11. 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?

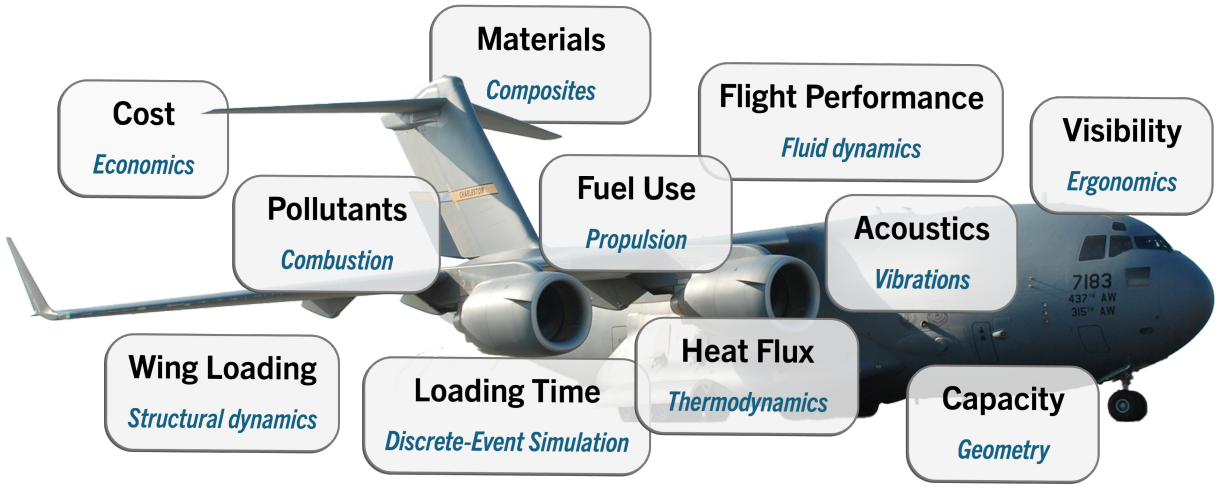


Figure 1.11: Illustration of information and relevant subsystem domain for a complex system. *Image by 1st Lt. Jen Richard [16].*

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.

1.6.1. 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:

$$T = 2\pi\sqrt{\frac{r}{g}} \tag{1.5}$$

Usually, when designing, we want to configure the pendulum to achieve a desired period T^*. To do this we can rearrange Eq. 1.5 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:

$$r^* = \frac{gT^{*2}}{4\pi^2} = \frac{(9.81)(2^2)}{4\pi^2} = 0.993 \text{ m} \tag{1.6}$$

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:

$$r = \frac{\frac{h_t}{2}m_t + \frac{h_b}{2}m_b}{m_t + m_b} \tag{1.7}$$

where the indices *t* and *b* stand for parameters for the *t*ether and *b*ob respectively. This results in the CHG in Figure 1.12.

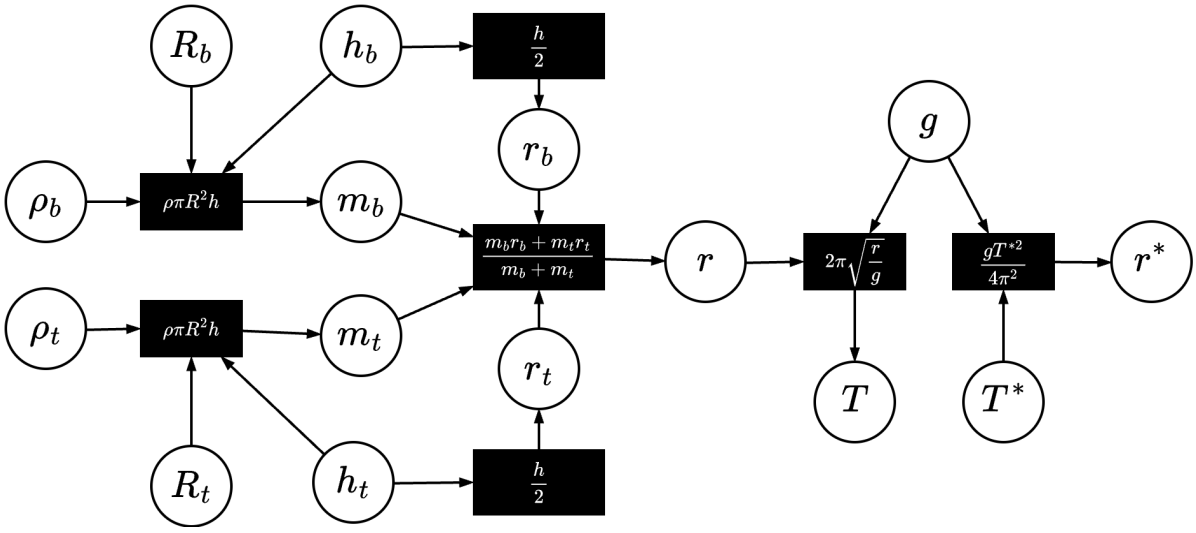


Figure 1.12: A CHG for a pendulum with a cylindrical tether and bob, where all the relationships are algebraic.

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, 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 1.13, where the red "SW" cube indicates a relation that requires

[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

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.

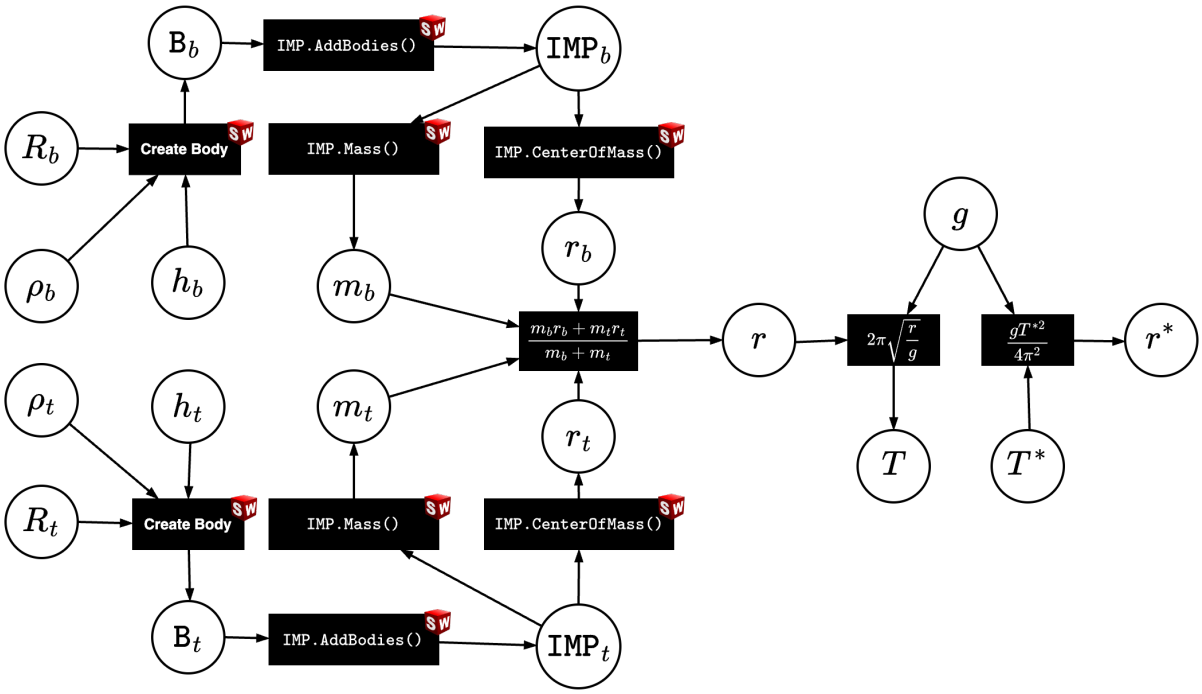


Figure 1.13: A CHG for a pendulum with a cylindrical tether and bob, where some relationships are calculated in SOLIDWORKS.

Figure 1.13 represents the integration of SOLIDWORKS with our dynamic models using updatable digital threads. You'll notice the graph in Figure 1.13 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 1.14) and details how model relations should be executed. One possible workflow for calculating the pendulum's desired length could be:

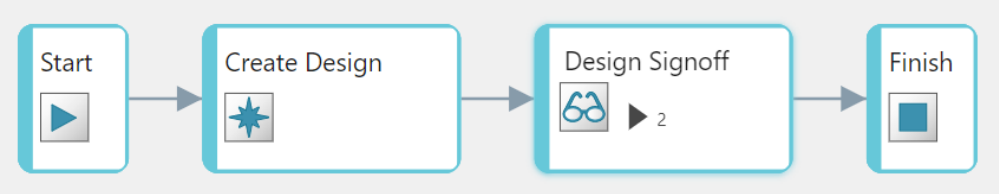


Figure 1.14: An example design workflow in Teamcenter, a product data management application owned by Siemens [17].

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.
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. simulating 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. More information on cross-platform simulation is given in Chapter 4.

1.6.2. 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 [18].

This description of a DT (described more fully in Chapter 5) 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 [19] 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 [18, 20–25], is that DTs provide methods for understanding the physical world. Whether the thing being represented is the world's climate [26], a river [27], a person [28, 29], or a machine [30, 31], 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. This is more fully explored in Chapter 6.

1.7. Conclusion

In this chapter 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 chapter focused on the applications of CHGs to creating foundations for model-based engineering and digital twins.

The rest of this dissertation reviews these major points in far more detail, providing the mathematical and logical premises that make these claims plausible. The reader is encouraged to use the remaining chapters to investigate any points of interest at a deeper level. There are also several in-depth examples of CHGs given in these chapters, including an elevator lift system with a PID controller in Chapter 2.6, a chaotic pendulum in Chapter 3.5, a crankshaft (including FEA and CAD calculated models) in Chapter 4.4, and a DT of a microgrid at the Naval Postgraduate School in Monterey, CA in Chapter 6.6.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

[8] W. Ross Ashby. *An Introduction to Cybernetics*. Internet. London: Chapman & Hall, 1956. http://pcp.vub.ac.be/books/IntroCyb.pdf (visited on 02/27/2025).

[9] 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).

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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).

[15] 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.

[16] 1st Lt. Jen Richard. *American Servicemembers, Helicopters Deploy to Haiti*. Mar. 21, 2010. https://www.af.mil/News/Photos/igphoto/2000379980/ (visited on 10/18/2025).

[17] 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).

[18] 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.

[19] 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.

[20] 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 01/19/2025).

[21] 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.

[22] 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.

[23] 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).

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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).

[30] 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).

[31] 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.

CHAPTER 2

Definition of Constraint Hypergraphs

This chapter is based on the paper "Unified System Modeling and Simulation via Constraint Hypergraphs," which was originally published in a special issue of ASME's Journal of Computing and Information Science in Engineering on networks and graphs for engineering systems and design [1].

Abstract: This paper describes the theory behind constraint hypergraphs: a novel modeling framework that can be used to universally represent and simulate complex systems. Multi-domain system models are traditionally compiled from many diverse frameworks, each based in a single domain. Incompatibilities between these frameworks prevent information from being shared resulting in data silos, duplicate work, and knowledge gaps. A constraint hypergraph addresses these problems by providing a universal modeling framework within which all model prescriptions can be expressed. This methodology expands mathematical structures previously explored in abstract mathematics and systems theory into a new executable framework. Each hypergraph expresses the holistic behavior of a system in a declarative paradigm that describes the relationships between system properties. In addition to modeling, it is shown how constraint hypergraphs can be used for universal, cross-cutting simulation through principles of function composition. The theoretical framework of a constraint hypergraph is demonstrated with a practical representation of a hybrid system, combining a discrete-event simulation and continuous PID controller into a single model of an elevator lift system.

2.1. Introduction

Everything in the universe is a system; in their efforts to understand those systems, scientists and engineers fabricate models that describe their behavior. These models are defined according to prescribed frameworks: the language of algebra, pseudocode, even visual diagrams such as circuit networks or architectural blueprints. Most frameworks are developed for a targeted domain such as geometry, artificial intelligence, or economics. Where these niche representations fall short is in the representation and simulation of systems spanning multiple domains [2], preventing modelers from understanding how a part of one domain, such as bird migrations, affects an object in another subsystem, such as the energy-production of a windmill.

To grapple with these cross-cutting interactions, systems theorists have devised general system modeling frameworks. Most of these, such as SysML or Modelica, employ an object-oriented paradigm [3, 4]. They classify objects in the system, as well as interfaces for sharing information between them [5]. Object-oriented modeling is often preferred because it is easier for humans to read and understand; the cost is that models get broken into isolated subsystems where interactions between domains may not be fully captured. Because of this, object-oriented models can be difficult to integrate, with simulation limited to restricted domains [6].

Another class of model frameworks is declarative. A model following a declarative paradigm establishes the relationships comprising a system without enforcing a preferred order, connecting all system components at the same level of representation [7]. Domain-specific modeling frameworks include bond graphs for dynamic systems, the constraint networks of computer science [8] and the factor graphs applied primarily to coding theory [9, 10]. Because they use functions as their underlying semantic unit, declarative models are far more interoperable than object-oriented frameworks, and consequently better suited for expressing system behavior [11]. This paper describes a new declarative framework for the universal modeling of heterogeneous systems, addressing the yet-unsolved problem of providing both generalized representation and execution of a system model. This framework is termed a constraint hypergraph (CH), borrowing phraseology from the field of constraint theory [8].

A CH is composed of nodes and edges. The nodes represent system properties such as position, speed, color, weight, etc. and are connected to each other via edges. Each edge represents an explicit

constraint that maps a set of nodes to a single value of another node. This can be read as: "given values for A and B, the edge constrains the value of C to be x." Because such constraints are often multidimensional (such as $C = A + B$), an edge constraint may be derived from the values of multiple nodes resulting in hyperedges in a hypergraph. A basic example of a CH consisting of five nodes and two constraints describing a mass-spring system is shown in Figure 2.1. The constraints for this dynamic system could also be represented algebraically as $\ddot{x} = \frac{k}{m}x$, where x, k and m denote displacement, stiffness, and mass respectively. The CH in the figure shows that $\ddot{x}$ can be calculated given known values for the nodes $\frac{k}{m}$ and x and that $\frac{k}{m}$ can be in turn calculated if k and m are known.

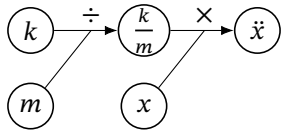


Figure 2.1: An example of a CH for mass-spring system.

CHs reveal how each property in the system affects another. This is not limited to a single system; CHs also describe how systems interact with other systems, and how models of two different systems can be combined into a single whole. The composition of two CHs is the union of the set of nodes in each, joining the hypergraphs along their shared nodes. This is illustrated in Figure 2.2, where the mass-spring system has been extended to include a damper.

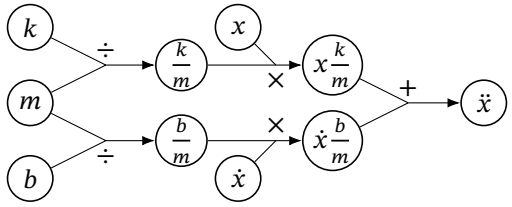


Figure 2.2: An example of a CH for mass-spring-damper system, extending the system in Figure 2.1.

In both of these figures, only one direction of the mapping has been made (showing relationships leading to the constraints on $\ddot{x}$). Relationships in CHs are implicitly causal; acausal constraints can only be included by adding additional inverted edges. This leads to more edges and can crowd the diagrams. This makes for an important point: in contrast to a language like SysML, CHs are not diagrammatic nor intended for visual modeling. They are formulated as a robust, mathematical structure for capturing

the fundamental relationships between system entities, allowing agents (both human and autonomous) to fully understand the behavior of the system and simulate its potential states.

CHs represent a new framework, but one whose structure is not entirely novel. The mathematics and methods of the framework amalgamate results from the fields of category theory, computer science, constraint theory, and logic programming. The contribution of this paper is to both describe how these findings can be combined to create a useful framework, and also demonstrate how the resulting strongly-coupled, multi-domain modeling scheme can be applied to the general representation and simulation of real-world systems. This is accomplished starting by providing a background of previous work on multi-domain system modeling (Section 2.2), followed by a robust definition of a CH. This definition is used to explore how CHs represent (Section 2.3) and simulate (Section 2.4) systems, as well as their limitations (Section 2.5). Finally, a case study using CHs to represent an elevator system is provided in Section 2.6.

2.2. Background

As long as engineers have considered systems, they have struggled to understand how objects of disjoint domains affect each other. From how forging affects the surface of a steel tool, to learning the ways in which music affects emotions, systems are by their nature heterogeneous, and consequently require methods of representing their heterogeneity. The section recounts only a fraction of the work in this field, with a focus on motivating CHs and their use by systems engineers.

2.2.1. System Modeling Frameworks

A system is an arrangement of things that together exhibit behavior that the individual components do not [12]. If systems are constructs of the real world, then their corollary in the virtual domain is a system model, which relates the information known about a reified system. A system model is composed of properties approximating attributes of the real system, which are commonly represented with variables [13]. Each property is allowed to vary over a set of values. The set of possible combinations of each property is known as the state space.

What establishes a system model as representing a system is the provision of a set of relationships between the system properties, such that the possible states for a system are restricted. The restriction

of system states is the definition of behavior given by Willems [6, 14], and is suitable for the purposes of this article. A system model is considered in this paper as a description of a system that provides rules governing the possible values for a set of properties. This general definition describes nearly every modeling framework as a system model. Geometric systems have properties of distance and orientation constrained by dimensions and geometric relationships, while finite element models have properties of nodal locations or thermal flow constrained by partial differential equations. Model frameworks applied to social [15], ecological [16], economic [17], and other domains also fit under this description.

The mathematics for describing system models are provided by category theory (see the introduction by Spivak and Fong [18]), which provides tools for defining the relationships inherent across systems. Efforts in this regard began with Fong in 2016 [19], who specifically introduced decorated cospans for deconstructing circuit networks into hypergraphs. This was built upon by Baez and Pollard, who applied decorated cospans to Petri nets [20], and by Patterson et al. in deconstructing dynamic systems [21, 22]. The categories used in these works are extensions of graphs, and generalize the connections between a set of things [23]. This similarity to systems–also composed of entities connected to each other by constraints–leads to many system modeling frameworks being graph-based in nature. A selection of graph-based frameworks along with their primary domain of use is shown in Table 2.1. The diversity of modeling schemes stems from the varied interpretations of nodes and edges, where each might be assigned domain-specific system elements, relationship types, or composition rules.

2.2.2. Model Interoperability

With such a diversity of modeling types, establishing interoperability becomes a critical challenge in multi-domain systems. When sub-systems are modeled within a domain-specific framework, inter-framework incompatibilities can result in modeling silos: where a holistic system is composed of individual subsystem models which cannot effectively share information. The primary result of sequestered models is the inability to capture relationships between elements in different silos. This is a critical challenge for decisions makers; for instance, a systems engineer may not be able to see how changing a sensor will influence the weight distribution in an aircraft because the sensor interfaces, electric circuits, and payload models are all siloed in incompatible frameworks. Secondary effects of model silos include information loss during data exchanges, duplication (where singular entities are represented multiple

Table 2.1: A non-exhaustive list of graph-based system modeling frameworks

Framework	System Domain	Source
Block diagrams	Dynamic	[24]
Bond graphs	Dynamic	[25, 26]
Linear graphs	Dynamic	[27, 28]
Stock and Flow Diagrams	Dynamic	[29, 30]
Organization charts	Knowledge	
Entity-Relationship Model	Knowledge	[31]
Circuit diagrams	Electronics	[19]
Flow charts	Processes	[32]
Petri nets	Discrete-event	[20]
Markov chains	Discrete-event	[33]
Bayesian networks	Stochastic	[34]
Causal models	Multi-Domain	[35]
Component objects (Paredis)	Multi-Domain	[36]
Composable Objects (COBS)	Multi-Domain	[37]
SysML	Multi-Domain	[38]
Constraint graphs	Multi-Domain	[39–41]
Factor graphs	Multi-Domain	[9, 10]

times throughout a system), and challenges modifying or scaling due to the inter-system complexities not being captured in the global model.

There are two general approaches to providing model interoperability: transformation or unification [42], which relate respectively to the competing concepts of object-oriented and declarative modeling [43]. In the former, subsystems are treated as independent black boxes. Each subsystem has the ability to transform certain signals delivered via a defined interface (a port) [44, 45]. This allows each subsystem to be uniquely defined according to its own domain-specific framework provided there is a standard interface for inter-component communication. The second approach is of system unification, where the components of each individual system are deconstructed and reconciled into a single holistic model. Of the two, transformer frameworks are more common, since it is simpler to define interfaces than fully deconstruct every member of a system.

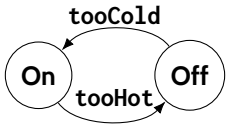
Transformative frameworks, such as block diagrams, stock and flow diagrams, or Entity-Relation models, focus on the objects in a system. Because each class of object is unique, transformative frameworks must define unique methods for sharing information among the system [5]. These systems are often easier for designers to work with, since objects correspond to how a designer deconstructs a sys-

tem [42]. Object-oriented solutions include SyDer [46], component objects [36], the Functional Mockup Interface (FMI) standard [47]–strictly for dynamic system simulations, though standards for hybrid systems are being developed [48]–and SysML, a multi-domain modeling framework developed in 2007 [38] with the objective of supporting systems engineering tasks [49]. These weakly-coupled frameworks may struggle to simulate reactive systems featuring complex relationships between many system components [50].

Strong coupling is provided by unifying models, which do not prescribe objects, but instead deal primarily with the system constraints and the properties they relate. The relationships of a unifying framework are functions, which are common across all modeling domains. Where object-oriented frameworks are nested, the use of functions results in unifying models being flat, with every relationship elevated to the status of "first-class citizen" [7]. This makes combining models simpler since information does not need to be transformed between system components. Friedman was the first to show how functional frameworks could be used to describe universal system behavior, writing about systems of mathematical expressions constraining a set of variables [39, 51]. These constraint models, as labeled by Friedman, were used to expose system complexity by representing the system behaviors, though they were not formulated to provide execution. CHs, the framework introduced in this paper, are an expansion of Friedman's work adapted to universally represent and simulate system behavior.

2.2.3. Motivation

To motivate how system unification is accomplished by CHs, consider the example given by Gomes of a continuous and discrete system [52], where the discrete model is a state machine of a simple thermostat regulating the temperature in a room according to the following state machine:



Here the states "On" and "Off" refer to the status of a heater controlled by the thermostat, and the transitions `tooHot` and `tooCold` are triggered when the temperature in the room exceeds some threshold. The change in the room temperature is modeled by the following continuous equation, where T is the temperature, $r > 0$ is a constant rate of change, $q \in \{0, 1\}$ indicates whether the heater is on or off,

and k is the rate of heating:

$$\dot{T} = -r(T - kq)$$

In their survey on cosimulation [52], Gomes found that the only established methods for coupling the discrete and continuous models into a hybrid system was to interpret subsystems from one domain in terms of the other, a messy translation process prone to information loss. Alternatively, by identifying the system properties and functional constraints, the system can be fully reconciled into the CH shown in Fig. 2.3.

Constraint hypergraphs are not a visual framework, but to illustrate their composition, the following diagrammatic schema is followed in Figure 2.3 and the rest of this paper. In the figure, each circular node represents a variable, and each black box describes the function for an edge. Arrows wire nodes to edges (showing the function domain) and edges to a node (the codomain). Multiple domain arrows indicate a hyperedge, where the domain is the Cartesian product of all linked nodes. When order matters in the arguments, labels may be written on the domain arrows in the form $a, b, \ldots, z$. Finally, nodes that have only a single domain arrow are removed, replaced with wires connecting edges to edges, similar to a short circuit in an electrical diagram.

Building a CH reveals the two variables shared by the systems: q and T, allowing the systems to be connected along the respective nodes. The result is a fully coupled system, such that the relationships between the continuous and discrete models are completely expressed by the CH, exposing the behavior of the holistic system. In considering the hypergraph in Figure 2.3, one notices that the discrete and continuous models are not easily distinguished. This is typical of a declarative modeling paradigm, where each model is fully deconstructed to the functional level. The categorization–and consequent isolation–of subsystems is largely a feature of object-oriented modeling, which treats each subsystem independently. Contrast this with a CH, where each function is considered and connected equally regardless of its subsystem of origin.

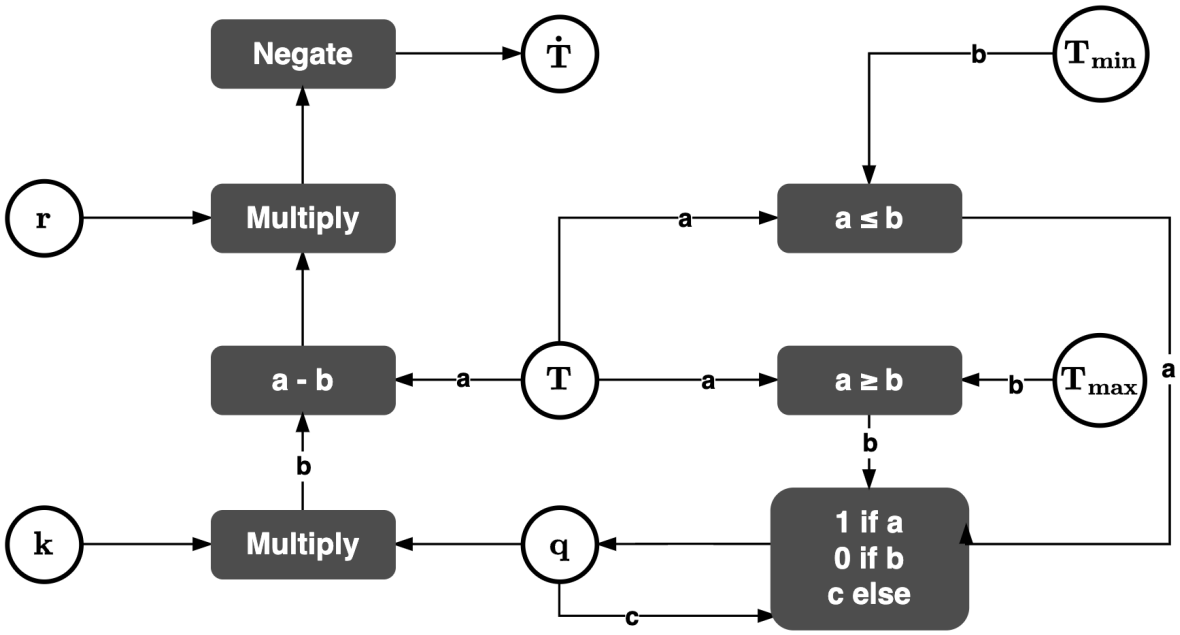


Figure 2.3: CHG for a hybrid system modeling a thermostat controlling the room temperature x.

2.3. Structure of a Constraint Hypergraph

2.3.1. Formulation

The general definition of a CH is given as

Definition 1 *Constraint hypergraph: a graph $\mathcal{H}$ composed of a set V as vertices and a set of functions E as edges, where each vertex $v \in V$ is a set, and each function $e \in E$ maps between the Cartesian product of a subset of V: $V' \subseteq V$ and another element of V: $T \in V$ such that $\prod_{V'} \xrightarrow{e} T$.*

Let $\prod_X$ as used above be given as the Cartesian product of all sets in the superset X such that

$$\prod_X \coloneqq x_1 \times x_2 \times \ldots \times x_n \text{ for } x \in X, n = ||X|| \tag{2.1}$$

The structure of a CH is more explicitly described using the syntax of category theory, although having an understanding of this abstract mathematical field is not necessary to understand their structure. Consequently, the limited discussions involving category theory are supplementary in nature. In that guise, a CH $\mathcal{H}$ is a subcategory of **Set** with objects derived from a set V where each element $v \in V$ is also a set. The objects $\mathrm{Ob}(\mathcal{H})$ are given by the Cartesian product of each element of the power set

of V, or $\mathrm{Ob}(\mathcal{H}) := \{v_1 \times v_2 \times ... \times v_n \; \forall \; v_i \in P \; \forall \; P \in \mathcal{P}(V)\}$. Since $\mathrm{Ob}(\mathcal{H})$ is a subcategory of **Set**, the morphisms of $\mathcal{H}$ are functions mapping $\mathrm{Ob}(\mathcal{H}) \rightarrow \mathrm{Ob}(\mathcal{H})$. Composition is then given by typical function composition, $\circ$, and the identity by the identity function on a set.

Definition 2 *Edge: Given two functions* cod *and* dom *and two objects* a, b *(not necessarily unique) taken from the same set, let an edge* e *be defined such that* $a \in \mathtt{dom}(e)$ *and* $b \in \mathtt{cod}(e)$ *for at least one such pair* (a, b)*.*

The functions cod and dom follow the definitions given by Mac Lane [23], with both mapping from $E \rightarrow \mathrm{Ob}(\mathcal{H})$, where E is a set of edges. In the sequel $\mathtt{dom}(e)$ is sometimes referred to as the *domain* of e, and similarly to $\mathtt{cod}(e)$ as the *codomain* of e. Similarly, the Cartesian product of each element in an edge's domain is referred to as S (the source set), and the codomain as T (the target set).

A hypergraph can be simply described as a collection of vertices connected by edges where the edges can connect any number of vertices. A CH adds to this general definition two important constraints:

1. Each vertex in V is a set; and
2. Each edge in E has only a single vertex in its codomain.

Note that here the definition of an edge is really that of a *directed* edge. This is more general than an undirected edge [53], and is important to the formulation of a CH. As non-directed graphs are not treated in this article, any such reference to a graph or hypergraph should be construed as referring to a *directed* graph and *directed* hypergraph.

The utility of a CH lies primarily in finding sequences between sets of vertices. Traversing a CH is similar to pathfinding in a graph, with the added complexity of multiple dependencies present at each edge whose domain has a cardinality greater than 1. Traversal methods are discussed in greater detail in Section 2.4.

In order for a CH to be traversable, it must be paired with a computational engine capable of identifying the mapping given by each $S \xrightarrow{e} T$. Though such an engine is not required for the CH to be valid, the utility of a CH stems largely from its traversals. As such, the computational engine is a significant part of the CH. This engine may perform lookups similar to a querying agent in a relational database, or it may be a calculator capable of computing some rule that encodes the function mapping.

2.3.2. Characteristics

In addition to the description given in Definition 1, a CH is endued with several characteristics that increase its functionality. These characteristics do not alter the mathematical structure given by the more general definitions. Their inclusion is motivated by the application of CHs.

Definition 3 *Ordered: There should be an injective function* $\mathtt{idx}_e : \mathit{dom}(e) \to \mathbb{N}_{||\mathit{dom}(e)||}$ *given for all edges* $e \in E$ *in a CH, where* $\mathbb{N}_i$ *is the set of natural numbers up to i.*

The purpose of ordering is so each vertex connected to an edge can be uniquely identified along a hyperedge during traversal of the hypergraph. The ordering of elements of S by f enables non-commutative relationships to be codified in an edge. For example, a function rule that does not commute such as $\div(A, B) \to C$ (mapping each element of $A \times B$ with the quotient of its ordered pair in C) requires its operands to be assigned in a specific order, i.e. $\div(A, B) \neq \div(B, A)$.

Borrowing again from the language of Category Theory, an ordered CH is one in which each set $\mathtt{dom}(e)$ is totally ordered, such that for any objects $a, b \in \mathtt{dom}(e)$, $a \leq b$ is given by $\mathtt{idx}_e(a) \leq \mathtt{idx}_e(b)$, with $\leq$ defined as the typical magnitudinal ordering operation for the set of natural numbers. Note that equivalence between a and b indicates a commutative relationship espoused by e between the two objects. Consequently, an edge whose mapping was determined by addition might have $\mathtt{idx}_e(a) = 1\ \forall a \in \mathtt{dom}(e)$. Hence, $\mathtt{idx}_e$ is injective, not bijective.

Definition 4 *Conditional Viability: For each edge* $e \in E$ *in a CH, there should be a function* $\mathtt{via}_e : \prod_S \to \mathbb{B}$*, where* $\mathbb{B} := \{0, 1\}$ *is the set of Booleans, and* $S := \mathit{dom}(e)$*.*

The value mapped by the function $\mathtt{via}_e$ determines whether e is viable. A non-viable edge is one that is known to exist, but for which the relationship is not currently sequenceable. This prevents traversals along a non-viable edge during simulation. A viable edge is equivalent with Definition 2.

As there are many relationships in the real world that exist for only a portion of their expressible dependent factors, such as a door permitting entrance only if it is unlocked, conditional viability has significant impact on the CH's ability to represent realistic relationships. In more practical terms, a conditionally viable edge is one whose traversability can change during sequencing. A secondary, but no less important effect of this is the ability to form sequencable cycles in the CH. In order for a cycle

to form a part of a path, there must be some mechanism by which the processing agent can exit a cycle based on values encountered at run-time. Such a requirement is fulfilled by conditional viability, with the values provided by the elements mapped from $\mathtt{dom}(e)$. A better understanding of the mathematical basis for conditional viability is covered in Section 2.4.

2.3.3. Universality

In addition to their formal definition, the authors aim to show that CHs can be used to represent any real system, either singular or covering multiple domains. Such a general objective can be reduced to showing the following conditions: (a) that the framework can represent any system property; and (b) that the framework can represent any interaction between the system properties.

A system property represents an identifiable phenomenon of an entity. Consider two assumptions: that every phenomenon can be represented by a set of values, and only a single one of these values may be manifest for any distinct frame of consideration. The second assumption stems from considering the system to be deterministic: where only one possible outcome may result for any set of initial inputs [13]. A system's evolution is characterized as the manifestation of different values across unique frames of consideration. The behavior of a system is the set of restrictions on the possible values a system property can manifest for a given configuration of other system properties, following the definition given previously by Willems [6]. The goal of system representation is to express these system properties and inter-property restrictions.

A CH is composed solely of these elements. By refraining from prescribing the types of properties that can be represented, a CH is able to represent any property that can be described by a set of values. Due to the finitude of information, this should include all phenomena that can be considered, satisfying the first condition of system representation.

The second condition, considering system interactions, is motivated by the notion that a system property may manifest only a single value when considered. Consequently, any interaction between system properties must be given as the prescription of a unique value. A function is the correct mathematical construct for representing these interactions. By mapping the values of a system property to a single, distinct value of another, a function enforces the causality assumption implicit in system modeling. A CH encodes all possible system interactions by using functions to represent not just relations

between a pair of properties, but also combinations of multiple properties. Consequently, any function that can be mapped between the properties of a system can be expressed by a CH, confirming the second condition.

This can be demonstrated by the example of the hybrid radiator system in Figure 2.3. In that figure, there are seven properties of the system relating to temperature, settings, power flow, etc., each one represented by a node in the hypergraph. The set of every possible state for the system is given by all combinations of these properties, or $\prod_V$ where V is the set of properties (following the definition of $\prod_X$ in Equation 2.1). The behavior of the system is the restriction of these combinations; for example, q can not be 1 if T is greater than T_{max}. These restrictions come as a direct result of the interaction of system components. As the components in a system evolve, they affect other parts, constraining the associated properties.

To represent a system, a modeling framework must capture every constraint imposed by the association of the system's parts. It can be difficult to do this in a procedural (object-oriented) framework: trying to describe how a controller changes the continuous temperature of a room with a block diagram is possible, but not without converting the discrete controller output into a continuous signal. Contrast this to a CH, where the constraints between system properties are the natural language of the framework. The result is that a CH can not only represent any kind of real, deterministic system, but also any behavior espoused by the system.

2.4. System Simulation

An important purpose of representing a system is the ability to perform simulations, with simulation defined here as the measurement of a system property made without having first observed that property in reality. Under this definition, nearly every decision made by a modeler requires simulation. Before a decision is made, the state of a system is unset and cannot be observed since it does not yet exist. An architect deciding on the location of a building cannot observe that location in reality as the building has not been built. Rather, the architect simulates the system to identify the optimal setting. Consequently, any system representation that does not afford simulation is nearly useless to a decision maker. One of the principle benefits of using a CH is the ability to simulate a system generally, exposing the possible ways to simulate a property anywhere in the system as long as it has been constrained by a constraint.

Simulation of a CH is focused not on translating visual diagrams into executable scripts, but on creating a mathematical structure that aligns execution with the underlying system behavior. The purpose of simulating a constraint hypergraph is to constrain a system such that system data becomes evident. If data is generated from constraints in a model, then it is artificial, and it is called simulated data. The data that are known prior to the simulation are referred to as inputs, and the data that need to be generated are referred to as outputs. This terminology should not be confused with expressing a CH in terms of inputs and outputs. A CH model is a set of constraints and relationships. The concepts of inputs and outputs apply only during a simulation when it becomes necessary to process relationships in a certain order. The directed edges in a CH describe which direction such sequencing may occur. Figure 2.4 shows an example of this: the model contains multiple different dependencies from different nodes, such that it's impossible to say which node depends on which. The exception to this is during a simulation, when a single node is set to be the input, and the graph describes which nodes can become possible outputs. The act of processing a CH from one node to another is referred to here as sequencing, and is synonymous with simulation.

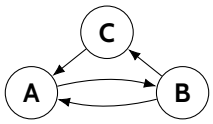


Figure 2.4: A cyclic graph describing how causality can only be assigned within the context of a simulation.

The expressiveness of a CH is founded in two principles: composition and determinism. Composition is enforced in a CH by representing relationships as function. In order to be a valid mapping, a function must show how each element in one set (the domain) is associated with an element in another set (the codomain) [54]. This guarantees that if a value of each source node is known, then the target node for the edge can always be calculated by the encoded function. The solved target node can then be utilized for simulation of additional nodes. Each step in the simulation composes with the previous, allowing an agent to readily form sequences of simulation steps throughout the graph.

CHs are also weakly deterministic, in that given a single input, each linked node is constrained to manifest only a single value. The two properties of composition and determinism are the foundation to forming simulatable sequences in a CH. As shown in Figure 2.5, any sequence starting in the leftmost

set is guaranteed to be able to arrive at the rightmost (composition), and given an element in either of the two leftmost sets, there is only one element in a set to the right corresponding to that value (determinism).

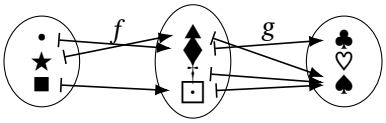


Figure 2.5: Two functions mapping between sets demonstrating composition and determinism.

The authors label CHs as only being weakly deterministic because the same set of inputs is not required constantly map to the same set of outputs [13]. Whether the simulation of a CH will be deterministic or not depends on how the functional relationships are defined. A deterministic function is one where every value in its domain is always mapped to the same value in its codomain. Though this is typically the case, the only mathematical requirement of the function is that it maps the values in its domain to a unique value in the codomain. If the mappings between inputs and outputs specified by a function change for repeated calls, then any simulation along the corresponding edge will be non-deterministic. This allows for stochastic modeling, including Monte Carlo simulations. In practice, deterministic models are simple to obtain by requiring all functions to provide a constant mapping.

2.4.1. Sequences

In a CH, the fundamental construct of a simulation is a path. A path (or a chain) in a graph is some sequence of distinct vertices connected in a sequence of non-repeating edges [53]. The corollary for a hypergraph is a hyperpath, which connects a given set of vertices (the source) to another single vertex (the target) by a chain of hyperedges. In order for a hyperpath to be traversable, a value must be known or generated for every node in the path. Each edge describes a rule whose execution generates a new value, thereby advancing the simulating agent along the path.

The actual execution of this rule–e.g. performing a table lookup, or executing a sequence of mathematical operations–is performed by a simulation engine paired with each function. Part of the process of preparing a constraint hypergraph is connecting each function rule to a simulation engine that can process it. During simulation, the engine is passed the value of the source set along with the rule to be calculated. It then returns the output value, which is assigned to the target node.

Structurally, hyperpaths are constructed as trees. In a hyperpath tree, branching occurs along every edge whose domain has a cardinality greater than one. By making the output node the root, and placing the source nodes as the leaves, the tree shows all the functions and nodes which must be traversed to complete the simulation. A hyperpath (as a tree) is shown in Figure 2.6 for a hypergraph with two source nodes S_1, and S_2 and a target node T.

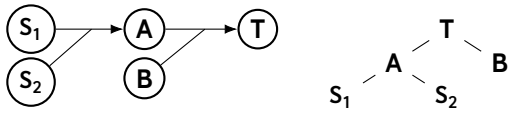


Figure 2.6: A hypergraph with a hyperpath from $\{S_1, S_2\}$ to T represented as a tree.

2.4.2. Pathfinding

In addition to sequencing a hyperpath, a means for determining which hyperpath to sequence must be provided. The philosophy of a CH is that the model contains all relevant relationships between elements of the system, which can create multiple ways to travel in between the same nodes. It is a nontrivial task to discover a valid path between nodes in a hypergraph, and even more complex to find an optimal one. To do this, the modeler needs to quantify the optimality of each edge, generally by assigning weights. Ausiello et al. described ways of traversing directed hypergraphs using a modification of Djikstra's algorithm [55, 56] to identify the minimum-cost path between all nodes in the hypergraph. Though this works well for simple hypergraphs, CHs are not simple due to their inclusion of two features: cycles and conditional edges. Ausiello's work must be adapted to handle these conditions. Since this adaptation is not straight forward, it is worthwhile to motivate the provision of these graph features in a CH.

Conditional Viability of Edges

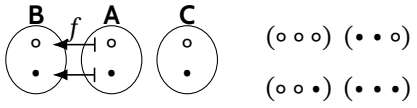
Consider the following example for a system of three switches with states on and off represented respectively as ∘ and •. There are eight possible states of the system as construed.

(∘ ∘ ∘) (∘ • ∘) (• ∘ ∘) (• • ∘)

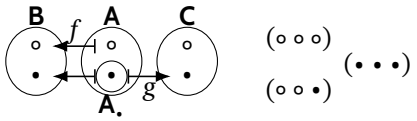
(∘ ∘ •) (∘ • •) (• ∘ •) (• • •)

The most intuitive way to represent these state values is by associating a variable with each switch, labeled A, B, C. Each variable will be represented by a node in a CH, with $A, B, C \coloneqq \{\circ, \bullet\}$.

Providing a constraint decreases the degrees of freedom for the system. For example, a function f constraining A and B to be equivalent reduces the degrees of freedom from three to two, and the number of possible system states from eight to four, as shown below.



Because the way system states are categorized is arbitrary, oftentimes a system may exhibit behavior that does not perfectly correlate to the prescribed variables. For example, if switch C is always off if switch A is off, then the system behavior cannot be correctly expressed by a function mapping $A \to C$, as C is not constrained by the $\circ$ state of A. Instead, this behavior should be represented by a function $g : A_{\bullet} \to C$, where $A_{\bullet}$ only represents the $\bullet$ state of A. Accordingly, $A_{\bullet}$ is a subset of A, as shown below.



Including $A_{\bullet}$ changes the paradigm of the CH by introducing nested nodes. The main value of nesting is that the intuitive sets of A, B and C can be maintained while also describing relationships between individual members of A. Duplication of entities is also avoided since every function that maps from A implicitly maps from $A_{\bullet}$ also, without requiring f to be redefined. Nested relationships represent constraints that only hold for a limited subset of the domain's values. This is such a common occurrence in system modeling that conditional edges were identified by Peak et al. as a fundamental requisite for system simulation [37]. An example is Hooke's law relating the force of a spring to its deflection, which is valid only if the deflection is in the spring's linear regime.

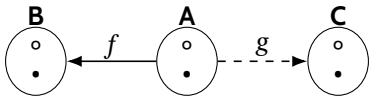
The complexity of conditional edges comes as a result of violating composition. Let X' be a subset of a set X. Any function f that points from X also points from every value in X'. However, the inverse of this is not true: if X' is the domain of a function g, then a simulation sequence arriving at X is not guaranteed to be ably to traverse g unless the assigned value for X happens to also be in X'. The result of this is

that the simulation path becomes conditional on the specific value solved for X at runtime. This greatly increases the computational expense of simulation, as a path through the CH must be rediscovered for each unique set of inputs, in contrast with the general paths found by Djikstra's algorithm that are optimal for any node values.

In order to enable sequencing, the use of a conditional viability function $\texttt{via}_e$ for any edge e was provided in Definition 4. This function is used to determine whether a value x is part of the domain of e, such that:

$$\texttt{via}_e(x) = 1 \iff x \in X \wedge X \in \texttt{dom}(e) \tag{2.2}$$

An edge is described as being conditionally viable if $\texttt{via}_e$ is not 1 for all values of $\texttt{dom}(e)$. The set of values for which $\texttt{via}_e$ is 1 is referred to as the *viable set of e*. Conditional viability can be used for path switching, where the node a simulation sequence next reaches changes based on the values of the latest inputs. In this case, there are two or more edges with disjoint viable sets, so that only one edge is valid for any given value. In this paper, conditionally viable edges are typically shown as dashed, as in the figure below (which expresses the switching example described previously).



Cycles

One of the most vexing issues of a graph is dealing with cycles: a path that begins and ends on the same node [57]. In a CH, a cycle in a simulation sequence would seemingly indicate that the value of a node is in someway constrained by itself, a violation of causality not found in the real world.

Despite this, cycles are still introduced as a matter of convenience to the modeler in performing simulations. This is due to many system variables being iterations of previous system states, especially as a system evolves through time. In a procedural model, such as those espoused by Wymore, a system has several independent states which repeat at each instance of time [58]. A CH, in contrast, has no sense of state. Each instance of a system variable is considered unique and independent, and must have some sequence of hyperedges connecting it to the input values to be successfully solved for. For

instance, a hypergraph representing a body moving at a constant velocity along a single dimension is shown at the top of Fig. 2.7, where the position at each time step i is given by x_i.

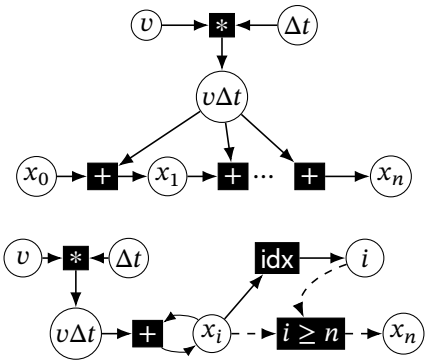


Figure 2.7: A hypergraph representing the position (x_i) of a moving body at multiple points in time as an extended path (top) and as a cycle (bottom).

Since a modeler may wish to simulate x at an arbitrary time step, this hypergraph should arguably be extended to infinity. This impossible expression can be enabled through the use of cycles, as shown at the bottom of Figure 2.7. In the cyclical hypergraph, the addition of the index i as well as a conditional edge $i \geq n$ provides a mechanism for exiting the cycle after n iterations. Note the essential functionality of the conditional edge, without which a pathfinder would not be able to identify a valid path that exits x_i for only a subset of the node's values.

Pathfinding Process

Having motivated the inclusion of conditional edges and cycles, the methods of pathfinding can now be properly described. Pathfinding is the act of tracing a hyperpath from a set of source nodes S to a target node T in the CH. Assuming that conditionally viable edges are present in the hypergraph, pathfinding must be performed for each unique set of input values. Consequently, the first step in the pathfinding sequence is to pare down the hypergraph to a subhypergraph consisting only of the nodes and edges that are possible candidates for a path. Given S and T, a solver can perform a basic search for a subhypergraph $\mathcal{H}'$, where each node in $\mathcal{H}'$ is reachable from at least one $s \in S$ and from which T is likewise reachable. A node b is reachable from (or connected to) another node a if there is a hyperpath from a to b [57].

Once $\mathcal{H}'$ has been prepared, a solver should be run using standard algorithms for pathfinding (such

as A*). At each node, the solver calculates a step along a hyperpath through $\mathcal{H}'$. In doing so, the solver creates a hyper-dimensional search tree. Each node in the search tree corresponds to a viable edge, as depicted in Fig. 2.6. Additional hyper dimensions indicate parallel paths established whenever a conditional edge is encountered. This suggests that the optimal path may run through the conditional edge, but because that path cannot be guaranteed to exist during runtime, other paths must also be explored. As the solver searches $\mathcal{H}'$, these parallel path are pruned from the search tree if either of the following conditions are met:

1. There are no unprocessed edges remaining from the root node (and the root node is not T).
2. The total cost of the branch is less optimal than another branch comprised of simple edges.

Pruning in such a way is a description of backtracking, the most common method of solving a constraint problem [40]. The result of such exploration is a tree of all possible candidates for an optimal path from S to T. If the lowest cost path is made up of simple edges then the tree will be a normal search tree optimal for any input values. Otherwise, during simulation, the solver will proceed through the tree. At any branches onto parallel paths, the solver should step down the lowest-cost branch until the solver either reaches T or the path becomes non-viable. In the later case, the solver resets to the latest solved node and traverses the next-most optimal path.

Although repeated pathfinding can become computationally expensive, it provides the unexpected benefit of gracefully handling discontinuities. Rather than aborting the simulation, solvers encountering a discontinuity (such as an ill-formed mapping or missing value) are able to switch to the next viable path as long as one is available.

2.4.3. Weighting Schemes and Model Selection

A heretofore undiscussed aspect of CHs is the application of weights to their edges. While weights are a common practice in graph theory, it will be useful to consider their interpretation with respect to modeling systems. Weights are primarily used to depict a cost of traversing an edge; in the classic traveling salesman problem, weights depict distances between cities on an imaginary map. Search algorithms require weights to discriminate between parallel paths, preferring the path with the lowest summed weight. In the case of a CH, the cost of traversing an edge is the cost of executing a constraint on the system model. This cost could be interpreted a number of ways, such as the computational cost

of calculation, the distance to the desired target node, the cost of excluding some alternate constraint, or even the uncertainty associated with the modeling constraint.

The inclusion of weights in a CH specifies that some edges in the model should be considered of greater precedence than another. Precedence only matters in instances where there are a plurality of edges offering competing routes for reaching a node. The act of identifying an optimal model among many options is typically described as model selection. Edge weights enable automatic model selection by providing a quantitative parameter for comparing edges.

A practical example is selecting a path that minimizes a simulation's uncertainty. The utility of interpreting edge weights as modeling uncertainty is not in the quantification; assigning uncertainty is still as arduous in a CH as any other framework. Instead, CHs provide an advantage in composing uncertainty for different simulations. A CH breaks down every relationship in a system model into a single, traceable function. For a systems engineer, each of these functions can be thought of as assumption. By listing each edge in a path, a modeler can systematically consider each assumption made in a simulation. Because each function is independent of all others, uncertainty can be assigned without having to consider side effects or duplicated calculations. And because the total uncertainty of any arbitrary path can be trivially calculated, a pathfinding algorithm can search for a simulation path that minimizes uncertainty.

When the edge weights are interpreted as computational costs, then the weighting scheme can be used to determine the least expensive simulation to compute. Weighting schemes are not exclusive either, they can be combined to compare simulation paths by different metrics. The resulting paths can then be optimized via a multi-optimization method to minimize multiple objectives.

2.5. Limitations

Though constraint hypergraphs have been used to great effect, they by no means represent a silver bullet to modeling challenges. It has already been mentioned that flat, complex system models can be visually overwhelming. Object-oriented frameworks, which provide methods for abstraction and encapsulation, are often better for decomposing a system for a human modeler. It is for this reason that this article has not focused on establishing hypergraph diagrams. This section aims to set forth some of their other known limitations.

Related to the challenges of visualization, CHs are not optimal for model development. Because of their generality, any relationship can be formulated in a CH, including impossible relationships. The specificity of domain-specific frameworks discourage modelers from creating invalid systems, such as connecting gravity to a battery terminal. There is no such restriction imposed by CHs. From this the conclusion is drawn that CHs are used most effectively to unify system models that have already been developed in more specialized frameworks.

Perhaps the greatest limitation of CHs is that of syntactic interoperability. Although the semantics of a system are perfectly captured by a CH, this is dependent upon syntactic agreement between models. For instance, it may be impossible to say whether the label of ”Speed” for a node in one model refers to the same data as the ”Velocity” label of another. Though the use of ontologies can help prevent naming conflicts, the lack of tools for reconciling nodal identities greatly inconveniences the adoption of CHs. The authors are intrigued by the possibility of providing syntactic interoperability through reasoners based on graph similarity metrics.

Another major barrier to convenient adoption is the heavy processing time of constraint hypergraphs. Because optimal simulation paths cannot be autonomously determined a priori, path searching often must occur during each simulation run. Though constraint programming has developed great tools and methods for searching, this processing overhead can limit CHs use in real-time environments. This is especially true considering that solving a constraint problem is NP-hard [59], and CH networks can become incredibly complex. There are some instances that paths can be pre-determined, though in such cases simulation success often cannot be guaranteed unless the hypergraph contains no conditionally viable edges.

Finally, constraint hypergraphs are very good at exposing system behavior; they are less good at hiding it. There are many instances in which a modeler may which to obscure the sensitive behavior of a subsystem in a shared model, such as with proprietary technology. While it is certainly possible to form black boxes in a CH (simply by aggregating nodes into a single node and reconnecting graph edges), such actions prevent subsystem properties from being connected with other system elements. This greatly reduces the efficacy of a CH. Encapsulation and abstraction are both primary features of object-oriented systems, but they reduce the expressiveness of the more functional CHs. If such features are needed, interface-based frameworks may need to be used; privacy-motivated black boxing is one of

the many use cases of FMIs [60, 61].

2.6. Case Study

A CH for an elevator lift system is provided to partially validate the constraint hypergraph structure. The unified model is an aggregation of a discrete-event simulation (DES), a dynamic Newtonian model, and continuous state space model (built around a PID controller). Descriptions for each node are provided in Appendix B.2. At the risk of repetition, the reader is reminded that CHs are not generally a good choice for visualizing system models due to their high complexity and abundance of lines. However, for the purpose of communicating their use, a diagram for the CH has been provided in Figure 2.8, with the model scoped to be as simple as possible without sacrificing functionality. The diagrammatic scheme follows that of Figure 2.3, with the added stylizations of zigzag hashing for nodes that are shown multiple times in the figure (for clarity) and double arrows ($\twoheadrightarrow$) for edges that increase the iteration of a node in a cycle. There should be one such edge for every cycle in the graph; Figure 2.8 has three cycles with corresponding iterative edges. The weight of every edge is set to one, resulting in the solver preferring the path with the minimum number of steps during simulation.

There are three primary subsystems represented in the CH: the dynamic system, the PID controller, and the DES of the passenger actions. The properties and edges for these subsystems are roughly gathered in the right, top left, and bottom left of the figure respectively. These classifiers for the various subsystems are only useful for a human modeler as the CH focuses solely on the holistic system. Consequently, it is not clear in Figure 2.8 where one subsystem ends and another begins–at least not as clear as it would be in an object-oriented framework. The mock elevator consists of a carriage moving between three floors referenced by integers 0, 1, and 2. The forces acting on the carriage include the empty weight as well as the summed weight of each passenger (with the assumption that each passenger weighs the same). The driving force of the elevator is given by a PID controller using a first-order Euler integrator with a fixed step-size.

The DES subsystem is formalized for four passengers referred to as `A` through `D`. There are three properties associated with a passenger: the floor they start on (`startX`), their desired destination floor (`goalX`), and whether they are on or off the elevator carriage (`onX`). Recall that the CH does not include a ”passenger” object, only these properties. Also note that it is required to explicitly model the prop-

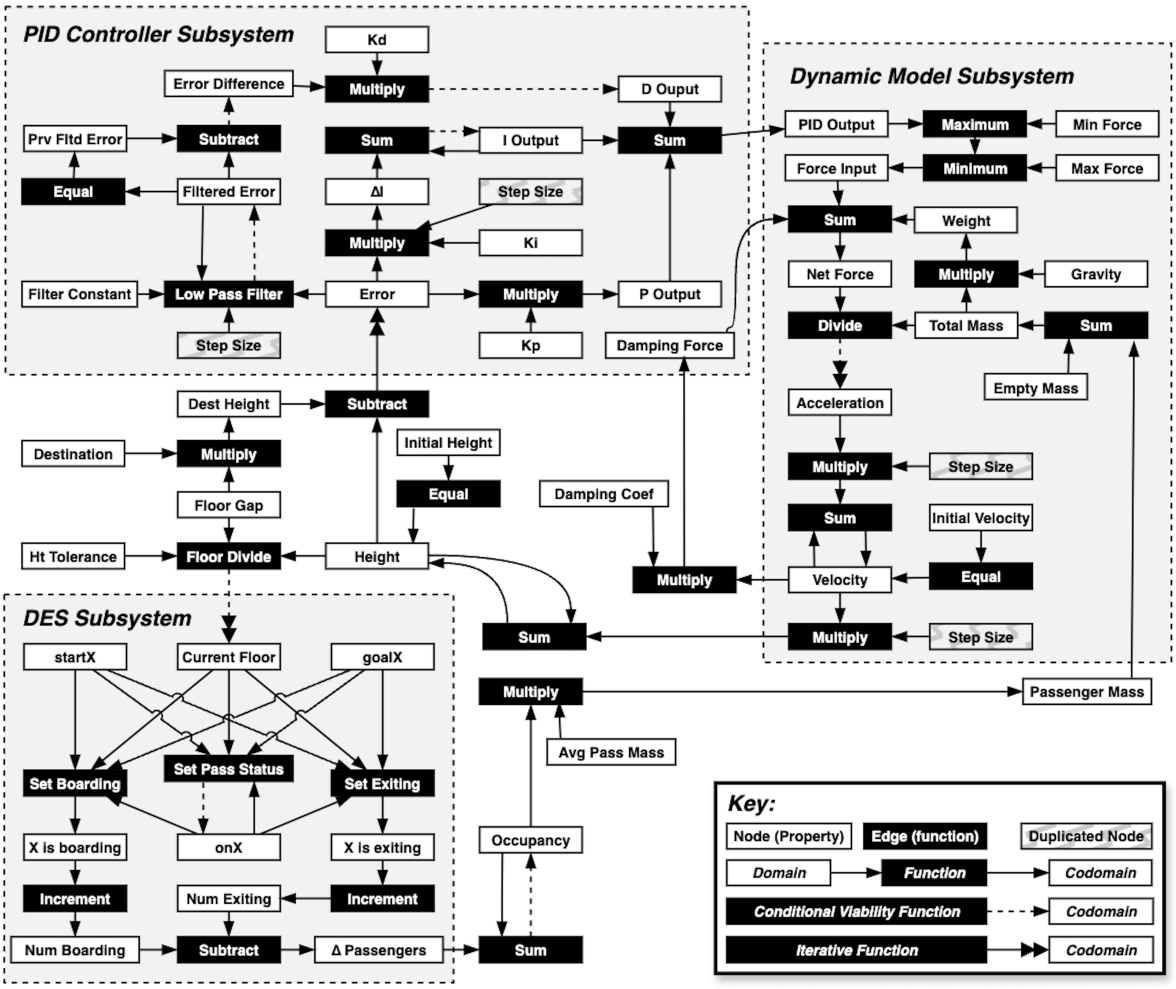


Figure 2.8: Constraint hypergraph model for a hybrid elevator lift system, integrating models marked by shaded regions.

erties for each passenger, as each new passenger extends the possible states (or degrees of freedom) of a system. When executed, each node should be duplicated as a new passenger is simulated. To avoid complicated, redundant figures (beyond the graph already depicted), the nodes for only a single passenger are expressed, contained by the dotted box in the lower left corner of Figure 2.8. As shown by that sequence, an ordered tuple `(onX, startX, goalX)` is made for each passenger. The `Set Boarding` function counts the tuples for which `onX` is `False` and `Start` is `Current Floor`. In this case there are no passengers meeting the criteria, so `Num Boarding` is zero. `Set Exiting` does the same except the conditions are `onX` being `True` and `Goal` equaling `Current Floor`. `Occupancy` is then derived as the previous value of `Occupancy` minus `Num Exiting` and added to `Num Boarding`. The `Set Pass Status` edge

Table 2.2: Initial values for properties associated with elevator passengers.

Person ID	Start Floor	Goal Floor	On Elevator?
A	1	2	False
B	1	2	False
C	0	2	True
D	2	0	False

shown in Figure 2.8 is given by the following conditional function:

$$\begin{gathered}\text{False if } \texttt{Current Floor} = \texttt{StartX} \\ \text{True if } \texttt{Current Floor} = \texttt{GoalX} \\ \texttt{onX} \text{ else}\end{gathered} \tag{2.3}$$

The number of properties comprising the system state is given by the number of nodes in the CH, which in this case is 45. There are several nodes for which constraints are not provided, these must be treated as inputs for a valid simulation, such as physical constants like gravitational acceleration. Initial properties for each passenger have also been given in Table 2.2. In the example the elevator initially starts on floor 0 and moves incrementally to floor 2 (only ascending for simplicity).

Having visualized the CH, the next goal of this case study is to demonstrate a full system simulation. Any node in the hypergraph can be chosen to be simulated, but selecting `Occupancy` allows a more interesting example. The goal of a simulation is to predict the value of `Occupancy` at every landing to which the elevator arrives. In practice, this simulation would be conducted using a computational tool, however, this study is a demonstration of theory. The enactment of this simulation is consequently performed by a theoretical agent capable of processing a CH, which will be referred to as CH AGENT. The inputs to CH AGENT are the graph in Figure 2.8, as well as a list of input values to several nodes, as tabulated in Appendix B.2.

Assuming the inputs have been seeded correctly, a pathfinding algorithm would attempt to find a valid path from a subset of the input nodes to `Occupancy`. Pathfinding is conducted with some searching strategy such as a depth-first or breadth-first search, with the search starting from nodes with known values (the input set). If all nodes in the source set of an edge are known, then the edge can be traversed. Traversing an edge results in CH AGENT solving for the value of the target node by performing the

function calculation represented by the edge. This new node is then added to the list of known values. The process repeats until CH AGENT is able to solve for the goal node, at which point the simulation terminates.

Because Occupancy is given as a known node, the initial simulation is trivial. However, Occupancy is included inside a cycle, meaning that additional iterations of the node can be solved for if all other values in the cycle are found. The cycle for Occupancy involves a hyperedge requiring the number of passengers boarding and exiting the carriage to be calculated, consequently, CH AGENT must solve for these nodes before it can find the next value of Occupancy, written here as $\texttt{Occupancy}_2$. The process for doing so is given by tracing the starting position of each passenger to determine whether they are on the carriage, boarding, or exiting while the carriage is at its initial floor, according to Equation 2.3. The result is a chain of equations that ultimately transforms X Start and X Goal into $\texttt{Occupancy}_2$, which when solved returns a value of two.

Solving for $\texttt{Occupancy}_3$ requires a much longer process. The conditional edge relating ΔPassengers to Occupancy requires that the iteration of the source of Occupancy be only one iteration greater than the iteration levels of Num Boarding and Num Exiting, but the path found by CH AGENT includes only $\texttt{Num Boarding}_1$ and $\texttt{Num Exiting}_1$. Because no iterative edge (indicated by ↠) was encountered in the simulation path found previously by CH AGENT, the path cannot be reused to solve for the node's next value. Tracing the cycle back in Figure 2.8, the iterative edge is a hyperedge relating $\texttt{Floor Gap}_1$, $\texttt{Ht Tolerance}_1$, and $\texttt{Height}_1$ to $\texttt{Elev Current Floor}_2$. In order to increment the state of each node in the DES simulation just traversed, CH AGENT must find a path that connects through this edge, so that each source node will also be in their second iteration upon solving for $\texttt{Occupancy}_3$.

To necessary path takes CH AGENT through the entirety of the hypergraph, starting from $\texttt{Occupancy}_2$ and ending on $\texttt{Height}_2$. After the iterative edge is traversed, the original path is retraced but at a higher iteration level, resulting in solving $\texttt{Occupancy}_3$. In this case, CH AGENT traverses 42 edges to reveal that the value of $\texttt{Occupancy}_3$ is unchanged from the previous iteration. Along the way, values are found for $\texttt{Height}_2$, $\texttt{PID Output}_1$, and other nodes that are necessary for the general simulation.

By structuring the hybrid elevator system as a constraint hypergraph, CH AGENT demonstrates the universality of simulation, moving easily between the various subsystems without concern for ports and type specifications. It should be emphasized that this is a theoretical demonstration; computational

complexity, run time, and other computing metrics are dependent upon the practical instantiation of CH AGENT and the specific search strategies employed by the encoded algorithms. For validation, a Python-based implementation of CH AGENT employing a breadth-first search strategy [62] was run to calculate the `Height` of the elevator over 100 iterations, returning the results shown in Figure 2.9. As a true instantiation of CH AGENT, no additional programming is employed apart from generic plotting software, conducting the simulation without any for or while loops, go-to statements, or conditional logic–all behavior of the system is encapsulated in the CH and executed by CH AGENT. While these results do not validate the models, which are overly simplified for demonstration purposes, they do indicate the ability of CH AGENT to integrate complex systems without relying on manually ported connections.

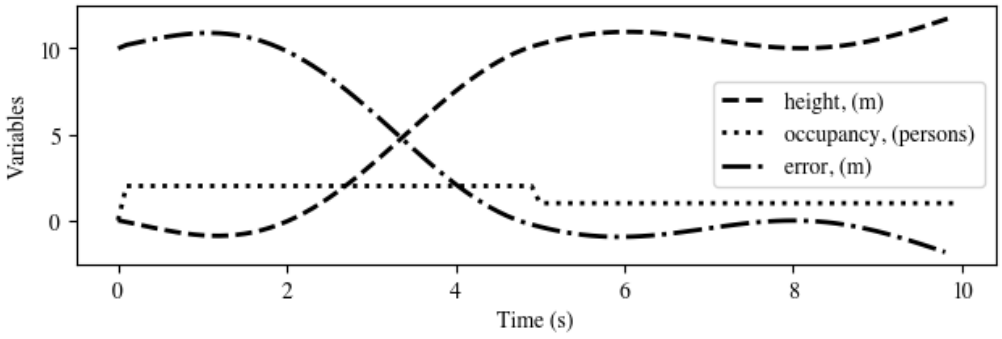


Figure 2.9: Results of a simulation for three variables covering 100 cycles.

2.7. Future Work

A robust, unified system model has many possible applications, some of which the authors hope to explore in future work; these include decision modeling and digital twins. Decision modeling applications stem from the fact that the decisions made by engineers and other agents often must be made before relevant information about a system is known; this is especially true early in the design stage. Constraint hypergraphs make it easier for the information that is known about a system to be understood and analyzed, but the process of quantifying uncertainty and validating simulation predictions remains a topic for further investigation.

Digital twins, defined as the virtual representation of some real system, are a confluence of data streams and models. Their popularity has driven use cases in a variety of systems, such as global environments [63] and national infrastructure [64, 65]. This rich variety of system domains requires the combination of disparate models as well as the processing of heterogeneous data streams. As digital twins grow in fidelity these multifarious facets only increase. Constraint hypergraphs may promote the interoperability of digital twins [66] on both of these fronts: by unifying siloed system models and also by enabling sequencing of data across the holistic system. The author's envision a constraint hypergraph expanding the role currently played by knowledge graphs from information banks [67] to system orchestrators: capturing and distributing information connected to system behavior with a multiplicity of connected agents.

2.8. Conclusion

The principle factors that contribute to a simulatable system model are composability and determinism. These two principles are embedded uniquely in various system models, but are robustly defined in constraint hypergraphs. Constraint hypergraphs, in their declarative description of property relationships, embody the structure of a system. The claim that CHs are suitable for general inter-system reconciliation was supported by showing that CHs support composability and determinism through functional composition, and further describing how every system behavior can be both represented and sequenced within the CH structure.

In the attempt to make this framework more practical, the mathematical structures have been related in the language of systems theory, showing how CHs communicate information and how the relevant notions of objects, states, and behavior are described. CHs are not only useful for semantic representation, they are also simulatable, and can be used to derive unknown system properties or solution spaces. It was shown that this depended on both the provision of functions and the representation of nodes as subsets of other nodes. A mechanism was then built out for building executable sequences and even cycles using edges with conditional viability.

In addition to establishing what CHs are, it was established how they can be used, including how weightings can represent uncertainty or computational costs, as well as applications to autonomous decision-making. These use cases were contrasted with some known limitations of CHs. Finally, these

principles were demonstrated with an example of an elevator lifting system, with multiple system domains unified by a single CH. The authors hope to build upon this work through additional refinement of the CH framework, as well as the provision of integrable tools that can assist with CH modeling in established engineering applications.

References

[1] 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.

[2] Kenneth Boulding. "General Systems Theory: The Skeleton of Science". *Management Science* 2.3 (Apr. 1956), pp. 197–208. http://www.panarchy.org/boulding/systems.1956.html (visited on 01/08/2025).

[3] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A Practical Guide to SysML: The Systems Modeling Language*. 3rd ed. Waltham: Elsevier, 2015. ISBN: 978-0-12-800202-5.

[4] Modelica Association. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling*. Version 1.4. Dec. 15, 2000. https://modelica.org/documents/ModelicaTutorial14.pdf (visited on 07/10/2024).

[5] Jonas Larsson. "Interoperability in Modeling and Simulation". Linköping: Linköpings Universitet, 2003. 178 pp. ISBN: 917373781X.

[6] 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.

[7] 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.

[8] Francesca Rossi, Peter van Beek, and Toby Walsh. "Constraint Programming". *Foundations of Artificial Intelligence*. Ed. by Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter. Vol. 3. Handbook of Knowledge Representation. Elsevier, Jan. 1, 2008, pp. 181–211. DOI: 10.1016/S1574-6526(07)03004-0.

[9] F.R. Kschischang, B.J. Frey, and H.-A. Loeliger. "Factor Graphs and the Sum-Product Algorithm". *IEEE Trans. Inform. Theory* 47.2 (Feb. 2001), pp. 498–519. ISSN: 00189448. DOI: 10.1109/18.910572.

[10] Hans-Andrea Loeliger. "An Introduction to Factor Graphs". *IEEE Signal Processing Magazine* 21.1 (Jan. 2004), pp. 28–41. ISSN: 1558-0792. DOI: 10.1109/MSP.2004.1267047.

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

[12] International Organization for Standardization. *Systems and Software Engineering - System Life Cycle Processes*. May 2023. https://www.iso.org/standard/81702.html (visited on 09/21/2024). Published.

[13] Edward A. Lee. "Determinism". *ACM Trans. Embed. Comput. Syst.* 20.5 (May 29, 2021), 38:1–38:34. ISSN: 1539-9087. DOI: 10.1145/3453652.

[14] 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.

[15] Robert M. Hauser and John Robert Warren. "Socioeconomic Indexes for Occupations: A Review, Update, and Critique". *Sociological Methodology* 27.1 (1997), pp. 177–298. ISSN: 1467-9531. DOI: 10.1111/1467-9531.271028.

[16] Alfred J. Lotka. "Analytical Note on Certain Rhythmic Relations in Organic Systems". *Proc Natl Acad Sci U S A* 6.7 (July 1920), pp. 410–415. ISSN: 0027-8424. PMID: 16576509. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1084562/ (visited on 09/25/2024).

[17] Sam Ouliaris. *Economic Models: Simulations of Reality*. Finance & Development. https://www.imf.org/external/pubs/ft/fandd/basics/models.htm (visited on 09/25/2024).

[18] Brendan Fong and David I. Spivak. *Seven Sketches in Compositionality: An Invitation to Applied Category Theory*. Oct. 12, 2018. DOI: 10.48550/arXiv.1803.05316. arXiv: 1803.05316 [math]. Pre-published.

[19] Brendan Fong. "The Algebra of Open and Interconnected Systems". PhD thesis. Oxford University: arXiv, Sept. 17, 2016. DOI: 10.48550/arXiv.1609.05382. arXiv: 1609.05382.

[20] John C. Baez and Blake S. Pollard. "A Compositional Framework for Reaction Networks". *Rev. Math. Phys.* 29.09 (Oct. 2017), p. 1750028. ISSN: 0129-055X. DOI: 10.1142/S0129055X17500283.

[21] Evan Patterson, David I. Spivak, and Dmitry Vagner. "Wiring Diagrams as Normal Forms for Computing in Symmetric Monoidal Categories". *Proceedings 3rd Annual International Applied Category Theory Conference 2020*. 3rd Annual International Applied Category Theory Conference 2020. Vol. 333. Electronic Proceedings in Theoretical Computer Science. Cambridge, USA: Open Publishing Association, Feb. 8, 2021, pp. 49–64. DOI: 10.4204/EPTCS.333.4. arXiv: 2101.12046 [cs].

[22] Evan Patterson et al. "A Diagrammatic View of Differential Equations in Physics". *MINE* 5.2 (2022), pp. 1–59. ISSN: 2640-3501. DOI: 10.3934/mine.2023036. arXiv: 2204.01843 [math-ph].

[23] 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.

[24] William J. Palm. "Block Diagrams, State-Variable Models, and Simulation Methods". *System Dynamics*. Third edition. New York, NY: McGraw-Hill Science, 2014, pp. 250–318. ISBN: 978-0-07-339806-8. https://highered.mheducation.com/sites/0073398063/information_center_view0/.

[25] Wolfgang Borutzky, ed. *Bond Graph Modelling of Engineering Systems: Theory, Applications and Software Support*. New York, NY: Springer, 2011. ISBN: 978-1-4419-9367-0. DOI: 10.1007/978-1-4419-9368-7.

[26] Henry M. Paytner. *The Gestation and Birth of Bond Graphs*. HMP and Bond Graphs. 2000. https://sites.utexas.edu/longoria/files/2020/10/Birth_of_-Bond_Graphs.pdf (visited on 06/17/2024).

[27] J. J. McPhee. "On the Use of Linear Graph Theory in Multibody System Dynamics". *Nonlinear Dyn* 9.1 (Feb. 1, 1996), pp. 73–90. ISSN: 1573-269X. DOI: 10.1007/BF01833294.

[28] Horace M. Trent. "Isomorphisms between Oriented Linear Graphs and Lumped Physical Systems". *The Journal of the Acoustical Society of America* 27.3 (May 1, 1955), pp. 500–527. ISSN: 0001-4966, 1520-8524. DOI: 10.1121/1.1907949.

[29] Irving Fisher. "What Is Capital?" *The Economic Journal* 6.24 (1896), pp. 509–534. ISSN: 0013-0133. DOI: 10.2307/2957184. JSTOR: 2957184.

[30] John Baez et al. "Compositional Modeling with Stock and Flow Diagrams". *Proceedings Fifth International Conference on Applied Category Theory*. Fifth International Conference on Applied Category Theory (ACT2023). Vol. 380. Electronic Proceedings in Theoretical Computer Science. Glasgow, United Kingdom: Open Publishing Association, Aug. 7, 2023, pp. 77–96. DOI: 10.4204/EPTCS.380.5. arXiv: 2205.08373,.

[31] Peter Pin-Shan Chen. "The Entity-Relationship Model—toward a Unified View of Data". *ACM Trans. Database Syst.* 1.1 (Mar. 1, 1976), pp. 9–36. ISSN: 0362-5915. DOI: 10.1145/320434.320440.

[32] Frank B. Gilbreth and Lillian M. Gilbreth. "Process Charts". Annual Meeting of The American Society of Mechanical Engineers. New York: American Society of Mechanical Engineers, Dec. 5, 1921. https://web.archive.org/web/20150509222833/https://engineering.purdue.edu/IE/GilbrethLibrary/gilbrethproject/processcharts.pdf (visited on 09/25/2024).

[33] Oliver Chukwudi Ibe. *Markov Processes for Stochastic Modeling*. 2nd edition. Elsevier Insights. London: Elsevier, 2013. ISBN: 978-0-12-407795-9.

[34] Rónán Daly, Qiang Shen, and Stuart Aitken. "Learning Bayesian Networks: Approaches and Issues". *The Knowledge Engineering Review* 26.2 (May 2011), pp. 99–157. ISSN: 1469-8005, 0269-8889. DOI: 10.1017/S0269888910000251.

[35] Judea Pearl. *Causality*. Cambridge University Press, Sept. 14, 2009. 487 pp. ISBN: 978-0-521-89560-6. Google Books: f4nuexsNVZIC.

[36] C.J.J. Paredis et al. "Composable Models for Simulation-Based Design". *EWC* 17.2 (July 1, 2001), pp. 112–128. ISSN: 1435-5663. DOI: 10.1007/PL00007197.

[37] Russel Peak et al. *The Composable Object (COB) Knowledge Representation: Enabling Advanced Collaborative Engineering Environments (CEEs)*. Technical Report. National Aeronautics and Space Administration, Oct. 31, 2005. https://www.eislab.gatech.edu/projects/nasa-ngcobs/COB_Requirements_v1.0.pdf (visited on 09/20/2024).

[38] Object Modeling Group. *OMG Systems Modeling Language (OMG SysML)*. Version 1.0. Needham, MA, Sept. 1, 2007. https://www.omg.org/spec/SysML/1.0/PDF (visited on 09/21/2024). Formal.

[39] 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.

[40] Rina Dechter. "Constraint Networks". *Encyclopedia of Artificial Intelligence*. Ed. by Stuart C. Shapiro. 2nd ed. Vol. 1. New York: John Wiley & Sons Inc, Jan. 1992, pp. 276–285.

[41] Christophe Lecoutre. *Constraint Networks: Targeting Simplicity for Techniques and Algorithms*. John Wiley & Sons, Mar. 1, 2013. 461 pp. ISBN: 978-1-118-61791-5.

[42] Rajarishi Sinha et al. “Modeling and Simulation Methods for Design of Engineering Systems”. *J. Comput. Inf. Sci. Eng* 1.1 (Mar. 1, 2001), pp. 84–91. ISSN: 1530-9827. DOI: 10.1115/1.1344877.

[43] John Morris, Gregory Mocko, and John Wagner. “Effects of Functional and Declarative Modeling Frameworks on System Simulation”. *Under review with J. Dyn. Sys., Meas., Control (ASME)* (June 2025).

[44] Christian Andersson. “Methods and Tools for Co-Simulation of Dynamic Systems with the Functional Mock-up Interface”. Doctoral Dissertation in Mathematical Sciences. Lund, Sweden: Lund University, May 4, 2016. ISBN: 978-91-7623-697-0. https://www.maths.lth.se/na/staff/chria/phdthesis.pdf (visited on 03/12/2024).

[45] Theo J. A. de Vries, Paul B. T. Weustink, and Johannes A. Cremer. “Improving Dynamic System Model Building Through Constraints”. *Computer Aided Conceptual Design*. Lancaster International Workshop on Engineering Design CACD’97. Lancaster, 1997, pp. 83–95. ISBN: 1-86220-026-2. https://ris.utwente.nl/ws/portalfiles/portal/169842019/DeVries1997improving.pdf (visited on 09/26/2024).

[46] Frank Feldkamp, Michael Heinrich, and Klaus Dieter Meyer-Gramann. “SyDeR—System Design for Reusability”. *AIEDAM* 12.4 (Sept. 1998), pp. 373–382. ISSN: 0890-0604, 1469-1760. DOI: 10.1017/S0890060498124095.

[47] T Blochwitz et al. “The Functional Mockup Interface for Tool Independent Exchange of Simulation Models”. *Proceedings 8th Modelica Conference*. Modelica Conference. Dresden, Mar. 20, 2011. http://www.functional-mockup-interface.org (visited on 01/13/2022).

[48] David Broman et al. “Requirements for Hybrid Cosimulation Standards”. *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*. HSCC ’15. New York, NY, USA: Association for Computing Machinery, Apr. 14, 2015, pp. 179–188. ISBN: 978-1-4503-3433-4. DOI: 10.1145/2728606.2728629.

[49] Sabine Wolny et al. “Thirteen Years of SysML: A Systematic Mapping Study”. *Softw Syst Model* 19.1 (Jan. 1, 2020), pp. 111–169. ISSN: 1619-1374. DOI: 10.1007/s10270-019-00735-y.

[50] D Harel and A Pnueli. “Reactive Systems”. Logics and Models of Concurrent Systems. Ed. by K Apt. Vol. 13. NATO ASI Series. Berlin, Heidelberg: Springer, Jan. 1985. https://link.springer.com/chapter/10.1007/978-3-642-82453-1_17 (visited on 09/17/2024).

[51] 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.

[52] Cláudio Gomes et al. “Co-Simulation: A Survey”. *ACM Comput. Surv.* 51.3 (May 23, 2018), 49:1–49:33. ISSN: 0360-0300. DOI: 10.1145/3179993.

[53] Claude Berge. *Graphs and Hypergraphs*. Trans. by Edward Minieka. North-Holland Mathematical Library 6. Amsterdam, New York: North-Holland Pub. Co., 1973. 528 pp. ISBN: 978-0-444-10399-4.

[54] Israel N. Herstein. *Topics in Algebra*. 1st ed. Waltham, MA: Blaisdell Publishing Company, 1964. ISBN: 978-0-536-00257-0.

[55] 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).

[56] Giorgio Ausiello and Luigi Laura. “Directed Hypergraphs: Introduction and Fundamental Algorithms—A Survey”. *Theoretical Computer Science* 658 (Part B Jan. 7, 2017), pp. 293–306. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2016.03.016.

[57] Reinhard Diestel. *Graph Theory*. Vol. 173. Graduate Texts in Mathematics. Berlin, Heidelberg: Springer, 2017. ISBN: 978-3-662-53622-3. DOI: 10.1007/978-3-662-53622-3.

[58] Albert Wayne Wymore. *Model-Based Systems Engineering: An Introduction to the Mathematical Theory of Discrete Systems and to the Tricotyledon Theory of System Design*. Systems Engineering Series. Boca Raton, Fla.: CRC Press, 1993. 710 pp. ISBN: 978-0-8493-8012-9.

[59] Francesca Rossi, Peter van Beek, and Toby Walsh, eds. *Handbook of Constraint Programming*. Foundations of Artificial Intelligence. Amsterdam Heidelberg: Elsevier, 2007. 955 pp. ISBN: 978-0-444-52726-4.

[60] Cláudio Gomes et al. “Semantic Adaptation for FMI Co-Simulation with Hierarchical Simulators”. *SIMULATION* 95.3 (Mar. 1, 2019), pp. 241–269. ISSN: 0037-5497. DOI: 10.1177/0037549718759775.

[61] Christian Wolf, Miriam Schleipen, and Georg Frey. “Secure Exchange of Black-Box Simulation Models Using FMI in the Industrial Context”. *Modelica Conferences*. The 15th International Modelica Conference. Linköping Electronic Conference Proceedings 204. Aachen, Germany: Linköping University, Dec. 22, 2023, pp. 487–496. ISBN: 978-91-8075-505-4. DOI: 10.3384/ecp204487.

[62] John Morris. *ConstraintHg*. Version 0.2.2. Nov. 23, 2024. DOI: 10.5281/zenodo.15278018.

[63] Jacqueline Le Moigne and Benjamin Smith. *Advanced Information Systems Technology (AIST) Earth Systems Digital Twin (ESDT) Workshop Report*. NASA, Oct. 28, 2022. https://esto.nasa.gov/files/ESDT_Workshop_Report.pdf (visited on 05/21/2024).

[64] Angela Walters. *National Digital Twin Programme*. Sept. 7, 2019. https://www.cdbb.cam.ac.uk/what-we-did/national-digital-twin-programme (visited on 04/13/2023).

[65] Andy Walker. *Singapore's Digital Twin – from Science Fiction to Hi-Tech Reality*. Infrastructure Global. May 4, 2023. https://infra.global/singapores-digital-twin-from-science-fiction-to-hi-tech-reality/ (visited on 05/22/2024).

[66] 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).

[67] Jethro Akroyd et al. *National Digital Twin of the UK – a Knowledge-Graph Approach*. Dec. 18, 2020. https://como.ceb.cam.ac.uk/media/preprints/c4e-preprint-264.pdf (visited on 05/23/2024). Pre-published.

CHAPTER 3

Methods for Enabling Declarative Simulation

This chapter is based on the paper "Effects of Functional and Declarative Modeling Frameworks on System Simulation," which is under review in ASME's Journal of Dynamic Systems, Measurement, and Control [1]. It was presented in October 2025 at the Modeling, Estimation, and Controls Conference (MECC) in Pittsburgh, PA [2].

Abstract: System modeling frameworks can be categorized into imperative and declarative paradigms. A model's paradigm effects its efficacy: imperative models allow simple execution, while declarative models capture the behavior of the underlying system. This paper compares these paradigms, as well as functional and object-oriented frameworks, in light of physics-based systems. This is done by exploring the principles of systems modeling and simulation. Simulation is shown to be the composition of functions representing system behavior. Simulatable frameworks can be differentiated by their ability to identify and compose these functions for a specific input and output pairing. The various frameworks are explored, applying concepts more typically studied in computer science to general systems engineering. The frameworks are investigated by comparing simulations of a driven double pendulum in various modeling languages. Observations include that functional, declarative models allow for greater reusability and holistic system simulation.

3.1. Introduction

Modelers, trying to represent the behavior of a complex system, are limited by the expressiveness of the available frameworks [3]. The ultimate goal of modeling a system is to understand its interactions, generally for the purpose of simulating its state. This is generally performed in niche frameworks tailored to a specific domain, such as a circuit diagram or ball-and-stick models. Frameworks for multi-domain systems–for instance, one considering the effects of material on power consumption–are often more limited, leading to calls for more advanced, rigorous, multi-domain formalisms [4]. Due to the prevalence of computer simulation, digital systems and counting methods have received the bulk of consideration over the past decades, focusing discussion of different modeling paradigms on the design of software systems. The purpose of this paper is to review how four of these paradigms effect the simulation of physics-based systems, namely imperative, declarative, functional, and object-oriented paradigms. This paper is not intended to specify categories for all system modeling frameworks; the ambiguity with such classifications would make any such activity futile. Rather, by exploring the structures associated with various frameworks, the authors hope to show how different aspects of a modeling paradigm contribute to its simulatability.

To support comparisons between frameworks, the authors have elected to consider a pendulum as a common system, preparing models in different languages based on the established dynamics [5]. These different representations are contrasted to show how each framework exposes the structure of the underlying system. This culminates in an example simulating a double pendulum in five different languages. The simulations show both where principles of computer programming can be applied to systems engineering and where there may be disagreement. Two specific observations are made in the course of the study: first, that declarative models are better suited for simulating heterogeneous systems due to their ability to adapt to changes in the system's scope; and second, that the efficacy of object-oriented paradigms for system simulation increases when the model is both purely functional and enforces sharing of a common global state. These observations derive from the notion that a system is determined by a collection of state variables with behavior given by a set of functions constraining system's state. The act of simulation is consequently the act of composing these functions into a chain that transforms a set of inputs to a desired, unknown output. Each framework can then be differentiated–

in terms of its simulatability–based on its method of identifying and composing the functions of the underlying system.

Note that functional modeling as used here does not refer to the deconstruction of a system into its *functionalities* [6, 7], a focus of systems engineering and design. Rather the functional modeling paradigm discussed in this paper is aligned with practices of computer science, such as described by [8] and [9], where the base unit of representation is an algebraic function. Programming languages that fall under this paradigm include Haskell, LISP, and Miranda, though other languages can approximate function-based modeling (as shown in Section 3.5).

3.2. Review of System Simulation

In designing a system, an engineer must describe how the system's elements should be arranged so that their interactions result in some desired behavior. Whether the elements are people in an airport, parts of an aircraft, or the metal grains of a wing-strut, an engineer must understand how bringing them together will cause people to board a plane that will fly without its wings failing mid-flight. To understand the systems that comprise the universe, engineers must create models that describe both the system elements and relationships between them [10].

Two independent objectives for modeling a system are described here: to describe the elements comprising a system and to simulate facts about the system a model represents. It is the view of the authors that these two aims include most, if not all, objectives of system modeling. Models of the first type are often visual diagrams, such as the free body diagram shown in Figure 3.1(a) or the SysML block definition diagram in Figure 3.1(b). These may be employed as an initial step in understanding a complex system or as a tool for communicating a system's constitution.

The second objective of modeling a system is its simulation, which derives from the interpretation of a system as a source of data [11]. Ashby conceived of a system as a list of variables, whose values are generally considered the state of a system [12]. Simulation is the prediction of a system's state–or at least a part of it–through reasoning on the behavior of the system [13]. The predicted value(s) are artificial, in that they are data corresponding to states of the system that were not observed, often for reasons such as lack of availability, sufficient measurement power, or even temporal existence [14]. Decisions about a system require simulation, especially during the design phase when the systems of

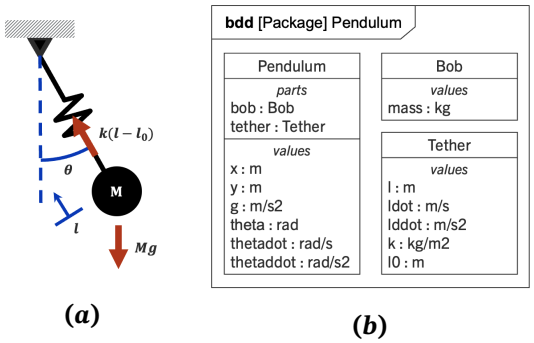


Figure 3.1: Two descriptive models of an elastic pendulum: 3.1(a) a free body diagram showing the forces of the system; and 3.1(b) a SysML block definition diagram showing the system elements.

interest are not yet physically realized [15].

A simulatable model is one that represents system behavior such that artificial observations on the system can be made. In this perspective, the model is the encoding of system behavior, and the simulation is the provision of data to a model such that other data (corresponding with unobserved system states) are revealed. Such a model can be treated as a morphism between a set of inputs and outputs [16, 17]. The morphism for mapping between sets of values is an algebraic function [18], meaning that a simulation model can be described more explicitly as providing a function **F** that maps between the set of state variables whose values are known (inputs) and the set representing another state variable whose value is unknown [19]. The calculation of **F** for some value in its domain constitutes a simulation. Every so-called model solver or simulation engine must provide a way of discovering and calculating such a function from the underlying model [20]. Some models express **F** explicitly, while others provide **F** by the composition of sub-functions $\mathbf{F} = f_n \circ f_{n-1} \circ \ldots \circ f_2 \circ f_1$, where the domain of each sub-function is either a known input or the codomain of a preceding sub-function, and the codomain of f_n is the desired output. This is shown in Figure 3.2, where a simulation of a pendulum's velocity is given by the composition of two functions.

These functions represent the behavior of the modeled system. This was shown by Willems when he described behavior as restrictions on the possible values a system state can exhibit [21]. Models are developed to capture the combinatorial interactions between variables [22]. A model is considered under constrained if the number of values a system variable can exhibit for a single frame of consideration is greater than one. Alternatively, because reality is consistent, models of real systems must be

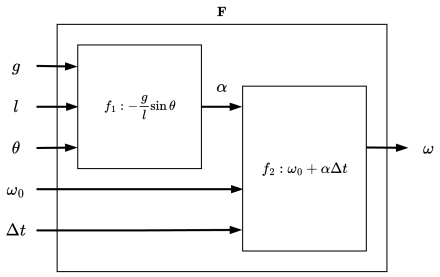


Figure 3.2: Example of a simulation **F** for calculating angular velocity ω from initial position θ for a pendulum. **F** is composed of two functions f_1 and f_2 each taking 3 inputs. By composition, **F** can be treated as a function with 5 inputs.

fully-constrained, so the system state exhibits a unique value for any frame of consideration. Because functions map to unique values in their codomain, any constraint of a real-world system can be represented by a function, and a fully-constrained system model can be considered as the set of functions constraining how a system evolves in response to stimulus [23].

3.3. Modeling Paradigms

The variety of ways to represent and compose functions results in a plurality of modeling languages [24], such as the examples given in Table 3.1. Note that the subjective nature of the designation of *objective* given informally by the authors only as a rough characterization of the language. Whatever a language's objective, it will consist of a set of rules for manipulating symbols that, when followed, enable a user to interpret the system represented by the model [25]. The rules, which describe how system components may be combined [26], are collectively referred to as a framework [8] or a formalism [11]. As a basic example, the components of a circuit diagram are visual symbols representing electrical elements such as wires, resistors, or capacitors. These symbols can only be connected together in specific ways: a resistor symbol, for instance, can only be placed on top of a wire, and not on top of a capacitor. By following the rules of the framework, a modeler can convey the behavior of an electronic circuit. Note that meaningful distinctions between languages result from differences in their frameworks, as symbols can be arbitrarily replaced without affecting the underlying interpretation. Indeed, it can be said that the framework rules provide the interpretation of a model [27]. This motivates the exclusive focus in this paper on modeling frameworks.

The manner in which a model expresses the system's behavior has a significant impact on its simu-

Table 3.1: Examples of system modeling frameworks, adapted from [1]

Framework	Domain	Objective	Source
Bond graphs	Dynamic	Description	[28, 29]
Block diagrams	Dynamic	Simulation	[30, 31]
Stock and Flow	Dynamic	Simulation	[32, 33]
Entity-Relationship	Knowledge	Simulation	[34]
Circuit diagrams	Electronics	Description	[17]
Flow charts	Processes	Description	[35]
Petri nets	Processes	Description	[16]
State machines	Processes	Simulation	[36]
Markov chains	Processes	Simulation	[37]
Gantt charts	Scheduling	Description	[38]
PERT diagrams	Scheduling	Simulation	[39]
Bayesian networks	Stochastic	Simulation	[40, 41]
Causal models	Multi-domain	Description	[42]
SysML diagrams	Multi-domain	Description	[43]
EXPRESS	Multi-domain	Description	[44]
Algebraic Expressions	Multi-domain	Description	
Constraint graphs	Multi-domain	Simulation	[1, 45–47]

latability. Two primary paradigms of providing a simulation function **F** are considered: imperative and declarative frameworks in Sections 3.3.1 and 3.3.2 respectively. Also considered here are secondary paradigms concerning how basic system structures are represented; frameworks that deconstruct a system into modular subsystems are considered object-oriented (Section 3.3.4), while models that express each relationship as a function are functional (Section 3.3.3). The use of these paradigms has considerable effect on how simulations may be conducted.

This section reviews how these paradigms are defined in connection with their relationship with system simulation. Much of the discussion on modeling paradigms is driven by computer scientists, likely due to the prevalence of the computer in defining and simulating systems. Applications of these principles to general systems engineering frameworks are given throughout the section, with a list of frameworks and the paradigms they primarily operate under tabulated in Table 3.2. The categorizations are illustrative of general focus of the framework. They are provided for demonstrative purposes only, as such designations are fairly subjective. MATLAB, for instance, as a programming language can be used to create models in all categories. It's specification as general-imperative indicates its lack of restrictions on models, which by default tend toward the imperative.

Table 3.2: Informal categorization of modeling frameworks as imperative or declarative, further distinguished by emphasized data representation.

	General	*Functional*	*Object-Oriented*
Imperative	Flowcharts MATLAB	Block Diagrams State Machines	C++ SysML
Declarative	Gantt charts PERT diagrams Markov Chains	Bond Graphs Constraint Graphs Bayesian Networks Petri Nets Stock and Flow	*Functional + OO* Modelica Circuit Diagrams ER Diagrams

An analogy of a model is presented here to motivate the discussion of imperative and declarative paradigms. Consider a map of railway connections between cities, such as the one in Figure 3.3a. In this analogy, cities represent state variables and rail lines represent the known functions relating them. This constitutes a model and describes the behavior of the underlying system. Simulation is akin to imagining a traveler moving from city to city. Every city that the traveler has visited becomes a known value for a state variable in the system (within the current frame of consideration). The simulation function $\mathbf{F} \coloneqq \{\text{LA}\} \rightarrow \{\text{Seattle}\}$ is explicitly given in the model as $f_{\text{LA}\rightarrow\text{SEA}}$, however $F \coloneqq \{\text{LA}\} \rightarrow \{\text{NYC}\}$ must be composed from other functions in the model: $f_{\text{CHI}\rightarrow\text{NYC}} \circ f_{\text{DEN}\rightarrow\text{CHI}} \circ f_{\text{LA}\rightarrow\text{DEN}}$.

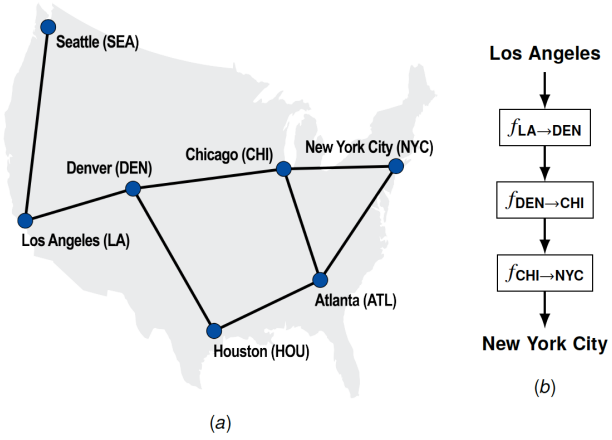


Figure 3.3: Two analogies of modeling: (a) a declarative model expressing every possible route (simulation) that can be connected between the cities (system variables); and (b) an imperative model showing the steps necessary to travel from Los Angeles (the input) to New York City (the output).

Here the map is a declarative model that declares the facts of the system. The map does not prescribe how to travel anywhere, rather it enables an external agent to use the stated facts to create ”routes” of simulation. This is contrasted with an imperative model, such as the one in Figure 3.3b. Models created in this paradigm look more like recipes, providing a list of steps that must be followed to simulate the desired output. A different imperative model must be provided for every simulation function, in contrast with the map which expresses every route in a single structure. Though both models can be used for simulation, the declarative model better captures the behavior of the underlying system.

3.3.1. Imperative Modeling

Any simulation can be described as a sequence mapping inputs to outputs. In a model used for computational simulation, this sequence is a set of tasks performed by the computer and typically termed a program [26]. Flowcharts are an example of an imperative model where the executor is not a computer, such as the flowchart in Figure 3.4 describing the process for calculating the angular acceleration α of a simple pendulum. Note the direct correspondence between the model and the action of simulation, with the model describing the steps needed to artificially calculate the unobserved value α in terms of known values l and θ (the tether length and angular position, respectively). Imperative models, also known as procedural models, are denoted by a change in state, where each line in the program updates the state of the system the model represents until the state arrives at the desired output [48].

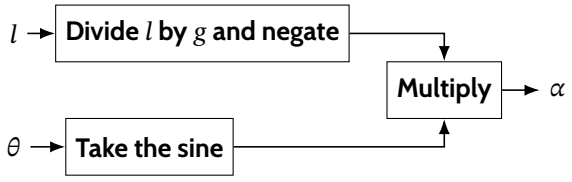


Figure 3.4: A flowchart showing how to calculate the angular acceleration α of a pendulum based on inputs of tether length l and angular position θ.

In prescribing the steps of a simulation, these models are immediately executable. An imperative model defines how a simulation should be calculated [49], a necessary provision before executing a process whether by a computer [50] or biological engine [51]. Consequently, all executable models are either imperative or must be paired with a mechanism for providing an imperative process.

Imperative models are generally the most simple to develop, as they only need to express system be-

havior for the specific fact they are purposed to simulate. Their simplicity, however, belies the difficulty in understanding the systems they communicate [52]. A process has no intrinsic points where it can be broken or reconfigured. Imperative models are consequently more difficult to connect and adapt [31]. This is evident in models of workflow processes, which function well for their prescribed sequences, but incapable of being adapted to new situations or unforeseen problems [53].

3.3.2. Declarative Modeling

While an imperative model describes the steps needed to calculate a single output, declarative models describe many possible values that could be obtained from a simulation. Declarativity exists on a spectrum. Any process that abstracts lower-level instructions may be thought of as declarative [50], making it necessary to establish a frame of reference. In this paper, the word declarative is used to refer to the degree of a framework's capacity to simulate a system's state given a model of its behavior. The more declarative a model is, the greater the portion of the system's state that can be simulated. That is, a declarative model possesses all the information necessary for the state of a system to be ascertained by a declarative solver. The test for a purely declarative language is whether expressions in the model can be evaluated without a specific order [49], a property often described as referential transparency. This is because declarative solvers must be able to rearrange expressions to form simulations–an expression that changes its output based on what was called previously cannot be arbitrarily arranged. Referentially transparent models consequently codify all possible ways to modify a system's state, rather than describing a specific mutation sequence.

The map analogy again proves useful; consider how each step of a route is provided with respect to an updated location of the traveler. However, the connections shown in Figure 3.3a never change regardless of the actual route taken by the passenger. It is assumed that an able traveler can form a sequence for traveling to their destination from the information declared by the map. Similarly, declarative models provide information without providing the procedures for simulating that information [54], leaving the work of routing the lower-level process to an execution agent [48]. This is evident in the parametric diagrams of SysML, where constraints must be solved by an external solver [55, 56].

Because of their generality, declarative models are better able to capture the holistic nature of a system. An imperative framework provides a single avenue for simulating a system, while every possible

way of simulating a system can be extracted from a declarative model. Circuit diagrams are declarative, in that an engineer can solve for the power at any node in the circuit. Likewise, algebraic expressions declare equivalencies as in, for instance, a linear system of equations $A\mathbf{x} = B$, which can be subsequently solved imperatively for $\mathbf{x}$ by an agent (either a computer or tired student) using matrix row operations. In many languages it is possible to describe both imperative and declarative models, often by focusing on features dedicated to one paradigm or another. Pure LISP, for example, is a declarative language until functions for updating list values are introduced [49].

Whatever the implementing language, simulation of a declarative model is contingent upon a mechanism for extracting information out of the model, turning inputs to outputs. These mechanisms may be constraint solvers (especially with logic programming [26]) or proprietary compilers (such as with Modelica [57]). Functional languages expose information by expressing each declared facts as a function, so that a compiler can chain functions together using function composition [58, 59]. The rigor and power of this method has made functional programming nearly synonymous with declarative modeling [48], though the two are decidedly distinct [49].

3.3.3. Functional Modeling

Because functions are the primitive element of a simulation, all models must identify and compose functions for simulation. However, this does not mean that all frameworks express these functions explicitly, as do those described as functional paradigms. Functional models represent a system as a set of basic functions with behavior defined by composition [60].

It is the fact that functions compose that give functional models their primary characteristics. While typical structures do not necessarily provide points of connection, any two functions that share a domain and codomain can be immediately joined together. The functions presented by a functional model form a set of reconfigurable building blocks that can be used by a modeler to simulate various aspects of the represented system [58]. Such functionality is sometimes referred to as composability [15, 61, 62], modularity [58, 63], extensibility [54, 64], or interoperability [65]. The rigor of function composition, supported by Church's lambda calculi [66], drove the development of functional languages such as LISP, Haskell, and Miranda [67].

Using function composition as the "glue" for bringing elements together brings significant benefits

for modeling systems [58]. It was shown in Section 3.2 that the fundamental representation of system behavior is a function mapping between sets. Because each element in a function's domain must be mapped to a unique value in its codomain, a function mapping represents an independently composable method for constraining a state variable [68]. In other words, a function describes the affect of the system on a single variable, such as an output desired in a simulation [9].

3.3.4. Object-Oriented Modeling

Creating objects while modeling is directly reflective of a system of systems worldview, where every system can be broken down into a collection of interacting subsystems [26]. Every object represents an independent system composed of elements that exhibit collective behavior. Objects in a model consequently contain elements (usually variables) and manifest behaviors encoded in a set of processes (often–but not always–functions) that relate the elements. These might be defined in a class template, of which each object adheres to as an instance of the class [49].

The tools for defining a system are provided by functional programming, consequently most object definitions adhere to the functional modeling paradigm [69]. Differences arise in consideration of how subsystems interact. Object-oriented paradigms encapsulate portions of a system, so that each encapsulated object (subsystem) has its own state distinct from the state of the global system [69]. The global system behavior is then described by coupling subsystems along predefined ports [15]. The resulting inter-subsystem coupling is often (though not always [19]) conducted imperatively [70], resulting in sequences of tasks sent between objects [71]. The SysML model shown in Figure 3.1(b) is an example of encapsulated objects, where the tether and bob must be connected to the pendulum values to form a holistic system [56]. In order to be both functional and object-oriented, a paradigm must provide some mechanisms for ensuring inter-object relationships are purely functional. Examples include requiring objects to sharing a collective state and preventing non-functional couplings.

3.4. Effects of Frameworks on System Simulation

The purpose of an ideal modeling framework is to express all the information associated with a system: what states it manifests both currently and in the future. Simulatability is the ability for expressions of these states to be calculated from some base model. Forming a simulation, as defined in

Section 3.2, requires the ability to rearrange functions into a process that can transform inputs into outputs. The arbitrary ordering of methods is akin to referential transparency, a property of declarative models. Consequently, simulatability is indicated by the declarativity of a model.

Defining system behavior by functions has implications for how these functions can be rearranged. The inputs of a function describe its dependencies. This concept is distinct from causality; as evidence, consider a function for calculating gravitational acceleration based on recording how long it takes an item to fall in a vacuum. This function indicates that calculating gravity is dependent upon fall time, not that falling causes gravity to behave as it does. Rearranging functions requires knowledge of their dependencies, a potential source of error for a modeling framework. If the dependencies of the system are not well identified, the model's simulatability will decrease. Diverging from the structure of flat, functional models can have unintended consequences of a model's expression of functional dependencies. These consequences and their sources are explored in the remainder of this section with imperative (Section 3.4.1) and object-oriented approaches (Section 3.4.2). The takeaways of this section are summarized in Table 3.3.

3.4.1. Limitations of Imperative Models

The least interoperable models are those constructed in an imperative framework, such as the flowchart shown in Figure 3.4. Though the same variables (g, l, θ, α) and relationships are included in the chart as in other models, the flowchart cannot be connected with any other diagram except on the provision of its output, α. This is because the chart describes behavior with processes, which cannot pass information except at their endpoints. There are three processes in the flowchart:

1. Divide l by g and negate;
2. Take the sine; and
3. Multiply.

It is not communicated to any interfacing agent what the results of the processes are. Consequently, it cannot be inferred from the model alone what effect there might be in rearranging them. The result is a brittle model that can be used to simulate the system only for the prescribed scope, rather than a flexible model for describing the relationships between variables [16].

Functional imperative models, such as the block diagram shown in Figure 3.7, are less inflexible.

A block diagram describes each relationship as a function, mapping a set of inputs to outputs. Because functions always reduce to unique values, a simulation can exhibit system facts anywhere in the diagram. Despite the benefits arising from representing behaviors as functions, they are still limited by their imperative natures [31]. The block diagram describes a process for calculating some output variable, with each step of that process given by some function transforming the outputs of the one before it. While the outputs of each function are able to be clearly expressed, the behaviors themselves are not interoperable, and the scope of the model remains fixed. One indication of this is that the inputs to the model cannot be changed–one could not discover gravitational acceleration g for instance by inputting observed values of angular acceleration α, even though those relationships are at least hinted at by the model. This is also true for state machines, which show how a state transforms according to a set of composing functions, but which cannot replicate that process if the system state must be adapted.

3.4.2. Simulatability with Object-Oriented Modeling

Both the benefits, and the limitations, of object-oriented programming stem from the encapsulation of subsystems into distinct objects. Although object-oriented systems are often considered universally more interoperable, this claim merits closer inspection [72]. By ”hiding” information from the rest of the model behind the object boundary, a modeler is able to describe which elements of the object are influential in determining the inter-object behavior of the global system [73]. This allows (indeed, requires) interfaces to be built by which objects can interact with each other [71]. Object interfaces enable greater composability in the same way a handle simplifies interactions with a door.

The pitfall of encapsulated models is when the real system deviates from the imposed worldview. While it is natural to decompose a system into distinct entities [74], such distinctions are inevitably arbitrary. It is often impossible to abstract systems that interact with other systems across many dimensions (often termed reactive systems [75]). Such a strongly encapsulated object might fail to express which variables are dependencies for other behaviors, losing the capacity to express interactions of the aggregate system. The key here is that reality generally does not decompose into independent subsystems. Even if the interaction between Brazilian butterflies and Texan tornados is slight, a model that isolates their behaviors will fail to capture the full system effects [76], what Nielsen et al. call the ”synergistic collaboration of constituents” [65]. A strongly encapsulated object-oriented framework with hidden

states and methods is more modular, but at the cost of limited ability to capture complex interactions.

Not all object-oriented paradigms require strong encapsulation. Some paradigms, such as circuit diagrams, enforce a global state that encompasses all objects in the system. This allows for modularization of a system without losing the ability to capture holistic behavior. There seems to be some confusion about this topic: nearly all sources agree that the key features of object-oriented modeling are the encapsulation of objects that maintain a private state, leading to interaction via sequences of messages communicated between objects [26, 49, 69–71]. However, the label ”object-oriented” is often applied whenever properties are merely grouped together, rather than fully encapsulated. For instance, a system of two simple, uncoupled pendulums is shown in Figure 3.5. Each pendulum is represented by the same four variables (θ, ω, α, and l) and share the gravitational acceleration g. Though each set of pendulum variables can be distinguished from the rest of the system, such grouping has no effect on the actual behavior of the system. This is because the states of the two pendulum are adopted by the global system, so that relationships could in theory be expressed between any subsystem variable. There is a case to be made that these subsystems should not be strictly considered objects, since there is no encapsulation nor serialized communication, although some works on systems modeling describe them as such [11, 19].

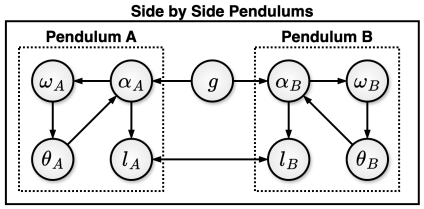


Figure 3.5: A system representation of two pendulums hung side by side (but not coupled). The variables associated with each pendulum are shown in the dashed boxes, while inter-variable algebraic relationships are shown with solid arrows.

The classic interpretation of object-oriented modeling requires subsystems to expose only a part of their local state through an interface. If every system fact is available to other agents, then identifying subsystems is merely a convenient organizational tool rather than a true-indicator of a class structure (though perhaps enabling such useful properties as inheritance and polymorphism). A model that exposes all state variables to functional relationships avoids the limits of imperative connections between encapsulated objects; these types of models can be written even in object-oriented paradigms such as

C++ or Modelica, though doing so limits whether certain features of these languages, such as inheritance schemes, can be expressed.

Encapsulated objects are not without their benefits in modeling. The rigid, hierarchical structures [77] of object-oriented paradigms enable a system to be statically defined so that its state is prescribed to evolve only in specified ways regardless of its environment. This allows objects (or the templates that define them) to be reused outside their intended use case. This is especially useful for software development, where modules and libraries are integrated into foreign scripts without need for adaptation. Other benefits include obscuring sensitive data (such as proprietary models) [78], and established interfaces. Object-oriented paradigms have resulted in significant advances in co-simulation, such as the Functional Mockup Interface used for system of systems simulation in Modelica [79, 80].

The authors argue though that the dynamic nature of systems engineering, where the behavior of the systems might change rapidly, negates many of these benefits. Rigid, inflexible interfaces do not provide modelers with the ability to adapt models to new environments requiring new behavioral interactions. For instance, the equation of motion for a simple pendulum can be expressed as $\alpha = -\frac{g}{l}\sin\theta$. However, if the pendulum is coupled with another pendulum (such as when forming a double pendulum), this equation no longer holds. Instead, the pendulums motion is influenced by the momentum of the other bob, and the equations of motion become more complex [5]. Expanding the scope of the system invalidates the model. If the pendulum is encapsulated, the model provides no indication that this should be the case, as encapsulated objects are expected to evolve independent of which systems they are coupled with. This is famously studied under the ”Expression Problem” [81], which showed the limits of procedurally connected objects to reflect changes in behavior [82]. Though solutions to this problem have been proposed [83], the underlying philosophy of independent subsystems clearly is at odds with the idea of emergent, holistic behaviors prevalent in systems theory.

3.5. Study of Simulation Methods by Paradigm

To demonstrate the claims made in Table 3.3, two simulations of a double pendulum system are conducted in five modeling frameworks. The object of the study is to show how the arrangement the modeling framework influences how a simulation is conducted. The methodology is to build a model in each paradigm that can be simulated in two ways. Each model is based on a set of common functions

Table 3.3: Overview of simulation in various modeling paradigms.

Paradigm	Definition	Strengths	Weaknesses
Imperative	Defines the specific process transforming inputs to outputs	Simple to develop	Manual simulation processes for each input/output
Declarative	Defines the relationships which are processed by a solver to compose the simulation	General simulation of a system	Requires solver to provide simulation processes
Functional	Reduces each relationship to an explicit algebraic function	Reconfigurable, exposes system behavior	Additional modeling steps
Object-Oriented	Structures subsystems into independent modules	Modular and exportable, especially for static systems	Encourages data hiding and imperative behavioral specification

describing the pendulum's behavior. With the system and simulation controlled across experiments, the differences in model structure can be attributed to the modeling framework. The variations in arranging the common functions are subsequently evaluated by their simulatability, measured by less subjective metrics of reuse and declarativity, as defined in Section 3.6.

3.5.1. System Description

The double pendulum system considered in the study consists of two linked pendulums, where the top pendulum A is driven at some predefined angular velocity and the bottom pendulum B is free to rotate around the first, as shown in Figure 3.6. Both pendulums are of unit length. The scope of this system is given by the following variables, while its behavior is described by the subsequent functions:

Variables, with values given for constants:

- $g = 9.81$ m/s^2, the gravitational acceleration;
- θ_A, θ_B (rad), the angular position of the bobs with respect to the vertical axis;
- ω_A, ω_B (rad/s), the angular velocity of the bobs;
- α_A, α_B (rad/s^2), the angular acceleration of the bobs;
- $\Delta t = 0.1$ s, the time step of the simulation; and
- $t = 0.1$ s, the current time for the frame of consideration.

Relationships:

$$f_{\alpha_A} : \{\omega_A, \omega_{A_0}, \Delta t\} \rightarrow \alpha_A = \frac{\omega_A - \omega_{A_0}}{\Delta t} \tag{3.1}$$

$$f_{\alpha_B} : \{\theta_A, \theta_B, \omega_A, \alpha_A, g\} \rightarrow \alpha_B$$
$$= -\ddot{x}\cos\theta_B - \sin\theta_B\,(\ddot{y} + g)$$

where: (3.2)

$$\ddot{x} = \alpha_A \cos\theta_A - \omega_A^2 \sin\theta_A \text{and}$$
$$\ddot{y} = \alpha_A \sin\theta_A + \omega_A^2 \cos\theta_A$$

$$f_{\omega_A} : \{t\} \rightarrow \omega_A = \begin{cases} -\frac{\pi}{4} & \text{if } t \,\%\, 4 < 2 \\ \frac{\pi}{4} & \text{otherwise} \end{cases} \tag{3.3}$$

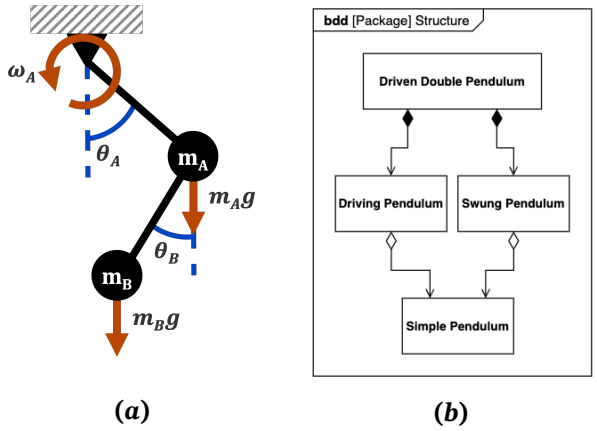
$$f_{\omega_B} : \{\alpha_B, \omega_{B_0}, \Delta t\} \rightarrow \omega_B = \alpha_B\left(\Delta t + \omega_{B_0}\right) \tag{3.4}$$

$$f_{\theta_A} : \{\omega_A, \theta_{A_0}, \Delta t\} \rightarrow \theta_A = \omega_A\left(\Delta t + \theta_{A_0}\right) \tag{3.5}$$

$$f_{\theta_B} : \{\omega_B, \theta_{B_0}, \Delta t\} \rightarrow \theta_B = \omega_B\left(\Delta t + \theta_{B_0}\right) \tag{3.6}$$

Equations 3.4, 3.5, and 3.6 are a first-order Eulerian integration of α and ω to yield ω and θ, while Eq. 3.1 is a first-order differentiation of α in terms of ω. While none of these are very robust, the simple relationships help show the functional nature of simulation without relying on proprietary numerical recipes. In addition to these differential terms, Eq. 3.2 and 3.3 result from the interaction of the driving pendulum with the freely swinging one.

Two simulations of the pendulum are considered: the first to simulate θ_B (Case 1), and the second to simulate the sum of θ_A and θ_B. These outputs are calculated when $t = \Delta t$, after the first time step. For comparison purposes, both cases are simulated with the same initial inputs of $\theta_{A_0}, \theta_{B_0} = \frac{\pi}{4}$, and $\alpha_{A_0}, \omega_{B_0} = 0$, in addition to the constants defined above. Each case is given by a simulation function, $\mathbf{F}_1$ and $\mathbf{F}_2$ respectively, which can be composed algebraically from the functions given by Eq. 3.1–3.6 in the following ways:



(a) **(b)**

Figure 3.6: Free body diagram of a driven double pendulum, where the upper bob rotates around the fixed point at a fixed angular velocity shown as 3.6(a) a free body diagram; and 3.6(b) a SysML block definition diagram showing the class deconstruction.

$$\mathbf{F}_1 : f_{\theta_B} \circ f_{\omega_B} \circ f_{\alpha_B} \tag{3.7}$$

$$\mathbf{F}_2 : f_{\theta_A} \circ f_{\omega_A} + f_{\theta_B} \circ f_{\omega_B} \circ f_{\alpha_B} \tag{3.8}$$

Each simulation is prepared[1] in five unique frameworks representative of different modeling paradigms:

Imperative, Non-functional: MATLAB

Imperative, Functional: Block diagrams (prepared in Simulink)

Imperative, Object-Oriented: C++

Declarative, Object-Oriented: Modelica

Declarative, Functional: Constraint hypergraphs

The authors reemphasize that there is no specification of a language as imperative versus declarative or functional versus object-oriented, it is the framework that should be considered in these categories. For instance, models in the MATLAB programming language could be imperative, functional, or even arranged in classes; it's use in representing purely procedural modeling is solely demonstrative.

The differences between the paradigms considered in the study are in the way they form the functions of $\mathbf{F}_1$ and $\mathbf{F}_2$ given in Eq. 3.7 and 3.8. A diagrammatic overview of how the functions in Eq. 3.1–3.6 are arranged in each of the frameworks is given in Figure 3.8.

[1]Full scripts for all simulations are made available on GitHub.

3.5.2. Imperative, Non-Functional

The most basic way to form the simulation is to construct it as an explicit process. As shown in the top row of Figure 3.8, a non-functional implementation ignores the behavioral building blocks given in Eq. 3.1–3.6. Instead, both $\mathbf{F}_1$ and $\mathbf{F}_2$ are given as black box functions, without any indication of the processes employed to transform inputs to outputs. Such a process is given in Block 3.1.

Block 3.1: Imperative, non-functional model simulating case 1 in MATLAB.

```
function thetaB = F1(alphA, omegaA, thetaA, thetB, lA, lB, g, step)
  xddot = lA * (alphA * cos(thetaA) - omegaA^2 * sin(thetaA));
  yddot = lA * (alphA * sin(thetaA) + omegaA^2 * cos(thetaA));
  alphaB = (xddot * cos(thetaB) + sin(thetaB) * (yddot + g)) / -lB;
  omegaB(2) = alphaB * step + omegaB;
  thetaB(2) = omegaB(2) * step + thetaB;
end
```

This script is not easy to rearrange due to the constant state mutations. Without expert knowledge of the model, it is difficult to know whether it is possible to put the third line before the first, or if `thetaB` could be calculated before `alphaB`. The dependencies are not explicitly provided, making the process all but impossible to modify unless a modeler is fully aware of the underlying model behavior.

3.5.3. Imperative, Functional

A functional paradigm, such as the block diagram shown in Figure 3.7, is comparatively more configurable due to the expression of the functions generating the process. The block diagram shown in the second row of Figure 3.8 is still imperative, but the inputs and outputs to the blocks show how they can be arranged. The result is that the modeler immediately knows that `thetaB` cannot be simulated before `alphaB`, because `alphaB` is an input to an upstream function.

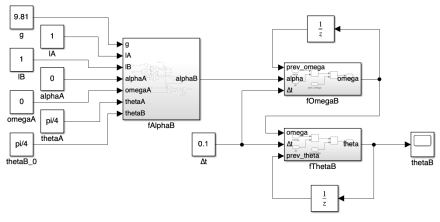


Figure 3.7: Imperative, functional model simulating case 1 in a block diagram, implemented in Simulink.

3.5.4. Imperative, Object-Oriented

Regarding simulation, the primary difference between the functional and object-oriented paradigms is the additional structure of the latter. An example of the class structure of the pendulum is described by the SysML diagram in Figure 3.6(*b*). This structure can be exploited by inheriting the variables and functions of the base `Pendulum` class into both the `SwungPendulum` and `DrivingPendulum` models. This is visualized in the third row of Figure 3.8, where the Pendulum class structure is described on the left, and its interface (composed of its member variables) shown inherited within each of the boxes describing `SwungPendulum` and `DrivenPendulum`. The formal definition for each of these classes is given in Block 3.2.

Block 3.2: Class structure for models in C++

```
class Pendulum {
  float alpha, omega, omega0, theta, theta0, g, step;

  float f_omega() {
    omega = omega0 + alpha * step;
  }

  float f_theta() {
    theta = theta0 + omega * step;
  }
};

class SwungPendulum : public Pendulum {
  float f_alpha(float alphaA, float omegaA, float thetaA) {
    float xdd = alphaA * cos(thetaA) - pow(omegaA,2) * sin(thetaA);
    float ydd = alphaA * sin(thetaA) + pow(omegaA,2) * cos(thetaA);
    alpha = -xdd * cos(theta) - sin(theta) * (ydd + g);
  }
};

class DrivingPendulum : public Pendulum {
  float f_alpha(float omega0) {
    alpha = (omega - omega0) / step;
  }

  float f_omega(float t) {
    float speed = PI / 4;
    omega = (fmod(t, 4) < 2) ? -speed : speed;
  }
};
```

Organizing construction of the composite system in this way allows a modeler to reuse behavior defined in a single place, under the assumption that the behavior within the class will not change, no matter the context it is implemented in. This allows the `Pendulum` class to be inherited into two different

contexts without loss of meaning. Behavior descriptions within a class are imperative processes that manipulate the state of the class (comprised of its variables). It remains the prerogative of the modeler to define how the system is to be simulated. For instance, although all the functions of Eq. 3.1–3.6 are accessible in `DrivenPendulum`, the modeler still has to define how they can be arranged to produce the simulation functions `F_1` and `F_2` shown in Block 3.3.

Block 3.3: Object-oriented simulations of case 1 and 2 in C++

```
class DrivenPendulum {
  DrivingPendulum A;
  SwungPendulum B;

  float F_1() {
    B.f_alpha(A.alpha, A.omega, A.theta);
    B.f_omega();
    B.f_theta();
    return B.theta;
  }

  float F_2(float t) {
    A.f_omega(t);
    A.f_alpha();
    A.f_theta();
    B.f_alpha(A.alpha, A.omega, A.theta,);
    B.f_omega();
    B.f_theta();
    return A.theta + B.theta;
  }
};
```

These models as shown are still purely functional in that every method modifies only a single variable in the object's state. This allows the system behavior (which is always expressible in terms of strict functions [21]) to be captured by the modeler, for instance in the arrangement of `F_1` and `F_2`. This is not the case if the methods begin modifying more than one state variable. As an example, consider the redefinition of the `Pendulum` class which imperatively combines f_{ω_B} and f_{θ_B} into a single method, written in Block 3.4 and described visually on the bottom left of the third row of Figure 3.8.

Block 3.4: Non-functional redefinition of `Pendulum` class in C++

```
class PendulumNonFunctional {
  float alpha, omega, omega0, theta, theta0, g, step;

  float f_omega_theta() {
    omega = omega0 + alpha * step;
    theta = theta0 + omega * step;
  }
};
```

As defined, the `PendulumNonFunctional` class limits the simulatability of the model. The function

`f_omega_theta` lumps together the behavior of ω and α. While this has negligible impact for simulating θ_B, it precludes the ability for ω to have deviant behavior. The difference between the driven (A) and swung (B) pendulums is that ω_A can be expressed as a function of t, as in Eq. 3.3. This is handled in the functional,-object-oriented framework by overriding `f_omega` in `DrivenPendulum`. This becomes impossible when incorporating non-functional methods–not that f_{ω_A} can no longer be defined, but rather that f_{θ_A} can no longer be called from the parent class. The result is that $\mathbf{F}_2$ can no longer be composed within the non-functional framework, which consequently fails to express the behavior of the driven bob.

3.5.5. Declarative, Object-Oriented

There are relatively few frameworks that are both explicitly declarative and object-oriented. Modelica, based originally on Dymola [84], allows for models to be declared as objects. The language's compiler employs a robust set of algorithms that can solve the differential equations in the model [85], providing a convenient platform for dynamic modeling. While the C++ functions described in Section 3.5.4 have to be manually connected, models written in the Modelica framework can be combined automatically by the underlying solver, the basic premise of declarative modeling.

As an equation solver, Modelica models must be fully constrained before a solution can be found (unlike the imperative process in Block 3.1, which only required the specific functions of Eq. 3.7). This means that the entire system must be solved concurrently–there is no command to solve for only a single variable in Modelica. Instead, as is typical of a declarative paradigm, the model is simulated with a simple call to the underlying engine (`simulate(DrivenPendulum);`). There is also no way to mandate the solution method used by the solver, which is why the Eulerian integrations are not specified in the model.

Because Modelica is object-oriented, the various types of pendulums can immediately extend the base class provided at the beginning of Block 3.5. Note however that the objects themselves are only ever coupled within a class, never connected via imperative message senders. This is true of every Modelica model, which has no ability to procedurally call system structures [19]. Because of this, the super class `DrivenPendulum` shares the same state as the subclasses it inherits. This means that there is no encapsulation of Modelica objects, although information can be hid by using access specifiers. The

difference means that inter-object connections are not imperatively abstracted the way messaging in other object-oriented paradigms.

Block 3.5: Declarative, object-oriented simulation in Modelica

```
class Pendulum "simple pendulum"
   parameter Real l=1;
   Real theta(start=0.7);
   Real omega(start=0.0);
equation
   der(theta) = omega;
end Pendulum;

class SwungPendulum "simple pendulum coupled to driving pendulum"
   extends Pendulum;
   Real xddot, yddot;
end SwungPendulum

class DrivingPendulum "driving pendulum"
   extends Pendulum;
   parameter Real speed=0.7;
equation
   if time mod 4 < 2 then
      omega = -speed;
   else
      omega = speed;
   endif;
end DrivenPendulum

class DrivenPendulum "driven double pendulum"
   parameter Real g=9.81;
   SwungPendulum swung(theta(start=0.7));
   DrivingPendulum driver(theta(start=0.7));
   Real sum_theta;
equation
   swung.xddot = driver.l * driver.alpha * cos(driver.theta) - driver.omega^2 * sin(driver.theta);

   swung.yddot = driver.l * driver.alpha * sin(driver.theta) + driver.omega^2 * cos(driver.theta);

   der(swung.omega) = (swung.xddot * cos(swung.theta) + sin(swung.theta) * (swung.yddot + g));

   sum_theta = swung.theta + driver.theta;
end DrivenPendulum;
```

3.5.6. Declarative, Functional

Imperative approaches successfully simulate the first calculated value of θ_B. However, any other output such as ω_A or even a value of θ_B at t_n for $n > 1$, is calculated from a different function composition chain than Eq. 3.7. Each unique process requires a corresponding imperative model to be configured. These procedural approaches are contrasted with the constraint hypergraph in the bottom row of Figure 3.8.

The hypergraph captures all possible function composition chains from the model of the pendulum system. This modeling framework expresses systems graphically, with variables as nodes and the functions relating them given as hyperedges. Each path in the hypergraph represents a valid way of composing functions in the system, consequently the set of all paths describes all possible simulations of a modeled system. A more detailed overview of this framework and its use in simulating systems is given by the authors in [1].

Because every simulation of the system can be represented by a path through the hypergraph, simulation construction can be autonomously performed by finding the lowest cost path between the nodes representing the inputs and output. Performing this pathfinding constitutes a declarative solver. Given a path exists in the model, a solver can discover a simulation between any set of variables.

3.6. Results

The measurement of a model's simulatability is more qualitative than quantitative. Subjectivity is addressed in the analysis of the case study by reducing simulatability into two metrics that can be measured more precisely: reuse and declarativity. These metrics were specifically selected for being pertinent to systems simulation. Definitions and the motivation for these metrics are given as:

1. Reuse: to what degree the structure of the first simulation is repeated in the second. Reusability is a partial capture of the extent to which system behavior can be expressed within the model. A reusable model captures the interactions in the system, so that the modeler can understand how behavior in one simulation influences other components in the system. A non-reusable framework enables no such knowledge capture, leaving the system's behavior entirely unknown to the modeler. Reusable frameworks enable simulations to be easily formed from the underlying system models. A reusable model, for instance, would automatically capture the formulation of $\mathbf{F_1}$ in the solution of $\mathbf{F_2}$.
2. Declarativity: the number of unique simulations that can be conducted on the model without additional user arrangement. This is related to black box extensibility, described by Zenger [64] as the ability for a model to be extended without explicit knowledge of the underlying system by the modeler. Expressive frameworks anticipate additional simulations without requiring added modeling effort.

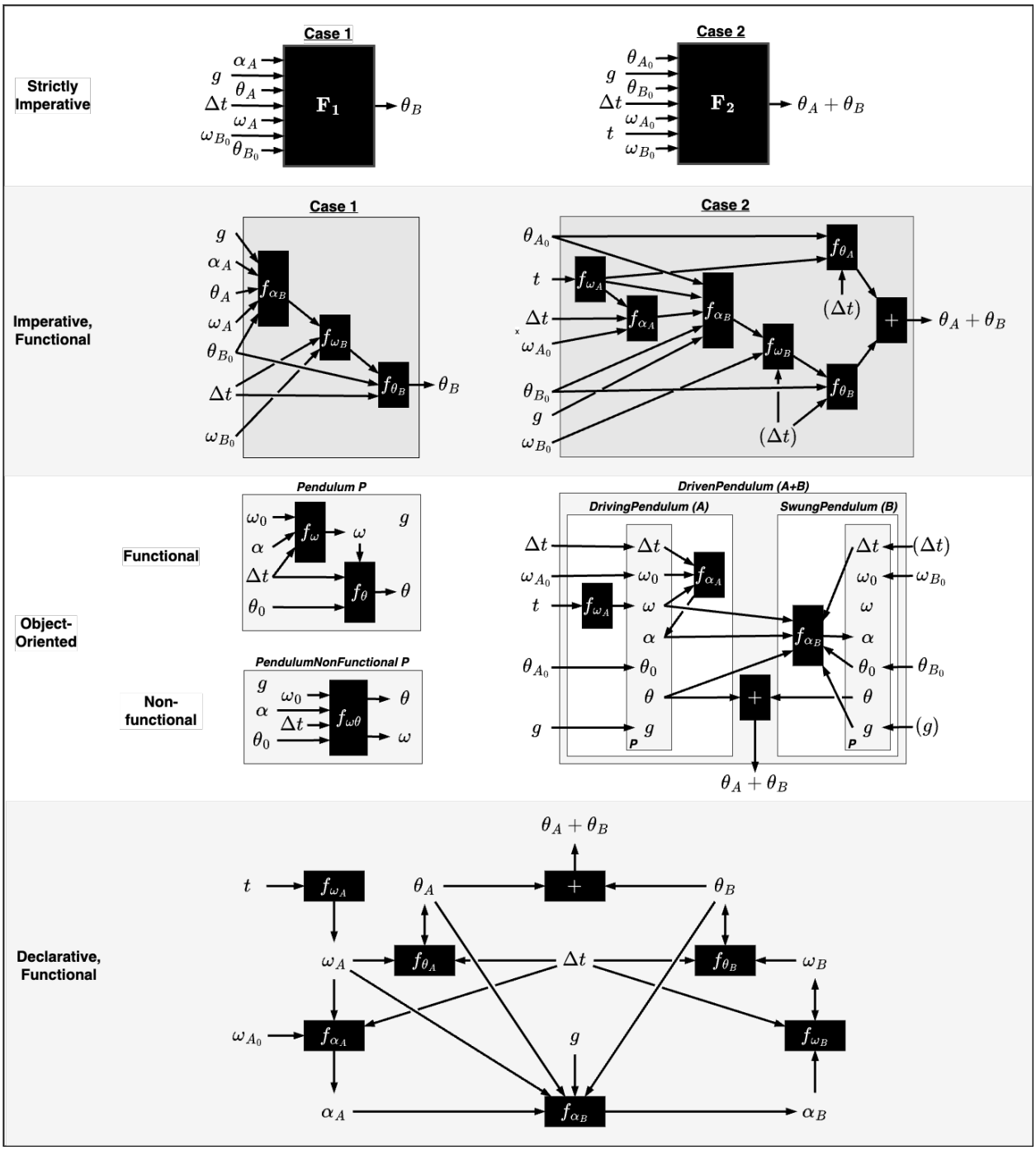


Figure 3.8: Overview of how the functions are arranged in each modeling approach to form simulations for each of the two cases. From top to bottom: strictly imperative, imperative-functional, object-oriented, and declarative-functional. In each, functions are shown as black boxes while variables are given as plain text. Inputs and outputs are described by wires with arrows pointing in or out of the black boxes respectively. The scope in all but the declarative-functional model is defined by alternating gray and white boxes, with global inputs and outputs indicated by wires crossing these boundaries.

The general conclusion of the analysis is that declarative models are much more extensible than imperative ones. Framework structures (object-oriented versus functional) follow this correlation with respect to their degree of declarativity: imperatively-coupled classes and procedures make it "very difficult to understand the relationship between a program and ... the function which it computes" [52].

3.6.1. Reuse

As a measure of how much the structure of the model is reused between common simulations, the diagrammatic description of the model's structure given in Figure 3.8 is the primary source of evaluating a framework's reusability. A result common to both metrics is the lack of simulatability in the imperative, non-functional framework (the control model). The first row of Figure 3.8 makes this evident, where the structures of the two simulations (indicated by $\mathbf{F}_1$ and $\mathbf{F}_2$) are entirely independent of each other. The imperative approach provides no components for rearranging, consequently there is nothing a modeler can imply from case 1 that assists in building the simulation for case 2. In other words, there is no ability for a modeler to form $\mathbf{F}_2$ given $\mathbf{F}_1$ as expressed as an imperative procedure. In contrast with the control framework, the degree of reusability is much greater for the subsequent rows. The functions f_{α_B}, f_{ω_B}, and f_{θ_B} are each reused in Case 2 for the diagrams of the imperative-functional, object-oriented, and declarative-functional frameworks. Additionally, in every framework the arrangement of these functions (in forming Eq. 3.7) is fully maintained between cases.

Further distinctions can be drawn between how the frameworks specify possible arrangement of the base functions. In the imperative-functional framework, a modeler can immediately take the f_{α_B} function black box and pass ω_A in as an input along the specified wire. Similarly, the output of the f_{θ_B} function black box can be wired to the + operation furnishing the result of $\mathbf{F}_2$. Reconfiguring the model in this way does not depend on the modeler understanding the processes embedded in the functions, only which inputs and outputs are required to form the second simulation. Black box use is not as possible for non-functional methods, which require the modeler to know what portions of the systems state are modified inside the method. The non-functional method `f_omega_theta`, for instance, cannot be employed unless the modeler knows how both ω and θ will be modified.

The options for rewiring are increased in the functional-object-oriented paradigm, if only because the model is more comprehensive. Each method defined for a class is an additional way of forming

a simulation. For instance, the parent class `Pendulum` is inherited twice in the construction of the `DrivingPendulum` and `SwungPendulum` classes, allowing the functions f_ω and f_θ to be readily reused across instantiations. This, again is done without requiring knowledge of the pendulum's functions by the modeler. Finally, the declarative-functional model displays which simulations can be reused and in what matter–every possible arrangement of the system functions.

3.6.2. Declarativity

The final metric of consideration is declarativity, considering the number of simulations that can be automatically composed from a given framework. This measure is a proxy for estimating how much information about a system is contained in the model framework, such that it can be made available without additional understanding required from the modeler. The measurement for this number is taken only on the model for case 2, which is the maximally defined model for all frameworks. A simulation is considered to be able to be autonomously formed if the inputs and outputs for a simulation function are declared in the model, so that the only action needed to execute a simulation are to pass these inputs and outputs without any rewiring. Simulations are considered identical if they contain the same input and output variables–that is they can be expressed using the same simulation function.

The counts for all frameworks are:

- Strictly imperative: 1 ($\mathbf{F}_2$)
- Imperative-functional: 1 ($\mathbf{F}_2$)
- Object-oriented-functional: 6 ($\mathbf{F}_2$, f_{α_A}, f_{ω_A}, f_{ω_B}, f_{θ_A}, and f_{θ_B})
- Object-oriented-non-functional: 4 (f_{α_A}, f_{ω_A}, $f_{\omega\theta_A}$, and $f_{\omega\theta_B}$)
- Declarative-object-oriented: 6 (simulations for dynamic variables α_A, α_B, ω_A, ω_A, θ_A, and θ_B)
- Declarative-functional: 25 (determined as the number of unique paths [86] in the constraint hypergraph, each of which form a valid simulation and can be declaratively called.)

The equality between the worst cases, strictly imperative and imperative-functional, demonstrates how neither function-based nor declarative models imply the other. The added structure of the object-oriented paradigms means that variables can be immediately assigned to local methods. Five of the behavioral equations (Eq. 3.1–3.4) were local methods, with only f_{α_B} requiring an imperative, inter-class connection. The non-functional paradigm as specified in Block 3.4 did not lead to a valid definition of

$\mathbf{F}_2$, hence its lower scoring.

The evaluation for the declarative-object-oriented framework (Modelica) is only an estimate, since the simulation solver is not expressed as part of the model. However, because Modelica's solvers must solve the entire system at once, the result of simulating a system model is information for all the dynamic variables. This was accounted as six distinct simulations for the six dynamic variables in the model.

The most expressive framework is the declarative-functional constraint hypergraph. The graphical arrangement gives the maximal expression of the double pendulum's behavior possible given Eq. 3.1–3.4. This is evidenced by the number of simulations that can be autonomously formed from the model structure. Note that such declarativity requires a global system state comprised of a flat system representation. All variables and functions in the constraint hypergraph are treated as first-class citizens [48]. The lack of encapsulation means that interactions between system phenomena (such as Δt on θ_A) can be expressed by the model, as well as provisioned by the autonomous solver.

3.7. Discussion

The pendulum example above was specifically tailored to highlight the differences with simulation between paradigms, as tabulated in Table 3.3. By showing the models, the strengths and weaknesses of each paradigm become more evident. The primary weakness of imperative models is the need to manually define a unique simulation process for every pairing of inputs and outputs. Although this enables simulation without requiring a more general model, the resulting simulation is typically ignorant of the system structure. The result is simulations that are difficult to adapt and reuse as the system scope changes. This is exacerbated if the state transformations are not expressed as functions, as procedural mutations cannot be easily rearranged. The fact that system behaviors are fundamentally expressed as functions begets the importance of functional frameworks for system modeling. Significantly, this does not preclude the use of object-oriented frameworks as long as the resulting models adhere to functional practices, as described in Section 3.3.3. Modelica is an example of an object-oriented paradigm enforcing functional rules.

Object-oriented frameworks have been lauded as the cure to system interoperability [72], and certainly the use of encapsulated objects increases the ease of development and program modularity [70].

Though imperative couplings are the default method for structuring object-oriented models, objects can be defined with purely declarative connections. However, the added requirements of maintaining functional inter-system relationships can make it more difficult to create these extensible object-oriented models. The benefits of natural system decomposition are maximized when describing static systems, where declarativity is less important. Software systems are an example, where the operating environment for a software application is unlikely to change significantly over the life of the program. Natural systems, on the other hand, exhibit a wide variety of ever-evolving behaviors. Scientists attempting to study these heterogeneous systems should be wary of imperatively defined models [69].

This becomes especially apparent with data-hiding, which is promoted in software development to help distinguish model independence and prevent inadvertent tampering [87]. However, for more reactive models, data hiding can cause inadvertent problems with understanding emergent behavior. For instance, doubly inheriting the `Pendulum` class in Block 3.2 causes variables Δt and g to be instantiated twice. While it might be apparent to a modeler that these should be equivalent between the `DrivenPendulum` and `SwungPendulum` classes, this must be manually specified to avoid redefinition. In other words, the private sense of state prevented the object-oriented framework from declaratively capturing the behavior of the system.

Ultimately, languages may be able to express models described in multiple paradigms. MATLAB for instance can handle functional models as well as class definitions, despite its use here as an imperative, non-functional framework. However, the orientation of a language (C++ towards objects, block diagrams toward imperative processes) greatly influences the ability of a modeler to create acceptable simulation models. In this regard the authors are encouraged by the use of constraint hypergraphs, which may prove instrumental in exposing system behavior and enabling general simulation due to their functional and declarative nature.

3.7.1. Future Work

The expression problem discussed in Section 3.3.4 is not one-sided, it is also true that functional models have difficulty expressing new data types when the scope of the system they represent expands. This is principally a problem of abstraction. For instance, any of the system representations in Section 3.5 would be unable to interoperate with a model of bacterial growth on the bob due to a lack

of shared parameters even if the two representations influenced each other behaviorally. This occurs when the level of abstraction of a model omits the specific function outputs needed to couple with another system. Though functional, declarative models gracefully enable simulation across a system, there are as yet no methods for determining whether they can correctly express system behavior after coupling with only the information contained in the model *a priori*. This remains the greatest challenge with system interoperability. Its solution has been called for especially in conjunction with providing platforms for digital twins [88, 89] and model-based systems engineering [4].

3.8. Conclusion

This paper is exploratory in nature, laying the groundwork for how the framework a model is developed under influences its utilization and especially its simulation. The principle differences discussed were between declarative and imperative modeling paradigms. These were compared by simulating the same system in different frameworks and observing each model's ability to express system behavior and form comprehensive simulation processes. The primary insight from the example was that imperative models do not account for the behavior of a system, and consequently require simulation processes to be provided by an external modeler. Declarative models embed the system structure so that simulation processes can be automatically discovered from their construction. Additional observations were made between object-oriented frameworks, which break a system into more modular subsystems, and functional frameworks where system behavior is represented solely by algebraic functions. Both of these have strengths in representing systems, though the authors claim functional frameworks have significant advantages in exposing the behavior between highly interactive systems, although this can be largely reclaimed by coupling objects declaratively rather than procedurally.

All of these observations are based on the notion that system simulation is ultimately the process of arranging functions so that their composition transforms a set of inputs to an output. Because of this, each modeling framework is differentiated by its process of identifying and composing these functions. Frameworks that expose system behavior ultimately are easier to adapt as the system scope changes, an important consideration in the modeling of complex systems.

References

[1] 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.

[2] John Morris, Gregory Mocko, and John Wagner. "Effects of Functional and Declarative Modeling Frameworks on System Simulation". The 5th Modeling, Estimation and Control Conference (MECC 2025). Pittsburgh, PA: IFAC, Oct. 7, 2025.

[3] Michael J. Beeson. "Towards a Computation System Based on Set Theory". *Theoretical Computer Science* 60.3 (Dec. 1988), pp. 297–340. ISSN: 03043975. DOI: 10.1016/0304-3975(88)90115-6.

[4] INCOSE. *Systems Engineering Vision 2035*. INCOSE, 2021. https://www.incose.org/2021-redesign/load-test/systems-engineering-vision-2035 (visited on 02/19/2025).

[5] Gregory L. Baker and James A. Blackburn, eds. *The Pendulum: A Case Study in Physics*. Oxford: Oxford University Press, 2010. 288 pp. ISBN: 978-0-19-856754-7.

[6] Boris Eisenbart, Kilian Gericke, and Luciënne Blessing. "An Analysis of Functional Modeling Approaches Across Disciplines". *AI EDAM* 27.3 (Aug. 2013), pp. 281–289. ISSN: 0890-0604, 1469-1760. DOI: 10.1017/S0890060413000280.

[7] Gerhard Pahl et al. *Engineering Design: A Systematic Approach*. Ed. by Ken Wallace and Luciënne Blessing. 3rd ed. London: Springer, 2007. 617 pp. ISBN: 978-1-84628-318-5.

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

[9] Richard Bird and Philip Wadler. *Introduction to Functional Programming*. Prentice Hall International Series in Computer Science. New York: Prentice Hall, 1988. 293 pp. ISBN: 978-0-13-484189-2.

[10] Albert Wayne Wymore. *Model-Based Systems Engineering: An Introduction to the Mathematical Theory of Discrete Systems and to the Tricotyledon Theory of System Design*. Systems Engineering Series. Boca Raton, Fla.: CRC Press, 1993. 710 pp. ISBN: 978-0-8493-8012-9.

[11] Bernard P. Zeigler, Alexandre Muzy, and Ernesto Kofman. *Theory of Modeling and Simulation*. Third. Elsevier, 1976. ISBN: 978-0-12-813370-5. DOI: 10.1016/B978-0-12-813370-5.00002-X.

[12] W. Ross Ashby. *An Introduction to Cybernetics*. Internet. London: Chapman & Hall, 1956. http://pcp.vub.ac.be/books/IntroCyb.pdf (visited on 02/27/2025).

[13] Cláudio Gomes et al. "Co-Simulation: A Survey". *ACM Comput. Surv.* 51.3 (May 23, 2018), 49:1–49:33. ISSN: 0360-0300. DOI: 10.1145/3179993.

[14] Jerry Banks. "Introduction to Simulation". *Proceedings of the 31st Conference on Winter Simulation: Simulation—a Bridge to the Future - Volume 1*. WSC '99. New York, NY, USA: Association for Computing Machinery, Dec. 1, 1999, pp. 7–13. ISBN: 978-0-7803-5780-8. DOI: 10.1145/324138.324142.

[15] C.J.J. Paredis et al. "Composable Models for Simulation-Based Design". *EWC* 17.2 (July 1, 2001), pp. 112–128. ISSN: 1435-5663. DOI: 10.1007/PL00007197.

[16] John C. Baez and Blake S. Pollard. "A Compositional Framework for Reaction Networks". *Rev. Math. Phys.* 29.09 (Oct. 2017), p. 1750028. ISSN: 0129-055X. DOI: 10.1142/S0129055X17500283.

[17] Brendan Fong. "The Algebra of Open and Interconnected Systems". PhD thesis. Oxford University: arXiv, Sept. 17, 2016. DOI: 10.48550/arXiv.1609.05382. arXiv: 1609.05382.

[18] Israel N. Herstein. *Topics in Algebra*. 1st ed. Waltham, MA: Blaisdell Publishing Company, 1964. ISBN: 978-0-536-00257-0.

[19] Peter Fritzson. *Introduction to Modeling and Simulation of Technical and Physical Systems with Modelica*. Piscataway, NJ: IEEE Press, Nov. 2, 2011. ISBN: 978-1-118-01068-6. DOI: 10.1002/9781118094259.

[20] Christopher Strachey. "Fundamental Concepts in Programming Languages". *Higher-Order and Symbolic Computation* 13.1 (Apr. 1, 2000), pp. 11–49. ISSN: 1573-0557. DOI: 10.1023/A:1010000313106.

[21] 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.

[22] Spencer Breiner, Peter O. Denno, and Eswaran Subrahmanian. "Categories for Planning and Scheduling". *NIST* 67.11 (Dec. 1, 2020). DOI: 10.1090/noti2186.

[23] Jan Willem Polderman and Jan C. Willems. "Dynamical Systems". *Introduction to Mathematical Systems Theory: A Behavioral Approach*. Vol. 26. Texts in Applied Mathematics. New York, NY: Springer, 1998, pp. 1–25. ISBN: 978-1-4757-2953-5. DOI: 10.1007/978-1-4757-2953-5_1.

[24] Rajarishi Sinha et al. “Modeling and Simulation Methods for Design of Engineering Systems”. *J. Comput. Inf. Sci. Eng* 1.1 (Mar. 1, 2001), pp. 84–91. ISSN: 1530-9827. DOI: 10.1115/1.1344877.

[25] Paul Pichler et al. “Imperative versus Declarative Process Modeling Languages: An Empirical Investigation”. *Business Process Management Workshops*. Ed. by Florian Daniel, Kamel Barkaoui, and Schahram Dustdar. Vol. 99. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 383–394. ISBN: 978-3-642-28107-5. DOI: 10.1007/978-3-642-28108-2_37.

[26] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. 17. print. The MIT Electrical Engineering and Computer Science Series. Cambridge: MIT Pr. [u.a.], 1993. 542 pp. ISBN: 978-0-262-01077-1.

[27] Dana Scott. “Mathematical Concepts in Programming Language Semantics”. *Proceedings of the November 16-18, 1971, Fall Joint Computer Conference on - AFIPS '71 (Fall)*. The November 16-18, 1971, Fall Joint Computer Conference. Las Vegas, Nevada: ACM Press, 1971, p. 225. DOI: 10.1145/1478873.1478903.

[28] Wolfgang Borutzky, ed. *Bond Graph Modelling of Engineering Systems: Theory, Applications and Software Support*. New York, NY: Springer, 2011. ISBN: 978-1-4419-9367-0. DOI: 10.1007/978-1-4419-9368-7.

[29] Henry M. Paytner. *The Gestation and Birth of Bond Graphs*. HMP and Bond Graphs. 2000. https://sites.utexas.edu/longoria/files/2020/10/Birth_of_-Bond_Graphs.pdf (visited on 06/17/2024).

[30] William J. Palm. “Block Diagrams, State-Variable Models, and Simulation Methods”. *System Dynamics*. Third edition. New York, NY: McGraw-Hill Science, 2014, pp. 250–318. ISBN: 978-0-07-339806-8. https://highered.mheducation.com/sites/0073398063/information_center_view0/.

[31] Michael Tiller. “Block Diagrams vs. Acausal Modeling”. *Introduction to Physical Modeling with Modelica*. Ed. by Michael Tiller. Boston, MA: Springer US, 2001, pp. 255–264. ISBN: 978-1-4615-1561-6. DOI: 10.1007/978-1-4615-1561-6_11.

[32] Irving Fisher. “What Is Capital?” *The Economic Journal* 6.24 (1896), pp. 509–534. ISSN: 0013-0133. DOI: 10.2307/2957184. JSTOR: 2957184.

[33] John Baez et al. “Compositional Modeling with Stock and Flow Diagrams”. *Proceedings Fifth International Conference on Applied Category Theory*. Fifth International Conference on Applied Category Theory (ACT2023). Vol. 380. Electronic Proceedings in Theoretical Computer Science. Glasgow, United Kingdom: Open Publishing Association, Aug. 7, 2023, pp. 77–96. DOI: 10.4204/EPTCS.380.5. arXiv: 2205.08373,.

[34] Peter Pin-Shan Chen. “The Entity-Relationship Model—toward a Unified View of Data”. *ACM Trans. Database Syst.* 1.1 (Mar. 1, 1976), pp. 9–36. ISSN: 0362-5915. DOI: 10.1145/320434.320440.

[35] Frank B. Gilbreth and Lillian M. Gilbreth. “Process Charts”. Annual Meeting of The American Society of Mechanical Engineers. New York: American Society of Mechanical Engineers, Dec. 5, 1921. https://web.archive.org/web/20150509222833/https://engineering.purdue.edu/IE/GilbrethLibrary/gilbrethproject/processcharts.pdf (visited on 09/25/2024).

[36] Paul Wach and Alejandro Salado. “Can Wymore's Mathematical Framework Underpin SysML? An Initial Investigation of State Machines”. *Procedia Computer Science* 153 (2019), pp. 242–249. ISSN: 18770509. DOI: 10.1016/j.procs.2019.05.076.

[37] Oliver Chukwudi Ibe. *Markov Processes for Stochastic Modeling*. 2nd edition. Elsevier Insights. London: Elsevier, 2013. ISBN: 978-0-12-407795-9.

[38] Henry Laurence Gantt. *Work, Wages, and Profits*. Engineering Magazine, 1913. 332 pp. Google Books: bncAtMf2EpkC.

[39] US Department of the Navy. *Program Evaluation Research Task Summary Report Phase 1*. Washington DC: Government Printing Office, July 1958. https://web.archive.org/web/20151112203807/http://www.dtic.mil/dtic/tr/fulltext/u2/735902.pdf (visited on 06/17/2023).

[40] Rónán Daly, Qiang Shen, and Stuart Aitken. “Learning Bayesian Networks: Approaches and Issues”. *The Knowledge Engineering Review* 26.2 (May 2011), pp. 99–157. ISSN: 1469-8005, 0269-8889. DOI: 10.1017/S0269888910000251.

[41] Michael G. Kapteyn, Jacob V. R. Pretorius, and Karen E. Willcox. “A Probabilistic Graphical Model Foundation for Enabling Predictive Digital Twins at Scale”. *Nat Comput Sci* 1.5 (May 2021), pp. 337–347. ISSN: 2662-8457. DOI: 10.1038/s43588-021-00069-0.

[42] Judea Pearl. *Causality*. Cambridge University Press, Sept. 14, 2009. 487 pp. ISBN: 978-0-521-89560-6. Google Books: f4nuexsNVZIC.

[43] Object Modeling Group. *OMG Systems Modeling Language (OMG SysML)*. Version 1.0. Needham, MA, Sept. 1, 2007. https://www.omg.org/spec/SysML/1.0/PDF (visited on 09/21/2024). Formal.

[44] International Organization for Standardization. *ISO 10303-11*. Version 2. Nov. 2004. https://www.iso.org/standard/38047.html (visited on 03/06/2025). Published.

[45] 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.

[46] Rina Dechter. “Constraint Networks”. *Encyclopedia of Artificial Intelligence*. Ed. by Stuart C. Shapiro. 2nd ed. Vol. 1. New York: John Wiley & Sons Inc, Jan. 1992, pp. 276–285.

[47] Christophe Lecoutre. *Constraint Networks: Targeting Simplicity for Techniques and Algorithms*. John Wiley & Sons, Mar. 1, 2013. 461 pp. ISBN: 978-1-118-61791-5.

[48] 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.

[49] John C. Mitchell. *Concepts in Programming Languages*. Cambridge: Cambridge University Press, 2003. ISBN: 978-0-521-78098-8.

[50] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, Feb. 20, 2004. 944 pp. ISBN: 978-0-262-22069-9. Google Books: _bmyEnUnfTsC.

[51] Luciano Floridi. *Information: A Very Short Introduction*. Very Short Introductions. Oxford: Oxford University Press, 2010. 116 pp. ISBN: 978-0-19-955137-8.

[52] Michael J. O’Donnell. *Computing in Systems Described by Equations*. Lecture Notes in Computer Science 58. Berlin ; New York: Springer-Verlag, 1977. 111 pp. ISBN: 978-0-387-08531-9.

[53] Maja Pešić. “Constraint-Based Workflow Management Systems: Shifting Control to Users”. PhD thesis. Eindhoven, Netherlands: Technische Universiteit Eindhoven, Jan. 1, 2008. DOI: 10.6100/IR638413.

[54] Chris Reade. *Elements of Functional Programming*. International Computer Science Series. Wokingham, England ; Reading, Mass: Addison-Wesley, 1989. 600 pp. ISBN: 978-0-201-12915-1.

[55] Russell S. Peak et al. “Simulation-Based Design Using SysML Part 1: A Parametrics Primer”. *INCOSE International Symp* 17.1 (June 2007), pp. 1516–1535. ISSN: 2334-5837, 2334-5837. DOI: 10.1002/j.2334-5837.2007.tb02964.x.

[56] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A Practical Guide to SysML: The Systems Modeling Language*. 3rd ed. Waltham: Elsevier, 2015. ISBN: 978-0-12-800202-5.

[57] Martin Otter and Hilding Elmqvist. “Modelica”. *Simulation News Europe* 10.29/30 (Dec. 2000), pp. 3–8. https://www.sne-journal.org/fileadmin/user_upload_sne/SNE_Issues_OA/SNE_10/sne.10.29-30.pdf (visited on 02/25/2025).

[58] J. Hughes. “Why Functional Programming Matters”. *The Computer Journal* 32.2 (Feb. 1, 1989), pp. 98–107. ISSN: 0010-4620, 1460-2067. DOI: 10.1093/comjnl/32.2.98.

[59] György Egon Révész. *Lambda-Calculus Combinators, and Functional Programming*. Cambridge Tracts in Theoretical Science 4. Cambridge: Cambridge University Press, 1988. ISBN: 978-0-521-34589-7.

[60] Gérard Huet, ed. *Logical Foundations of Functional Programming*. The UT Year of Programming Series. Reading, Mass.: Addison-Wesley, 1990. 491 pp. ISBN: 978-0-201-17234-8.

[61] Kristopher Brown, Tyler Hanks, and James Fairbanks. *Compositional Exploration of Combinatorial Scientific Models*. June 7, 2022. DOI: 10.48550/arXiv.2206.08755. arXiv: 2206.08755 [cs, math]. Pre-published.

[62] Julian Hedges. “Towards Compositional Game Theory”. PhD thesis. London: Queen Mary University of London, Jan. 16, 2018. http://qmro.qmul.ac.uk/xmlui/handle/123456789/23259 (visited on 09/10/2024).

[63] David I. Spivak. *The Steady States of Coupled Dynamical Systems Compose According to Matrix Arithmetic*. Dec. 2, 2015. DOI: 10.48550/arXiv.1512.00802. arXiv: 1512.00802. Pre-published.

[64] Matthias Zenger. “Programming Language Abstractions for Extensible Software Components”. PhD thesis. Lausanne, Switzerland: EPFL, 2004. DOI: 10.5075/epfl-thesis-2930.

[65] Claus Ballegaard Nielsen et al. “Systems of Systems Engineering: Basic Concepts, Model-Based Techniques, and Research Directions”. *ACM Comput. Surv.* 48.2 (Sept. 24, 2015), 18:1–18:41. ISSN: 0360-0300. DOI: 10.1145/2794381.

[66] 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.

[67] Hendrik Pieter Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics 103. Amsterdam New York Oxford: North-Holland, 1981. ISBN: 978-0-444-85490-2.

[68] David I. Spivak. *Category Theory for Scientists*. Sept. 18, 2013. DOI: 10.48550/arXiv.1302.6946. arXiv: 1302.6946. Pre-published.

[69] 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.

[70] William R. Cook. “Object-Oriented Programming versus Abstract Data Types”. Foundations of Object-Oriented Languages. Ed. by J. W. de Bakker, W. P. de Roever, and G. Rozenberg. Vol. 489. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1991, pp. 151–178. ISBN: 978-3-540-46450-1. DOI: 10.1007/BFb0019443.

[71] Lewis J. Pinson and Richard S. Wiener. *An Introduction to Object-Oriented Programming and Smalltalk*. Reading, Mass.: Addison-Wesley, 1988. 502 pp. ISBN: 978-0-201-19127-1.

[72] Daniel R. Dolk and Jeffrey E. Kottemann. “Model Integration and a Theory of Models”. *Decision Support Systems. Model Management Systems* 9.1 (Jan. 1, 1993), pp. 51–63. ISSN: 0167-9236. DOI: 10.1016/0167-9236(93)90022-U.

[73] D. L. Parnas. “On the Criteria to Be Used in Decomposing Systems into Modules”. *Commun. ACM* 15.12 (Dec. 1, 1972), pp. 1053–1058. ISSN: 0001-0782. DOI: 10.1145/361598.361623.

[74] Tim Rentsch. “Object Oriented Programming”. *SIGPLAN Not.* 17.9 (Sept. 1982), pp. 51–57. ISSN: 0362-1340, 1558-1160. DOI: 10.1145/947955.947961.

[75] D Harel and A Pnueli. “Reactive Systems”. Logics and Models of Concurrent Systems. Ed. by K Apt. Vol. 13. NATO ASI Series. Berlin, Heidelberg: Springer, Jan. 1985. https://link.springer.com/chapter/10.1007/978-3-642-82453-1_17 (visited on 09/17/2024).

[76] Edward Lorenz. “Predictability: Does the Flap of a Butterfly’s Wings in Brazil Set Off a Tornado in Texas?” 39th Meeting of the American Association for the Advancement of Science (Washington DC). Dec. 29, 1972. https://mathsciencehistory.com/wp-content/uploads/2020/03/132_kap6_lorenz_artikel_the_butterfly_effect.pdf (visited on 02/28/2025).

[77] Cláudio Gomes et al. “Semantic Adaptation for FMI Co-Simulation with Hierarchical Simulators”. *SIMULATION* 95.3 (Mar. 1, 2019), pp. 241–269. ISSN: 0037-5497. DOI: 10.1177/0037549718759775.

[78] Christian Bertsch, Elmar Ahle, and Ulrich Schulmeister. “The Functional Mockup Interface - Seen from an Industrial Perspective”. The 10th International Modelica Conference, March 10-12, 2014, Lund, Sweden. Mar. 10, 2014, pp. 27–33. DOI: 10.3384/ecp1409627.

[79] Christian Andersson. “Methods and Tools for Co-Simulation of Dynamic Systems with the Functional Mock-up Interface”. Doctoral Dissertation in Mathematical Sciences. Lund, Sweden: Lund University, May 4, 2016. ISBN: 978-91-7623-697-0. https://www.maths.lth.se/na/staff/chria/phdthesis.pdf (visited on 03/12/2024).

[80] Marcus Wiens, Tobias Meyer, and Philipp Thomas. “The Potential of FMI for the Development of Digital Twins for Large Modular Multi-Domain Systems”. *Proceedings of the 14th International Modelica Conference*. 14th International Modelica Conference. Linköping, Sweden, Sept. 27, 2021, pp. 235–240. DOI: 10.3384/ecp21181235.

[81] Shriram Krishnamurthi, Matthias Felleisen, and Daniel P. Friedman. “Synthesizing Object-Oriented and Functional Design to Promote Re-Use”. *Proceedings of the 12th European Conference on Object-Oriented Programming*. 12th European Conference on Object-Oriented Programming. Vol. 1445. ECCOP ’98. Berlin, Heidelberg: Springer-Verlag, July 20, 1998, pp. 91–113. ISBN: 978-3-540-64737-9. DOI: 10.1007/BFb0054088.

[82] John C. Reynolds. “User-Defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction”. *Programming Methodology: A Collection of Articles by Members of IFIP WG2.3*. Ed. by David Gries. New York, NY: Springer, 1978, pp. 309–317. ISBN: 978-1-4612-6315-9. DOI: 10.1007/978-1-4612-6315-9_22.

[83] Philip Wadler. *The Expression Problem*. E-mail. Nov. 12, 1998. https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt (visited on 01/27/2025).

[84] 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.

[85] François E. Cellier and Ernesto Kofman. *Continuous System Simulation*. New York: Springer, 2006. 643 pp. ISBN: 978-0-387-26102-7. DOI: 10.1007/0-387-30260-3.

[86] Claude Berge. *Graphs and Hypergraphs*. Trans. by Edward Minieka. North-Holland Mathematical Library 6. Amsterdam, New York: North-Holland Pub. Co., 1973. 528 pp. ISBN: 978-0-444-10399-4.

[87] Craig Larman. “Protected Variation: The Importance of Being Closed”. *IEEE Software* (May 2001).

[88] 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.

[89] 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).

CHAPTER 4

Integrating Software with Multiphysics

This chapter is based on the paper "Declarative Integration of CAD Software into Multi-Physics Simulation via Constraint Hypergraphs," which was originally presented at the 2025 ASME International Design Engineering Technical Conferences and Computers and Information in Engineering Conference (IDETC-CIE) in Anaheim, CA [1] as is currently under review with the ASME Journal of Computing and Information Science in Engineering.

Abstract: Declarative modeling frameworks, such as Modelica, are often used to represent systems that require reusable and interoperable models. Simulation in such a framework requires a solver that is capable of transforming the model into an executable process. However, most declarative solvers are insular, in that they are unable to integrate with software applications needed for simulating complex systems, limiting their usability for simulating multi-domain models. This paper describes a process for creating declarative models that can integrate with an external tool by deconstructing its Application Programmer Interface (API) into a set of functions arranged into a constraint hypergraph. The constraint hypergraph's solver is shown to be capable of automatically parsing these functions to simulate arbitrary pairs of inputs and outputs. The result is a holistic modeling framework that allows for flexible simulation of a complex system, integrates directly with otherwise sequestered platforms, and reveals cross-cutting interactions between system elements. This is demonstrated by integrating the solid modeling capabilities of Onshape with a dynamic model of a crankshaft from a piston engine, showing how a geometric model can be integrated with independently-defined dynamic models. This validates the framework on a limited scale, setting the foundation for work for fully integrating disparate tools into multi-domain, multi-physics modeling and simulation.

4.1. Introduction

Modeling a complex system is a difficult task–a trivial observation of a non-trivial problem. Engineers must synthesize the coupled effects of hundreds or thousands of different parameters, reconcile disharmonious experimental observations, and handle the uncertainty glowering over every assumed fact. Adding to this is the work of integrating the isolated software tools that perform the required high-fidelity, multi-physics calculations of modern simulation [2]. Traditional system modeling frameworks create simulations by defining exact workflows that prescribe the order in which information should be passed between applications. This results in simulations that only present one perspective of the base system–a singular description of how a system can behave.

In contrast to the inflexibility of procedural simulation, this paper discusses a declarative framework for systems modeling and simulation, previously introduced in [3]. Termed a constraint hypergraph (CHG), this framework reduces a system to a set of state variables related by mathematical functions. Each function shows how one variable evolves in response to changes in other variables, in effect mapping a set of inputs to an output. A modeler identifies which tools are to be used for calculating these rules; for example, using a geometric modeling tool to calculate the mass of a solid body.

The collection of variables connected by functions forms a hypergraph, whose paths correspond to valid mappings between inputs and outputs. While a traditional simulation framework defines a single process for simulating an unknown value, the CHG defines all known processes as a cohesive model. These processes can be extracted for any connected pairing of input and output nodes, with the functions connecting them describing the series of calculations composing the simulation. CHGs additionally provide mechanisms allowing for the autonomous construction of a simulation process, providing far greater flexibility and interoperability when modeling and simulating complex systems.

The objective of this paper is to describe how declarative system models can be formed and simulated across independent simulation software by integrating them via a CHGs. When using a CHG the mechanisms of integration remain unchanged from standard methods, primarily involving calls of the Application-Programmer Interface (API) [4]. Instead of proposing an updated interface between models and software, CHGs describe how software can be reconciled into a single, strongly-coupled model. These declarative models allow modelers to focus on how a system behaves rather than how it will be

simulated.

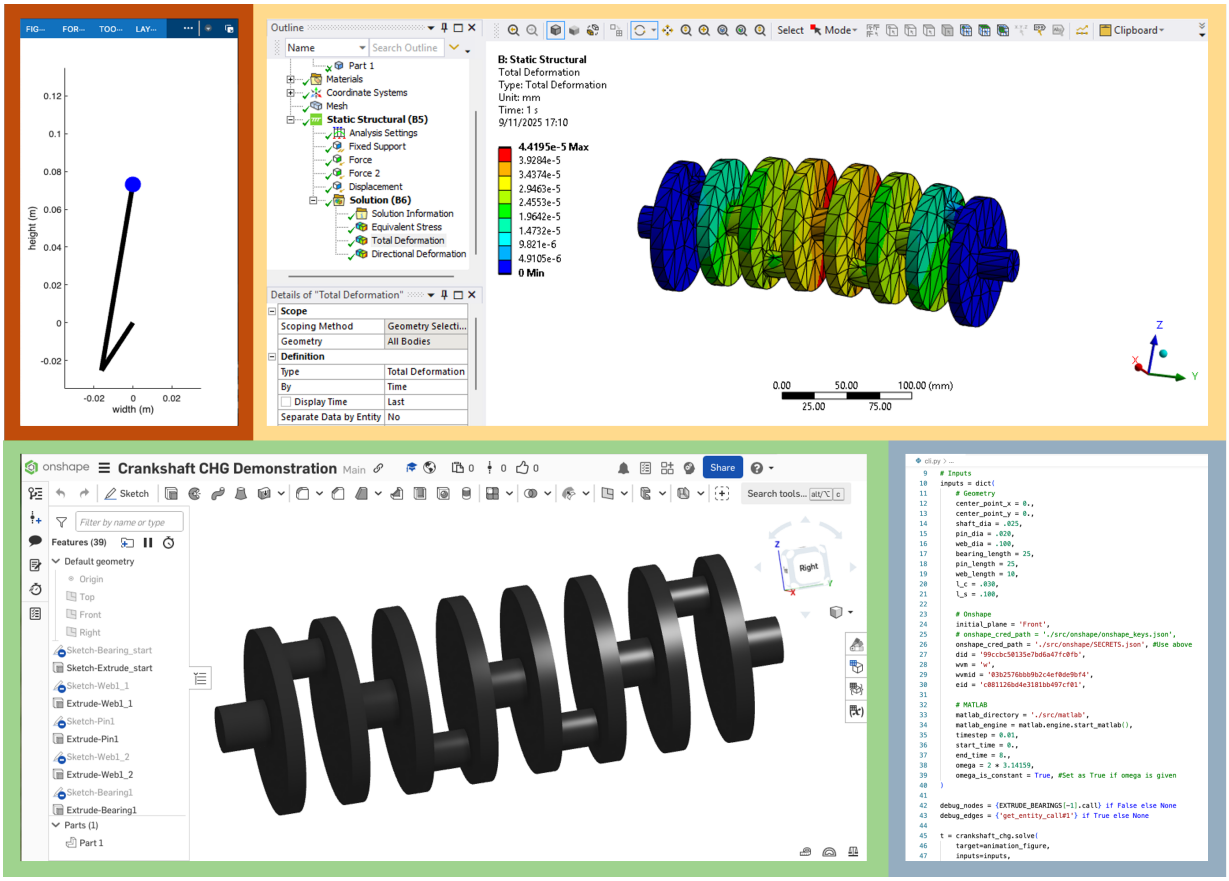


Figure 4.1: Overview of simulation declaratively integrated across four platforms (clockwise from top left): MATLAB (kinematic analysis), Ansys Mechanical (FEA), Python (general purpose), and Onshape (solid modeling).

These claims are demonstrated by integrating a solid model of the crankshaft with a representation of the dynamics of a piston engine. The system model unifies aspects of the crankshafts dynamic behavior with the mass properties defined by its geometry. As shown in Figure 4.1, a solid body model is generated by coupling the model with Onshape, a cloud-based Computer-Aided Design (CAD) platform. The system model allows for full generation of the solid-body model for a variety of different inputs. These outputs are then processed using MATLAB to reveal the cross-cutting behavioral interactions between the crankshaft's mass and kinematic motion. The system finally calculates load information using Ansys Mechanical to perform Finite-Element Analysis (FEA). To the authors' knowledge, this is the first time a fully declarative modeling framework has been used to integrate engineering applications such as CAD and FEA together in system simulation. To better focus on this integration, the

authors have employed models of a piston engine that vary in their validity and base assumptions. By so doing the authors have attempted to demonstrate the methods of declarative system integration using a CHG, rather than document the practical characterization of a slider-crank mechanism.

4.2. Review of System Modeling and Simulation

A system is an arrangement of things, such that the things exhibit some specific behavior [5]. The work of an engineer or decision-maker in any field is to provide a system whose behavior achieves a specific value-adding objective [6]. Similarly, the goal of a scientist is to characterize the behavior of the complex, interconnected system that is the universe [7]. From both cases it can be seen that nearly all human operations require some method for understanding the behavior of a system [8], which is referred to here as a model without digressing into the ontological definitions of modeling. Whether a model is a simplified analog or informational structure, its purpose is to enable a user to understand a more complex system [9].

4.2.1. System Modeling

What makes a plurality of things a system is their interactions [10]. As a foil for considering the nature of these relations, consider initially a group of things that do not interact. In such a collection, the behavior of each individual thing is independent of the rest of the group. Consequently, the rest of the group is not needed to understand the behavior of any individual thing, and can be ignored. This motivates the definition of a system as things that interact: ignoring any individual thing in the system prevents a modeler from understanding the behavior of the system as a whole. From this it can be understood that the behavior of a system is a characterization of how all the individual things, or elements, in the system effect each other.

To describe all the interactions of a system, a modeler must first have some sense for what it means for an element to be affected. The evolutions of an element occur over some phenomenon that is exhibited by the element and identified by an observer. By assigning unique values to the phenomenon, an observer can distinguish between changes in the element [11]. Knowing, for example, the difference between something *rotating* or *not-rotating* allows an observer to distinguish how running a piston engine influences the state of the crank shaft. The set of all values that an element might exhibit for a certain

phenomenon is a variable. A system can be entirely characterized by identifying a specific datum for every variable associated with the system [12]. All the values expressed concurrently comprise a system's state, with the system interactions describing the evolutions of a system's state between distinct frames of consideration.

The evolution of a system's state constitutes the system's behavior [13]. This was defined by Willems, who showed that system behavior can be represented as a set of constraints describing the affect the state of a system has on a single variable [14]. Each constraint can be described by a function mapping between two sets: one set corresponding to the variable being affected, and another of the values affecting it [15] (the word function here is denotes a mathematical morphism [16] rather than a role of a system as used in design theory [17]). Comparing this with the original definition of a system, it can be said that to describe a system, a model must be able to express all the variables comprising the system's state, as well as the functions that describe how those variables are related.

4.2.2. System Simulation

The whole purpose of system modeling is to allow information contained in a model to be extracted and used by some agent [8]. This is generally referred to as simulation, with an objective of identifying the value of at least one state variable without needing to observe the variable through experimentation [18]. In practice, this can be achieved only because the functions defining the system behavior constrain the variables being simulated. Consequently, simulation is explicitly the process of using functions to calculate the value of an unknown variable, creating a computable chain (or *trace* [19]) connecting some known inputs to the unknown outputs [20, 21].

For instance, consider the following kinematic model of a slider-crank mechanism:

$$\theta = \cos^{-1}\left(\frac{y^2 + l_c^2 - l_s^2}{2yl_c}\right) \tag{4.1}$$

$$y = l_c \cos\theta + \sqrt{l_s^2 - l_c^2 \sin^2\theta} \tag{4.2}$$

$$\omega = \frac{d\theta}{dt} \tag{4.3}$$

$$\dot{y} = \frac{dy}{dt} \tag{4.4}$$

where y is the distance of the tip of slider from the center of rotation, θ is the angle of rotation of the crank, and l_s and l_c are the lengths of the slider and crank arms respectively, as shown in Figure 4.2. The velocities of the system $\dot{y}$ and ω are the derivatives of y and θ. This model is expressed as a set of equations, not functions. For simulation to occur, an agent must manipulate the model to discover a chain of functions mapping inputs to outputs. For instance, if ω was given as a constant value, then y could be solved for by calculating the following functions:

$$f(\omega) \to \theta \coloneqq \int \omega dt \tag{4.5}$$

$$g(\theta, l_c, l_s) \to y \coloneqq l_c \cos\theta + \sqrt{l_s^2 - l_c^2 \sin^2\theta} \tag{4.6}$$

Because the domain of g is given by the codomain of f (and inputs l_c and l_s), y can be calculated as the composition of functions $g \circ f(\omega)$. This demonstrates how simulation is accomplished by identifying a chain of functions mapping a set of known inputs to the desired outputs. In this paper, these chains of composed functions are referred to as simulation processes [22].

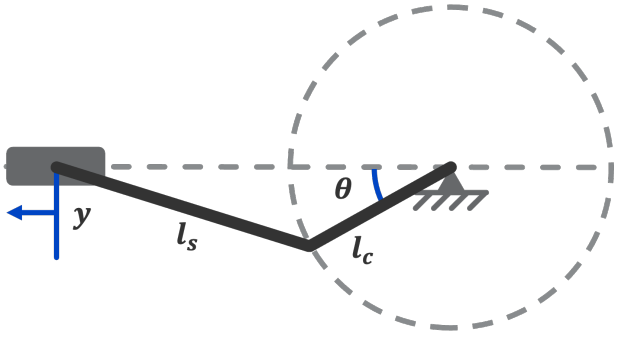


Figure 4.2: Kinematic diagram of a slider-crank mechanism.

Though there are many mechanisms for constructing simulation processes [23], several modeling frameworks (or formalisms [24, 25]) employ similar strategies, affecting how they can be simulated. One such strategy is often referred to as procedural modeling, where models describe only the steps pertaining to a single simulation process rather than the full behavior of the underlying system. These are also termed imperative models, since each expression in the model is a command for how to advance the simulation [26]. Imperative models provide a black-box representation of a system, one that hides

the system behavior inside an opaque process that only connects at its beginning and end [27].

Other modeling frameworks do not prescribe a specific simulation process, but allow for different processes to be constructed by interpreting the model structure. These frameworks are known as declarative, in that they declare the system's structure, leaving the work of assembling simulation processes to a separate mechanism [28, 29]. Equations (4.1–4.4) are declarative, with the simulation functions specified in Eqs. (4.5) and (4.6) generated by an independent agent (in this case a human). Another example of a declarative model is a map, which shows all possible routes between cities. This is contrasted with an imperative model, which would only describe the steps for traveling between two cities.

Declarative models expand the degree of a system that can be simulated. This is not to say that imperative models are limited from a system's scope; both paradigms allow for every state variable to simulated in a simulation. Rather, this statement describes the amount of orders permitted by the modeling framework. An imperative model can only convey a single ordering of behavioral functions. This is demonstrated by the imperative model of a slider-crank mechanism written in MATLAB and shown in Block 4.1:

Block 4.1: Imperative model of a slider-crank, with ω as an input.

```
% Inputs
l_c = 30;
l_s = 100;
timestep = 0.01;
time = 0:timestep:4;
omega = 2*pi;

% Simulation process
theta = zeros(size(time));
for i = 2:length(time)
    theta(i) = theta(i-1) + omega * timestep;
end

y = arrayfun(@(th) piston_height(th,l_c,l_s), theta);

function y = piston_height(th, l_c, l_s)
    y = l_c * cos(th) + sqrt(l_s^2 - l_c^2 * sin(th)^2);
end
```

The model in Block 4.1 represents a single ordering of the model. The simulation depends on an initial input of ω. If a different input were given, say y instead of ω, then the model would need to be completely rewritten. This is shown in Block 4.2, where the model connects an input of y to solve for θ:

Block 4.2: Imperative model of a slider-crank, with y as an input.

```
% Inputs
```

```
l_c = 30;
l_s = 100;
timestep = 0.01;
time = 0:timestep:4;
y = l_c*cos(time*10) + l_s;

% Simulation process
ydot = zeros(size(time));
for i = 2:length(time)
    ydot(i) = (y(i) - y(i-1)) / timestep;
end

th = arrayfun(@(y, ydot) crank_pos(y,ydot,l_c,l_s), y, ydot);
animatePiston(time, y, th, l_c, timestep);

function th = crank_pos(y, ydot, l_c, l_s)
    th = acos((y^2 + l_c^2 - l_s^2) / (2*y*l_c));
    if ydot > 0 %Correct arccos domain
        th = -th;
    end
end
```

These two models in Blocks 4.1 and 4.2 represent the same system with the same behavior, and yet are veritably incompatible. This is the quandary of imperative modeling: because imperative models are not interoperable, a modeler must specify a unique process by which the system is to be simulated for each pairing of input and output. For a system with n state variables, the maximum number of input/output pairings is given as the sum of all possible combinations of multi-variable input sets to a single output variable, or, algebraically:

$$\sum_{i=1}^{n-1}(n-i)\binom{n}{i} \tag{4.7}$$

The exponential growth rate of Eq. 4.7 relative to n, as shown in Figure 4.3, means that it is generally impractical for a modeler to fully describe all the ways a system can be interrogated. For instance, the slider-crank system at hand, which is defined for seven state variables, could potentially be simulated 441 different ways, while that number balloons to over 400 million simulations for a system of 25 variables. This limits imperative frameworks to either simple systems (with few state variables) or to be used with the expectation that only a small subset of behavioral interactions will be represented by the simulation [26].

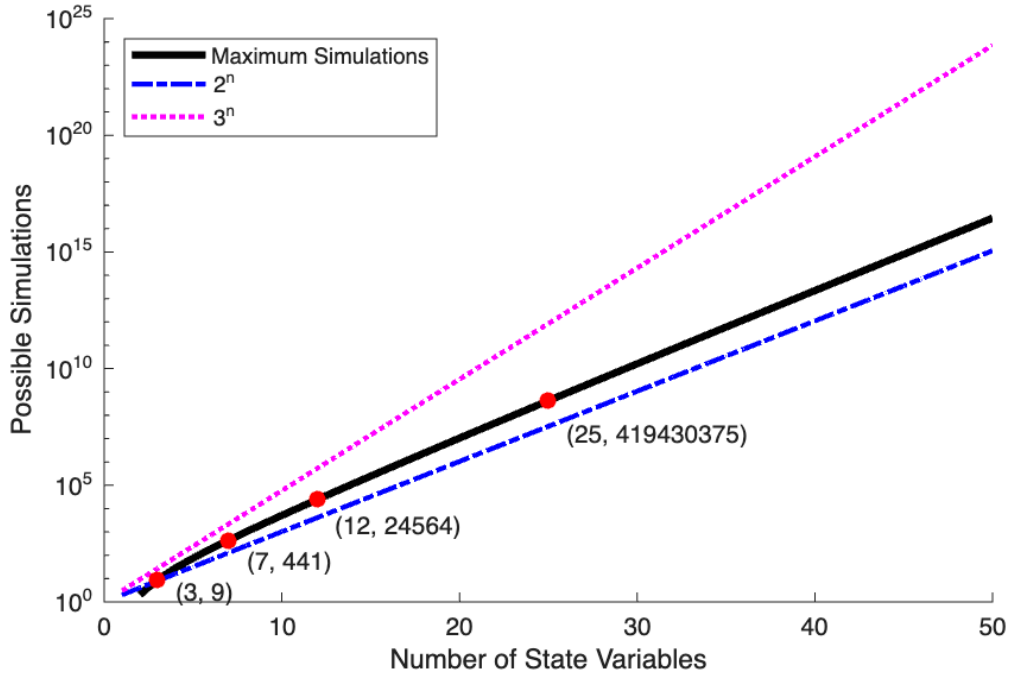


Figure 4.3: Exponential growth of maximum input/output pairings in a system as a function of the number of system variables.

4.3. Declarative Simulation via Constraint Hypergraphs

Declarative frameworks are motivated by a need to fully simulate complex systems, so that all system behaviors are accounted for. A review of how declarative frameworks contrast with imperative paradigms was given previously in [30]. This paper builds upon this review by demonstrating CHGs, a specific declarative formalism introduced in [3].

4.3.1. Overview of Constraint Hypergraphs

A CHG represents a system as a graph, with nodes corresponding to system variables and edges representing the functions that relate them. A CHG is a hypergraph because system functions are often multiple-arity. Variants of CHGs have been employed under various names such as model graphs [31, 32], or categorical sheaves [33, 34]. If the edges of a CHG are limited to unary functions then a CHG becomes similar to a Bayesian network in the sense employed in [35]. A CHG for the kinematic model of the slider-crank given in Blocks 4.1 and 4.2 is shown in Figure 4.4, with variables as circular nodes and the functions of each hyperedge given in the black boxes.

CHGs are particularly adept at handling multi-domain, multi-physics simulations. By represent-

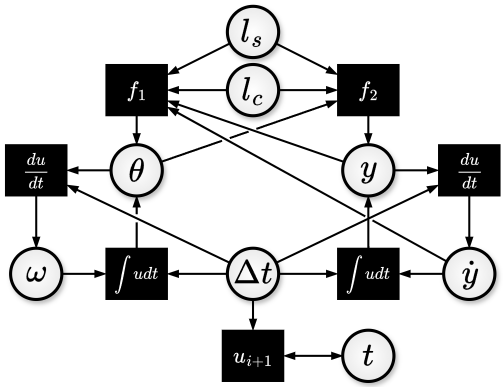


Figure 4.4: CHG model of slider-crank kinematics, with f_1 and f_2 given by Eqs. (4.1) and (4.2).

ing a system as a set of variables connected by functions, the holistic CHG explicitly captures both the system's state and behavior. While CHGs are not well-attuned for describing systems (visually CHGs tend to be busy and difficult to interpret), the decomposition of a system into independent functions allows for automatic construction of simulation processes. This greatly reduces the number of procedural simulations that must be written for a given model, as individual simulation pairings can be derived autonomously from the graph structure. Specific simulations of the systems are then construed as paths through the graph, connecting the nodes representing known inputs to the nodes corresponding to the desired outputs. This is shown in Figure 4.5, depicting two different paths drawn through the CHG from Figure 4.4 representing the simulation process given in Blocks 4.1 and 4.2. Each simulation process starts from a set of inputs (bolded in the figure) and ends on a set of outputs (blue outlines). Because of the multidimensional aspect of the aspects, a path through a hypergraph can be represented as a tree, with the inputs as leaves and a single output as the root, as shown in Figure 4.6.

The reason simulation processes can be formed automatically is due to the composition of functions. A function maps every element of its domain to its codomain. Consequently, two functions whose codomain and domain overlap are guaranteed to compose for all values of the first functions domain. A simulation process is then formed by chaining together consecutive edges into a chain of composing functions where the domain of the first function in the sequence is an input to the simulation, and the codomain of the last function is the desired output. Because of the structure imposed by the CHG formalism, these sequences can be automatically compiled by a pathfinding algorithm such as A* or a Depth-First Search. This capability for autonomous simulation construction makes CHGs purely

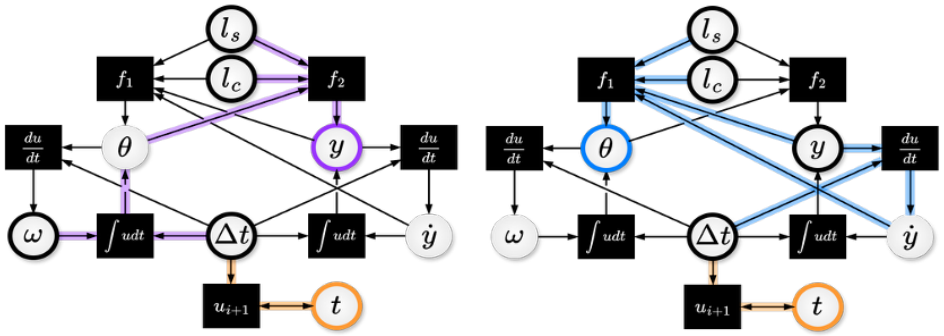


Figure 4.5: CHG from Figure 4.4 showing the two simulation processes imperatively given by Blocks 4.1 (left) and 4.2 (right) as paths through the graph connecting inputs (black, bolded outline) with outputs (colored, bolded outline).

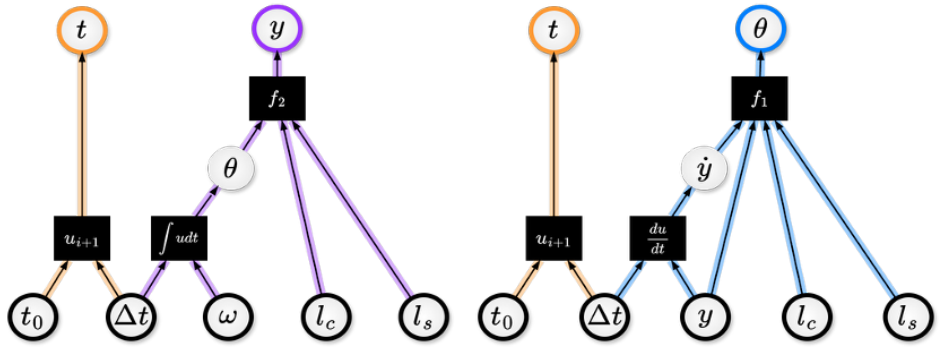


Figure 4.6: Paths through the CHGs shown in Figure 4.5 shown as trees, with a unique tree for each output.

declarative.

In addition to its declarative nature, a CHG also enables system simulation through its generality. The difficulty in establishing an engine for autonomously forming simulation processes means that most declarative languages are restricted to a singular domain. Bond graphs, for instance, can be used for arbitrary simulation of the energy-like entities in dynamic systems [36], but cannot query a relational database management system (RDBMS). Modelica's solver similarly solves differential equations [37], but is not used to construct geometric models despite its goal of multi-domain system modeling [38]. SysML was intended as a general modeling language for all systems, but has struggled to be adopted for hybrid simulation [39, 40]. This is not to limit the usefulness of these languages; it might even be said that the usefulness of each comes from specialization for a specific domain [41]. This is contrasted with CHGs, which are not specialized for any particular domain, but rather system simu-

lation in general. Rather than representing specific subsystems, CHGs break down a system into its primitive constituents: the state variables, and the relationships between those variables (represented as functions). As a result, CHGs can (in theory) represent any possible system, such that a model of a system expressed in any framework can be reconfigured as a CHG. In this a CHG maximizes the very essence of a system: generality [42]. The drawback is that CHGs do not provide formal aids to modelers, in the same way that the rigidity of a circuit diagram, for instance, helps guide a modeler to a suitable representation of an electronic circuit.

4.3.2. General, Declarative Simulation

Fully declarative system simulation can be provided when a system is represented using a CHG. This is provided by encoding the information of how a simulation process should be formed into the graph structure. Where imperative modeling requires a human modeler to arrange the models into a simulation process, the arrangement of a CHG allows these processes to be discovered autonomously. An agent creating a simulation process only needs to identify a path from the set of known inputs to the node representing the desired, unknown information.

One measurement of the significance of this capability is the reduction in modeling complexity. Eq. 4.7 gives the exponentially growing upper bound for the number of imperative programs that must be written for a system with n variables. However, because a CHG can create programs from composing edges in the hypergraph, the number of relationships required to capture the system's complexity is drastically reduced. For instance, given a system with three variables A, B and C, we can expect nine different pairings of inputs to outputs: six edges going from one node to another, such as $A : B$ and $B : A$, and three hyperedges, e.g. $\{A, B\} : C$. All nine of these pairings can be discovered on a CHG with only three edges: $A \to B$, $B \to C$, and $C \to A$. This is because of composition; for example, the simulation for $A : C$ can be computed as the path $A \to B$ composed with $B \to C$.

In general, the most efficient modeling structure is a minimally connected CHG–one where there is a single edge connecting every node, and all nodes are reachable from any other node. Such a CHG has only n edges (one leading from every node), a major contrast with the exponentially bounded complexity of imperative systems. Though minimally connected CHGs are not often encountered in practice, representing a system as a CHG nearly always results in linear growth in the number of relations de-

fined. The reduction in complexity is entirely due to the ability for a declarative solver to reuse functions, rather than every combination of functions needing to be explicitly defined.

4.3.3. Integration with Other Applications

Most multi-domain simulations require integration with software tools optimized for the domains expressed in the model. The insular nature of most software tools often limits their ability to connect with other modeling agents. The most egregious forms of insulation result in model silos, where information is not exchanged [43]. Methods of breaking down silos and integrating models are often imperative [4, 44, 45], often indicated when coupling is described by a flowchart or workflow. Imperatively coupled models establish processes by which different software subsystems may exchange messages [46]. This is a hallmark of encapsulation, where a subsystem has a local state that is distinct and unaffected by the global program [47]. Encapsulated, imperatively-coupled simulations can still be highly useful–as evidenced by recent usage of the Functional-Mockup Interface [48, 49]–yet lack the ability to fully simulate a system, as demonstrated in Section 4.2.2 and discussed further in [30].

Declarative model integration requires a reframing of how inter-tool simulation is understood. Instead of framing simulation in terms of passing information to software tools, multi-domain simulation should instead been seen as using tools to process relationships. The former viewpoint is imperative, with the tool forming a step in a simulation procedure. The latter is more aligned with a function-based understanding of a system: rather than defining steps, tools are identified as being able to map certain inputs to outputs. A declarative solver can then select the tools needed for a specific simulation process. In other words, there should be some tool capable of calculating each functional relationship in a system. In a CHG, each edge in a path represents a function that results in a valid transformation of inputs to outputs. The solver must first identify such a path, and then calculate the mapping associated with each function. The role of an external application is to perform this calculation. The functionality to do so is encoded to the CHG by embedding API calls into the specific mappings of each edge. A declarative solver calls these tools as it executes a simulation process, passing to the tool the function inputs and receiving in return the calculated output.

CHGs can be compared with other declarative solvers such as Modelica, whose calculations must remain native to the platform. This simulation in Modelica occurs external to the model structure, in

that the numerical integration of a system is never defined by the modeler [50]. Instead, each model is fitted to Modelica's interface, so that system inputs can be automatically passed to the declarative backend solver [8]. Because the numerical solver is not accounted for in the models, any emergent behavior from extending the system to other platforms cannot be predicted by the modeler. In contrast, a CHG requires no specific numerical integration method. A modeler encodes the specific strategy to be used into the model. In so doing, the numerical integration can be integrated with the greater system, and any emergent behaviors are captured in the composition of paths. Furthermore, this allows the modeler to specify how each relation is to be calculated. If an external application can be called by the declarative solver, then the consequent behavior will be fully captured in the simulation. To reemphasize this point, the activity of the declarative solver in a CHG is not executing the model, but rather arranging the simulation functions into an executable process, allowing system behavior to fully captured in the model structure.

This is readily apparent with basic algebra. The system shown in Figure 4.4 needs a solver that can perform arithmetic operations, trigonometry, and integration and differentiation. Execution of a simulation process, such as either process shown in Figure 4.5, could be performed with any tool providing these capabilities–such as a human agent equipped with a scientific calculator. A CHG solver might also choose to call a specific tool to provide additional capability, such as a modeler utilizing MATLAB's suite of Runge-Kutta algorithms for performing numerical integration. To do so the modeler must indicate that the rule for calculating the integration functions in Figure 4.4 should be executed using MATLAB. If the solver encounters the integration function while constructing a simulation path, it can then automatically pass the inputs of the current sequence to MATLAB (through its API). The output of the calculated function are then returned to the solver, which matches them with the next function in the sequence.

By treating software as a calculation tool, rather than an information handler, models can be solved declaratively. The claim of this paper is that coupling systems along a system's behavioral functions allows for an automatic method of performing inter-tool simulation. This is especially true as systems change. If the scope of the kinematic model simulated in Block 4.1 changed, for instance by including masses for the two arms, then the model would need to be rewritten. Additionally, any imperative coupling between software would need to be rewritten, since new information would now need to be

passed between the applications. This demonstrates the incredible fragility of imperatively coupled systems: they are entirely dependent upon the scope of the system they represent. This is contrasted with a CHG, for which the system representation is entirely independent of the calculation performed by the tool. In other words, which nodes are present in a CHG model and how they are connected does not influence the ability for the solver to call an external software tool because each the calculation of each function is independent of the system's scope.

Providing robust models for systems in flux (which, to a certain extent, includes all systems [51]) is one of the most challenging aspects of model-based engineering [52–54]. While CHGs do not answer every issue for interoperability, for instance, they do not provide mechanisms for syntactic interoperability [55], the authors find they address many of the thorny issues of software integration. These include simulating all parts of a system, redefining a system without having to rewrite all inter-tool connections, and the ability to connect with other systems without loss of meaning.

4.4. Multi-Domain Modeling of Crankshaft

These claims are demonstrated through simulating a multi-physics model of a crankshaft with form as given in Figures 4.7 and 4.8, with parameters explained in Table C.3. This model extends the simple kinematic system shown in Figure 4.4, with the crankshaft corresponding to the crank arm of the slider-crank mechanism. The kinematic model contained only eight state variables, the expanded model represents a system with a complexity several orders of magnitude greater. Representing this system as a CHG illustrates how the issues of inter-tool interoperability are addressed by the declarative framework. The example involves a case where an engine designer needs to know the mass moment of inertia J_c of the crankshaft, for instance to calculate the input power necessary for the crankshaft to accelerate to a specified rotational velocity in a given time. While the kinematic relationships given in Eqs. (4.1)–(4.4) still hold true, the addition of mass requires a more detailed model of the crankshaft.

4.4.1. Model Integration

J_c becomes increasingly difficult to compute algebraically as the geometry becomes more specialized. The common alternative is to use solid modeling software, which can readily compute moments of inertia by taking advantage of a point-based representation of a solid body. A major advantage of

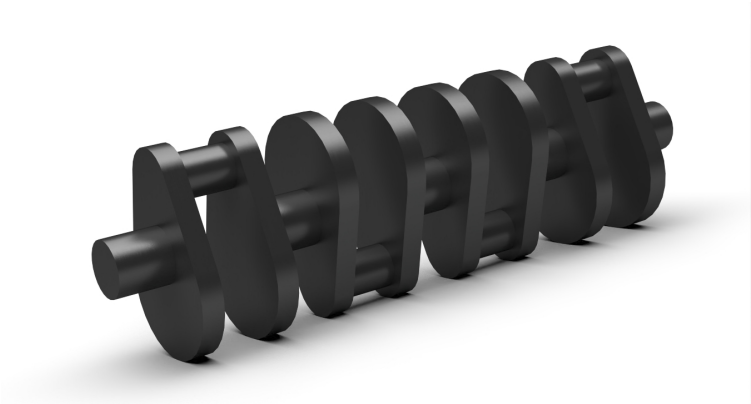


Figure 4.7: Image of the modeled crankshaft.

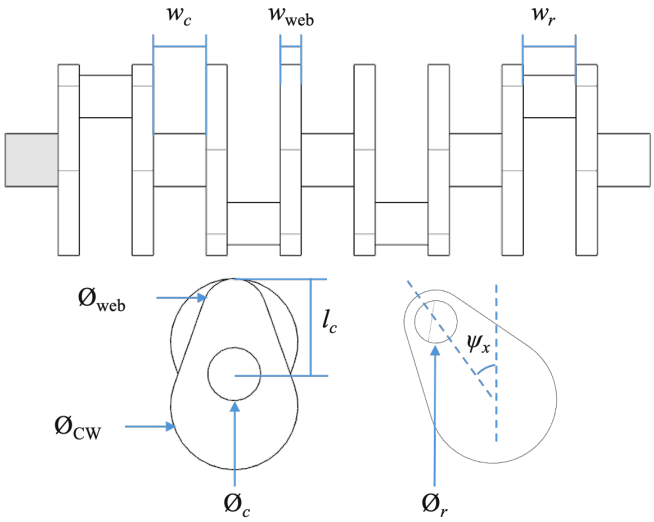


Figure 4.8: Geometric parameters for the crankshaft, as described in Table C.3, with the initial main bearing surface built in Figure 4.10 shaded in gray.

employing a CAD platform to calculate J_c is that any change to the physics of the crankshaft (such as adding machining features, new materials, keys, lubrication passages, etc.) will correspondingly update J_c. Providing a function in a CHG for calculating J_c using a CAD platform enables a true-model centric form of model-based engineering, where every value used by a decision-maker is represented by a single-node, and all the models for calculating or updating that node are given as paths in the CHG. This includes values that might be used in technical drawings, engineering analyses, or manufacturing data; all information and generating models corresponding to the piston engine could theoretically be captured in the CHG, providing the full integration of the disparate software into a model-based plat-

form. However, this more limited case study explores only how CAD software can be integrated with a dynamic model solver. The CHG for this integration is shown in Figure 4.10, though only covering a portion of the graph due to the scale of the system. The development process follows that given in Figure 4.4: represent each system variable as a node, then relate nodes to each other through multidimensional edges representing functions.

In addition to the diagrams shown in the section, the actual model is written in the Python programming language.[1] The declarative engine that solves the model is the open-source package ConstraintHg [56], which employs a breadth-first search algorithm to find relevant simulation paths connecting a pair of inputs to an output. ConstraintHg, written by the authors, is still in development. Although its performance is sufficient for its use in this study, its interface and execution speed are still being improved. As such, this paper focuses on demonstrating that pathfinding in a CHG can provide declarative model integration, rather than characterizing the precise algorithms used for that pathfinding, which will be focused on in later works.

4.4.2. Process

The process of forming a CHG, as described in Figure 4.9, begins with a modeler first identifies the values of the system that can be represented. Each of these is represented by a node in the CHG. The most readily identifiable nodes are the geometric parameters given in Table C.3. Other values include the variables needed to construct the body in a CAD application, such as planes of reference, boundary types, and geometric constraints. Finally, the hypergraph must include the actual values required by the API, including feature identifiers, access tokens, URIs (Uniform Resource Indicators), and HTTP (HyperText Transfer Protocol) calls. The inclusion of these later nodes allow the declarative solver to autonomously handle client-server interactions–often a messy part of functional languages [57]. A partial CHG with these nodes included is shown in Figure 4.10.

Each of these values are connected in the hypergraph by functions that determine their value. Many of these functions will be algebraic, such as determining the total length of the crankshaft by summing the length of its individual sections. Others will need to be calculated by the various applications. In this study the solid body model of the crankshaft is assembled in Onshape, a cloud-based CAD ap-

[1]Full scripts are provided under the MIT license at https://github.com/jmorris335/tool-interoperability-scripts/.

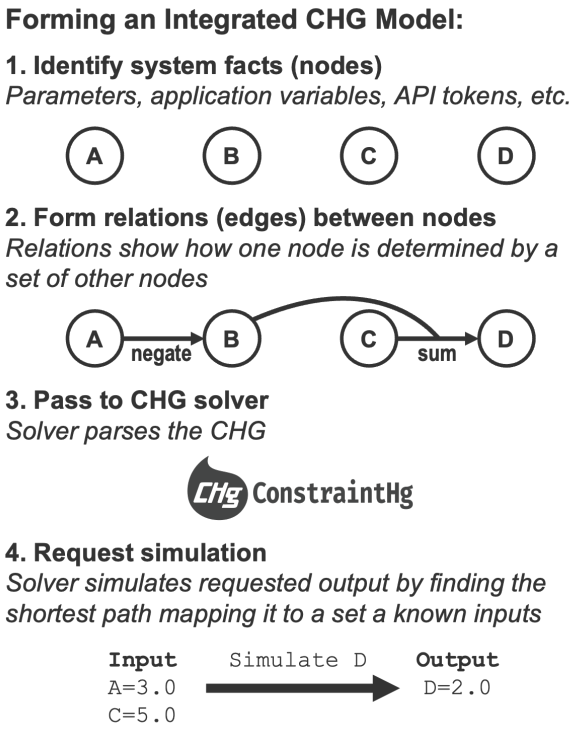


Figure 4.9: Process of forming and simulating a CHG, incorporating the demonstration CHG solver *ConstraintHg* [56].

plication offering an established API [58]. Onshape was chosen for its general availability as well as to demonstrate how CHGs can facilitate HTTP (HyperText Transfer Protocol) connections. It's worth restating an important point here: the innovation of using a CHG is not due to changing the *mechanism* by which software integration occurs–the method of interfacing with Onshape remains along predefined API calls. Instead, the innovation is on how that mechanism is structured into the system representation. CHGs enable a system model to fully capture system behavior, as opposed to just system operations. Wrapping the API calls into model functions shows how the behavior of the system is simulated by the interfacing software. This is the same pattern followed for integrating MATLAB and Ansys Mechanical into the simulation process.

4.5. Results

Once prepared, the CHG contains a full model of the system that can be immediately simulated. The user prepares a caller file such as the one shown in Block 4.3. This file does three primary things:

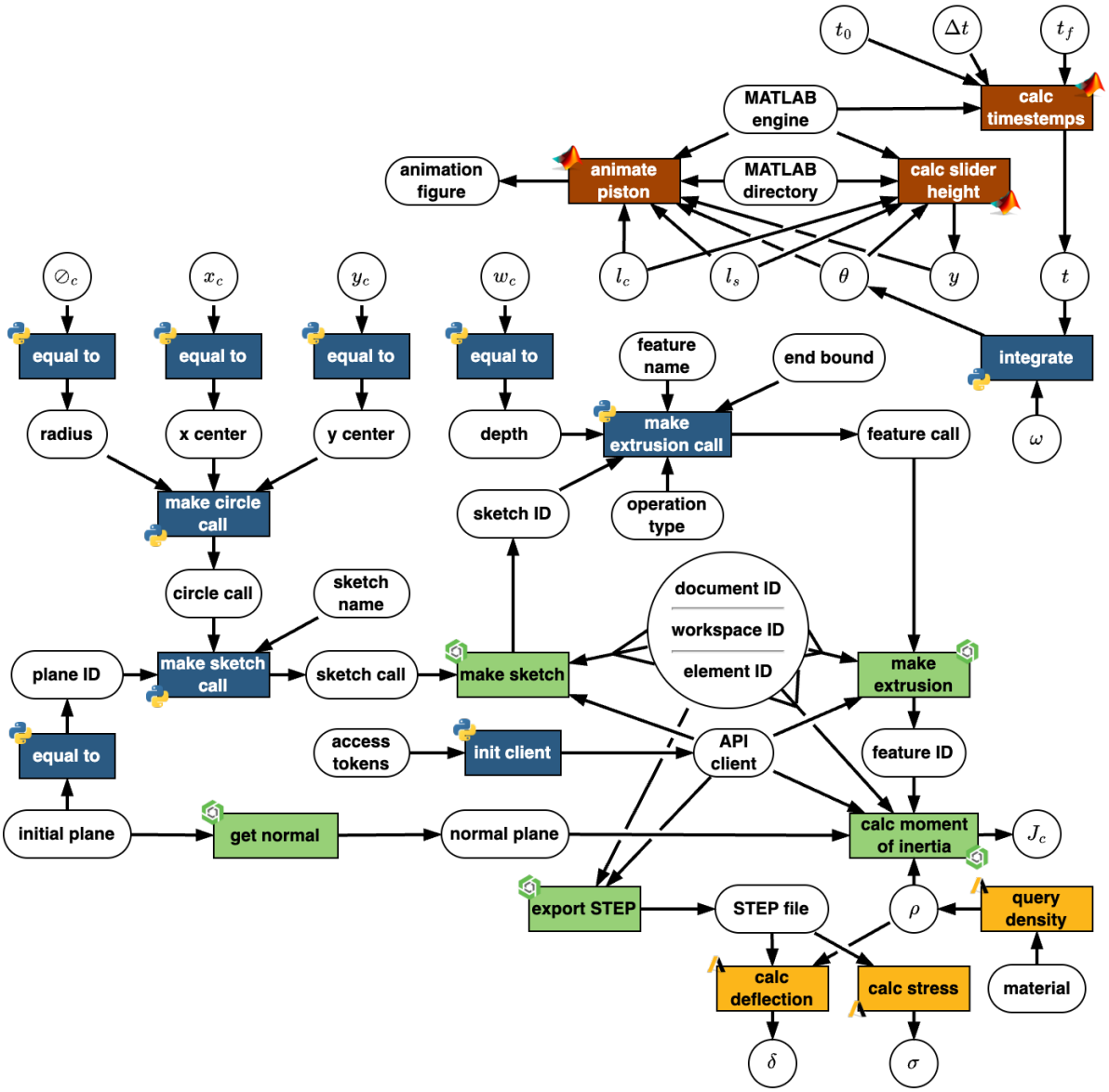


Figure 4.10: A subset of the full CHG (reduced for brevity) showing FEA simulation in Ansys, kinematic animation in MATLAB, and solid modeling of the crankshaft's primary bearing(shown as shaded in Figure 4.8) in Onshape. Basic connections and algebraic relationships are calculated in Python.

first, it initiates the API clients of the used software applications; second, it declares the list of known variables (inputs); and third, it calls the CHG solver, passing to it the inputs as well as the desired output. Notice that nowhere in the caller script does the user need to specify the behavior of the system, since the system behavior is decoupled from the simulation call.

Block 4.3: Example python script calling simulation of the CHG

```
import matlab.engine
import constrainthg

from src.matlab.matlab_chg import *
from src.onshape.onshape_chg import *
from src.ansys.ansys_chg import *

crankshaft_chg = matlab_chg + onshape_chg + ansys_chg

inputs = dict(
  shaft_dia = .025,
  pin_dia = .020,
  web_dia = .100,
  l_c = .030,
  l_s = .100,
  length_units = 'm',

  # Onshape
  initial_plane = 'Front',
  did = '99ccbc50135e7bd6a47fc0fb',
  wvmid = '03b2576bbb9b2c4ef0de9bf4',
  eid = 'c081126bd4e3181bb497cf01',

  # Ansys
  material = 'Carbon Steel',
  load_force = 84.,
  load_x_location = 0.05,
  load_y_location = 0.,
  fixed_face = 'JXB',

  # MATLAB
  matlab_directory = './src/matlab',
  matlab_engine = matlab.engine.start_matlab(),
  timestep = 0.01,
  omega = 2 * 3.14159,
)

crankshaft_chg.solve(target=sigma, inputs=inputs)
```

Executing the caller script engages the CHG solver, which seeds the CHG with the values of the passed inputs. Though many pathfinding methods are viable, the ConstraintHg algorithm used in the case study specifically uses a breadth-first search to discover valid simulation paths. The solver traces out the edges from each discovered node, building a graph of possible traces. As each edge is searched,

the solver executes its corresponding function rule. Algebraic functions will be calculated by calls to the Python interpreter, while software-specific functions will be passed to respective platform by the wrapped API, as shown in Figure 4.10. The result of the execution call is saved to the target node of the processed edge, expanding the set of discovered nodes. The process terminates either when the solver solves for the value of the desired output node, or when there are no more edges that can be viably traversed.

4.6. Discussion

There are several additional benefits to representing a system with a CHG no already discussed in the paper. For software integration, one primary advantage is the ability to isolate usable parts of the model. Because both the CHG and every subgraph of the CHG are valid models–able to be fully simulated–a modeler can choose a subset of the CHG edges from which to include in possible simulations. There are many situations in which model separation has advantages, such as when a modeler does not have access to all the software platforms utilized in a model. In such a case, the portions of the model that do not reference the inaccessible application are still executable. For instance, running the script in Block 4.3 without enabling the Onshape or Ansys client will still provide access to the motion study in MATLAB, since the edges in that simulation do not depend on outputs from the other software.

Similarly, using a CHG promotes simulatability at all stages of modeling. Because the validity of the model does not change as edges are added or removed, modelers can use a single CHG for all steps in the design process. Early stage designs might employ more abstracted variables and lower fidelity models. As the design grows in complexity, nodes representing more focused variables and higher fidelity models can be added to the graph. Adding more edges increases the number of simulations that can be conducted–corresponding to the increased knowledge about the system–but does not affect the model's validity. Consequently, the universality of the CHG extends both across domains and also across the system's evolution.

CHGs also allow competing models by automating model selection processes. Consider the case where design team specifies two separate models for calculating the bending stress of the crankshaft: one employing a high-fidelity, computationally expensive FEA analysis; and another based on a surrogate, linearized model that can be quickly executed using Python. Depending on the use case of the

simulation, a modeler might prefer the one model to the other. To describe this preference, the modeler can assign edge weights to the model, indicating the cost of executing the model. The pathfinding can then automatically select the lower-weighted edge when building the simulation process.

Perhaps the most difficult part of using CHGs to integrate models is discovering the functional relationships between model parameters. In a graphical solid modeling environment, such as using Onshape through its browser-based client, a modeler only needs to specify a single chain of parameter relationships. The resulting CAD model is imperatively defined, taking a set of defined inputs and mapping them to the final body. Contrast this with a declarative model, which must provide translations between multiple pairings of inputs and outputs. To accomplish this, the relationships of the procedural definition must be generalized and new functions defined–an often laborious process.

For example, each main journal on the crankshaft is defined to be concentrically aligned with each other. An imperative model might establish this relationship by making the center of each journal equal to the journal defined before it. The modeling kernel must consequently process the first journal before calculating the placement of the subsequent cylindrical sections. But the behavior of the crankshaft requires no such explicit ordering, only that the journals are all concentrically aligned. The procedural definition can be expanded to capture this more general behavior by modeling the journal centers in a CHG, as shown in Figure 4.11. To do so, additional relationships must be added between all the center points, not just a singular one as in the procedural case. The resulting subgraph is fully connected (or complete), implying the centers of all journals can be calculated if any one of them is provided as an input. This is true regardless of the order in which the journals are solved for. This simple case of concentric shafts is indicative of declarative modeling. By expressing how all variables are related, the CHG better captures the behavior of the crankshaft, at the expense of needing additional relationships to be defined by the modeler.

The effort of decomposing Onshape's API into composable functions results in a declarative language for solid modeling. This language is purely a wrapper, with its symbols comprised of the actual API calls and syntax provided by the modeler. While other declarative languages for solid modeling have been proposed before [59], wrapping the existing interface better takes advantage of the functionality provided by the software. Onshape uses the Parasolid modeling kernel to construct geometries procedurally. But by abstracting out the functions of Onshape's API, the CHG solver can rearrange op-

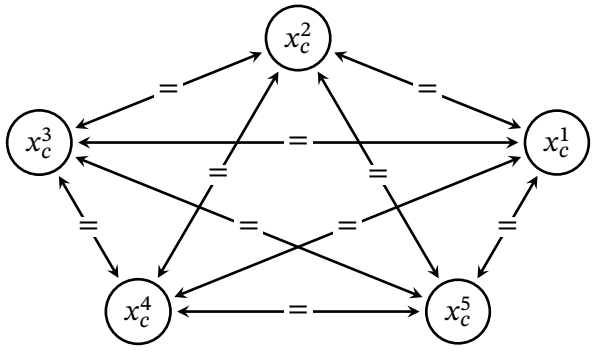


Figure 4.11: Declarative representation for horizontal location of main journals on crankshaft as a CHG where all edges indicate equivalency.

erations as needed to compose the eventual model geometry. This converts a traditionally imperative process into a declarative one, so any CAD model constructed from the singular CHG could possess a unique, yet consistent feature tree.

The consistency of the CHG model must be ensured by the modeler. For instance, a Boolean union operation cannot be applied to a single body. Any edge in a CHG representing this operation must be provided with inputs corresponding to the multiple bodies to be combined. This is enforced by the modeler, whose task is to define the domain and codomain of each function in the CHG. This, in many aspects, summarizes the work of modeling a system: the arrangement of functions showing how certain variables influence other variables.

The advantage of using a CHG is not eliminating the labor of modeling a system. Rather, it is that this effort is fully captured when composing simulations, rather than being limited to a single, procedural interpretation. A declarative language provides more efficient translation from a real system to the constructions created to represent it. While traditional model integration attempts relies on procedural calls along established API scripts [4], a CHG allows for arbitrary cosimulation of a system. One economic consideration is that effort invested into developing models yields far greater returns with the declarative models of a CHG, which can be reused as many times, and in as many ways, as required by the organization [60]. The mechanisms for model composability can also drive distributed simulation, yielding benefits for execution time and resource management [19, 60].

4.6.1. Limitations

CHGs are considered to be closed-world, that is, the CHG assumes that nothing exists except that which declared in the model. This stems from how CHGs capture information generally: because all information in the system can be related within a CHG, it does not make sense to prescribe additional information external to a CHG's scope. Though a CHG's connectedness is certainly one of its strengths, the resulting closed-world framework can mask the inconsistencies between the CHG and the real world system it approximates. For example, a force applied to the crankshaft could be modeled as being related to the accelerations of the piston heads. Though capturing a significant part of the crankshaft's behavior, the model implies that the pistons are the only applied load on the crankshaft. This assumption is implicitly given by the scope of the CHG, and as such may not be immediately apparent to the modeler. The factors and relationships not expressed in the CHG may have significant influence on the system's behavior. The method for discovering these unmodeled factors likely lies, as with other modeling frameworks, in verifying the model against observations in the real world.

There are other obstacles preventing the immediate adoption of CHGs as a method of general system modeling. One is that creating the necessary inter-variable relationships is challenging when the interfacing application is primarily designed for interaction through a graphical user interface (GUI). Selecting geometric entities such as faces or edges is difficult in a CHG. Difficult, but necessary, since the primary benefit of a CHG is in its automatic execution, so that there must be some function allowing the CHG solver to perform the tasks normally undertaken by a human agent interfacing with the GUI. The authors address this by noting that, though CHGs are excellent at expressing models, they are less suitable for initial development. If the behavior of a system is initially unknown, then the modeler is advised to make use of the available frameworks that have been developed for the purpose of model development. In other words, a modeler should start in a CAD environment, working with the GUI. Or they should start by drawing a circuit diagram, or a bond graph. Only after teasing out the system behavior should these models be reconstructed into a unifying CHG.

A second limitation to general adoption is the CHG's reliance on a software exposing its functionality through an API. It takes considerable labor to wrest an application into a collection of functions that can subsequently be arranged by a modeler. This labor is liable to be wasted if the API is significantly

modified or taken offline. Even more unworkable is when an application does not provide an API in the first place, preventing integration via a CHG. For instances of CAD, this can be somewhat resolved by building an CHG interface around a modeling kernel such as Parasolid. Extracting the functionalities of Parasolid can provide a better basis for generating solid models due to Parasolid's time-tested (and somewhat static) nature. Additionally, its use in a variety of CAD applications means that such a declarative wrapper might be useful for more than just one software.

4.6.2. Future Work

General adoption of CHGs is dependent upon a robust CHG solver being made available to practitioners. The authors have discussed one such solver currently in development [56]. The analysis of this solver, including its runtime and consistency, require further consideration in a future article. Once a robust solver is released, additional work would likely be merited to build toolboxes for integrating modeling kernels into CHGs. These could be extended to other software systems: finite element methods, Computer-Aided Manufacturing (CAM), Product Data Management (PDM), Enterprise Resource Planning (ERP) and other systems. For the latter two, the focus shifts from dynamic systems to maintaining digital threads. Though these may seem like different paradigms, in reality both are systems that require modeling and simulation, and consequently that could benefit from being represented as CHGs.

4.7. Conclusion

This paper showed how declarative modeling can be performed with multi-domain models, even when the simulation of those models requires otherwise sequestered analytical software. The key solution was the use of a CHG as a holistic model formalism that captured the full system behavior, with software tools integrated into the CHG as edges in the hypergraph. This is paradigmatically different from traditional modeling couplers, which pass messages between distinct models rather than unifying them into a single framework.

This work builds upon previous articles that discussed the mathematics of CHGs [3] and how they engender declarative, functional modeling [30]. It was shown that the purpose of system modeling is to perform simulations, where facts about the system of interest can be artificially observed through

the relationships of the model. One point critical to this paper was that the process of simulating a model is equivalent to constructing a chain of functions mapping a set of known inputs to the unknown outputs being simulated. The result is the provision of a simulation process: a series of calculable steps describing how an output is constrained by the inputs.

How these simulation processes are constructed has significant differences for modelers, especially when it comes to integrating between software tools. Models in traditional, imperatively-coupled frameworks express a single simulation process. The process contains at which points information should be passed and received from the software in the ecosystem, often through an API. The resulting simulation may be holistic, but it is inflexible, only describing a single behavioral trace of the system being represented. Imperative simulations make it difficult for modelers to understand the different interactions of a system, as only a single set of inputs can every be prescribed.

Alternatively, CHG models avoid connecting software along procedures. Instead, applications are treated as tools for calculating function rules. A modeler first deconstructs an application's interface into a set of functions, then arranges the functions to describe how the state variables of a system affect one another. The model's structure allows for a path-finding engine to arbitrarily solve the resulting CHG for any connected pairing of inputs and outputs, allowing for universal, automatic simulation of the connected system.

This is demonstrated by forming a CHG of a crankshaft that integrates kinematic and dynamic models with a solid model formed in the CAD platform Onshape. The full process of connecting with Onshape, including file structures and authorization, is handled by the CHG. This provides fully autonomous parsing of the resulting system model. Various configurations of the crankshaft can be formed by specifying various inputs. Simulation relations best calculated in specialized packages such as CAD or FEA are processed using the API for the respective tool, with the declarative simulation occurring in the autonomous selection of which relations to simulate and what order they be simulated in. Though this work demonstrates integration between only four software platforms (Python, MATLAB, Ansys Mechanical, and Onshape), the theory shows how declarative simulation can be provided between any software which exposes its functionality via an API.

References

[1] John Morris et al. “Declarative Integration of CAD Software into Multi-Physics Simulation via Constraint Hypergraphs”. *Proceedings of the ASME 2025 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*. ASME IDETC-CIE 2025. Anaheim, CA: ASME, Aug. 17–20, 2025.

[2] 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.

[3] 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.

[4] Andrew Montalbano, Gregory Mocko, and Gang Li. “Integration of Vehicle Dynamics with Finite Element Composite Structure Simulation”. *Proceedings of the 2024 Ground Vehicle Systems Engineering and Technology Symposium (GVSETS)*. 2024 NDIA Michigan Chapter Ground Vehicle Systems Engineering and Technology Symposium. Novi, MI: NDIA, Aug. 13, 2024.

[5] International Organization for Standardization. *Systems and Software Engineering - System Life Cycle Processes*. May 2023. https://www.iso.org/standard/81702.html (visited on 09/21/2024). Published.

[6] Edward A. Lee. “Determinism”. *ACM Trans. Embed. Comput. Syst.* 20.5 (May 29, 2021), 38:1–38:34. ISSN: 1539-9087. DOI: 10.1145/3453652.

[7] Kenneth Boulding. “General Systems Theory: The Skeleton of Science”. *Management Science* 2.3 (Apr. 1956), pp. 197–208. http://www.panarchy.org/boulding/systems.1956.html (visited on 01/08/2025).

[8] 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.

[9] Bernard P. Zeigler. *Multifacetted Modelling and Discrete Event Simulation*. London: Academic Press, 1984. 372 pp. ISBN: 978-0-12-778450-2.

[10] Benjamin S. Blanchard and W. J. Fabrycky. *Systems Engineering and Analysis*. 3rd ed. Upper Saddle River, N.J: Prentice Hall, 1998. 738 pp. ISBN: 978-0-13-135047-2.

[11] Luciano Floridi. *Information: A Very Short Introduction*. Very Short Introductions. Oxford: Oxford University Press, 2010. 116 pp. ISBN: 978-0-19-955137-8.

[12] W. Ross Ashby. *An Introduction to Cybernetics*. Internet. London: Chapman & Hall, 1956. http://pcp.vub.ac.be/books/IntroCyb.pdf (visited on 02/27/2025).

[13] Jan Willem Polderman and Jan C. Willems. “Dynamical Systems”. *Introduction to Mathematical Systems Theory: A Behavioral Approach*. Vol. 26. Texts in Applied Mathematics. New York, NY: Springer, 1998, pp. 1–25. ISBN: 978-1-4757-2953-5. DOI: 10.1007/978-1-4757-2953-5_1.

[14] 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.

[15] David I. Spivak. *Category Theory for Scientists*. Sept. 18, 2013. DOI: 10.48550/arXiv.1302.6946. arXiv: 1302.6946. Prepublished.

[16] Israel N. Herstein. *Topics in Algebra*. 1st ed. Waltham, MA: Blaisdell Publishing Company, 1964. ISBN: 978-0-536-00257-0.

[17] Boris Eisenbart, Kilian Gericke, and Luciënne Blessing. “An Analysis of Functional Modeling Approaches Across Disciplines”. *AI EDAM* 27.3 (Aug. 2013), pp. 281–289. ISSN: 0890-0604, 1469-1760. DOI: 10.1017/S0890060413000280.

[18] François E. Cellier and Ernesto Kofman. *Continuous System Simulation*. New York: Springer, 2006. 643 pp. ISBN: 978-0-387-26102-7. DOI: 10.1007/0-387-30260-3.

[19] Cláudio Gomes et al. “Co-Simulation: A Survey”. *ACM Comput. Surv.* 51.3 (May 23, 2018), 49:1–49:33. ISSN: 0360-0300. DOI: 10.1145/3179993.

[20] Evan Patterson, David I. Spivak, and Dmitry Vagner. “Wiring Diagrams as Normal Forms for Computing in Symmetric Monoidal Categories”. *Proceedings 3rd Annual International Applied Category Theory Conference 2020*. 3rd Annual International Applied Category Theory Conference 2020. Vol. 333. Electronic Proceedings in Theoretical Computer Science. Cambridge, USA: Open Publishing Association, Feb. 8, 2021, pp. 49–64. DOI: 10.4204/EPTCS.333.4. arXiv: 2101.12046 [cs].

[21] John C. Baez and Blake S. Pollard. “A Compositional Framework for Reaction Networks”. *Rev. Math. Phys.* 29.09 (Oct. 2017), p. 1750028. ISSN: 0129-055X. DOI: 10.1142/S0129055X17500283.

[22] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, Feb. 20, 2004. 944 pp. ISBN: 978-0-262-22069-9. Google Books: _bmyEnUnfTsC.

[23] Rajarishi Sinha et al. "Modeling and Simulation Methods for Design of Engineering Systems". *J. Comput. Inf. Sci. Eng* 1.1 (Mar. 1, 2001), pp. 84–91. ISSN: 1530-9827. DOI: 10.1115/1.1344877.

[24] Bernard P. Zeigler, Alexandre Muzy, and Ernesto Kofman. *Theory of Modeling and Simulation*. Third. Elsevier, 1976. ISBN: 978-0-12-813370-5. DOI: 10.1016/B978-0-12-813370-5.00002-X.

[25] Michael Tiller. *Introduction to Physical Modeling with Modelica*. The Kluwer International Series in Engineering and Computer Science SECS 615. Boston: Kluwer Academic Publishers, 2001. 344 pp. ISBN: 978-0-7923-7367-4.

[26] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. 17. print. The MIT Electrical Engineering and Computer Science Series. Cambridge: MIT Pr. [u.a.], 1993. 542 pp. ISBN: 978-0-262-01077-1.

[27] 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.

[28] Chris Reade. *Elements of Functional Programming*. International Computer Science Series. Wokingham, England ; Reading, Mass: Addison-Wesley, 1989. 600 pp. ISBN: 978-0-201-12915-1.

[29] John C. Mitchell. *Concepts in Programming Languages*. Cambridge: Cambridge University Press, 2003. ISBN: 978-0-521-78098-8.

[30] John Morris, Gregory Mocko, and John Wagner. "Effects of Functional and Declarative Modeling Frameworks on System Simulation". *Under review with J. Dyn. Sys., Meas., Control (ASME)* (June 2025).

[31] 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.

[32] 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.

[33] Patrick Schultz, David I. Spivak, and Christina Vasilakopoulou. *Dynamical Systems and Sheaves*. Mar. 15, 2019. DOI: 10.48550/arXiv.1609.08086. arXiv: 1609.08086 [math]. Pre-published.

[34] Gioele Zardini et al. "A Compositional Sheaf-Theoretic Framework for Event-Based Systems (Extended Version)". *Electron. Proc. Theor. Comput. Sci.* 333 (Feb. 8, 2021), pp. 139–153. ISSN: 2075-2180. DOI: 10.4204/EPTCS.333.10. arXiv: 2005.04715 [eess].

[35] Michael G. Kapteyn, Jacob V. R. Pretorius, and Karen E. Willcox. "A Probabilistic Graphical Model Foundation for Enabling Predictive Digital Twins at Scale". *Nat Comput Sci* 1.5 (May 2021), pp. 337–347. ISSN: 2662-8457. DOI: 10.1038/s43588-021-00069-0.

[36] P.C. Breedveld. "Concept-Oriented Modeling of Dynamic Behavior". *Bond Graph Modelling of Engineering Systems: Theory, Applications and Software Support*. Ed. by Wolfgang Borutzky. New York: Springer. ISBN: 978-1-4419-9367-0. DOI: 10.1007/978-1-4419-9368-7.

[37] Sven Erik Mattsson, Hilding Elmqvist, and Martin Otter. "Physical System Modeling with Modelica". *Control Engineering Practice* 6.4 (Apr. 1, 1998), pp. 501–510. ISSN: 0967-0661. DOI: 10.1016/S0967-0661(98)00047-1.

[38] Martin Otter and Hilding Elmqvist. "Modelica". *Simulation News Europe* 10.29/30 (Dec. 2000), pp. 3–8. https://www.sne-journal.org/fileadmin/user_upload_sne/SNE_Issues_OA/SNE_10/sne.10.29-30.pdf (visited on 02/25/2025).

[39] Sabine Wolny et al. "Thirteen Years of SysML: A Systematic Mapping Study". *Softw Syst Model* 19.1 (Jan. 1, 2020), pp. 111–169. ISSN: 1619-1374. DOI: 10.1007/s10270-019-00735-y.

[40] Wu Xinquan et al. "Simulating Hybrid SysML Models: A Model Transformation Approach under the DEVS Framework". *J Supercomput* 79.2 (Feb. 1, 2023), pp. 2010–2030. ISSN: 1573-0484. DOI: 10.1007/s11227-022-04654-6.

[41] Karl Johan Åström, Hilding Elmqvist, and Sven Erik Mattsson. "Evolution of Continuous-Time Modeling and Simulation". 12th European Simulation Multiconference. Manchester, UK, June 16, 1998.

[42] Brian R Gaines. "General Systems Research: Quo Vadis?" *General Systems: Yearbook of the Society for General Systems Research* 24 (1979), pp. 1–9.

[43] Hou Yee Quek et al. "Dynamic Knowledge Graph Applications for Augmented Built Environments Through "The World Avatar"". *Journal of Building Engineering* 91 (Aug. 2024), p. 109507. ISSN: 23527102. DOI: 10.1016/j.jobe.2024.109507.

[44] Alessandro Margara et al. "Towards an Engineering Methodology for Multi-Model Scientific Simulations". *2015 IEEE/ACM 1st International Workshop on Software Engineering for High Performance Computing in Science*. 2015 IEEE/ACM

1st International Workshop on Software Engineering for High Performance Computing in Science. Florence, Italy, May 19, 2015, pp. 51–55. ISBN: 978-1-4673-7082-0. DOI: 10.1109/SE4HPCS.2015.15.

[45] Pascal Cantot. *Simulation and Modeling of Systems of Systems*. Ed. by Pascal Cantot and Dominique Luzeaux. London: John Wiley & Sons, 2011. 378 pp. ISBN: 978-1-84821-234-3.

[46] Lewis J. Pinson and Richard S. Wiener. *An Introduction to Object-Oriented Programming and Smalltalk*. Reading, Mass.: Addison-Wesley, 1988. 502 pp. ISBN: 978-0-201-19127-1.

[47] 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.

[48] T Blochwitz et al. "The Functional Mockup Interface for Tool Independent Exchange of Simulation Models". *Proceedings 8th Modelica Conference*. Modelica Conference. Dresden, Mar. 20, 2011. http://www.functional-mockup-interface.org (visited on 01/13/2022).

[49] Marcus Wiens, Tobias Meyer, and Philipp Thomas. "The Potential of FMI for the Development of Digital Twins for Large Modular Multi-Domain Systems". *Proceedings of the 14th International Modelica Conference*. 14th International Modelica Conference. Linköping, Sweden, Sept. 27, 2021, pp. 235–240. DOI: 10.3384/ecp21181235.

[50] Peter Fritzson. *Introduction to Modeling and Simulation of Technical and Physical Systems with Modelica*. Piscataway, NJ: IEEE Press, Nov. 2, 2011. ISBN: 978-1-118-01068-6. DOI: 10.1002/9781118094259.

[51] D. J. Wagg et al. "Digital Twins: State-of-the-Art and Future Directions for Modeling and Simulation in Engineering Dynamics Applications [Special Section]". *ASME J. Risk Uncertainty Part B* 6.3 (May 12, 2020), 030901:1–030901:18. ISSN: 2332-9017. DOI: 10.1115/1.4046739.

[52] Daniel R. Dolk and Jeffrey E. Kottemann. "Model Integration and a Theory of Models". *Decision Support Systems. Model Management Systems* 9.1 (Jan. 1, 1993), pp. 51–63. ISSN: 0167-9236. DOI: 10.1016/0167-9236(93)90022-U.

[53] 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.

[54] INCOSE. *Systems Engineering Vision 2035*. INCOSE, 2021. https://www.incose.org/2021-redesign/load-test/systems-engineering-vision-2035 (visited on 02/19/2025).

[55] International Organization for Standardization. *Information Technology-Cloud Computing-Interoperability and Portability*. International Standard. Version 2017-12. Dec. 2017. DOI: 10.3403/30313036U.

[56] John Morris. *ConstraintHg*. Version 0.2.2. Nov. 23, 2024. DOI: 10.5281/zenodo.15278018.

[57] Andrew D. Gordon. *Functional Programming and Input/Output*. [Repr. der Ausg.] 1994. Distinguished Dissertations in Computer Science. Cambridge: Cambridge Univ. Press, 2008. 155 pp. ISBN: 978-0-521-47103-9.

[58] Onshape. *Glassworks API*. Version v10. Onshape. https://cad.onshape.com/glassworks/explorer (visited on 03/15/2025).

[59] Chandrakana Nandi et al. "Functional Programming for Compiling and Decompiling Computer-Aided Design". *Proc. ACM Program. Lang.* 2 (ICFP July 30, 2018), pp. 1–31. ISSN: 2475-1421. DOI: 10.1145/3236794.

[60] Simon J. E. Taylor. "Distributed Simulation: State-of-the-Art and Potential for Operational Research". *European Journal of Operational Research* 273.1 (Feb. 16, 2019), pp. 1–19. ISSN: 0377-2217. DOI: 10.1016/j.ejor.2018.04.032.

CHAPTER 5

Review of Digital Twins

Abstract: The definition of a digital twin (DT) is typically given as ”a virtual representation of real-world entities and processes, synchronized at a specified frequency and fidelity” [1]. Though this definition is useful for discussing DTs, it only encapsulates what digital twins are composed of, without expressing their purpose. Because of this ambiguity, research on DTs tend to morph their meaning into convenient forms based on the use case of the DT considered by the study. In order to set forth unambiguous and multidisciplinary conclusions concerning DTs, it is necessary to supplement the definition through expounding the purpose of a DT in a way that is applicable to all use cases. Critically, such an exposition greatly clarifies the value of constraint hypergraphs in representing DTs, making this exercise an essential part of the dissertation.

5.1. Purpose of a Digital Twin

Since being first used as a technical term [2], DTs have been continually redefined [3], especially as media teams have employed the term for marketing purposes. Entropy of overuse has further disseminated its core meaning into a variety of related terms such as ”device shadow,” ”virtual prototype,” and ”mirrored system” [4]. A more exact approach is to define a DT not by the parts it is made of, but rather by the functions it performs. This is best established when considering DTs as systems, deriving a definition from principles of systems theory.

The starting point is considering the digital nature of a DT. There are two universal systems (worlds) that must be considered when dealing with a DT, as shown in Figure 5.1. The first is the real world, comprised of entities (things) that constitute the reality we exist in. Interaction between these entities defines the behavior of the real world. The goal of an agent–an intelligent entity–is to influence these entities in such a way as to enact some outcome consisting of a desired state of reality.

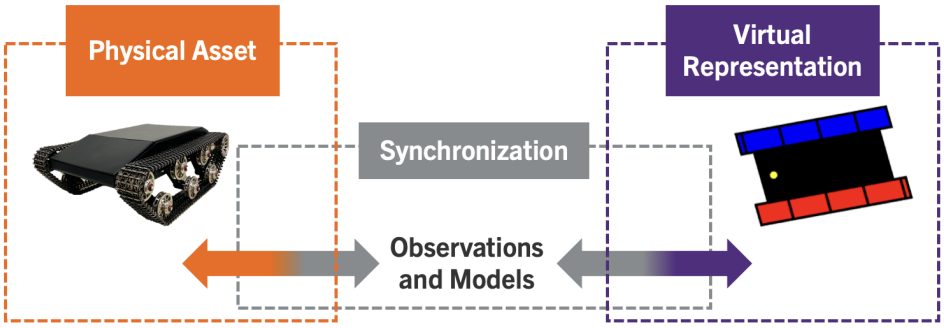


Figure 5.1: Major domains of a digital twin.

To obtain a specific outcome, the agent must understand the ways in which entities may interact. This is evident in a bird sensing wind speeds before landing on a specific branch, or a driver following a route in order to arrive at a specific destination. In each case, the agent must properly predict how their actions will result in their desired outcome, and consequently how the entities they interact with will behave.

The key to an agent’s intelligence is representing the system as information. This comprises the second of the two universes: the virtual domain whose monad is information characterizing the real world. Though the virtual world is a pattern generated from the real world, the two domains are con-

trasted by the finitude of information. Although every entity in the real world is capable of generating an infinite amount of information related to its state, because information is finite, the virtual world can only be an approximation of the real one. The process of transforming system properties into useful information is known as observation.

For instance, an infinite number of system elements could be observed from photograph shown in Figure 5.2. Examples include observing that the subject of the photograph is a single ship, and that the color of the water is blue. Having made those observations we can further process them, making inferences about the ship's motion, the weather depicted, and more.

Figure 5.2: Photograph of the USCGS Midgett of the coast of California. *Image by PH1 S. Smith [5].*

Because observations are finite constructs of an infinite domain, they are necessarily only approximations of the state of the real world. Observing the water to be blue approximates the varying shades of light as a single color. This may be useful to us as agents, but it also demonstrates that observations are always wrong at some level; the virtual world never perfectly describes the real one. The collected observations form the basis of the virtual representation of the real world entity.

Another limiting factor of observations is that we can only make a finite number of them. The virtual domain is subsequently sparse, aligning to reality only at the instances that have been observed. It is impossible to make enough finite observations to perfectly represent the behavior inherent in a

real-world entity, leaving perpetual gaps in the observed virtualization–areas where information about the real system is lacking.

Besides taking more observations, the only method for filling these gaps is through constructing models. Models serve as proxies of the real world that are used to generate information about a real world system. Such generated information is an approximation,[1] or simulated information, since it is artificially fabricated via the model.

The process of producing simulated information is known as simulation, and any kind of mechanism capable of interpreting a model and extracting the fabricated information can be referred to as a simulation engine. By analogy, a simulation engine ”observes” a model to add information to the virtual domain, supplementing any actual observations of the real-world. The utility of a model stems from being able to provide information with greater ease than corresponding observations of the real world.

To be able to serve as a real world proxy, models must exist in the same form as a real system. The fundamental form of a model describes the relationships between entities within a set of categories,[2] where a category is a collection of entities that share a common property. Such a form can be visually represented by a graph, such as the model shown in Figure 5.3.

Models have lots of different valid definitions, as described in [8], however most technical models can be represented in this form. For instance, the relationships in a geometric model are largely typified as relative positions between points and lines. Dynamic models relate how the state variables of a system relate to other entities such as forces, energy transfer, or motion. Regression models show the relationships between factors and outcomes. These relationships might be thought of more rigorously as a mapping, where a model maps entities from one set to another.

It should be noted that the categorizations a model relates are not real, they are virtual suppositions of order imposed by the modeler. Consequently, a model could represent any number of real systems. Model categories are necessarily abstract in the same way that any sub-system of the universal system is an abstract, arbitrary subdivision. Phrased a different way, the categories that are established by a modeler only pertain to the domain of the model. What constitutes the ship of Theseus depends on

[1] It is mandatory when talking about models to include the famous quote by George Box that ”all models are wrong, some are useful” [6].

[2] Similar or equivalent words for entity include thing, object, element, and instance; synonyms for category include class, group, set, and identity.

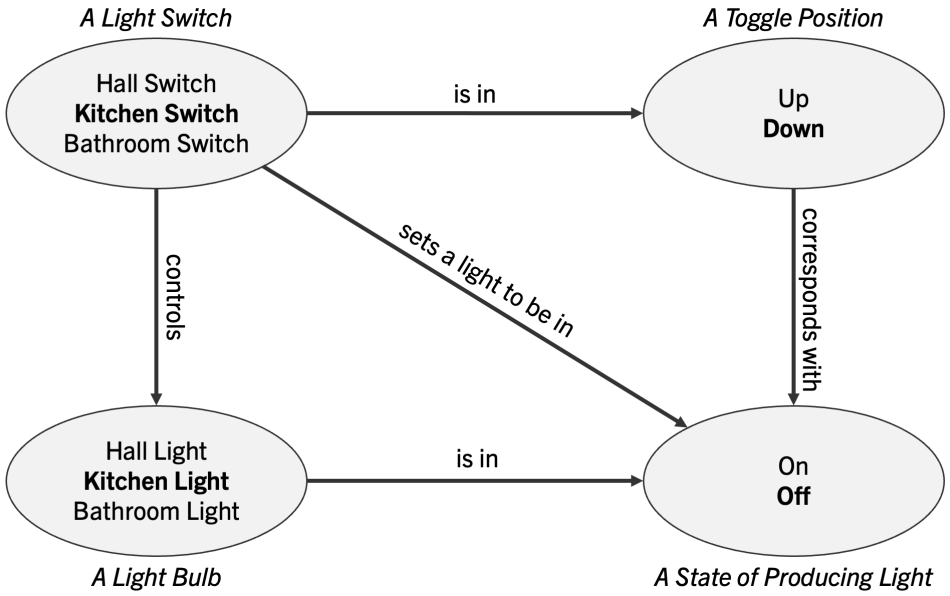


Figure 5.3: A model of a system of a light switch and bulb, with categories as the gray ellipses labeled above in *italics*, specific entities as text in the ellipses, relationships as the text above each directed arrow, and an observation in **bold face**. This style of diagramming is known as an olog [7].

the definition of the ship given by the ancient Athenians. This is semantically understood when we say that a model represents *a thing*, while an observation is always taken of *that thing*. Observations are always specific versus models which are always abstract.

Observations and simulations are two methods for obtaining information about a system, respectively using the real world and a model as bases. Two important differences between the two are their accuracy and convenience. Observations may be very difficult to perform, such as observing an asteroid crashing into the earth. Simulations, on the other hand, are more convenient but do not represent the real system, only an approximate model.

Blending the specificity and accuracy of observations with the generality and convenience of models and simulations serves as a basis for a digital twin. Digital twins are virtual representations of a specific real world system. Both models and observations are required to produce these virtualizations. A cube, such as the one in Figure 5.4, serves as a simple example of this.

As a geometric concept, a cubic model consists of a series of constraints that, when enforced, define a six-shaded shape. We could use a cube to represent something real, such as the wooden block shown in the figure. To do so we would have to take some observations of the block, such as the length of one

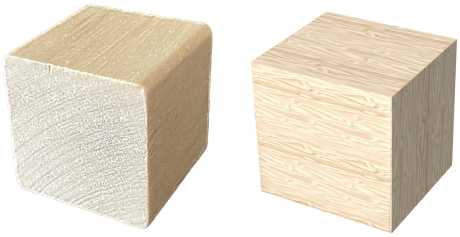


Figure 5.4: An image of a wooden block and a CAD model of a cube.

of its sides. An applied model can be constructed that approximates the geometry of the real block. The model can then be simulated to estimate information that was not observed, such as the distance between two vertices.

The combination of observations and models to represent the real wooden block constitutes a digital twin. Though the digital twin is not perfectly faithful–it doesn't reflect the actual world information, such as the fillets on the side of the block or the divots in the grain structure–it exhibits all the virtual information associated with the block. This reflects the primary purpose of a DT, which is to transform a specific, real entity into the virtual (and therefore understandable) domain. From this established purpose can be derived a definition of a DT as the best possible virtual representation of a specific, real system, exhibiting equivalent behaviors as its real counterpart.

The specificity of a DT is actually a broad requirement. For the cube model to be a digital twin it must not represent any wooden block, but the specific wooden block shown in Figure 5.4. This is practically accomplished through some form of synchronization, where the DT updates in response to changes in the real world entity. The importance of synchronization (or equivalently, real-time updating) was stressed by Wagg et al. [9], who called it a "key functionality" of a DT. It is specificity that distinguishes an applied model, a virtual representation augmented with observed data, from a digital twin. Kritzinger et al. further differentiated digital twins as possessing an automatic synchronization between the real and virtual domains [10], a position also espoused by Juarez et al. [11].

Revealing the synchronized behavior of an entity could be done ideally by either taking all observations of the real entity or building models with perfect predictive power. Since neither of these tasks can be practically performed, a DT exists as a best-possible combination of the two, a confluence of observations and models that reveals real world behavior. The effectiveness of a DT can be shown across two dimensions: the degree of the states we are able to observe, and the degree of states we are able to

predict via models. These two dimensions are shown in Figure 5.5.

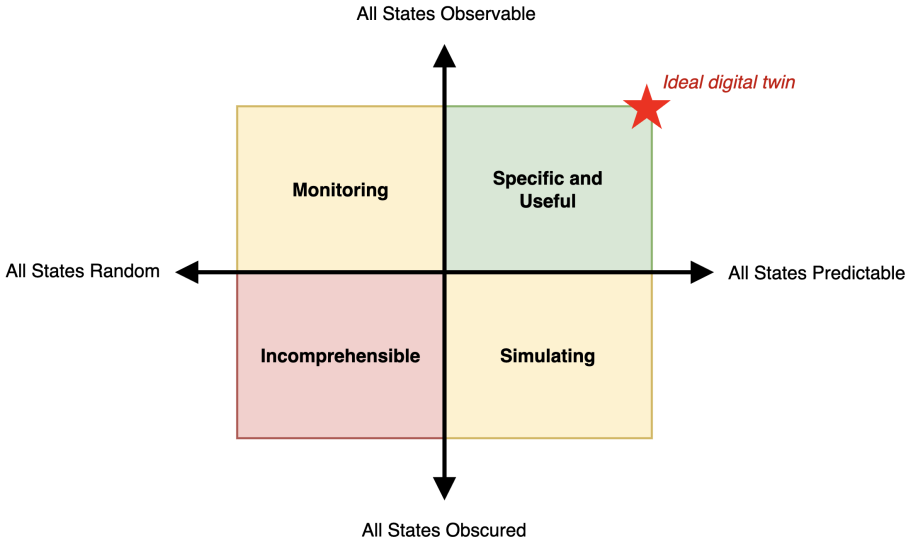


Figure 5.5: Typification of interrogations of a system provided by a digital twin, with observability on the vertical axis, and behavioral prediction (provided by models) on the horizontal.

This definition of a DT is similar to definitions that have been given before, such as by the Digital Twin Consortium [1]. Though these definitions are succinct, they can lend themselves to misrepresentation. To clarify several of these more common misconceptions, a few corollaries about DTs are given:

The first is that the real counterpart does not have to be physical. While the real domain does contain all physical entities, such as ships and people and cities, it also contains real metaphysical entities including supply chains, economies, political systems, processes, and thought patterns. Consequently, DTs can be built for entities that do not exist tangibly, as long as that DT represents the current state of the real entity.[3] When a DT is made of a physical entity, it can be spoken of as an ”atoms to bits” transformation [12].

Second, a DT is an information-generating device, providing the information necessary for an agent to make useful insights into the real world. This information can be used for three different types of tasks:

1. Monitoring, where the state of the entity is relayed and tracked in the current environment;

[3]It is in this sense that DTs are said to be built for products that are still in development. In this case the product exists as a real idea or design which the DT represents, not as an actual physical device. When the product is released, then more individual DTs can be made for each released item.

2. Prediction, where the state of the entity is estimated given an arbitrary environment; and
3. Actuation, where the state of the entity is influenced to achieve some expected outcome.

Each of these tasks can be accomplished without a DT, but a DT adds valuable information that can aid their execution. Indeed, the definition of a DT states that an ideal DT provides the most information possible for the execution of these tasks. But it should not be posited that the performance of these tasks constitutes a DT, nor should a DT be constrained as having to perform all these tasks. When this is so the definition of a DT becomes use-case driven, resulting in the numerous definitions given by the disparate industries in which they are employed. This confusion can be avoided when DTs are more appropriately conceived.

Lastly, a DT is not exclusively a model. A model is abstract, while a DT represents a specific, real entity. We might say a model represents *a* ship, generally, while a DT represents *that* ship. Models exist solely in the virtual domain, while DTs are joined to the real world through observations of their real twin. This is an important point. While models can be used to estimate behaviors with sufficient accuracy, they almost always must be combined with real-world observations.

Similarly, but on the other side of the scale, a digital twin is not solely composed of observations. Such concepts are referred to as digitization.[4] Digitizations are digital reproductions of physical assets, such as a scanned document. A key part of this is that digitizations are static, while Digital Twins are live-updating [9].

Although it is an especially common mistake to describe digitizations as digital twins, in most cases the offense is neglecting to understand how the observed information is augmented with models. For instance, a recent article by Matterport, a company that produces 3D point clouds of buildings, defined digital twins as ”virtual 3D models of physical space” [13].[5] In this case the point cloud constitutes the observations, and the notions that relate those points to walls, windows, and other structures are the models. The same is often true when labeling IoT (Internet of Things) systems as digital twins. Though data collection is an important component, just collecting data does not constitute a DT.

By clarifying the purpose of a DT, it is hoped that implementing organizations will be better able to

[4]Not to be confused with digitalization, the work of transforming an organization’s processes from physical to digital systems.

[5]It is another misconception that DTs are or must include 3D models. There is nothing in the definition a DT that mandates a geometric model versus any other model.

recognize and employ DTs, recognizing the value that a DT can bring to any decision maker.

5.2. History of Digital Twins

The history of the Digital Twin starts implicitly with modeling and digital simulation. In 1991, Gelenter described using computers to simulate the known universe [14]. This echoed the idea of harnessing computers to simulate data on huge scales, sentiment expressed at least since Asimov wrote of Multivac in the 1950s [15].

Despite this interest, DTs could not be developed pragmatically until there was sufficient computational power available. This limitation made it so that early actors were often large organizations with access to advanced computing systems. Consequently, John Vickers, a technologist at NASA, was the first person to explicitly use the term "Digital Twin" in 2003 [16, 17], and NASA became the first organization to issue an official definition of the concept in 2010 [18, 19]. This was expanded by Grieves in his 2012 book [2].

As computational power and connectivity became more available, DTs began to become more widespread. This as also led to their affiliation with marketing platforms, where they are hyped by companies such as Amazon [20], IBM [21], Microsoft [22, 23], MathWorks [24], MapleSoft [25], Siemens [26], and Dassault Systèmes [27]. Greater attention on DTs as a cornerstone of the Fourth Industrial Revolution[6] [29] also means increased confusion, especially as marketing campaigns stretch the meaning of DTs to incentivize their products.

As much a victim of the hype cycle as any other product, DTs have gradually revealed the wicked problems [30] that prevent their easy adoption. These include the difficulty in developing a DT [31] and the high cost to do so [32, 33], the opaque nature of a DTs ROI [34, 35], the perpetual effort required to maintain them [36], and the difficulty of integrating them with other systems [37].

Part of the response of invested stakeholders has been the production of standards guiding the adoption of DTs. The publication of ISO 23247 [38] in 2021 was one of the first international standards directed at DT architectures. It was followed in 2022 by a ITU-T Y.3090, a standard from the International Telecommunication Union specifying how to structure systems of DTs to enable intra-

[6]The Fourth Industrial Revolution, also abbreviated 4IR, is the notion that data and connectivity will transform the way industry is carried out. It was first introduced by Philbeck and Davis in 2018 [28].

communication [39]. Later reviewers such as [40] and [41] summarized other applicable standards dealing with communication and networking, distributed systems, and information representation–all important components of making DTs function in modern enterprises.

5.3. Use Cases of Digital Twins

The wide applicability of DTs across multiple industries is one driver of the varied forms DTs can take. In 2019 Tao et al. found four industries with ongoing DT applications [42]. By 2022, Mihai et al. set the number of disrupted industries at seven [29]. Though what constitutes an industry is rather subjective, what is not speculative is the increasing number of use cases for DTs. Though in each use case the DT might take on different forms, many of the functions and challenges remain the same across implementation scenarios.

The most data-driven DTs are likely those developed for commercial buildings. Facility management and construction companies were early adopters of the Internet of Things (IoT). Their established data streams meant they were well positioned to integrate DTs into their Building Lifecycle Management (BLM) [43]. Services like Azure Digital Twins from Microsoft[7] and Amazon Web Services' TwinMaker offer DT functionality while being primarily focused on the data capture of IoT networks.

Many DTs that start with IoT exist as primarily data-driven observations partially augmented with basic models. These simple DTs are not as useful for performing predictive tasks, though this can be remedied through the use of machine learning on the extant data streams. At worst, data-driven DTs are incorrectly labeled, and should otherwise be referred to as digitizations (see Section 5.1). This includes efforts to digitize locations for virtual audiences, such as with real estate [13], virtual museums [45], or the case of preserving the cultural legacy of Tuvalu [46].

The opposite approach has been followed by large engineering firms that are more likely to have model-driven processes. NASA, perhaps the first real adopter of DTs, first employed physical twins as part of its validation process for the late Apollo missions [47]. Digital twins that mirrored spacecraft in real time came later, though whether they should be labeled as such is a matter of debate [48]. Aerospace and automotive sectors followed in NASA's footsteps, producing DTs of airplanes [49], jet engines [50],

[7]Microsoft also released an open-source language for defining DTs that integrates with Azure Digital Twins known as the Digital Twin Definition Language (DTDL) [44].

and vehicles [51] that were largely focused on connecting models to data. The nature of engineering enterprises means that many of these DTs correspond to products that are still under development, in this case the DTs mirror product concepts rather than an actual physical entity. Such a DT was labeled a "Digital Twin Prototype" by Grieves [52]. Information about such a non-tangible entity might come from requirements documentation, prototypes, or system diagrams to name a few sources [53]. DTs might also be used during the product design stage to represent environments [54] or other entities in the product family [55].

Another model-centric use case that has found significant support from DTs in recent years is health monitoring. In this aspect there are two angles of representation: representing the human body generally, such as with Siemens' Emma Twin [56], and representing individual human bodies to create a personal DT [57]. The former is generally used for research, where medicinal testing and experimental procedures can be performed virtually rather than on real persons [58]. On the other side of health monitoring, DTs of individuals maximize the amount of personal health information that can be made available for a person's health care. This might be used to recommending individualized treatments or performing diagnoses. Considerable ethical and private challenges remain to be addressed with this sensitive information [59].

As complex as the human biome is to represent, global environments remain the consummate challenge of terrestrial modelers. Weather, climate change, and natural disasters all evolve on terribly complex systems that necessitate equally involved DTs to represent virtually [60]. Geopolitical entities are also building DTs of cities and countries to better understand the effects of public policy and infrastructure. Examples include the United Kingdom [61] and Singapore [62].

The most common use case for DTs remains manufacturing, representing an estimated 50 percent of all published research on DTs [54]. Like facilities, manufacturers are often ideal practitioners for DTs due to having previously established digital threads from engagement in Industry 4.0 [8] initiatives [64]. The entities most likely to be mirrored by a DT are generally long-lived, well understood manufacturing assets such as machines [65], robotics [66], warehouses [67], or processing lines [68]. The controlled environments of a manufacturing system also lead to DTs of entire shop floors [69–71]. Benefits of

[8] Industry 4.0 is a new paradigm for general industry and manufacturing specifically first referenced in Germany in 2013 [63] that espouses interconnectivity between machines and well-integrated data streams.

implementing DTs include product health monitoring [42, 72] (and predictive maintenance [35, 73]), operation optimization [74], and virtual commissioning [75].

5.4. Digital Twin Ecosystems

Unfortunately, the benefits of a DT are only available to organizations with sufficient resources to develop DTs, which generally precludes small to medium enterprises (SMEs) [76]. This pay-to-play conundrum with digital twins leaves smaller organizations unable to leverage Industry 4.0 effectively. Some prohibitive costs of developing a DT include the following:

- Digital Thread: the system connecting information and artifacts throughout a company.
- Infrastructure such as server/edge computing devices, sensors, communication networks (such as a 5G wireless network), and PLM systems, all of which are used to support the DT's many components;
- IT (Information Technology) department, which maintains data streams and technology used in the DT, especially an IIoT (Industrial Internet of Things) network.
- Multi-domain expertise, which is required for developing statistical and physics-informed models, and machine learning algorithms.
- Computational platforms, including both hardware and software for executing numerical simulations, generating 3D models, and other computationally expensive tasks.
- Time; because DTs require significant verification, they often need significant work-hours to create, test, and launch [77].
- Non-allocated funds, which can be invested into a DT project in expectation for eventual returns.

The impact of this precludes the adoption of DTs in the organization. Like other long term investments, this pay-to-play factor of implementing DTs provides their benefits only to larger institutions. Indeed, companies leading out in deploying DTs are mostly massive companies and even geopolitical entities. These include GE Aviation [50], Tesla [51], the United Kingdom [61], and NASA [19].

Human factors such as a lack of awareness, unsupportive leadership, or a general lack of training also prevents DTs from being successfully implemented in SMEs. Change2Twin,[9] an organization of the

[9]The overall objective of the Change2Twin project is "to ensure that 100% of manufacturing companies in Europe have access to 100% of technologies needed to deploy a digital twin" [78].

European Union, suggested that these were actually the most pertinent to a DTs success [31]. Although these areas of study are critically important, this research focuses instead on the similarly significant technical challenges in DT deployment.

One approach to mitigating these costs is to outsource the development of a DT via a Manufacturing as a Service (MaaS) model as a many-to-one commissioning [79]. This occurs is when an external group creates a DT that can be deployed to multiple implementing organizations. The system reduces the cost of development for each organization while taking advantage of the economies of scale. This business model is similar to one pursued by GE digital, though within a fixed infrastructure system [80]. A MaaS DT can be referred to as redeployable.[10] Redeployability comes with the trade off that each DT must be capable of being integrated into a range of digital systems.

For a DT to be robustly redeployable it must also be interoperable with the elements of whichever digital system it is deployed into. This might include networking protocols and APIs, labeling schemes, PLM/ERP systems, and other DTs. Interoperability, a more generalized term in relation to redeployability, can be defined as a DT's ability to co-operate with other twinning components within a networked system. It should be noted that interoperability has been given many conflicting definitions: Klar et al. [82] defined it as the mutual collaboration of peer DTs, while Ricci et al. [83] defined it as the use of twinning resources in an open systems perspective. A widely cited white paper produced by Budiardjo and Migliori for the Digital Twin Consortium [84] used the term more generally as the ability to share information between two distinct entities, citing a ISO standard on interoperability in cloud computing [85].

These two terms together are used here to denote a modular DT, one that espouses an essential plug-and-play modality. A modular DT is one that functions with different systems. It can be redeployed from one system to another and maintain its interoperability. The value and the cost of modularity is integrating DTs into unknown systems. In order for a MaaS organization to produce a modular DT for deployment in an SME, the DT must be able to function with the arbitrary digital system that the SME has in place. Significant work still needs to be done to properly understanding these systems and enable MaaS DT development.

[10]Other similar terms include "domain-neutral," [81] and "reusable" [79]. This article uses "redeployable" due its clearer interpretation.

For example, there is considerable confusion in the current literature about how DTs should engage with other elements in a system. The most prevailing, though arguably incorrect, view espoused by [82, 83, 86, 87] treats DTs as primitive elements of a larger DT system. This is contrasted with the view that DTs are themselves composed of submodules that may need to be shared with other DTs. In this paradigm, connections are needed between DT components, and DTs can be formed arbitrarily by aggregating various DT submodules. This view is espoused (in varying degrees) in [75, 79, 88]. It is also possible that the first view is a more general depiction of the latter, and that submodular interfacing is implied by peer-to-peer DT systems; this is likely in the case of the ISO 23247 standard [38], which mentions peer-to-peer DT communication but does not define how such communication should take place.

Regardless of the perspective, the goal of forming DTs into systems is very prevalent in literature (and commercial thought pieces). The diversity of thought on the subject is exemplified by the variety of names assigned to the subject, such as ”Digital Twin Aggregates” [52], ”Web of Digital Twins” [83], ”Digital Twin Network” [89], ”Digital Twin Environment” [79], ”Digital Twin Systems-of-Systems” [90] and ”Digital Twin Ecosystems” [91–93]. The National Twin Program in the United Kingdom calls their digital twin of the globe a ”World Avatar”, comprised of an ecosystem of computational agents and a knowledge graph of concepts and data [94]. The author prefers to use digital twin ecosystem (DTE) since it conveys a sense of components that interact with each other without specifying a singular purpose (an interpretation employed, for instance, by [95]). This relates well to a system of DTs, which should be able to fulfill as many arbitrary functions as is required of the information it provides, and which (by the latter of the two worldviews expressed above) contains not just discrete DTs, but also many DT elements.

A species in a DTE is something that potentially comprises a DT, though critically it does not need to. Zambrano et al. identified three principle species (under the term ”fundamental building blocks”): models, algorithms and microservices, and data [79]. To this list the author adds simulation engines and services, where a simulation engine is any program capable of extracting virtual information from a model. When a model is coupled to a simulation engine it can be referred to as a simulation unit [96]. Services, meanwhile, are programs that provide visualizations, user interfaces, storage, security, or other supportive roles critical to a DT’s operation. The inclusion of services in a DT ecosystem was

also made by Human et al. [97]. One clear observation is that none of these building blocks are specific to a single twinnable entity; a model of a tire can be used just as effectively in the DTs of various cars, a rubber manufacturing line, or a road-surface representation.

Budiardjo and Migliori [84] discussed how the best way to understand the reusability of these components is through the perspective that everything is a system of systems. Like in systems engineering, defining the boundary of a DT is largely subjective–a designer could choose a DT to represent a tire and suspension system or make a DT of an entire car. In either case, the DT would consist of similar building blocks. This leads to a unique understanding of DT ecosystems not as sets of collaborating DTs, but rather sets of modular digital elements that can be composed together to form arbitrary DTs. García et al. reached similar conclusion in their study on the subject [92], though their agents considered consisted of models and data streams exclusively. The important takeaway here is that creating modular DTs depends on recognizing which species the DTs must interface with. A DTE provides a valuable conceptual model of all the species that must be accommodated by an integrating DT.

These species, as described above, are given explicitly as:

- Models: agents that are capable of describing some facet of unobserved reality.
- Simulation engines: agents capable of producing simulated information from a model. This definition includes the algorithms of Zambrano et al. [79].
- Data streams: agents capable of producing information from a data store, such as a database management system.
- Services: agents that support the organization, connection, distribution, and other secondary functions of other agents in the DTE.

5.5. Digital Twin Interoperability

Despite their benefits, there are several challenges associated with DTEs. Fifteen of these were elucidated by Michael et al. in 2022 [90], and included barriers with integration, protection of intellectual property, establishing access rights for integrating models, and resolving state conflicts. Integration is certainly the principle challenge, as DTs nearly always consist of heterogeneous information.

The problem of integration can be decomposed into two separate issues, termed semantic and syntactic interoperability. Semantic interoperability is the ability for an agent to understand the meaning

of exchanged information, while syntactic interoperability references understanding the format of exchanged information [85]. Syntactic issues are highly significant with heterogeneous data streams, as pointed out by the National Digital Twins Programme [98].

One well-researched method of providing syntactic interoperability is the use on ontologies [99]. An ontology is an object-oriented structure that explicitly establishes the meaning of a syntax over a given domain. They are often used to with DTs to provide explicit understandings for what data streams represent [100–102], as well as to represent relationships between entities [103].

Semantic interoperability is often more difficult to provide. The Functional Mockup Interface (FMI), first introduced in 2011 [104], is a free, open standard that enables interoperability between dynamic models (characterized by ordinary differential equations). Models are composed of Functional Mockup Units (FMUs), which can then pass state evolutions and parameter data between each other according to the FMI standard. FMUs come in three different types, differentiated by their what information they expose to a simulation engine. This may be a full model (Model Exchange interface), portions of a model (Scheduled Execution), or a combined model and an executable simulation engine as a simulation unit (Co-Simulation) [105]. In this last case, the FMU only exposes the results of the simulation to an interfacing tool. Wiens described how FMI can be used to define interoperable models within a DT using Co-Simulation FMI [106].

The Modelica Association, which governs the FMI standard, also provides a standard for specifying the subsystem of models (or FMUs) interfacing with simulation engines known as System Structure Package (SSP) [107]. SSP and similar standards work well for connecting dynamic models to numerical integrators largely because the methods for simulating and coupling dynamic models are well established. Proprietary software packages such as Amesim, MATLAB/Simulink, and Dymola all can connect with FMUs. However, there are no general solutions for connecting non-dynamic models to simulation engines. In these cases, manual integration of models via APIs[11] remains the de facto interface method; an inflexible approach that is incredibly difficult to scale (as the ecosystem grows) and maintain (as APIs and models change). The same integration problems exist for agents interfacing with data streams.

The full realization of API-driven information exchange is an insular computing platform. Compa-

[11] Application Programmer Interface

nies such as MathWorks, Ansys, Dassault Systèmes, and Siemens have all worked over time to develop a full-stack platform of computational tools that interface directly with each another. The goal is that a DTE could be entirely represented within the software platform of the vendor, allowing the vendor to handle semantic integration via tight intra-software connections. A hallmark of such an approach is the accumulation of diverse simulation engines that expand the capabilities of the platform, such as the software shown in Figure 5.6. The utility of a holistic computing environment is dependent upon the combined capabilities of the platform. Additionally, insular software environments can be difficult to connect to other platforms if an external capability is desired. This restricts the efficacy of a DT expressed on any such platform.

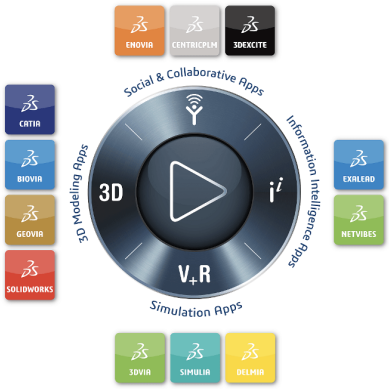


Figure 5.6: The 3DEXPERIENCE platform from Dassault Systèmes is arguably the most explicit example of a close-loop platform, describing itself as a "unified environment" based around a portfolio of 12 brands [108]. *Image credit: Dassault Systèmes*

The success and failures of FMUs are examples of a popular strategy of standardizing syntax in such a way as to fully expose the established semantics of a dynamic model. Such an approach could be referred to as universal interfacing, the work of having developers transform their agents into a uniform format agreed upon by all other developers. Although universal interfaces work well when fully executed, obtaining agreement from all possible vendors is incredibly difficult to do. The work of creating a universal data exchange and charging port for electronic devices over the past several decades[12] is testament to the difficulty of the problem. Universal modeling languages are consequently

[12]Such as in the development of the Universal Serial Bus, which only recently is moving to a truly universal configuration with its USB-C series.

plentiful but rarely fully adopted. Examples with varying levels of utilization (and focuses) include EXPRESS [109], APMonitor [110], Rebeca [111], ASCEND [112], and Modelica [113].

If the scope of integrating DTEs is broadened to include more diverse systems, then additional strategies can be found in the domain of computer science, where operating systems and distributed architectures must deal with a variety of heterogeneous information. One such area is distributed simulation, where a simulation is executed across multiple different simulation platforms. Distributed simulation is governed by IEEE 1516 standard espousing High Level Architectures (HLAs) [114]. HLAs force constituent simulation systems (known as federates) to adhere to a set of rules that enable publish-subscribe communication of information. Communication, synchronization, and sequencing are controlled by a Run-Time Interface (RTI) that connects each federate.

Distributed simulation in the context of DTs is not a heavily studied topic. It was alluded to by Taylor in his general overview of HLAs [115], and again by Budiardjo and Migliori in their often referenced work on DT interoperability [84]. Explicit use of the standard in the realm of DTs was shown by Longo et al., who also described how distributed simulation enabled key DT functions [116]. Song et al. provided another application of IEEE 1516 to DTs, though they focused primarily on data acquisition and messaging [117].

Another approach to achieving semantic and syntactic interoperability is to identify a basis transformation between each agent. The use of the word ”basis” here is especially meaningful: in linear algebra a basis serves as a frame for expressing any combination of ordered information (vectors). In this analogy, a universal interface would prescribe a singular basis that all information must be expressed in. Alternatively, a basis transformation prescribes operations that can transform a vector from one basis to any other. While the practicality of a universal interface is dependent upon its degree of adoption, a basis transformation provides full utility no matter the number of adopters. A universal set of operations for transforming models and data streams and simulation engines from one syntax to another, while preserving semantic meaning, is the Holy Grail of interoperability.

The key takeaway here is the need for a set of operations that can transform informational elements in a DTE between arbitrary syntaxes. The word element here is used to indicate a primitive entity shared among any representation format. These elements and relationships form a foundational structure which might then be expressed uniquely in a modeling language or simulation environment. Such a

structure is commonly represented as a network, with elements as nodes and relationships as edges.

Research on networks stems from a wide variety of disciplines employing graphs for information integration. Model integration was a major topic in the two decades prior to 2000 [118], with strong contributions in graph-based paradigms given by Geoffrion [119] and Jones [120]. Their early work in establishing modeling paradigms, schemas, and model components led to modeling languages that sought to deconstruct how entities related to each other in pragmatic ways, such as in engineering design decision-making [121].

This research has led to a broad adoption of knowledge graphs, entity-relation (ER) diagrams, and networking flows to DTs and DTEs. Microsoft Azure uses a graph to represent the IoT elements of a DT [122]. Du & Luo looked at creating graphs of a DTE that used correlation coefficients to relate between variables [123]. Perhaps the most common use of a graph is a knowledge graph, which represents semantic relationships between elements in a DT. The World Avatar, led by Computational Modeling Group at the University of Cambridge, is a significant example of this type of representation [124, 125], as shown in Figure 5.7. Other projects and researchers employing semantic knowledge graphs include [94, 126, 127].

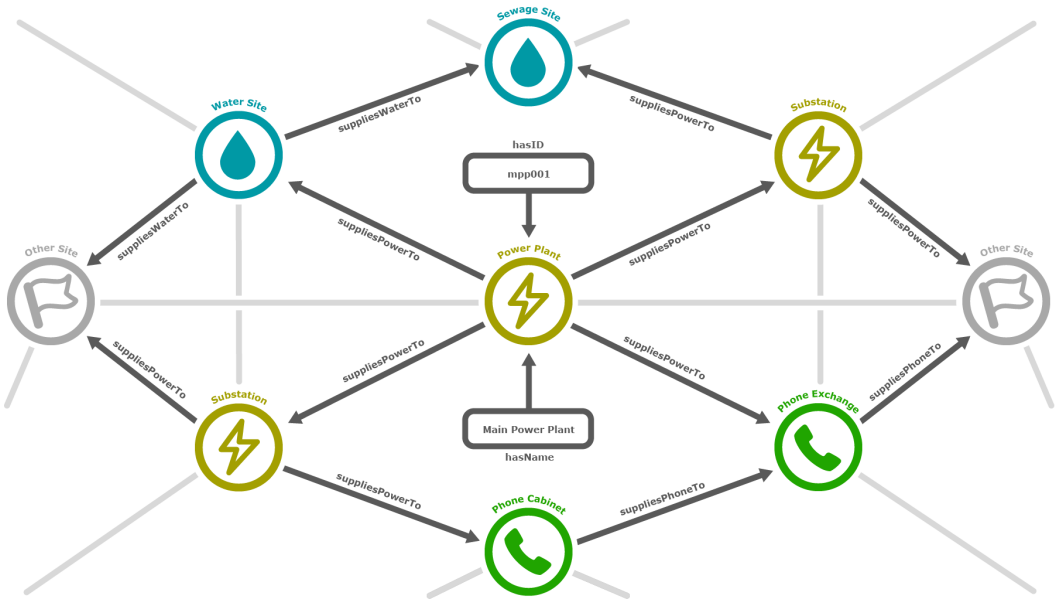


Figure 5.7: Example of a typical knowledge graph used in the World Avatar project for a global digital twin.

The basic theory of a knowledge graph is that knowledge (in the form of information) can be represented as a relationship between two entities [128]. A critical difference with constraint hypergraphs is the independence of information. In a knowledge graph, each edge implies a characterized relationship between entities. Two entities may be linked by more than one edge, because they might be related in more than one way. For instance, the knowledge graph in Figure 5.8 shows two relations between a person and an employee. These two edges are not competitive, rather they show different ways to relate the two entities. The interpretation of these relationships is not given by a function, but rather through some schema that interprets the "is a" and "works for a" labels.

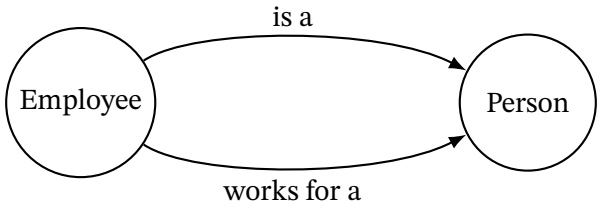


Figure 5.8: Simple knowledge graph showing two possible relationships between entities.

The use of a knowledge graph in this way does not encode the system's behavior into the model. While knowledge graphs are an excellent format for describing systems, their lack of mathematical rigor leaves them unable to be executed, and therefore inadequate for doing system interrogation. Chapter 6 will show how constraint hypergraphs enable more robust footings for representing digital twins.

References

[1] Digital Twin Consortium. *Definition of a Digital Twin*. Object Management Group, Inc. Dec. 3, 2020. https://www.digitaltwinconsortium.org/initiatives/the-definition-of-a-digital-twin.htm (visited on 10/18/2021).

[2] Michael Grieves. *Virtually Perfect*. Cocoa Beach: Space Coast Press, LLC, 2011. ISBN: 978-0-9821380-0-7.

[3] David Jones et al. "Characterising the Digital Twin: A Systematic Literature Review". *CIRP Journal of Manufacturing Science and Technology* 29 (May 1, 2020), pp. 36–52. ISSN: 1755-5817. DOI: 10.1016/j.cirpj.2020.02.002.

[4] Adil Rasheed, Omer San, and Trond Kvamsdal. "Digital Twin: Values, Challenges and Enablers from a Modeling Perspective". *IEEE Access* 8 (2020), pp. 21980–22012. ISSN: 21693536. DOI: 10.1109/ACCESS.2020.2970143.

[5] PH1 S. Smith. *230505-G-G0000-115*. https://www.history.uscg.mil/Our-Collections/Photos/igphoto/2003216550/ (visited on 07/08/2023).

[6] George E. P. Box and Norman Richard Draper. *Empirical Model-Building and Response Surfaces*. Wiley Series in Probability and Mathematical Statistics. New York: Wiley, 1987. 669 pp. ISBN: 978-0-471-81033-9.

[7] David I. Spivak. *Category Theory for Scientists*. Sept. 18, 2013. DOI: 10.48550/arXiv.1302.6946. arXiv: 1302.6946. Prepublished.

[8] Roman Frigg and James Nguyen. "Scientific Representation". *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Winter 2021. Metaphysics Research Lab, Stanford University, 2021. https://plato.stanford.edu/archives/win2021/entries/scientific-representation/ (visited on 05/16/2024).

[9] D. J. Wagg et al. "Digital Twins: State-of-the-Art and Future Directions for Modeling and Simulation in Engineering Dynamics Applications [Special Section]". *ASME J. Risk Uncertainty Part B* 6.3 (May 12, 2020), 030901:1–030901:18. ISSN: 2332-9017. DOI: 10.1115/1.4046739.

[10] Werner Kritzinger et al. "Digital Twin in Manufacturing: A Categorical Literature Review and Classification". *IFAC-PapersOnLine*. 16th IFAC Symposium on Information Control Problems in Manufacturing INCOM 2018 51.11 (Jan. 1, 2018), pp. 1016–1022. ISSN: 2405-8963. DOI: 10.1016/j.ifacol.2018.08.474.

[11] Maria G. Juarez, Vicente J. Botti, and Adriana S. Giret. "Digital Twins: Review and Challenges". *JCISE* 21.030802 (Apr. 2, 2021). ISSN: 1530-9827. DOI: 10.1115/1.4050244.

[12] Shoumen Palit Austin Datta. "Emergence of Digital Twins - Is This the March of Reason?" *Journal of Innovation Management* 5.3 (Nov. 29, 2017), pp. 14–33. ISSN: 2183-0606. DOI: 10.24840/2183-0606_005.003_0003.

[13] Matterport Editorial Team. *How (and Why) to Create Digital Twins in Real Estate*. Matterport. May 26, 2023. https://matterport.com/learn/digital-twin/real-estate (visited on 05/16/2024).

[14] David Hillel Gelernter. "Mirror Worlds?" *Mirror Worlds: Or the Day Software Puts the Universe in a Shoebox... How It Will Happen and What It Will Mean*. New York: Oxford University Press, Nov. 14, 1991. ISBN: 978-0-19-506812-2.

[15] Isaac Asimov. "The Last Question". *Science Fiction Quarterly* 4.5 (Nov. 1956). https://archive.org/details/Science_Fiction_Quarterly_New_Series_v04n05_1956-11_slpn/page/n5/mode/2up (visited on 05/21/2024).

[16] Matt Zborowski. "Finding Meaning, Application for the Much-Discussed "Digital Twin"". *Journal of Petroleum Technology* 70.06 (June 1, 2018), pp. 26–32. ISSN: 0149-2136. DOI: 10.2118/0618-0026-JPT.

[17] Michael Grieves. *Digital Twin: Manufacturing Excellence through Virtual Factory Replication*. Dassault Systèmes, 2014.

[18] Mike Shafto et al. *Draft Modeling, Simulation, Information Technology & Processing Roadmap*. TA11-27. Washington DC: National Aeronautics and Space Administration, Nov. 2010. https://www.nasa.gov/pdf/501321main_TA11-MSITP-DRAFT-Nov2010-A1.pdf.

[19] Edward H. Glaessgen, D. S. Stargel, and D. S. Stargel. "The Digital Twin Paradigm for Future NASA and U.S. Air Force Vehicles". *53rd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference - Special Session on the Digital Twin*. Honolulu, Apr. 16, 2012. DOI: 10.2514/6.2012-1818.

[20] Jay Naves. *Using AWS IoT TwinMaker to Build a Spacecraft Digital Twin*. AWS Public Sector Blog. Apr. 14, 2023. https://aws.amazon.com/blogs/publicsector/using-aws-iot-twinmaker-build-spacecraft-digital-twin/ (visited on 05/21/2024).

[21] Mukesh Dialani. *Digital Twins — Transforming Supply Chains and Operations*. IBM, Mar. 2022. https://www.ibm.com/downloads/cas/LVJKXXNA (visited on 05/21/2024).

[22] Shyam Varan Nath and Pieter van Schalkwyk. *Building Industrial Digital Twins: Design, Develop, and Deploy Digital Twins Solutions for Real-World Industries Using Azure Digital Twins*. Birmingham Mumbai: Packt, Nov. 2, 2021. 266 pp. ISBN: 978-1-83921-907-8. https://industrialdigitaltwinbook.com/.

[23] Microsoft Corporation. *Azure Digital Twins Documentation - Tutorials, API Reference*. Microsoft Docs. https://docs.microsoft.com/en-us/azure/digital-twins/ (visited on 11/29/2021).

[24] MathWorks. *Digital Twins for Predictive Maintenance*. MathWorks, Inc. https://explore.mathworks.com/digital-twins-for-predictive-maintenance/landing-36US-760M7.html (visited on 10/14/2021).

[25] Maplesoft. *Virtual Commissioning with a Model-Driven Digital Twin*. Waterloo Maple Inc. https://www.maplesoft.com/solutions/engineering/AppAreas/Virtual-Commissioning.aspx (visited on 01/13/2022).

[26] Siemens. *Manufacturing Process & Production Simulation*. 2021. https://resources.sw.siemens.com/en-US/e-book-manufacturing-simulation-medical-device-production-digital-twin (visited on 05/21/2024).

[27] Dassault Systèmes. *Beyond Digital Twin: Small and Medium-Sized Manufacturers Enjoy Big-Enterprise Benefits and ROI with Virtual Twin Experience on the Cloud.* 2020. https://discover.3ds.com/beyond-digital-twin (visited on 05/21/2024).

[28] Thomas Philbeck and Nicholas Davis. "The Fourth Industrial Revolution: Shaping a New Era". *Journal of International Affairs* 72.1 (2018), pp. 17–22. ISSN: 0022-197X. JSTOR: 26588339. https://www.jstor.org/stable/26588339 (visited on 01/11/2022).

[29] Stefan Mihai et al. "Digital Twins: A Survey on Enabling Technologies, Challenges, Trends and Future Prospects". *IEEE Commun. Surv. Tutorials* 24.4 (Win. 2022), pp. 2255–2291. ISSN: 1553-877X, 2373-745X. DOI: 10.1109/COMST.2022.3208773.

[30] Horst W. J. Rittel and Melvin M. Webber. "Dilemmas in a General Theory of Planning". *Policy Sci* 4.2 (June 1, 1973), pp. 155–169. ISSN: 1573-0891. DOI: 10.1007/BF01405730.

[31] Paolo Pileggi et al. *Overcoming Digital Twin Barriers for Manufacturing SMEs*. Position paper. Change2Twin, Apr. 2021. https://www.syncontwin.eu/insights/10-overcoming-9-digital-twin-barriers-for-manufacturing-smes/ (visited on 09/04/2025).

[32] Jan Stentoft et al. "Drivers and Barriers for Industry 4.0 Readiness and Practice: Empirical Evidence from Small and Medium-Sized Manufacturers". *Production Planning & Control* 32.10 (July 27, 2021), pp. 811–828. ISSN: 0953-7287. DOI: 10.1080/09537287.2020.1768318.

[33] Timothy D. West and Mark Blackburn. "Is Digital Thread/Digital Twin Affordable? A Systemic Assessment of the Cost of DoD's Latest Manhattan Project". *Procedia Computer Science*. Complex Adaptive Systems Conference with Theme: Engineering Cyber Physical Systems, CAS October 30 – November 1, 2017, Chicago, Illinois, USA 114 (Jan. 1, 2017), pp. 47–56. ISSN: 1877-0509. DOI: 10.1016/j.procs.2017.09.003.

[34] Jürg Meierhofer et al. "Digital Twin-Enabled Decision Support Services in Industrial Ecosystems". *Applied Sciences* 11.23 (Dec. 2, 2021), p. 11418. ISSN: 2076-3417. DOI: 10.3390/APP112311418.

[35] Conner Eddy et al. "Usefulness and Time Savings Metrics to Evaluate Adoption of Digital Twin Technology". WCX SAE World Congress Experience. SAE International, Jan. 24, 2023, p. 14. DOI: 10.4271/2023-01-0111.

[36] Rosario Davide D'Amico, Sri Addepalli, and John Ahmet Erkoyuncu. "Industrial Insights on Digital Twins in Manufacturing: Application Landscape, Current Practices, and Future Needs". *Big Data and Cognitive Computing* 7.3 (3 Sept. 2023), p. 126. ISSN: 2504-2289. DOI: 10.3390/bdcc7030126.

[37] Clement Fortin et al., eds. *Product Lifecycle Management in the Digital Twin Era: 16th IFIP WG 5.1 International Conference, PLM 2019, Moscow, Russia, July 8–12, 2019, Revised Selected Papers*. Vol. 565. IFIP Advances in Information and Communication Technology. Cham: Springer International Publishing, 2019. ISBN: 978-3-030-42250-9. DOI: 10.1007/978-3-030-42250-9.

[38] International Organization for Standardization. *Automation Systems and Integration — Digital Twin Framework for Manufacturing*. Version 2021. Switzerland, Oct. 2021. https://www.iso.org/obp/ui/en/#iso:std:iso:23247:-1:ed-1:v1:en (visited on 02/02/2024). Published.

[39] ITU-T. *Digital Twin Network - Requirements and Architecture*. Version 1.0. Feb. 13, 2022. https://www.itu.int/rec/T-REC-Y.3090-202202-I (visited on 05/22/2024). Approved.

[40] Kai Wang et al. "A Review of the Technology Standards for Enabling Digital Twin". *Digital Twin* 2.4 (Mar. 28, 2022). DOI: 10.12688/digitaltwin.17549.1.

[41] JaeSeung Song and Franck Le Gall. "Digital Twin Standards, Open Source, and Best Practices". *The Digital Twin*. Ed. by Noel Crespi, Adam T. Drobot, and Roberto Minerva. Cham: Springer International Publishing, 2023, pp. 497–530. ISBN: 978-3-031-21342-7. DOI: 10.1007/978-3-031-21343-4_18.

[42] Fei Tao et al. "Digital Twin in Industry: State-of-the-Art". *IEEE Transactions on Industrial Informatics* 15.4 (Apr. 1, 2019), pp. 2405–2415. DOI: 10.1109/TII.2018.2873186.

[43] Ibrahim Yitmen et al. "An Adapted Model of Cognitive Digital Twins for Building Lifecycle Management". *Applied Sciences* 11.4276 (2021). DOI: 10.3390/app11094276.

[44] *Digital Twins Definition Language*. Microsoft, May 9, 2022. https://github.com/Azure/opendigitaltwins-dtdl/blob/b3f36cf3fb6062a9da68ed4b085a17c7a1144763/DTDL/v2/dtdlv2.md (visited on 05/12/2022).

[45] Wolfram Luther et al. "Digital Twins and Enabling Technologies in Museums and Cultural Heritage: An Overview". *Sensors (Basel)* 23.3 (Feb. 1, 2023), p. 1583. ISSN: 1424-8220. DOI: 10.3390/s23031583. PMID: 36772623.

[46] Pita Ligaiula. *Could a Digital Twin of Tuvalu Preserve the Island Nation before It's Lost to the Collapsing Climate? | PINA*. Sept. 30, 2022. https://pina.com.fj/2022/09/30/could-a-digital-twin-of-tuvalu-preserve-the-island-nation-before-its-lost-to-the-collapsing-climate/ (visited on 04/13/2023).

[47] B. Danette Allen. "Digital Twins and Living Models at NASA". Digital Twin Summit. Nov. 4, 2021. https://ntrs.nasa.gov/citations/20210023699 (visited on 05/22/2024).

[48] Michael Grieves. *Physical Twins, Digital Twins, and the Apollo Myth*. Sept. 8, 2022. https://www.linkedin.com/pulse/physical-twins-digital-apollo-myth-michael-grieves/ (visited on 05/22/2024).

[49] Eric J. Tuegel et al. "Reengineering Aircraft Structural Life Prediction Using a Digital Twin". *International Journal of Aerospace Engineering* (Jan. 2011), pp. 1–14. ISSN: 16875966. DOI: 10.1155/2011/154798.

[50] Chelsey Levingston. *No Two Alike: Each Jet Engine Delivery Comes With Its Own Digital Thumbprint*. GE Aerospace. Mar. 15, 2021. https://www.geaerospace.com/news/articles/product-technology/no-two-alike-each-jet-engine-delivery-comes-its-own-digital-thumbprint (visited on 04/10/2024).

[51] Hyper Think. *The Digital Twin*. CSC Blogs. Sept. 8, 2016. https://web.archive.org/web/20161110134238/https://blogs.csc.com/2016/09/08/the-digital-twin/ (visited on 12/02/2021).

[52] Michael W. Grieves. "Virtually Intelligent Product Systems: Digital and Physical Twins". *Complex Systems Engineering: Theory and Practice*. Vol. 256. Progress in Astronautics and Aeronautics. American Institute of Aeronautics and Astronautics, Inc., Jan. 2019, pp. 175–200. ISBN: 978-1-62410-564-7. DOI: 10.2514/5.9781624105654.0175.0200.

[53] Fei Tao et al. "Digital Twin-Driven Product Design, Manufacturing and Service with Big Data". *Int Journal Advanced Manufacturing Technology* 94 (2018), pp. 3563–3576. DOI: 10.1007/s00170-017-0233-1.

[54] C. K. Lo, C. H. Chen, and Ray Y. Zhong. "A Review of Digital Twin in Product Design and Development". *Advanced Engineering Informatics* 48 (Apr. 1, 2021), p. 101297. ISSN: 1474-0346. DOI: 10.1016/J.AEI.2021.101297.

[55] Kendrik Yan Hong Lim et al. "A Digital Twin-Enhanced System for Engineering Product Family Design and Optimization". *Journal of Manufacturing Systems* 57 (Oct. 1, 2020), pp. 82–93. ISSN: 0278-6125. DOI: 10.1016/J.JMSY.2020.08.011.

[56] 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).

[57] Radhya Sahal, Saeed H. Alsamhi, and Kenneth N. Brown. "Personal Digital Twin: A Close Look into the Present and a Step Towards the Future of Personalised Healthcare Industry". *Sensors* 22.15 (15 Jan. 2022), p. 5918. ISSN: 1424-8220. DOI: 10.3390/s22155918.

[58] 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.

[59] Derrick de Kerckhove. "The Personal Digital Twin, Ethical Considerations". *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 379.2207 (Aug. 16, 2021), p. 20200367. DOI: 10.1098/rsta.2020.0367.

[60] Jacqueline Le Moigne and Benjamin Smith. *Advanced Information Systems Technology (AIST) Earth Systems Digital Twin (ESDT) Workshop Report*. NASA, Oct. 28, 2022. https://esto.nasa.gov/files/ESDT_Workshop_Report.pdf (visited on 05/21/2024).

[61] Angela Walters. *National Digital Twin Programme*. Sept. 7, 2019. https://www.cdbb.cam.ac.uk/what-we-did/national-digital-twin-programme (visited on 04/13/2023).

[62] Andy Walker. *Singapore's Digital Twin – from Science Fiction to Hi-Tech Reality*. Infrastructure Global. May 4, 2023. https://infra.global/singapores-digital-twin-from-science-fiction-to-hi-tech-reality/ (visited on 05/22/2024).

[63] Henning Kagermann, Wolfgang Wahlster, and Johannes Helbig. *Recommendations for Implementing the Strategic Initiative INDUSTRIE 4.0*. Germany: acatech, Apr. 8, 2013. https://en.acatech.de/publication/recommendations-for-implementing-the-strategic-initiative-industrie-4-0-final-report-of-the-industrie-4-0-working-group/ (visited on 03/20/2023).

[64] Ron S. Kenett and Jacob Bortman. "The Digital Twin in Industry 4.0: A Wide-Angle Perspective". *Quality and Reliability Engineering International* 38.3 (2022), pp. 1357–1366. ISSN: 1099-1638. DOI: 10.1002/qre.2948.

[65] Schneider Electric. *Schneider Electric Launches Digital Twin Software Solution*. June 3, 2022. https://www.businesswire.com/news/home/20220603005094/en/Schneider-Electric-Launches-Digital-Twin-Software-Solution (visited on 05/22/2024).

[66] Minh Duc Vu et al. "A Conceptual Digital Twin for Cost-Effective Development of a Welding Robotic System for Smart Manufacturing". *2nd Annual International Conference on Material, Machines and Methods for Sustainable Development (MMMS2020), 12-15 Nov. 2020*. Proceedings of the 2nd Annual International Conference on Material, Machines and Methods for Sustainable Development (MMMS2020). Lecture Notes in Mechanical Engineering (LNME). Cham, Switzerland: Springer International Publishing, 2021, pp. 1018–25. DOI: 10.1007/978-3-030-69610-8_134.

[67] Marie-Jane Bélanger, Robert Pellerin, and Samir Lamouri. "A Literature Review on Digital Twins in Warehouses". *Procedia Computer Science*. CENTERIS – International Conference on ENTERprise Information Systems / ProjMAN – International Conference on Project MANagement / HCist – International Conference on Health and Social Care Information Systems and Technologies 2022 219 (Jan. 1, 2023), pp. 370–377. ISSN: 1877-0509. DOI: 10.1016/j.procs.2023.01.302.

[68] Bernhard Wallner et al. "Digital Twin Development and Operation of a Flexible Manufacturing Cell Using ISO 23247". *Procedia CIRP*. 56th CIRP International Conference on Manufacturing Systems 2023 120 (Jan. 1, 2023), pp. 1149–1154. ISSN: 2212-8271. DOI: 10.1016/j.procir.2023.09.140.

[69] Young Jun Son, Richard A. Wysk, and Albert T. Jones. "Simulation-Based Shop Floor Control: Formal Model, Model Generation and Control Interface". *IIE Transactions* 35.1 (Jan. 1, 2003), pp. 29–48. ISSN: 0740-817X. DOI: 10.1080/07408170304428.

[70] João Vitor Arantes Cabral, Efraín Andrés Rodríguez Gasca, and Alberto Jose Alvares. "Digital Twin Implementation for Machining Center Based on ISO 23247 Standard". *IEEE Latin America Transactions* 21.5 (May 2023), pp. 628–635. ISSN: 1548-0992. DOI: 10.1109/TLA.2023.10130834.

[71] Alp Akcay et al. "Maintenance and Operations of Manufacturing Digital Twins". *Proceedings of the 2023 Winter Simulation Conference*. 2023 Winter Simulation Conference. San Antonio, TX: IEEE Press, Dec. 13, 2023, pp. 1888–1899. ISBN: 979-8-3503-6966-3. https://www.nist.gov/publications/maintenance-and-operations-manufacturing-digital-twins (visited on 02/07/2024).

[72] Guodong Shao. *Use Case Scenarios for Digital Twin Implementation Based on ISO 23247*. 400–2. National Institute of Standards and Technology, May 4, 2021. DOI: 10.6028/NIST.AMS.400-2.

[73] Yang Fu et al. "Digital Twin for Integration of Design-Manufacturing-Maintenance: An Overview". *Chinese Journal of Mechanical Engineering* 35.1 (June 23, 2022), p. 80. ISSN: 2192-8258. DOI: 10.1186/s10033-022-00760-x.

[74] Thomas H.J. Uhlemann, Christian Lehmann, and Rolf Steinhilper. "The Digital Twin: Realizing the Cyber-Physical Production System for Industry 4.0". *Procedia CIRP* 61 (Jan. 1, 2017), pp. 335–340. ISSN: 2212-8271. DOI: 10.1016/J.PROCIR.2016.11.152.

[75] Guodong Shao and Moneer Helu. "Framework for a Digital Twin in Manufacturing: Scope and Requirements". *Manufacturing Letters* 24 (Apr. 1, 2020), pp. 105–107. ISSN: 2213-8463. DOI: 10.1016/j.mfglet.2020.04.004.

[76] Thomas H.-J. Uhlemann et al. "The Digital Twin: Demonstrating the Potential of Real Time Data Acquisition in Production Systems". *7th Conference on Learning Factories (Clf 2017)*. Ed. by J. Metternich and R. Glass. Vol. 9. Amsterdam: Elsevier Science Bv, 2017, pp. 113–120. DOI: 10.1016/j.promfg.2017.04.043.

[77] Guodong Shao, Joe Hightower, and William Schindel. "Credibility Consideration for Digital Twins in Manufacturing". *Manufacturing Letters* 35 (Jan. 1, 2023), pp. 24–28. ISSN: 2213-8463. DOI: 10.1016/j.mfglet.2022.11.009.

[78] Change2Twin. *Objectives*. Change2Twin Project. https://www.change2twin.eu/about/objectives/ (visited on 04/10/2024).

[79] Valentina Zambrano et al. "Industrial Digitalization in the Industry 4.0 Era: Classification, Reuse and Authoring of Digital Models on Digital Twin Platforms". *Array* 14 (2022), p. 100176. ISSN: 25900056. DOI: 10.1016/j.array.2022.100176.

[80] Amazon Web Services BrandVoice. *The Future Of Energy: Using Digital Twins As A Strategic Asset At GE Digital*. Innovation. Dec. 7, 2021. https://www.forbes.com/sites/amazonwebservices/2021/12/07/the-future-of-energy-using-digital-twins-as-a-strategic-asset-at-ge-digital/ (visited on 04/10/2024).

[81] Goran Savić et al. "Towards a Domain-Neutral Platform for Sustainable Digital Twin Development". *Sustainability* 15.18 (18 Jan. 2023), p. 13612. ISSN: 2071-1050. DOI: 10.3390/su151813612.

[82] Robert Klar, Niklas Arvidsson, and Vangelis Angelakis. "Digital Twins' Maturity: The Need for Interoperability". *IEEE Systems Journal* 18.1 (Mar. 2024), pp. 713–724. ISSN: 1937-9234. DOI: 10.1109/JSYST.2023.3340422.

[83] 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.

[84] 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).

[85] International Organization for Standardization. *Information Technology-Cloud Computing-Interoperability and Portability*. International Standard. Version 2017-12. Dec. 2017. DOI: 10.3403/30313036U.

[86] Yining Huang et al. "Semantic Interoperability of Digital Twins: Ontology-based Capability Checking in AAS Modeling Framework". *2023 IEEE 6th International Conference on Industrial Cyber-Physical Systems (ICPS)*. 2023 IEEE 6th International Conference on Industrial Cyber-Physical Systems (ICPS). May 2023, pp. 1–8. DOI: 10.1109/ICPS58381.2023.10128003.

[87] Flavio Corradini et al. "Towards a Digital Twin Modelling Notation". *2022 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*. 2022 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech). Sept. 2022, pp. 1–6. DOI: 10.1109/DASC/PiCom/CBDCom/Cy55231.2022.9927827.

[88] Akram Hakiri et al. "A Comprehensive Survey on Digital Twin for Future Networks and Emerging Internet of Things Industry". *Computer Networks* 244 (May 1, 2024), p. 110350. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2024.110350.

[89] Yiwen Wu, Ke Zhang, and Yan Zhang. "Digital Twin Networks: A Survey". *IEEE Internet of Things Journal* 8.18 (Sept. 2021), pp. 13789–13804. ISSN: 2327-4662. DOI: 10.1109/JIOT.2021.3079510.

[90] Judith Michael et al. “Integration Challenges for Digital Twin Systems-of-Systems”. *Proceedings of the 10th IEEE/ACM International Workshop on Software Engineering for Systems-of-Systems and Software Ecosystems*. SESoS ’22. New York, NY, USA: Association for Computing Machinery, Nov. 9, 2022, pp. 9–12. ISBN: 978-1-4503-9334-8. DOI: 10.1145/3528229.3529384.

[91] Henrique Diogo Silva, Miguel Azevedo, and António Lucas Soares. “A Vision for a Platform-based Digital-Twin Ecosystem”. *IFAC-PapersOnLine*. 17th IFAC Symposium on Information Control Problems in Manufacturing INCOM 2021 54.1 (Jan. 1, 2021), pp. 761–766. ISSN: 2405-8963. DOI: 10.1016/j.ifacol.2021.08.088.

[92] Álvaro García, Anibal Bregon, and Miguel A. Martínez-Prieto. “Towards a Connected Digital Twin Learning Ecosystem in Manufacturing: Enablers and Challenges”. *Computers & Industrial Engineering* 171 (Sept. 1, 2022), p. 108463. ISSN: 0360-8352. DOI: 10.1016/j.cie.2022.108463.

[93] Hayley Bennett et al. *Towards Ecosystems of Connected Digital Twins to Address Global Challenges*. London: The Alan Turing Institute, May 18, 2023. DOI: 10.5281/zenodo.7840266.

[94] Jethro Akroyd et al. *National Digital Twin of the UK – a Knowledge-Graph Approach*. Dec. 18, 2020. https://como.ceb.cam.ac.uk/media/preprints/c4e-preprint-264.pdf (visited on 05/23/2024). Pre-published.

[95] Irene Karaguilla Ficheman and Roseli de Deus Lopes. “Digital Learning Ecosystems: Authoring, Collaboration, Immersion and Mobility”. *Proceedings of the 7th International Conference on Interaction Design and Children*. IDC ’08. New York, NY, USA: Association for Computing Machinery, June 11, 2008, pp. 9–12. ISBN: 978-1-59593-994-4. DOI: 10.1145/1463689.1463705.

[96] Cláudio Gomes et al. “Semantic Adaptation for FMI Co-Simulation with Hierarchical Simulators”. *SIMULATION* 95.3 (Mar. 1, 2019), pp. 241–269. ISSN: 0037-5497. DOI: 10.1177/0037549718759775.

[97] C. Human, A. H. Basson, and K. Kruger. “A Design Framework for a System of Digital Twins and Services”. *Computers in Industry* 144 (Jan. 1, 2023), p. 103796. ISSN: 0166-3615. DOI: 10.1016/j.compind.2022.103796.

[98] James Hetherington and Matthew West. *The Pathway Towards an Information Management Framework - a ‘Commons’ for Digital Built Britain*. Centre for Digital Built Britain, May 28, 2020. DOI: 10.17863/CAM.52659.

[99] Mary Bone et al. “Toward an Interoperability and Integration Framework to Enable Digital Thread”. *Systems* 6.4 (Dec. 18, 2018), p. 46. ISSN: 2079-8954. DOI: 10.3390/systems6040046.

[100] Amit V. Deokar and Omar F. El-Gayar. “A Semantic Web Services-Based Architecture for Model Management Systems”. *Proceedings of the 41st Annual Hawaii International Conference on System Sciences (HICSS 2008)*. Proceedings of the 41st Annual Hawaii International Conference on System Sciences (HICSS 2008). Jan. 2008, pp. 95–95. DOI: 10.1109/HICSS.2008.37.

[101] Milos Drobnjakovic et al. “Towards Ontologizing a Digital Twin Framework for Manufacturing”. *Advances in Production Management Systems. Production Management Systems for Responsible Manufacturing, Service, and Logistics Futures*. APMS 2023 IFIP International Conference. Vol. 689. IFIPAICT. Trondheim, NO: Springer, Cham, Sept. 21, 2023, pp. 317–329. ISBN: 978-3-031-43665-9. DOI: 10.1007/978-3-031-43666-6_22.

[102] D. Arena, F. Ameri, and D. Kiritsis. “Skill Modelling for Digital Factories”. *Advances in Production Management Systems. Smart Manufacturing for Industry 4.0*. Ed. by Ilkyeong Moon et al. Vol. 536. Cham: Springer International Publishing, 2018, pp. 318–326. ISBN: 978-3-319-99707-0. DOI: 10.1007/978-3-319-99707-0_40.

[103] Andrew D. Spear, Werner Ceusters, and Barry Smith. “Functions in Basic Formal Ontology”. *AO* 11.2 (June 22, 2016), pp. 103–128. ISSN: 18758533, 15705838. DOI: 10.3233/AO-160164.

[104] T Blochwitz et al. “The Functional Mockup Interface for Tool Independent Exchange of Simulation Models”. *Proceedings 8th Modelica Conference*. Modelica Conference. Dresden, Mar. 20, 2011. http://www.functional-mockup-interface.org (visited on 01/13/2022).

[105] Modelica Association Project FMI. *Functional Mock-up Interface Specification*. Specification. Version 3.0.2. Nov. 27, 2024. https://fmi-standard.org/docs/3.0.2/ (visited on 09/05/2025).

[106] Marcus Wiens, Tobias Meyer, and Philipp Thomas. “The Potential of FMI for the Development of Digital Twins for Large Modular Multi-Domain Systems”. *Proceedings of the 14th International Modelica Conference*. 14th International Modelica Conference. Linköping, Sweden, Sept. 27, 2021, pp. 235–240. DOI: 10.3384/ecp21181235.

[107] Modelica Association. *System Structure and Parameterization Standard*. Version 1.0.1. July 25, 2022. https://ssp-standard.org/publications/SSP101/SystemStructureAndParameterization101.pdf (visited on 03/11/2024).

[108] Dassault Systèmes. *The 3DEXPERIENCE Platform: Sustainable, Collaborative Innovation*. 2021. https://www.3ds.com/sites/default/files/2021-11/3dexperience-ebook-final.pdf (visited on 06/14/2024).

[109] International Organization for Standardization. *Part 11: Description Methods: The EXPRESS Language Reference Manual*. Version 2. Nov. 2004. https://www.iso.org/standard/38047.html (visited on 06/14/2024).

[110] John D. Hedengren et al. “Nonlinear Modeling, Estimation and Predictive Control in APMonitor”. *Computers & Chemical Engineering* 70 (Nov. 2014), pp. 133–148. ISSN: 00981354. DOI: 10.1016/j.compchemeng.2014.04.013.

[111] Marjan Sirjani et al. *Rebeca Modeling Language*. Rebeca Research Group, 2002. https://rebeca-lang.org/ (visited on 06/14/2024).

[112] P. C. Piela et al. “ASCEND: An Object-Oriented Computer Environment for Modeling and Analysis: The Modeling Language”. *Computers & Chemical Engineering* 15.1 (Jan. 1, 1991), pp. 53–72. ISSN: 0098-1354. DOI: 10.1016/0098-1354(91)87006-U.

[113] Sven Erik Mattsson and Hilding Elmqvist. “Modelica - An International Effort to Design the Next Generation Modeling Language”. *IFAC Proceedings Volumes*. 7th IFAC Symposium on Computer Aided Control Systems Design (CACSD ’97), Gent, Belgium, 28-30 April 30.4 (Apr. 1, 1997), pp. 151–155. ISSN: 1474-6670. DOI: 10.1016/S1474-6670(17)43628-7.

[114] *IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)– Framework and Rules*. Aug. 2010. DOI: 10.1109/IEEESTD.2010.5553440. Inactive-Reserved Standard.

[115] Simon J. E. Taylor. “Distributed Simulation: State-of-the-Art and Potential for Operational Research”. *European Journal of Operational Research* 273.1 (Feb. 16, 2019), pp. 1–19. ISSN: 0377-2217. DOI: 10.1016/j.ejor.2018.04.032.

[116] Francesco Longo et al. “Distributed Simulation for Digital Twins: An Application to Support the Autonomous Robotics for the Extended Ship”. *2022 IEEE/ACM 26th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. 2022 IEEE/ACM 26th International Symposium on Distributed Simulation and Real Time Applications (DS-RT). Sept. 2022, pp. 179–186. DOI: 10.1109/DS-RT55542.2022.9932057.

[117] Eugene Y. Song et al. “IEEE 1451 Smart Sensor Digital Twin Federation for IoT/CPS Research”. *2019 IEEE Sensors Applications Symposium (SAS)*. 2019 IEEE Sensors Applications Symposium (SAS). Mar. 2019, pp. 1–6. DOI: 10.1109/SAS.2019.8706111.

[118] Daniel R. Dolk and Jeffrey E. Kottemann. “Model Integration and a Theory of Models”. *Decision Support Systems*. Model Management Systems 9.1 (Jan. 1, 1993), pp. 51–63. ISSN: 0167-9236. DOI: 10.1016/0167-9236(93)90022-U.

[119] Arthur M. Geoffrion. “An Introduction to Structured Modeling”. *Management Science* 33.5 (May 1, 1987), pp. 547–588. DOI: 10.1287/mnsc.33.5.547.

[120] Christopher V. Jones. “An Introduction to Graph-Based Modeling Systems, Part I: Overview”. *ORSA Journal on Computing* 2.2 (May 1990), pp. 136–151. ISSN: 0899-1499. DOI: 10.1287/ijoc.2.2.136.

[121] Ahsan Qamar et al. “Dependency Modeling and Model Management in Mechatronic Design”. *Journal of Computing and Information Science in Engineering* 12.041009 (Dec. 11, 2012). ISSN: 1530-9827. DOI: 10.1115/1.4007986.

[122] Microsoft Corporation. *Digital Twins and Their Twin Graph*. Azure Digital Twins. Jan. 3, 2024. https://learn.microsoft.com/en-us/azure/digital-twins/concepts-twins-graph (visited on 06/14/2024).

[123] Jiadi Du and Tie Luo. *Digital Twin Graph: Automated Domain-Agnostic Construction, Fusion, and Simulation of IoT-Enabled World*. Apr. 19, 2023. arXiv: 2304.10018 [cs]. http://arxiv.org/abs/2304.10018 (visited on 06/13/2024). Pre-published.

[124] Jethro Akroyd et al. “Universal Digital Twin - A Dynamic Knowledge Graph”. *Data-Centric Engineering* 2 (Jan. 2021), e14. ISSN: 2632-6736. DOI: 10.1017/dce.2021.10.

[125] Hou Yee Quek et al. “Dynamic Knowledge Graph Applications for Augmented Built Environments Through “The World Avatar””. *Journal of Building Engineering* 91 (Aug. 2024), p. 109507. ISSN: 23527102. DOI: 10.1016/j.jobe.2024.109507.

[126] Maryna Waszak et al. “Let the Asset Decide: Digital Twins with Knowledge Graphs”. *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*. 2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C). Mar. 2022, pp. 35–39. DOI: 10.1109/ICSA-C54293.2022.00014.

[127] Gilles Privat. “Graph Models for Systems-of-Systems Digital Twins” (2021). DOI: 10.13140/RG.2.2.23047.93602.

[128] Aidan Hogan et al. “Knowledge Graphs”. *ACM Comput. Surv.* 54.4 (July 2, 2021), 71:1–71:37. ISSN: 0360-0300. DOI: 10.1145/3447772.

CHAPTER 6

A Universal Foundation for Digital Twins

This chapter is based on the paper "Constraint Hypergraphs as a Unifying Framework for Digital Twins," which is under review in the IEEE Systems Journal [1]. It is available as a preprint on arXiv [2].

Abstract: Digital twins, used to represent physical systems, have been lauded as tools for understanding reality. Complex system behavior is typically captured in domain-specific models crafted by subject experts. Contemporary methods for employing models in a digital twin require prescriptive interfaces, resulting in twins that are difficult to connect, redeploy, and modify. The limited interoperability of these twins has prompted calls for a universal framework enabling observability across model aggregations. Here we show how a new mathematical formalism called a constraint hypergraph serves as such a framework by representing system behavior as the composition of set-based functions. A digital twin is shown to be the second of two coupled systems where both adhere to the same constraint hypergraph, permitting the properties of the first to be observable from the second. Interoperability is given by deconstructing models into a structure enabling autonomous, white-box simulation of system properties. The resulting digital twins can interact immediately with both human and autonomous agents. This is demonstrated in a case study of a microgrid, showing how both measured and simulated data from the aggregated twins can be provided regardless of the operating environment. By connecting models, constraint hypergraphs supply scientists and modelers robust means to capture, communicate, and combine digital twins across all fields of study. We expect this framework to expand the use of digital twins, enriching scientific insights and collaborations by providing a structure for characterizing complex systems.

©2025 IEEE. Reprinted, with permission, from John Morris et al. "Constraint Hypergraphs as a Unifying Framework for Digital Twins". *Under review with IEEE Systems Journal* (July 2025). DOI: 10.48550/arXiv.2507.05494.

6.1. Introduction

Every thing in the world is a system, and every scientific endeavor is fundamentally concerned with describing those systems and their behaviors. A system is a collection of things that affect the world in unique ways when arranged together. Whether seed or rainforest, child or civilization, transistor or satellite, a scientist works to understand how individual parts combine to produce the actions of the whole [3]. Explanations of system behavior are expressed in modeling languages such as the algebraic models used to represent systems of numbers, hierarchical charts for organizations, and diagrams of electric circuits. By default, a model represents only a virtual system, meaning the system it describes exists solely as information. For example, a model of child development describes not a specific child, but a virtual child representing all children. But additional, specialized information can be revealed by building models that describe a single, physical system. Such specific representations are termed digital twins (DTs), and are employed to expose the complex interactions of real systems, allowing previously indiscernible relationships to be described. DTs leveraging modern data processing capabilities have been proposed for solving problems in both scientific and industrial communities such as medical diagnostics [4], global weather forecasting [5], and machine maintenance [6].

Because of the complexity of the systems they represent, high fidelity DTs end up being excruciatingly intricate. Their convolutions result in fragile representations that are difficult to maintain [7], resistant to reuse and redeployment [8], and prone to failure [9]. More disconcerting is the difficulty of connecting manually configured DTs with external agents, including other DTs [10]. As the scope of studied systems increases to include more domains, DTs must aggregate into systems of systems, sharing information and models with other DTs [11]. The challenge of forming interoperable DTs has been described as one the most significant facing modelers [12, 13], prompting calls for a universal representation schema for DTs [14–16] that "is reusable across multiple domains, supports multiple diverse activities, and serves the needs of multiple users" [17].

Though many frameworks have been proposed [13, 15, 16, 18–26], the authors observe that each is inhibited by the dualism of expressiveness at the cost of interpretability. Knowledge graphs [13, 16, 24, 25] represent the former end of the spectrum. With their connections defined arbitrarily, knowledge graphs can represent any set of objects and their relationships [27]. However, to interpret the meaning

of a relationship requires an extensive ontology, restricting their ability to externally integrate [20, 28]. Alternatively, more explicit frameworks may fully convey meaning, but only apply to systems within a singular domain [18, 20], or with limited types of relationships such as Bayesian probability models [26, 29].

The purpose of this paper is to propose a new framework for DTs that solves the paradox of generalized interpretation by deconstructing system behaviors into the composition of set-based functions. The resulting schema is mathematically rigorous, enables deterministic simulation of systems from and across any domain, and provides mechanisms for ready integration of models with external agents. This framework is demonstrated through building a DT for a microgrid, following the validation methodology of Pedersen et al. [30]. Although the application and formulations of this framework are new, its methods will be familiar to mathematicians and computer scientists versed in the principles of the lambda calculi [31] and functional programming [32]. It is the hope of the authors that applying these principles to DTs will empower scientists and engineers in every field to solve the complex systems framing society's most critical challenges.

6.2. Overview of Digital Twins

The concepts of systems modeling are found in many fields, often resulting in competing interpretations of common words. The purpose of this section is to precisely define some of these terms as employed by the authors in the main paper. These interpretations are specific to this paper and intended to be supplementary in nature; they are not meant to serve as a formal standard or supersede existing definitions provided by works better suited for this purpose [11, 17, 33–38]. The traditional definition of a DT is of a virtual representation of a physical system [33, 34, 39]. However, this definition conflicts with the worldview where actions in the physical domain can only be carried out by physical entities. Modern interpretations of DTs ascribe a verbose range of services for which a DT might be used for: decision-making [35], receiving [40] and recording [41] system data, validating [42] and updating [37, 43] system models, generating [44] and predicting [11] system states, controlling physical systems [45], and communicating signals [46, 47]. To perform these physical actions, a DT must have some physical aspect to it beyond being solely defined as a virtual entity. Carefully exploring the ontological nuances of a DT allows greater consideration of how they can best represented.

The goal of any agent is to affect the world to achieve some specific outcome. This is true whether the agent is a swallow landing on a delicate branch, a synthetic automaton milling a workpiece, or a scientist trying to prevent the rise of global temperatures. To properly influence the physical world requires understanding of its complex network of interacting elements. Framing and characterizing the real world creates structures that are virtual, not physical; composed not of matter, but of information. Although entities in these two domains do not interact with each other, systems in the physical world are better influenced by agents who understood them virtually.

Definition 5 (Virtual Domain) *The domain whose quanta are information (always taken to be discrete), the associations of which describe phenomena observed in the physical domain, and whose bounds are finite.*

Definition 6 (Physical Domain) *The domain existing in space and time, the phenomena of which are described virtually. Used synonymously with "real world."*

The physical domain is taken as including elements characterized as *cyber*, including electromagnetic signals, transistor networks, and visual displays.

Definition 7 (Phenomenon) *Some distinguishable portion or aspect of the physical domain.*

Note that the physical domain includes both the tangible and intangible, such as phenomena that exist only in the future. This is important when considering DTs that represent these phenomena despite their immateriality. A key aspect of virtually representing elements of the real world is distinguishing the changes that result from interaction. This necessitates a definition of *state*:

Definition 8 (State) *The description of a phenomenon by assigning a unique datum to each distinctive arrangement, such that the phenomenon is characterized by only one value within any single frame of consideration. Used synonymously with "state variable".*

Note how the authors' use of *state* differs from the collective sense of the word as used in phrases such as the "the total state of a system." The authors refer to this as the *state vector*.

Definition 9 (Observation) *The provision of a datum for a system state belonging to a single frame of consideration.*

The values of a state are termed *data*. Because the primary aspect of a datum is that it can be distinguished from another datum, the authors equate these data with the *information* given as the quanta of the virtual domain [48]. The arrangement of states forms a system, defined in part here although a more comprehensive and philosophical treatment of the topic was given by Cellier [49].

Definition 10 (System) *A general description of some phenomena, consequently a virtual entity that represents the physical domain as a collection of states. The term "physical system" refers to the phenomena being described, while the term "model" refers to the virtual depiction.*

Definition 11 (System Behavior) *The restriction of the state values a system can exhibit within a unique frame of consideration [50].*

With this terminology, the goal of an agent interacting with the physical domain can be expressed as bringing some system to a set of desired states. A DT should assist with this task, providing some advantage in either understanding or manipulating the complexities of a physical system. There are three processes by which a DT can do this: measurement, control, and simulation.

Definition 12 (Measurement) *An observation made of the physical domain following specific interaction with it by an agent.*

Definition 13 (Control) *The manipulation of the physical domain so that it exhibits a specific set of states.*

Definition 14 (Simulation) *An observation of a physical system by an agent that is not coupled along the state to be observed.*

Each of these tasks are physical, and consequently must be performed in the physical domain by a second physical system, called a twin, that is coupled with the SOI.

Definition 15 (Coupling) *The synchronization of two phenomena such that changes in one engender changes in the other. These changes occur across relationships between synchronized states.*

The need for bidirectional coupling is well recognized as a requirement for a DT [34, 37], as each process is enabled by a distinctive form of synchronization. Measurement is conducted by coupling

the states of the SOI to the twin, so that changes in the first can be measured by observing the latter. The reverse of this process is control, where arrangements of the twin cause the SOI to form a desired configuration. Simulation is based on the independent evaluation of the twin, yet still requires coupling with the SOI across the initial states from which the twin evolves. Additionally, simulation is only effective if the two systems behave similarly, so that observing of the twin informs the agent about the unmeasured state of the SOI.

The advantages in evaluating a twin versus the SOI come only if the twin has greater observability. Twins should also be highly configurable, so that the cost of matching the behavior of the SOI is minimized. Digital systems, whose states are described by Boolean variables, optimally provide these traits. Because of this, the twin is nearly always implemented on a digital computer, and is consequently termed a DT. DTs under this interpretation are physical rather than virtual, generally composed of transistors characterized by Boolean states. The example in Figure 6.1 provides an intuitive demonstration a DT, where the digital counter allows the states of the lightbulb system to be understood either through measurement (where the lightbulb system informs the digital one) or simulation (where the DT infers the state of the SOI). This motivates the following definition:

Definition 16 (Digital Twin) *A digital system coupled to another system so that each can affect the states of the other, configured so that measurement of the digital system engenders observation of all desired states of the other system.*

Because these processes are performed via the DT, it is always possible to enact them by manipulating and observing the DT's states. Both measured and simulated values are relayed to an agent through observation, while control sequences are triggered by certain states and verified by others. From this, the authors posit that for a framework to represent a DT it is sufficient to provide mechanisms for setting and observing the state of a DT.

Given this minimal interpretation, it is necessary to clarify the usage of DTs in practical deployments. Although control is not the primary purpose of a DT, it is often used to represent the control aspects of a real system [51]. For each of the functionalities of a DT listed at the beginning of the section there is some aspect of the physical domain that must be influenced, whether transistors in persistent memory, actuation of a manufacturing machine, or illuminating pixels on a display. A virtual entity

does not perform these physical actions, rather it describes the system behaviors that lead to these actions occurring. An agent that understands the virtual description can then configure "the physical system [to] emulate the model" [52]. For instance, the microgrid DT in Figure E.2 could be used to turn on and off the diesel generators (as shown in Figure 6.5) when the balance of the grid is overloaded. While the actual automation would occur along some coupling between the digital system and diesel generators, the representation of this action would be given by relating the balance of the grid (a state) with the status of the diesel generators (another state), the first serving as the trigger and the second one verifying that the action was performed correctly. Observing these values and enforcing their relationship is equivalent to performing the control task, assuming that the twin and SOI are coupled bidirectionally.

Semantic clarity is obtained by realizing that services provided by a DT (such as automation or visualization) are simply phenomena of the real world. Including these phenomena in the SOI allows them to be measured and influenced by the DT. The authors have demonstrated this separately by performing automating solid geometry modeling by expressing the functions of Computer-Aided Design (CAD) software within a constraint hypergraph [53]. It is consequently their belief that the minimalistic interpretation of a DT as a tool for measurement, control, and simulation includes all possible services that can be configured on a digital platform; a generalization rather than an exclusion that will hopefully allow DTs to find use in a variety of fields and applications.

6.3. Nature of Digital Twins

To maximize the application of the proposed framework, the authors have elected to consider DTs as generally as possible, focusing on the essential functionalities that a DT must provide. The definition of a DT is commonly given as a virtual representation of a physical system [34, 38, 39], used to inform an agent (whether human or automaton) about the state of some system of interest (SOI) in lieu of direct measurement. Their usefulness stems from the enhanced observability of a digital system, where information about the system is obtained with greater ease from the DT than the SOI [17]. Exposing system states allows agents to make decisions within the context of the SOI [44], converting virtualizations into actions influencing the real world. A state is a property of a system that is distinguishable from something else [48], such as the status of a lightbulb or the fuel level of a generator, and which can vary

(or evolve) over a range of values [50]. In this way an agent can distinguish between the lightbulb being *on* or *off*, or fuel level being *full* versus *empty*. The goal of an agent is to observe a set of state values exhibited from the SOI. If all states can be directly measured, then no DT is necessary. This is, however, not predominantly the case, necessitating the construction of a DT that informs the agent where direct observation of the SOI is untenable. The DT performs this through one of two mechanisms: direct coupling, where the DT updates in accordance with corresponding properties of the SOI; or simulation, where the DT evolves under a prescribed manipulation from some initial conditions until it approximates the unmeasured facts [49]. The combination of these allow an agent to observe the SOI's state through indirect measurements of the DT. Both mechanisms require a connection from the SOI to the DT enabling the latter to reflect the states of the former [11]. The authors posit that a digital system providing these mechanisms, along with the necessary intersystem connection, constitutes a DT.

A mock example of a DT is introduced in Figure 6.1 to motivate this limited definition. In the figure is an electrical circuit consisting of two lightbulbs connected by a bimodal switch such that only one lightbulb is illuminated at any instance. The disjointed bulbs form the SOI for some agent who desires to know which light is illuminated. To this end, the agent assigns two values to each lightbulb corresponding to whether the lightbulb is recognized as illuminated (*on*) or not (*off*). If the SOI is fully observable then discovering the current status of each lightbulb is trivial. A more significant case to consider is if the lightbulbs are obscured, preventing direct measurements. A solution involves the construction of the DT shown on the right of Figure 6.1, consisting of a box containing a pair of counters each capable of showing boolean integers zero or one. This DT is coupled to the SOI (perhaps via photoresistive sensors) so that the counters correspond to the various states of each lightbulb: one for *on* and zero for *off*. In this context, coupling refers to connecting the signals from one system to another, so that part of the state of one system is shared by the other. The degree of coupling determines whether the prescribed states of the SOI can be identified through observations of the DT.

It is, however, more likely that a SOI cannot be fully coupled to a DT, preventing some facts of the SOI from being exposed to the DT. To exhibit these data, the DT must be configured to have the same behavior as the SOI. Then the SOI and the DT will evolve commensurately, and the facts of the SOI can still be exposed. In the mock example, the behavior of the SOI is shown at the top of Figure 6.1, which describes the mutual exclusion of the states for the first and second lightbulbs. This behavior

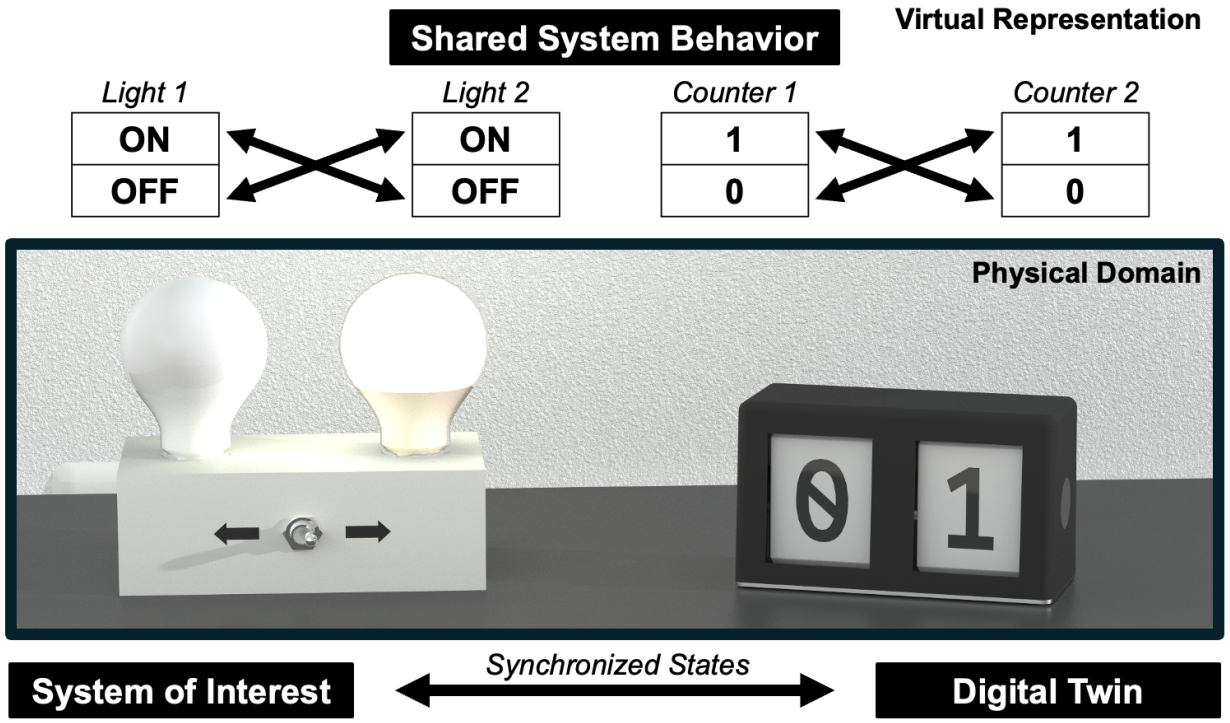


Figure 6.1: A visual description of a DT, with the system of interest and digital twin shown on the left and right respectively in the bottom (physical domain), and a virtual representation of shared behavior of the two systems at the top (virtual domain). For this example, the system of interest consists of two lightbulbs connected by a bimodal toggle switch such that only one bulb illuminates in either position of the switch. The digital analog similarly possesses two digital counters capable of displaying only two boolean numbers, 0 or 1.

can be mimicked by connecting the boolean counters of the DT so that the left counter only displays the opposite value of the right. After implementing this behavior, the DT needs only to observe a single light to manifest the states of the entire system. Note that DT and SOI do not share any common physical properties: the SOI is electrical; the DT is mechanical, with digital counters. Yet the interpretation of the two systems results in the same virtual specification of a set of states (represented by two boolean variables) and the behavior relating these states.

6.4. Representing System Behaviors

This example shows that a DT is a physical, digital system configured in such a way that, when observed, facts of another system can be understood. By necessity, this digital system is programmed to behave similarly to the SOI, so that experimenting on the DT produces the same output as corresponding

experiments of the SOI [54]. Any framework proposed for constructing a DT must explicitly reproduce this shared behavior, motivating consideration of how system behavior is defined for simulation.

Simulation is only possible if the agent can discover a relationship between known and unknown facts. These relationships describe a system, and the agent's description of these relationships is a system model [49]. Each relationship describes the effect of one variable on another. Willems [50] showed that the specification of all such relationships constitutes a system's behavior, and further that the effect of any behavior can be described as the restriction of the affected variable's possible values. For example, each light bulb in Figure 6.1 is constrained to only manifest the alternate state of its neighbor. Collecting restrictive relationships together results in an underdetermined model of a virtual system.

Reality is generally much more deterministic [52]. Merely reducing a variable's exhibitable values is not indicative of a reified system, which must be temporally consistent, meaning that each variable exhibits only a single value within any frame of consideration [52]. To simulate an unknown fact, an agent must discover relationships that reduce the set of possible values for a variable to a single datum. A function is the algebraic mechanism for mapping a value of one set to a value of another, consequently, a simulatable system model is one composed of functions. This can be illustrated by assigning variables to the bimodal lightbulb system in Figure 6.1, namely `light1` and `light2`, which each can exhibit the states *on* or *off*. The behavior of the first bulb is given by the constraint between `light2` and `light1`, described by a function f in Equation 6.1:

$$\texttt{light2} \xrightarrow{f} \texttt{light1} \coloneqq \begin{bmatrix} \texttt{on} \\ \texttt{off} \end{bmatrix} \not\pitchfork \begin{bmatrix} \texttt{on} \\ \texttt{off} \end{bmatrix} \tag{6.1}$$

As long as the SOI and DT in Figure 6.1 share states and behavior, equivalent state variables and functions can be applied to the digital display, as in Equation 6.2, where g is equivalent to f.

$$\texttt{counter2} \xrightarrow{g} \texttt{counter1} \coloneqq \begin{bmatrix} 1 \\ 0 \end{bmatrix} \not\pitchfork \begin{bmatrix} 1 \\ 0 \end{bmatrix} \tag{6.2}$$

The collection of all known prescriptive functions can be arranged as edges in a graph, with each edge connecting nodes that represent the system variables they relate. In multiple-arity relationships the function maps each combination of the domain variables to a unique value in the codomain. The

inclusion of multiple variables (or nodes) in the domain makes the relationship a hyperedge, and the resulting holistic structure is termed a constraint hypergraph (CHG).[1] A technical definition is provided in [55], which establishes that any explicit system behavior can be represented in a CHG. This is important for a potential DT framework, which must reconcile any models used to stipulate the behavior shared between the DT and SOI into a unified representation. While hypergraphs have been used for representing cyber-physical systems before [56–60], they have not been configured to describe system behaviors.

An example of a CHG is shown in Figure 6.2, where the arrows (edges) show how the state variable comprising the edge's codomain can be constrained given values for each node in the edge's domain. Simulation is conducted by traversing a path from a set of nodes with known values to a node whose value is desired to be simulated.

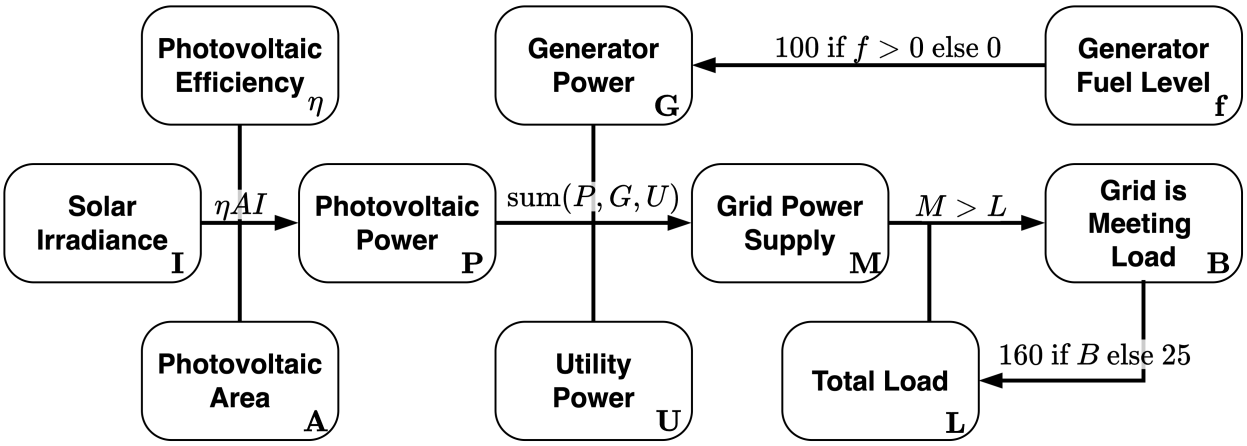


Figure 6.2: A simple CHG of a microgrid, with nodes for three grid actors (a photovoltaic array *P*, diesel generator *G*, and utility connection *U*) along with associated nodes. The behavioral functions that relate them are given by the arrows (pointing to the target) labeled with the algebraic rule governing the mapping.

6.5. Implementation of Digital Twins

Redefining a DT as a physical system sharing the behavior of another system clarifies what essential actions a framework must provide to implement a DT. In a system whose scope is not changing, the authors posit that there are two primary actions that a DT framework must support: (a) coupling the

[1] CHGs are described most succinctly as a partial subcategory of **Set**.

SOI to the DT, so that the DT has a shared-state with the SOI; and (b) configuring the DT to share the behavior of the SOI. Additionally, if the scope of the twinned system is changing, a framework must additionally (c) provide mechanisms ensuring the twinned representation is valid in the redefined context.

The first action is perhaps the most commonly provided functionality, generally implemented by sensors observing the SOI to the DT, such as an Internet of Things (IoT) network. Such a platform allows measurements of the SOI's state to be reflected in some repository, establishing a fully serviceable DT only if all information of the SOI is collected in the database–a rarity for representations of complex systems. To manifest information that cannot be measured, a DT must also twin the SOI's behavior. This occurs through the provision of models to the DT, which describe how observed inputs may be transformed into unobserved state values. CHGs adeptly perform these two tasks, storing all considered facts as nodes in the systems, and deconstructing models into the edges relating them. A serviceable DT further provides mechanisms for simulating the behavioral models. In addition to fulfilling the first two actions, the primary advantage of using a CHG to represent a DT is this ability to autonomously compose simulation processes. The combination of coupled states and simulatable models allows all desired information of the SOI to be observed via the DT, whether that data is measured or simulated, fulfilling the purpose of a DT.

While this alone merits consideration of a CHG as a formal framework for DTs, additional advantages derive from their ability to be redefined in different contexts. This last action required for representing a DT–proving a valid representation following arbitrary changes of the SOI's scope–is often characterized as interoperability [12] or extensibility [61], with the challenge of providing it being described the most significant barrier to widespread adoption of DTs [62]. Adapting a definition typically employed for software development to a DT, extensibility is the twin's ability to validly manifest a SOI's facts despite changes to the SOI's scope [63]. A model referencing the weight of a vehicle, for instance, might be extensible for changes of the vehicle's simulated location only as long as the location is on the Earth.

Providing extensibility for DTs adds additional complexity due to the expected use of a DT in forming system of systems representations [16, 45]. Combining DTs into arbitrary aggregate systems is akin to modifying the scope of the original SOI; for example, a DT of a tire and of a car might be combined,

expanding the scope of the SOI to the tire + car system. An effective framework ensures the representation of the individual systems remains valid across such scope changes, or else highlights when the representation is potentially invalid. This is especially important given the notion that systems are continuously changing, necessitating DTs that can evolve their virtual representations alongside the physical SOI [37, 64].

Decomposing a system into facts and behavioral constraints reveals the two vectors along which system scopes can change: first, adding or removing system data; and second, updating the relationships between that data. The two form the opposing sides of a modeling dilemma referred to as the "Expression Problem" [65, 66], which describes the difficulty of guaranteeing consistency after arbitrary updates to a model. Famously, a framework can generally be configured to be extensible for changes to either behaviors (relationships) or data, but not both [67]. Such configured frameworks are respectively described as either functional or procedural [68]. CHGs are intrinsically functional. This means that, while adding or removing facts in the system can indeed affect a CHG's consistency, modifications of a CHG's edges do not affect the validity of its representation, a characteristic referred to as having no side effects [32]. This is especially useful for DTs, which generally concern an explicitly defined system. Behavioral consistency also allows users to update the models of a DT without needing *a priori* knowledge of the global system, greatly reducing the sensitivity of maintaining a convoluted DT architecture. Examples and further discussion on these points are provided in the Methods section.

6.6. Overview of Microgrid Demonstration

The claim of CHGs being used as a framework for deploying DTs was validated following the methodology of Pedersen et al. [30] by specifically demonstrating a CHG-based DT made for microgrid system. A microgrid is an energy grid that operates semiautonomously from a standard utility network, provide additional flexibility to organizations whose needs go beyond the capabilities of established grids, such as renewable energy producers [69]. Microgrids are composed primarily of energy sources and sinks, here referred to as actors. The system in question, overviewed in Figure 6.3, consists of an arrangement of grid actors including two diesel generators, a photovoltaic array, a battery energy storage system (BESS), and several energy loads scaled at three possible levels. The system also accounts for connections to the utility grid. Determining the behavior of the microgrid, including the power pro-

duced or consumed by its various actors, requires knowledge of loads on the system, the viability of all intermediary connections, and live information on energy fluctuations, such as the availability of solar-generated power. A DT might be used for monitoring purposes as well as testing, with specific importance placed on describing the resilience of the grid to cases of random failure or sabotage [70, 71].

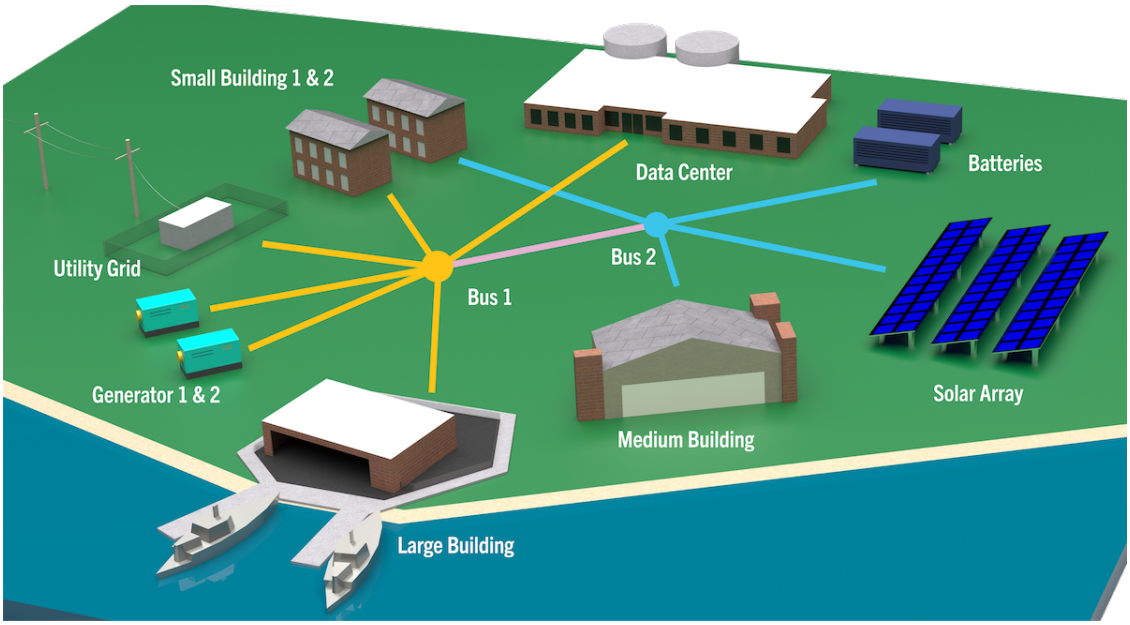


Figure 6.3: Overview of the physical microgrid system showing ten grid objects connected along two busses: five capable of providing a supply (generators, utility grid, solar cells, and the batteries) and six objects that may receive power (buildings, data center, and batteries).

The microgrid DT was individually validated by amending it to represent a test-scale microgrid that was operated over a three-day span in Monterey, California in the United States. This bench-top microgrid included a photovoltaic array, a BESS, one diesel generator, and an applied resistive load of approximately 0.7 kW. The DT represented these grid actors by using data inputs from the solar cells and load to simulate the states of the BESS and generator. Comparisons between the test run [72] and simulated values are shown in Figure 6.4, visually demonstrating its fidelity to the real system.

6.6.1. Development

The broader CHG was derived from standalone, generic models originally developed by researchers associated with the U.S. Navy [73–75]. These models were developed in largely imperative formats, such

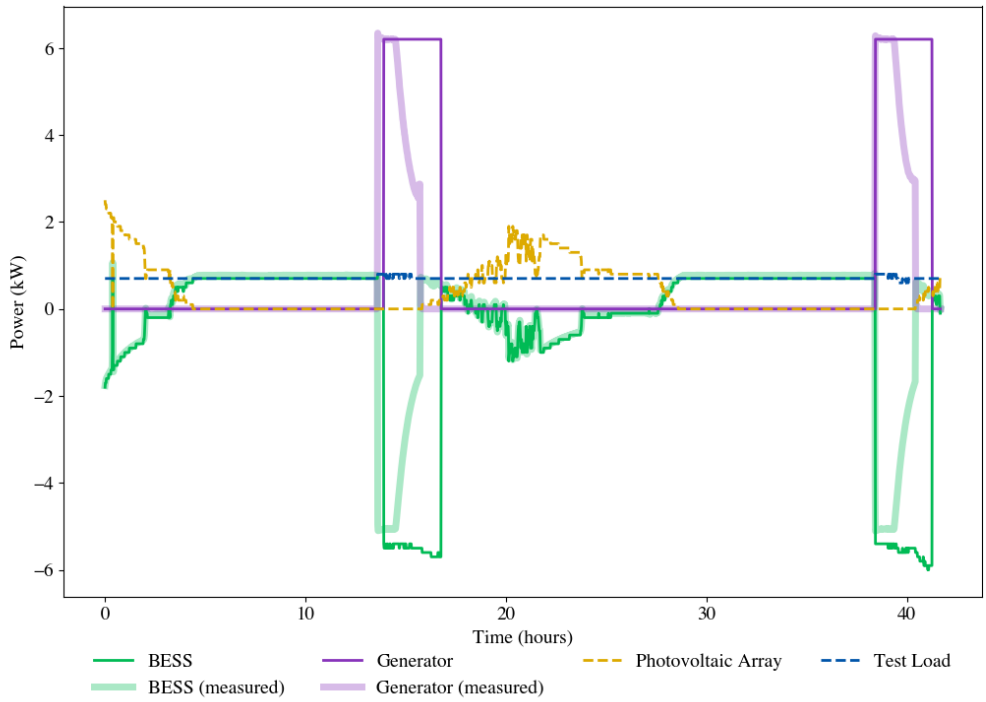


Figure 6.4: Validation of microgrid DT Model comparing data of the states of the four grid actors both simulated and compared to values observed during a 43-hour test run of a limited scale microgrid in Monterrey, California in April 2025. The dashed lines indicate observed values (directly coupled to the DT), while the solid lines indicate simulated data. Simulated data is overlaid on top of translucent lines indicating the data observed from the validation run.

as Microsoft Excel workbooks, MATLAB scripts, and Python modules. By deconstructing these into a CHG, the models could be simulated declaratively, rendering an effective DT. The process of conversion from a procedural model to a CHG involved four steps:

1. **Identify facts from the system.** Facts are any piece of information identified in the modeling process, including variables, parameters, or outputted data. These are listed as the nodes of the CHG in-preparation. Facts that were measured from the real system, such as solar irradiance or building usage rates, were set as initial values of their corresponding nodes, establishing the important connection between the virtual model and the physical SOI [37].
2. **Identify relationships between these facts.** Each relationship is described as an edge in the CHG. The partial hypergraph is visualized in Figure E.2, with edges representing static database queries, discrete-event simulation, linear system solving, stochastic relationships, and file input/output.

3. **Organize the model.** The described nodes and edges are parsed and stored into a persistent collection.
4. **Interrogate the model.** An autonomous agent capable of searching the graph provides observations of the underlying SOI by connecting known inputs with desired outputs along discovered paths in the graph.

594 variables were identified for Step 1, corresponding to various parameters and states of the microgrid system, as well as settings for the simulation itself. These are tabulated in the Table D.4. The collection of all variables forms a state vector, and encodes every dimension along which the system (as described) may evolve. These states blend common interpretations of properties, attributes, and class variables. For instance, the states of a BESS include its current charge level, capacity, but also properties such as its maximum output. If any of these are observable to the modeler–such as from vendor specification sheets–they can be provided as inputs to the CHG, in which case the simulation of that node is the trivial provision of the input node's value (where input equals output). Other properties in the system include system facts conveying simulation settings, data directories, and timing variables. The latter is an interesting aspect of CHGs, where time is considered an observed state of a system, rather than as a universal property (such as in system formalisms given by Wymore [76] or Ziegler [77]). The benefit of this is that time can be considered uniquely for composing subsystems. Reconciling two models is contingent upon reconciling time steps and counters, a process captured in reconciling the CHGs.

The edges for each system are based on imperative models written by Peterson [75] that were converted to a functional structure. The relationships are of a varied nature: some are algebraic, such as the relationship between the number of seconds in a minute and the number of seconds in a year, others are more procedural, such as the functions calculating whether the BESS should be charging based on its charge levels and usage requirements. In every case, functions represent the lowest level of abstraction in the system–a black box within which the solver does not consider system complexity. These black boxes can be incredibly complex if needed; the function relating which grid members should be utilized over a given time step, for example, is calculated via a series of Python methods. By abstracting these steps into a single function, all the sub-variables and relationships in the methods are abstracted away so that they cannot be connected to other nodes in the graph. This is the unfortunate necessity

of any system model, which must include a scope beyond which entities are not considered. Functions give this primitive level of abstraction.

6.6.2. Use of the Microgrid Digital Twin

Every path in the hypergraph is a potential simulation pairing a set of known inputs to a unique output. Deploying a CHG following its construction requires an engine capable of storing a CHG, parsing its members, and searching for a path mapping a set of inputs to an output. Such an engine may accomplish these tasks through any means, with its specific implementation affecting its efficiency of computation. Note that the performance of an engine is only tangentially related to the theoretical proposals here, and is consequently not explored in this study. The engine used by the authors for this article was a custom solver written in the Python programming language titled *ConstraintHg*[78], which employs a breadth-first search algorithm. Measured values (system information that is provided as an input) are given by discovering the trivial loop leading from a node to itself. In this case the engine acts as a database-management system: locating the fact's corresponding node and returning the stored value. Non-trivial paths supply simulated information, such as the projected cost of fuel for running the generator. In this case the agent seeds possible paths starting from each input, extending each path (a tree with inputs as its leaves) until one reaches the node corresponding to the desired output. The edges in the path represent the series of functions that, when composed, map the input set of the path's leaves to the path's root. The result of calculating each of these functions is the desired output.

After building the CHG and integrating it with the solving engine, the microgrid model was used to demonstrate several possible use cases of a DT:

1. **Data collection:** Relational database querying is perhaps the most similar form of system modeling, where each table represents the functional mapping between sets of values. In this case, the inputs for solar irradiance were taken from data sets compiled by the National Renewable Energy Laboratory (NREL) for Monterey, California, USA over the years 2000 to 2010 [79, 80]. All aspects of data querying including file locations, column names, and primary keys were included as nodes in the hypergraph, allowing for full data retrieval and visualization, as shown with the dashed lines in Figure 6.4.
2. **Control:** Although the DT was implemented post-data collection as a mock example, feedback

control was demonstrated by providing a controller strategy for the generator and BESS systems that prioritized lower-cost energy sources (such as the photovoltaic cells). Figure 6.4 shows the BESS switching from receiving to producing power, as well as the generator simulated as turning on to provide power to the grid when needed. Coupling the digital system to the microgrid actors would enable such real-time feedback to be implemented in the SOI. Methods for implementing such coupling have been widely shown, including programmable logic controllers (PLCs), MQTT messaging services, or built-in microcontrollers.

3. **System Interrogation:** Interfacing with the DT is the act of observing a state of the SOI. The microgrid DT enables universal interrogation of the microgrid's states by autonomously preparing simulation processes. This is programmatically conducted by calling the `solve(<state>)` function in the *ConstraintHg* package, passing it the name of the node representing the state to be queried. The takeaway from this is that an agent needs only to know the name of a state and have access to the `solve` method to be able to make full use of the DT. Serialized data, such as the time series of an actor's power output, can be provided by returning each value solved for along a path. Plots of serialized data are given in Figure 6.5 and 6.4.

6.6.3. Limitations

Many of the limitations of using CHGs to frame DTs are born from the nascent status of the supporting technologies. The custom solver used in this study, *ConstraintHg*, is limited in its solution speed and usability. In particular, the breadth-first search conducted within *ConstraintHg* to compose simulation paths is exhaustive and consequently computationally expensive. However, because these practical limitations are not tied to the theoretical foundations of CHGs, the authors suppose that future developments can address these early issues. Real time monitoring and control with DTs will require a more optimized approach, possibly leveraging search heuristics that allow quicker convergence to the optimal path.

Another reason limiting the scalability of systems is that pathfinding must be performed for every new input-output pairing. This is because a CHG is only a partial category, such that two functions that share a codomain and domain are not guaranteed to compose. Due to this partiality, a path in the hypergraph is only guaranteed to be solvable for its given set of inputs. For instance, a model within

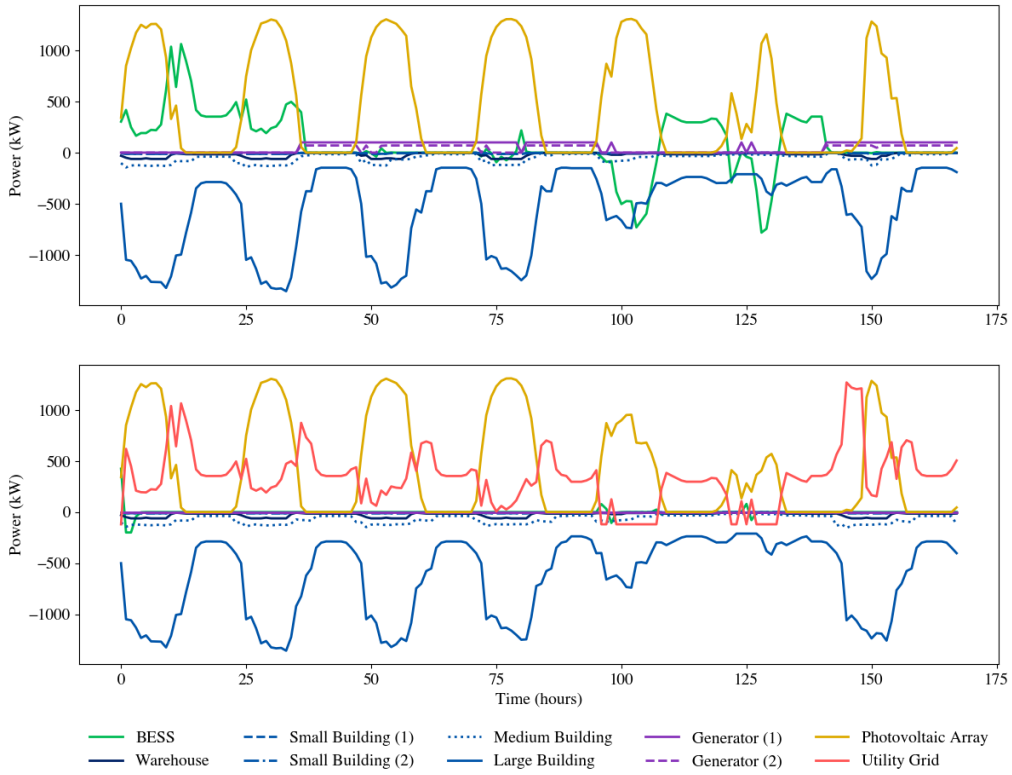


Figure 6.5: Demonstration of Microgrid CHG showing two simulations of the states of all grid actors over the same week-long span, with the top plot showing states of actors with the utility grid not connected (islanded) and the bottom plot showing the opposite.

the microgrid for calculating the power received from the utility grid is to check if the utility network is connected to the microgrid. If it is not, then the power received is zero. This model can be explicitly written as an edge e between two nodes A and B, the first representing the connection status of the utility grid, and the second one relating the power it is providing. Note that e is only valid if the A is set to be `False`; there is no mapping provided from A to B by e if A is `True`. This extends to any path containing e. A pathfinding algorithm that encounters e cannot then know whether e can be composed into a valid path until it is supplied with a value for A, provided either by simulating an edge connected to A or passed as an input to the search sequence, precluding *a priori* traversal optimization by common methods such as Djikstra's algorithm [81]. The authors address this in *ConstraintHg* by simulating every encountered edge in the search, so that the domain for every edge is always known to the solver. While this is guaranteed to be convergent, it is undeniably an untenable solution for scaled systems. It is the

hope of the authors that this, as well as the problem of scope changes described in greater detail in Supplementary Information 6.8, can be addressed through continuing studies into the mathematics of CHGs.

6.7. Handling Uncertainty with Constraint Hypergraphs

Uncertainty is inherent and omnipresent in representing reality [82], making how to reconcile with uncertainty an essential question for any modeling endeavor [83]. As a modeling framework, a CHG neither adds nor removes information to models adhering to its formalisms. However, deconstructing model elements into the nodes and edges of a CHG can provide significant insight into the uncertainty pertaining to a system representation. The tools provided in understanding uncertainty are best understood when framed against the three types of uncertainty described by Isukapalli et al. [84]: natural variability, model uncertainty, and data uncertainty. This section describes each of these as well as how they can be expressed and understood within the syntax of a CHG.

Background

Natural variability is the uncertainty in a model due to the imperfect approximation of physical phenomenon. As described in Chapter 2, information is a bounded, finite representation of the physical domain. Such discrete descriptions are inevitably insufficient for capturing the complex interactions of reality. For instance, while the connection between Brazilian butterflies and tornados in Texas could be identified [85], expressing the interaction as a direct relationship fails to identify the other features salient to the system's behavior. While the demonstrative relationship between butterflies and tornadoes is highly variable, similar variability is inherent in any considered model (though ideally on a reduced scale). The incongruence between the physical phenomena and its virtual description is intractable, meaning that it cannot be reduced by the efforts of a modeler [86]; no matter how careful modeler measures the butterfly wing beats, the resulting model will fail to perfectly explain tornado formation. This is described in Figure 6.6, which shows an illustrative plot of a model predicting solar irradiance based on the time of year. The relationship between these two phenomena is stochastic rather than deterministic, and is described by the probability density function. This is shown as the color map of the plot where brighter colors are more likely to be measured as real values. No model

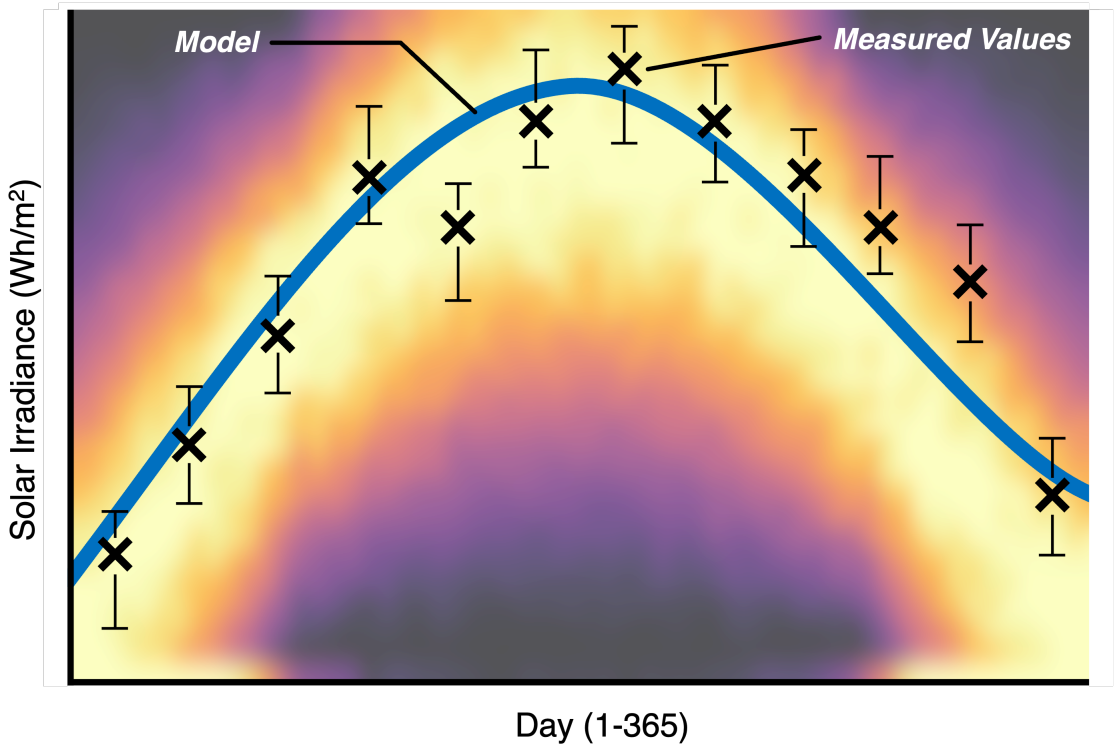


Figure 6.6: Illustration of modeling uncertainty for a relationship between day of the year and the average solar irradiance. The true value for irradiance is given probabilistically as the color map with brighter pixels indicating a higher likelihood of measuring a value in the given range. Twelve measured values are given by the black X's, each of which have some associated error that is often described using a measurement tolerance. A model, given by the blue line, is fit to the measured values with visible discrepancy between the modeled and measured values. Note that all data in the figure is synthetic and given for illustrative purposes only.

will perfectly capture this stochastic distribution.

Distinct from natural variation is data uncertainty, which is associated with the fidelity between the inputs provided to the model and the state of the physical system they represent. Each input is a datum that represents the specific measurement of some real world phenomenon. Measurement error, resulting for instance from the resolution of a measuring device or process, forms a key aspect of data uncertainty. An example is given by the measurements of solar irradiance shown by the black Xs in Figure 6.6, which cannot be guaranteed to align in the exact position provided. The data uncertainty is described by the error bars describing the tolerance of each measurement.

The final form of uncertainty is model uncertainty, describing unknown information associated with the structure of the model. The formation of every model is based on assumptions. For instance,

the interpolation of the measured data points in Figure 6.6 assumes that the solar irradiance follows a polynomial curve. Assumptions for calculating the energy production of a diesel generator might include the fuel being available and that the combustion cycle performs normatively. Each assumption is an aspect where the model might disagree with the real world, resulting in error. In addition to assumptions in model relationships, the modeler makes assumptions as to how data should be represented. While data uncertainty is the consideration of whether the diesel generator's fuel level is *full* or *empty*, model uncertainty is manifest as the assumption that fuel levels can only be *full* or *empty*. These two forms of uncertainty can be reduced and even eliminated through the efforts of the modeler [87], such as considering states with a higher degree of resolution or making more refined assumptions.

Expression in a Constraint Hypergraph

The structure of a CHG–its nodes and edges–provides an excellent framework for considering uncertainty in a simulation, particularly that attributed to data and model uncertainty. Nodal elements qualify a system's scope, while their values describe its resolution. Edges represent the assumptions inherent in the model that might need to be accepted to simulate specific information. More specific details can be found by describing how each form of simulation uncertainty is expressed by these two elements.

The fact that natural variability is not reducible relates to its inability to be represented in a model. It is instead the variation inherent in the form of representation selected by the modeler. As such it is not captured within the CHG, but it can be framed through use of the CHG. For instance, although the variability in the relationship shown in Figure 6.6 between the time of year and solar irradiance cannot be shown in the model, it can be estimated by comparing measured values against repeated simulations run over a sampling space (such as via Monte Carlo methods). These efforts benefit from a CHG, whose structure defines the explicit space for all simulations as well as their construction via the pathfinding measures described in the Methods section.

When an agent prescribes a value for a node they introduce information to the system from outside the model. This represents data uncertainty–the unknown accuracy of an assigned input. Like natural variability, data uncertainty is generally not captured explicitly in a model, as the information is derived external to the model's scope. However, the flexibility of a CHG allows quantified values

of uncertainty, such as the tolerance stack of a measurement process or the confidence interval for a simulation, to be included as a node in the graph representing the specific datum. Treating uncertainty parameters as states in the system reveals the relationships between a model's uncertainty and its simulated outcomes–an important aspect of using models for decision-making and risk estimation.

Contrasted with the first two forms of simulation uncertainty, model uncertainty is intrinsically captured in the structure of the CHG. Nodes encode the set of all distinguishable ways some phenomena can be characterized. As such, each node provides the resolution of the model corresponding to the specific phenomena considered. Furthermore, the scope of the system is explicitly given by the collection of nodes pertaining to the CHG. Assumptions in the model are always made in reference to this scope. The microgrid model, for instance, contains no nodes relating to the states of transformers or inverters on the grid. This consequently limits the model's fidelity, or ability to capture the effects of these actors on the rest of the system. If the exclusion of such phenomena affects the output node's value significantly, then a method of recourse is to include a node in the graph that aggregates unconsidered phenomena into a single value that accounts for the noise between simulated and measured values [37].

Edges represent the relationships prescribed between phenomena. Descriptions of relationships are virtual arrangements rather than physical constructs, and are consequently based entirely on virtual assumptions of the behavior of the real world. These assumptions are implicitly given by each edge, such that conducting a simulation implies acceptance of the underlying assumptions for each edge in the simulation path. For instance, Figure 6.4 shows an obvious discrepancy between the simulated and observed outputs of the diesel generators in the validation study. This is due to the assumption present in the DT model that the discharge profile of the diesel generators was uniform, while the controller in the real-world test applied a non-linear reduction at the end of each discharge cycle. Like with data uncertainty, specific quantified values of uncertainty (such as comparisons to measured data) can be included as nodes in the graph, showing the relationships of uncertainty parameters to the simulated outputs.

Assumptions of the real world relate to data uncertainty as well, in that the modeler assumes that an input is appropriate within the validity frame of the model. This is distinct from verifying whether an inputted value lies in the domain of an edge in the simulation path–instead it is the assumption that

the provided input accurately represents the corresponding phenomena. As with other assumptions, claiming an input to be appropriate can be expressed by an edge in the hypergraph. However, rather than relating two distinct nodes, assumptions on inputs are captured in loops: edges that have the same node as their source and target. This is the identity element in the category of a CHG, and its existence permits the expression of all modeling assumptions to be expressed solely by the edges in the graph.

Assumptions are the fundamental construct of any virtual representation, but are uniquely framed by a CHG. By representing system behavior as functions, relationships can be decomposed into chains of smaller edges. This allows each assumption to be isolated to a singular modeling construct. In addition to the increased precision in identifying assumptions, CHG models can be uniquely arranged to target or avoid specific assumptions depending on the needs of a modeler. Situations that require greater accuracy or execution time might prefer accepting different models. A CHG allows every simulation, with its inherent assumptions, to be compared by the modeler.

6.8. Digital Twin Composition with Constraint Hypergraphs

Figure E.2 holistically describes the microgrid system. However, each collection of nodes within the greater CHG–along with the edges that relate them–form a sub-CHG, equally valid if not more limited in scope. These sub-CHGs often relate to specific components of the microgrid system, such as grid actors or transmission infrastructure, and can be consequently perceived as forming a DT of that identified subsystem. Integrating these subsystem DTs into an aggregate DT illustrates one of the greatest challenges with providing DTs: interoperability [62]. Interoperability is an agent's ability to exchange information with another agent in a manner that preserves the meaning of the information [88]. The case for why DTs need to be interoperable to usefully represent a system is based on two reasons: the first is that all systems are composed of systems and are themselves subsets of other systems. An adequate representation of such systems of systems requires a DT that can interface with other DTs [16, 45]; e.g. a DT of a car with a DT of a tire, a tire DT with one of a road, etc. The second reason–known at least as early as Heraclitus–is that all systems change. As a consequence, all DTs must be able to adapt when the behaviors or scope of the SOI evolves [37, 64].

Maintaining meaning in a DT is not as simple as prescribing a defined interface by which all prospective DT connections must abide. The microgrid is an excellent illustrative example, since the behavior

of each grid actor is affected by the other actors to which it is connected. A rigid interface for a DT of a battery would need to define *a priori* how connecting to a diesel generator or utility grid would affect its output. This is only possible if the behavior of the DT is independent of any connected system. An independent system is typically encapsulated; because its behavior does not change, the only thing required for simulation is serialized messages concerning the state of connected objects, often passed along ports [89]. Such *a priori* independence is an untenable requirement for DTs representing the reactive systems typical of reality. Key to this conundrum is understanding that connecting DTs does not form a system of independent systems, but rather a new aggregate system that consumes the interfacing models. Because of this, the problem of interoperability does not concern interface connections, but rather changes to the scope of the system representation. For two DTs to interoperate, their interactions must be captured in the aggregate system.

Forming an aggregate system is the result of either adding or removing information, the last of the three operations performed on persistent data. Because a CHG fully captures a system description, the types of changes typical of DT interoperability can be described by the possible modifications that can be made to a CHG. In this sense the term interoperability is equivalent to extensibility [65], as employed in the development of programming languages.

There are two entities in a CHG: edges and nodes. Adding or removing an edge in a CHG amounts to redefining the behavior of the represented system, e.g. modifying the relationship between solar irradiation and power supply, $f \coloneqq (\eta, I, a) \to P$ from $P = \eta I a$ to $P = \eta I a(1-\eta)$ defines a new interaction between the domain (composed of the Cartesian product of η, I and a) and codomain P. Because CHGs are function-based in the same sense as languages such as Haskell and pure LISP, behavioral modifications of a CHG's edges do not affect the consistency of the graph, a characteristic referred to as having no side effects [32]. In other words, observations of a fact in the system are not affected by adding edges to the graph. The same is true for removing edges, as long as the observation is not exclusively dependent upon a simulated path along the removed edge. Additionally, a CHG's functional nature allows it to reveal a type of emergent behavior resulting from non-localized interactions. If the cost of running the microgrid is influenced by the power output of the photovoltaic array, and a separate model describes how that power output is affected by the day's weather, then combining the two graphs along the shared variable allows the interaction between weather on operational cost to emerge.

Behavioral interoperability–changes along a system's relationships–is the first type of interoperability. The second involves adding or removing system facts corresponding to the nodes in a CHG. This type of interoperability is less well-defined in functional models which struggle with adding new variants of information. This result is studied under the label the "expression problem" [65, 66], with Wadler famously describing it as a table, where extending the rows requires fixing the columns and vice versa. Allowing both rows (behaviors) and columns (facts) to be extended without *a posteriori* modifications is famously difficult [90].

With CHGs, there are two potential issues with adding or removing nodes from the system. The first is highly tractable, in that adding or removing nodes disrupts the definition of any connected edge, which must be redefined to accommodate the new nodes. This problem is highly localized; because edges can be modified without side effects, the disruption of modifying a node is limited to its connected edges, and does not propagate further in the graph. The second issue is that changing the considered scope of the model potentially invalidates the assumptions undergirding each model relationship. For example, the relationship calculating the maximum power generation capacity of the microgrid might be defined as summing the individual capacities of each grid actor. Such a relationship might be based on the assumption that all actors capable of generating power form part of the domain of the edge. This assumption is valid in the scope of the system as defined. However, if the scope changes to include, for example, a wind turbine on the microgrid, the assumption is invalidated, and the demonstrative edge would need to be modified by someone *a posteriori* in order to accommodate the proposed scope change. There are several potential solutions to this problem (some reviewed in [65]), which are the subject of additional research.

6.9. Conclusion

The behavior of a system is a virtual concept, and can be represented as a set of explicit constraint functions acting on the state variables of a system. These functions and variables can be composed into a mathematical structure called a CHG, a declarative model of a system that can be used in constructing analogously-behaving DTs. CHGs do not replace other modeling systems; rather they integrate them into a cohesive, global model. This model can be used to interrogate an SOI, expressing values that were either measured or else calculated within the CHG itself. These cross-cutting measures ensure

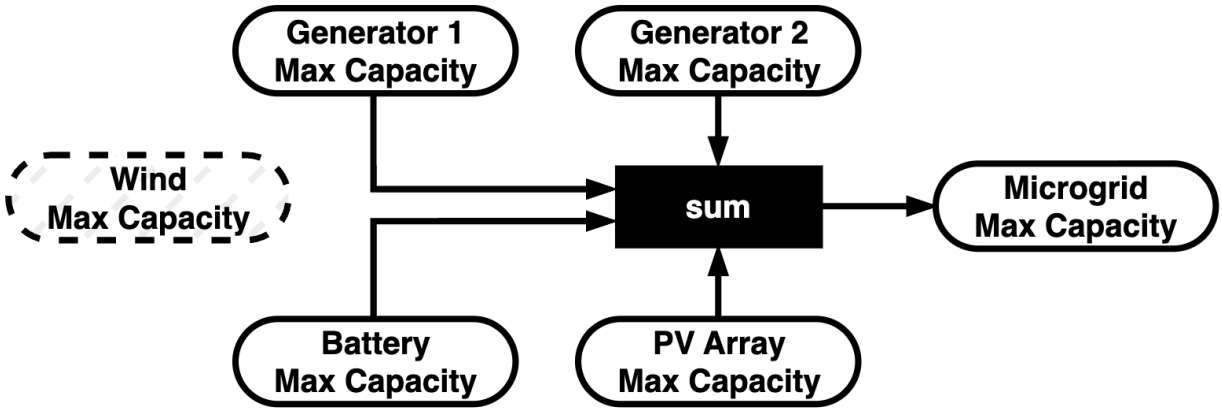


Figure 6.7: Example relation in the microgrid CHG invalidated by a scope change, where the relation sums the capacities of each actor to calculate the maximum grid capacity. This relationship would be invalidated by the addition of another capacity (shown here for a demonstrative wind turbine as the dashed node).

that DTs built upon CHGs are universally accessible, and generally far more extensible, scalable, and redeployable than traditional DTs.

This work introduces both theoretical and practical measures for developing DTs for fields as diverse as engineering, organization management, medicine, and ecology. The resulting DTs capture meaning across all domains, providing a mathematically robust framework for representing the complex and integrated systems constituting the world in which we live.

References

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

[2] John Morris et al. *Constraint Hypergraphs as a Unifying Framework for Digital Twins*. July 2025. DOI: 10.48550/arXiv.2507.05494. arXiv: 2507.05494. Pre-published.

[3] International Organization for Standardization. *Systems and Software Engineering - System Life Cycle Processes*. May 2023. https://www.iso.org/standard/81702.html (visited on 09/21/2024). Published.

[4] Radhya Sahal, Saeed H. Alsamhi, and Kenneth N. Brown. “Personal Digital Twin: A Close Look into the Present and a Step Towards the Future of Personalised Healthcare Industry”. *Sensors* 22.15 (15 Jan. 2022), p. 5918. ISSN: 1424-8220. DOI: 10.3390/s22155918.

[5] 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.

[6] Yang Fu et al. “Digital Twin for Integration of Design-Manufacturing-Maintenance: An Overview”. *Chinese Journal of Mechanical Engineering* 35.1 (June 23, 2022), p. 80. ISSN: 2192-8258. DOI: 10.1186/s10033-022-00760-x.

[7] Jussi Koskinen, Henna Lahtonen, and Tero Tilus. *Software Maintenance Cost Estimation and Modernization Support*. Information Technology Research Institute, June 19, 2003. https://static.aminer.org/pdf/PDF/000/364/724/estimating_the_costs_of_software_maintenance_tasks.pdf (visited on 06/21/2023).

[8] Valentina Zambrano et al. “Industrial Digitalization in the Industry 4.0 Era: Classification, Reuse and Authoring of Digital Models on Digital Twin Platforms”. *Array* 14 (2022), p. 100176. ISSN: 25900056. DOI: 10.1016/j.array.2022.100176.

[9] Paolo Pileggi et al. *Overcoming Digital Twin Barriers for Manufacturing SMEs*. Position paper. Change2Twin, Apr. 2021. https://www.syncontwin.eu/insights/10-overcoming-9-digital-twin-barriers-for-manufacturing-smes/ (visited on 09/04/2025).

[10] 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.

[11] 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.

[12] 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).

[13] 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.

[14] 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 01/19/2025).

[15] 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.

[16] 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.

[17] 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.

[18] International Organization for Standardization. *Automation Systems and Integration — Digital Twin Framework for Manufacturing*. Version 2021. Switzerland, Oct. 2021. https://www.iso.org/obp/ui/en/#iso:std:iso:23247:-1:ed-1:v1:en (visited on 02/02/2024). Published.

[19] John Osho et al. "Four Rs Framework for the Development of a Digital Twin: The Implementation of Representation with a FDM Manufacturing Machine". *Journal of Manufacturing Systems* 63 (Apr. 1, 2022), pp. 370–380. ISSN: 0278-6125. DOI: 10.1016/j.jmsy.2022.04.014.

[20] Milos Drobnjakovic et al. "Towards Ontologizing a Digital Twin Framework for Manufacturing". *Advances in Production Management Systems. Production Management Systems for Responsible Manufacturing, Service, and Logistics Futures*. APMS 2023 IFIP International Conference. Vol. 689. IFIPAICT. Trondheim, NO: Springer, Cham, Sept. 21, 2023, pp. 317–329. ISBN: 978-3-031-43665-9. DOI: 10.1007/978-3-031-43666-6_22.

[21] 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.

[22] ITU-T. *Digital Twin Network - Requirements and Architecture*. Version 1.0. Feb. 13, 2022. https://www.itu.int/rec/T-REC-Y.3090-202202-I (visited on 05/22/2024). Approved.

[23] Francesco Longo et al. "Distributed Simulation for Digital Twins: An Application to Support the Autonomous Robotics for the Extended Ship". *2022 IEEE/ACM 26th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. 2022 IEEE/ACM 26th International Symposium on Distributed Simulation and Real Time Applications (DS-RT). Sept. 2022, pp. 179–186. DOI: 10.1109/DS-RT55542.2022.9932057.

[24] Jethro Akroyd et al. "Universal Digital Twin - A Dynamic Knowledge Graph". *Data-Centric Engineering* 2 (Jan. 2021), e14. ISSN: 2632-6736. DOI: 10.1017/dce.2021.10.

[25] Maryna Waszak et al. "Let the Asset Decide: Digital Twins with Knowledge Graphs". *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*. 2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C). Mar. 2022, pp. 35–39. DOI: 10.1109/ICSA-C54293.2022.00014.

[26] Michael G. Kapteyn, Jacob V. R. Pretorius, and Karen E. Willcox. "A Probabilistic Graphical Model Foundation for Enabling Predictive Digital Twins at Scale". *Nat Comput Sci* 1.5 (May 2021), pp. 337–347. ISSN: 2662-8457. DOI: 10.1038/s43588-021-00069-0.

[27] Abheek Chatterjee et al. "A Comparison of Graph-Theoretic Approaches for Resilient System of Systems Design". *Journal of Computing and Information Science in Engineering* 23.030906 (Apr. 19, 2023). ISSN: 1530-9827. DOI: 10.1115/1.4062231.

[28] Aidan Hogan et al. *Knowledge Graphs*. Synthesis Lectures on Data, Semantics and Knowledge 22. Cham: Springer, 2022. 237 pp. ISBN: 978-3-031-01918-0.

[29] Michael. G. Kapteyn et al. "Data-Driven Physics-Based Digital Twins via a Library of Component-Based Reduced-Order Models". *International Journal for Numerical Methods in Engineering* 123.13 (2022), pp. 2986–3003. ISSN: 1097-0207. DOI: 10.1002/nme.6423.

[30] Kjartan Pedersen et al. "Validating Design Methods & Research: The Validation Square". *DETC 2000 ASME Design Engineering Technical Conferences*. DETC2000. Baltimore, MD: ASME, Sept. 10, 2000.

[31] Hendrik Pieter Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics 103. Amsterdam New York Oxford: North-Holland, 1981. ISBN: 978-0-444-85490-2.

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

[33] International Organization for Standaridization. *Digital Twin: Concepts and Terminology*. International Standard. Version 2023-11. Geneva, Switzerland, Nov. 2023. https://www.iso.org/standard/81442.html (visited on 06/11/2025). Published.

[34] Digital Twin Consortium. *Definition of a Digital Twin*. Object Management Group, Inc. Dec. 3, 2020. https://www.digitaltwinconsortium.org/initiatives/the-definition-of-a-digital-twin.htm (visited on 10/18/2021).

[35] AIAA Digital Engineering Integration Committee. *Digital Twin: Definition and Value*. American Institute of Aeronautics and Astronautics and Aerospace Industries Association, Dec. 2020. https://www.aiaa.org/docs/default-source/uploadedfiles/issues-and-advocacy/policy-papers/digital-twin-institute-position-paper-(december-2020).pdf (visited on 02/06/2024).

[36] Michael W. Grieves. "Virtually Intelligent Product Systems: Digital and Physical Twins". *Complex Systems Engineering: Theory and Practice*. Vol. 256. Progress in Astronautics and Aeronautics. American Institute of Aeronautics and Astronautics, Inc., Jan. 2019, pp. 175–200. ISBN: 978-1-62410-564-7. DOI: 10.2514/5.9781624105654.0175.0200.

[37] D. J. Wagg et al. "Digital Twins: State-of-the-Art and Future Directions for Modeling and Simulation in Engineering Dynamics Applications [Special Section]". *ASME J. Risk Uncertainty Part B* 6.3 (May 12, 2020), 030901:1–030901:18. ISSN: 2332-9017. DOI: 10.1115/1.4046739.

[38] Mike Shafto et al. *Draft Modeling, Simulation, Information Technology & Processing Roadmap*. TA11-27. Washington DC: National Aeronautics and Space Administration, Nov. 2010. https://www.nasa.gov/pdf/501321main_TA11-MSITP-DRAFT-Nov2010-A1.pdf.

[39] Michael Grieves. *Digital Twin: Manufacturing Excellence through Virtual Factory Replication*. Dassault Systèmes, 2014.

[40] Rosario Davide D'Amico, Sri Addepalli, and John Ahmet Erkoyuncu. "Industrial Insights on Digital Twins in Manufacturing: Application Landscape, Current Practices, and Future Needs". *Big Data and Cognitive Computing* 7.3 (3 Sept. 2023), p. 126. ISSN: 2504-2289. DOI: 10.3390/bdcc7030126.

[41] Shoumen Palit Austin Datta. "Emergence of Digital Twins - Is This the March of Reason?" *Journal of Innovation Management* 5.3 (Nov. 29, 2017), pp. 14–33. ISSN: 2183-0606. DOI: 10.24840/2183-0606_005.003_0003.

[42] Guodong Shao, Joe Hightower, and William Schindel. "Credibility Consideration for Digital Twins in Manufacturing". *Manufacturing Letters* 35 (Jan. 1, 2023), pp. 24–28. ISSN: 2213-8463. DOI: 10.1016/j.mfglet.2022.11.009.

[43] Akram Hakiri et al. "A Comprehensive Survey on Digital Twin for Future Networks and Emerging Internet of Things Industry". *Computer Networks* 244 (May 1, 2024), p. 110350. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2024.110350.

[44] Alberto Villalonga et al. "A Decision-Making Framework for Dynamic Scheduling of Cyber-Physical Production Systems Based on Digital Twins". *Annual Reviews in Control* 51 (Jan. 1, 2021), pp. 357–373. ISSN: 1367-5788. DOI: 10.1016/j.arcontrol.2021.04.008.

[45] Judith Michael et al. "Integration Challenges for Digital Twin Systems-of-Systems". *Proceedings of the 10th IEEE/ACM International Workshop on Software Engineering for Systems-of-Systems and Software Ecosystems*. SESoS '22. New York, NY, USA: Association for Computing Machinery, Nov. 9, 2022, pp. 9–12. ISBN: 978-1-4503-9334-8. DOI: 10.1145/3528229.3529384.

[46] Kendrik Yan Hong Lim, Pai Zheng, and Chun-Hsien Chen. "A State-of-the-Art Survey of Digital Twin: Techniques, Engineering Product Lifecycle Management and Business Innovation Perspectives". *J Intell Manuf* 31.6 (Aug. 1, 2020), pp. 1313–1337. ISSN: 1572-8145. DOI: 10.1007/s10845-019-01512-w.

[47] C. Human, A. H. Basson, and K. Kruger. "A Design Framework for a System of Digital Twins and Services". *Computers in Industry* 144 (Jan. 1, 2023), p. 103796. ISSN: 0166-3615. DOI: 10.1016/j.compind.2022.103796.

[48] Luciano Floridi. *Information: A Very Short Introduction*. Very Short Introductions. Oxford: Oxford University Press, 2010. 116 pp. ISBN: 978-0-19-955137-8.

[49] 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.

[50] 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.

[51] Jonathan Lussier et al. "Differentiating Between Digital Twins and Control Systems for Complex Systems". *[Manuscript submitted for review in the Journal of Computing and Information Science in Engineering]* (2025).

[52] Edward A. Lee. "Determinism". *ACM Trans. Embed. Comput. Syst.* 20.5 (May 29, 2021), 38:1–38:34. ISSN: 1539-9087. DOI: 10.1145/3453652.

[53] John Morris et al. "Declarative Integration of CAD Software into Multi-Physics Simulation via Constraint Hypergraphs". *Proceedings of the ASME 2025 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*. ASME IDETC-CIE 2025. Anaheim, CA: ASME, Aug. 17–20, 2025.

[54] François E. Cellier and Ernesto Kofman. *Continuous System Simulation*. New York: Springer, 2006. 643 pp. ISBN: 978-0-387-26102-7. DOI: 10.1007/0-387-30260-3.

[55] 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.

[56] László Nagy et al. "Hypergraph-Based Analysis and Design of Intelligent Collaborative Manufacturing Space". *Journal of Manufacturing Systems* 65 (Oct. 1, 2022), pp. 88–103. ISSN: 0278-6125. DOI: 10.1016/j.jmsy.2022.08.001.

[57] Luis F. Rivera et al. "Using Dynamic Knowledge Hypergraphs Toward Proactive AlOps through Digital Twins". *Proceedings of the 32nd Annual International Conference on Computer Science and Software Engineering*. CASCON '22. USA: IBM Corp., Nov. 15, 2022, pp. 163–168. DOI: 10.5555/3566055.3566074.

[58] Sagar Srinivas Sakhinana et al. "Joint Hypergraph Rewiring and Memory-Augmented Forecasting Techniques in Digital Twin Technology" ().

[59] Liqiao Xia et al. "Residual-Hypergraph Convolution Network: A Model-Based and Data-Driven Integrated Approach for Fault Diagnosis in Complex Equipment". *IEEE Transactions on Instrumentation and Measurement* 72 (2023), pp. 1–11. ISSN: 1557-9662. DOI: 10.1109/TIM.2022.3227609.

[60] Wang Zuoxu et al. "A Hypergraph-Based Knowledge Representation Model for Smart Product-Service System Development". ASME 2021 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference. American Society of Mechanical Engineers Digital Collection, Nov. 17, 2021. DOI: 10.1115/DETC2021-66732.

[61] Carlos Henrique Duarte. "Proof-Theoretic Foundations for the Design of Extensible Software Systems". PhD thesis. London: University of London, Nov. 1, 1998. DOI: 10.13140/RG.2.2.29584.46080.

[62] Vartan Piroumian. "Digital Twins: Universal Interoperability for the Digital Age". *Computer* 54.1 (Jan. 2021), pp. 61–69. ISSN: 1558-0814. DOI: 10.1109/MC.2020.3032148.

[63] Matthias Zenger. "Programming Language Abstractions for Extensible Software Components". PhD thesis. Lausanne, Switzerland: EPFL, 2004. DOI: 10.5075/epfl-thesis-2930.

[64] Paul G. Carlock and Robert E. Fenton. "System of Systems (SoS) Enterprise Systems Engineering for Information-Intensive Organizations". *Systems Engineering* 4.4 (2001), pp. 242–261. ISSN: 1520-6858. DOI: 10.1002/sys.1021.

[65] Matthias Zenger and Martin Odersky. "Independently Extensible Solutions to the Expression Problem". Foundations of Object-Oriented Languages (FOOL 2005). Long Beach, CA, USA: École Polytechnique Fédérale de Lausanne, Jan. 15, 2005. https://homepages.inf.ed.ac.uk/wadler/fool/program/10.html.

[66] Shriram Krishnamurthi, Matthias Felleisen, and Daniel P. Friedman. "Synthesizing Object-Oriented and Functional Design to Promote Re-Use". *Proceedings of the 12th European Conference on Object-Oriented Programming*. 12th European Conference on Object-Oriented Programming. Vol. 1445. ECCOP '98. Berlin, Heidelberg: Springer-Verlag, July 20, 1998, pp. 91–113. ISBN: 978-3-540-64737-9. DOI: 10.1007/BFb0054088.

[67] John C. Reynolds. "User-Defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction". *Programming Methodology: A Collection of Articles by Members of IFIP WG2.3*. Ed. by David Gries. New York, NY: Springer, 1978, pp. 309–317. ISBN: 978-1-4612-6315-9. DOI: 10.1007/978-1-4612-6315-9_22.

[68] 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.

[69] Robert Lasseter et al. *Integration of Distributed Energy Resources: The CERTS Microgrid Concept*. Consultant Report LBNL–50829, 799644. Berkeley, CA, USA: Consortium for Electric Reliability Technology Solutions, Apr. 1, 2002, LBNL–50829. DOI: 10.2172/799644.

[70] Abheek Chatterjee, Amira Bushagour, and Astrid Layton. “Resilient Microgrid Design Using Ecological Network Analysis”. *The Proceedings of the 2023 Conference on Systems Engineering Research*. Ed. by Dinesh Verma et al. Cham: Springer Nature Switzerland, 2024, pp. 603–617. ISBN: 978-3-031-49178-8. DOI: 10.1007/978-3-031-49179-5_41.

[71] William W. Anderson Jr and Douglas L. Van Bossuyt. “Foundations of Microgrid Resilience”. *Microgrids*. John Wiley & Sons, Ltd, 2024, pp. 655–679. ISBN: 978-1-119-89088-1. DOI: 10.1002/9781119890881.ch27.

[72] Richard Alves and Douglas Van Bossuyt. *Spanagel Microgrid Experimental Run*. CSV. Version 1.0. Zenodo, June 16, 2025. DOI: 10.5281/zenodo.15675037.

[73] Olive Grace A Oliveros. “Test Model for Power Distribution on U.S. Naval Installations”. Monterey, CA: Naval Postgraduate School, June 2024. HDL: 10945/73198. https://hdl.handle.net/10945/73198 (visited on 12/13/2024).

[74] Daniel Reich and Leah Frye. “Microgrid Planner: An Open-Source Software Platform”. *INFORMS Journal on Computing* (Aug. 29, 2024). ISSN: 1091-9856. DOI: 10.1287/ijoc.2023.0336.

[75] Christopher J Peterson. “Systems Architecture Design and Validation Methods for Microgrid Systems”. MA thesis. Monterey, CA: Naval Postgraduate School, Sept. 19, 2019. HDL: 10945/63493. https://hdl.handle.net/10945/63493 (visited on 12/13/2024).

[76] Albert Wayne Wymore. *Model-Based Systems Engineering: An Introduction to the Mathematical Theory of Discrete Systems and to the Tricotyledon Theory of System Design*. Systems Engineering Series. Boca Raton, Fla.: CRC Press, 1993. 710 pp. ISBN: 978-0-8493-8012-9.

[77] Bernard P. Zeigler, Alexandre Muzy, and Ernesto Kofman. *Theory of Modeling and Simulation*. Third. Elsevier, 1976. ISBN: 978-0-12-813370-5. DOI: 10.1016/B978-0-12-813370-5.00002-X.

[78] John Morris. *ConstraintHg*. Version 0.2.2. Nov. 23, 2024. DOI: 10.5281/zenodo.15278018.

[79] National Renewable Energy Laboratory. *National Solar Radiation Database (NSRDB)*. Meteorological Statistical Model 3 (MTS3). Version 1991-2010 Update. https://github.com/jmorris335/MicrogridHg/tree/main/src/solar_data: GitHub, Aug. 2012. https://nsrdb.nrel.gov/data-viewer (visited on 06/16/2025).

[80] Stephen Wilcox. *National Solar Radiation Database 1991–2010 Update: User’s Manual*. Technical Report NREL/TP-5500-54824. Golden, CO, USA: National Renewable Energy Laboratory, Aug. 2012. https://docs.nrel.gov/docs/fy12osti/54824.pdf.

[81] 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).

[82] George E. P. Box and Norman Richard Draper. *Empirical Model-Building and Response Surfaces*. Wiley Series in Probability and Mathematical Statistics. New York: Wiley, 1987. 669 pp. ISBN: 978-0-471-81033-9.

[83] George J Klir. *Uncertainty and Information*. Hoboken, NJ, USA: John Wiley & Sons, Inc, 2006. ISBN: 978-0-471-74867-0.

[84] S. S. Isukapalli, A. Roy, and P. G. Georgopoulos. “Stochastic Response Surface Methods (SRSMs) for Uncertainty Propagation: Application to Environmental and Biological Systems”. *Risk Analysis* 18.3 (1998), pp. 351–363. ISSN: 1539-6924. DOI: 10.1111/j.1539-6924.1998.tb01301.x.

[85] Edward N. Lorenz. *The Essence of Chaos*. The Jessie and John Danz Lectures. Seattle: University of Washington Press, 1993. 227 pp. ISBN: 978-0-295-97270-1.

[86] Hae-Jin Choi et al. “An Inductive Design Exploration Method for Hierarchical Systems Design under Uncertainty”. *Engineering Optimization* 40.4 (Apr. 2008), pp. 287–307. ISSN: 0305-215X, 1029-0273. DOI: 10.1080/03052150701742201.

[87] Ayan Sinha et al. “Uncertainty Management in the Design of Multiscale Systems”. *Journal of Mechanical Design* 135.1 (Dec. 21, 2012), p. 011008. ISSN: 1050-0472. DOI: 10.1115/1.4006186.

[88] International Organization for Standardization. *Information Technology-Cloud Computing-Interoperability and Portability*. International Standard. Version 2017-12. Dec. 2017. DOI: 10.3403/30313036U.

[89] 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.

[90] Philip Wadler. *The Expression Problem*. E-mail. Nov. 12, 1998. https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt (visited on 01/27/2025).

[91] David I. Spivak. *Category Theory for Scientists*. Sept. 18, 2013. DOI: 10.48550/arXiv.1302.6946. arXiv: 1302.6946. Prepublished.

Appendices

Appendix A. Associated Research Activities

A.1. Collaborating Researchers

Name	Institution	Location	Role
Xiaoxue Xue	Alan Turing Institute	London, UK	Collaborator
Joe Gregory	Uni. of Arizona	Tuscon, AZ	Co-author
Wayne Goddard	Clemson Uni.	Clemson, SC	Collaborator
Matthew Macauley	Clemson Uni.	Clemson, SC	Collaborator
Gregory Mocko	Clemson Uni.	Clemson, SC	Advisor, co-author
Satchit Ramnath	Clemson Uni.	Clemson, SC	Collaborator, co-author
John Wagner	Clemson Uni.	Clemson, SC	Advisor, co-author
Luigi Ponti	ENEA	Rome, Italy	Collaborator
Lance Sherry	George Mason Uni.	Fairfax, VA	Collaborator
George Simmons	IDEMS International	Reading, UK	Collaborator, reviewer
David Stern	IDEMS International	Reading, UK	Collaborator
Jackson Weaver	Kroeger Marine	Seneca, SC	Collaborator
Jason Bickford	Naval Postgrad. School	Monterrey, CA	Collaborator
Douglas Van Bossuyt	Naval Postgrad. School	Monterrey, CA	Mentor, co-author
Spencer Breiner	NIST	Gaithersburg, MD	Collaborator
Abheek Chatterjee	NIST	Gaithersburg, MD	Co-author
Duncan Gibbons	NIST	Gaithersburg, MD	Co-author
Shengyen Li	NIST	Gaithersburg, MD	Collaborator
Yan Lu	NIST	Gaithersburg, MD	Collaborator
Guodong Shao	NIST	Gaithersburg, MD	Mentor, collaborator
Lisha White	NIST	Gaithersburg, MD	Collaborator
Paul Witherwell	NIST	Gaithersburg, MD	Mentor, co-author
David Wagg	Uni. of Sheffield	Sheffield, UK	Collaborator
Matthew Bonney	Swansea Uni.	Swansea, UK	Collaborator
Astrid Layton	Texas A&M	College Station, TX	Co-author
Tim Hosgood	Topos Institute	Oxford, UK	Collaborator
Geoffrey Kerr	Virginia Tech	Blacksburg, VA	Collaborator
Paul Wach	Virginia Tech	Blacksburg, VA	Collaborator

Table A.1: Constraint hypergraphs and their related research have benefitted from inputs from a variety of people. The individuals listed in this table have engaged significantly with the research in this dissertation, either through providing conceptual reviews, extended discussion, or co-authoring papers.

A.2. Supporting Activities in the Product Lifecycle Management Center

The use of constraint hypergraphs was supported and greatly influenced by the Product Lifecycle Management Center at Clemson University (PLMC), where I worked as the applications engineer under the direction of Dr. John Wagner and Dr. Gregory Mocko. Starting in 2019, the PLMC led research

initiatives into digital twins, constructing digital twins for HVAC systems and ground vehicles. When I assisted with this objective upon joining in 2021, leading a team of undergraduate students in building a twin of a tracked, robotic vehicle.

At the same time these research projects were occurring, I was given the chance to prepare training materials for digital engineering. I prepared hour-long seminars teaching concepts like data analytics, engineering change management, cybersecurity, model-based systems engineering, GD&T, PDM, and project management, as well as workshops teaching SOLIDWORKS, NX, and Teamcenter. The over 100 seminars I instructed gave me valuable exposure to the techniques and challenges associated with engineering informatics.

In 2023, I reached out to a local manufacturer to start a research project making a digital twin of a chop saw used for cutting aluminum extrusion. This project quickly grew quite intensive. The model ecosystem, shown in Figure A.1, required the position of the saw to be tracked by a computer using two cameras. The position was fed into a Python script that calculated the depth of cut, which determined the loading of the blade and eventually the likelihood of blade failure due to fatigue, all of had to be outputted to a web dashboard in real time.

As I struggled through moving and transforming information from OpenCV to MySQL to Python back again, I recognized that the system we were building was incredibly fragile. It only worked if the position input from the cameras was perfectly transformed through every step of the long simulation chain. Furthermore, if the manufacturing company we were assisting had any new information–a new extrusion profile, or different location to cut material–that would require an entirely new model to be created.

Frustrated with the inflexibility of the program, I started investigating model structures. David Spivak's excellent *Category Theory for Scientists* [91] introduced me to the notion of categories and morphisms (there's a fair bit of similarity between constraint hypergraphs and the ologs he describes in the book), which led directly to the representation of models and simulations using nodes and edges in a hypergraph. The rest of the research in this dissertation comes out of these moments, and should be attributed to the work of the PLMC, especially in conducting inter-software simulations of complex digital twins.

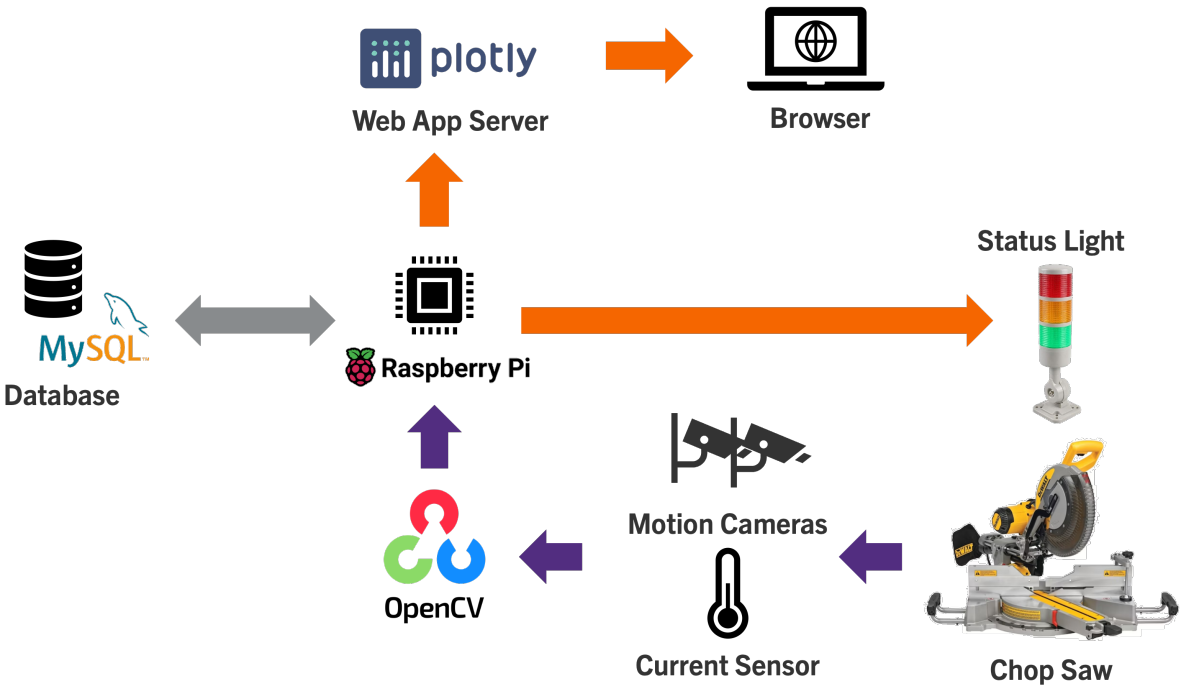


Figure A.1: Overview of the model ecosystem for a digital twin of a chop saw.

Appendix B. List of Variables in Elevator Case Study

Node Name	Description	Subsystem	Input Value	Units
Δ Passengers	Change in carriage occupancy	DES		Persons
Elev Current Floor	Current floor of carriage	DES	0	Floor
Num Boarding	Number of passengers boarding carriage	DES		Persons
Num Exiting	Number of passengers exiting carriage	DES		Persons
Occupancy	Number of passengers in carriage	DES	0	Persons
X Goal	Goal floor for passenger X	DES		Floor
X is Boarding	True if passenger X is currently boarding carriage	DES		Boolean
X is Exiting	True if passenger X is currently exiting carriage	DES		Boolean
X is Riding	True if passenger X is currently on the carriage	DES		Floor
X Start	Starting floor for passenger X	DES		Floor
Acceleration	Current acceleration of carriage	Dynamics		m/s^2
Avg Pass Mass	Average mass of a passenger	Dynamics	75	kg
Counterweight	Mass of counterbalancing weight	Dynamics	850	kg
Damping Coef	Damping coefficient	Dynamics	10	kg/s
Damping Force	Force due to system damping	Dynamics		N
Empty Mass	Mass of empty carriage	Dynamics	1000	kg
Gravity	Gravitational Acceleration	Dynamics	9.8	m/s^2
Height	Current position of carriage	Dynamics		m
Initial Height	Initial position of the carriage	Dynamics	0	m
Initial Velocity	Initial velocity of carriage	Dynamics	0	m/s
Net Force	Total force on carriage	Dynamics		N
Passenger Mass	Total mass of passengers on carriage	Dynamics		kg
Total Mass	Total mass of carriage	Dynamics		kg
Velocity	Current velocity of carriage	Dynamics		m/s
Weight	Weight of carriage	Dynamics		N
Step Size	Time between simulation steps	Many	0.1	s
ΔI	The change in integrative outputs	PID		N
D Output	Force output set by derivitive controller	PID		N
Dest Height	Height of destination floor	PID		m
Destination	Current floor carriage is moving to	PID	1	m
Error	Distance between current and goal locations	PID		m
Error Difference	Difference between current and previous step	PID		m
Filter Constant	Low-pass filter constant	PID	0.5	s
Filtered Error	Current filtered error signal	PID		m
Floor Gap	Distance between floors	PID	4	m
Force Input	Force input given to motor	PID		N
Ht Tolerance	Distance where carriage is considered at a floor	PID	0.2	m
I Output	Force output set by integrative controller	PID		N
Kd	Derivitive gain for PID controller	PID	-0.79	s
Ki	Integral gain for PID controller	PID	35	s^{-1}
Kp	Proportional gain for PID controller	PID	271	
Max Force	Maximum force reachable by the PID controller	PID	10000	N
Min Force	Minimum force reachable by the PID controller	PID	-1000	N
P Output	Force output set by proportional controller	PID		N
PID Output	Total force ouput of the PID controller	PID		N
Prv Fltd Error	Filtered error signal of previous step	PID		m

Table B.2: Descriptions of nodes of the CH shown in Figure 2.8 in Chapter 2. Input values, useful for simulation, are provided for nodes if applicable.

Appendix C. List of Inputs for the Crankshaft Case Study

Symbol	Description	Units
$\oslash_c$	Diameter of crankshaft	m
$\oslash_{\text{cw}}$	Diameter of counterweight on web	m
$\oslash_{\text{web}}$	Diameter of web (about crank pin)	m
$\oslash_r$	Diameter of crank pin	m
l_c	Throw length (crankshaft centerline to crank pin)	m
l_c	Slider length (crank pin to slider tip)	m
w_c	Width of crankshaft journal surface	m
w_r	Width of crank pin surface	m
w_{web}	Width of crank web	m
$\psi_1, \ldots \psi_4$	Angle of crank pin to vertical when first piston is TDC	rad
J_c	Moment of inertia for crankshaft	$\text{kg}\,\text{m}^2$
y	Slider vertical position	m
$\dot{y}$	Slider vertical velocity	m/s
θ	Crank angular position	rad
ω	Crank angular velocity	rad/s
t	Current simulation time	s
t_0	Simulation start time	s
t_f	Simulation end time	s
Δ_t	Simulation time step	s
ρ	Material density	kg/m^3
σ	Von Mises stress	MPa
δ	Total deformation	mm

Table C.3: Input parameters for crankshaft system model given in Chapter 4.

Appendix D. List of Nodes in Microgrid CHG Case Study

Node	Units	Description
Constants		
days in year	day	Number of days in a year
days in leap year	day	Number of days in a leap year
hours in day	hr	Number of hours in a day
hours in year	hr	Number of hours in a year
hours in leap year	hr	Number of hours in a leap year
seconds in minute	s	Number of seconds in a minute
minutes in hour	min	Number of minutes in an hour
seconds in hour	s	Number seconds in an hour
tolerance		Float tolerance
General Settings		
use random date	*bool*	True if the start date is set randomly
has random failure	*bool*	True if random failure is allowed
island mode	*bool*	true if all utility grids are disconnected
min year	year	Minimum year of simulation data
max year	year	Maximum year of simulation data
Timing Variables		
time	s	Total seconds elapsed in simulation
time step	s	seconds since last time calculation
hour	hr	Current hour (0-23)
day	day	Current day of the year (1-366)
year	year	Current year
start year	year	Starting year for simulation
start day	day	Starting day for simulation (1-366)
start hour	hr	Starting hour for simulation (1-24)
elapsed minutes	min	Number of minutes that have passed
elapsed hours	hr	Number of hours that have passed
num leap years	year	Number of leap years encountered
is leap year	*bool*	True if the current year is a leap year
hour index	hr	Current hour of the year (1-8784)
General Simulation		
state vector	*list*	State from each actor on the grid
names	*list*	Ordered names of actors on the grid
failing actors	*list*	List of failing components
is islanded	*bool*	True if the utiliy grid is disconnected
connectivity matrix	*list*	Cell $ij \rightarrow$ actor$_i$ receives from actor$_j$
States for a General Grid Actor (X)		
name X	*string*	Label for X
state X	kW	Power receiving (-) or providing (+)
is connected X	*bool*	True if X is connected to grid
X receives from Y	*bool*	True if X receives power from Y
X receiving from Y	*bool*	True if X is actively receiving from Y
is failing X	*bool*	True if X is failing
prob failing X		Probability of X failing
prob fixed X		Probability of failing X getting fixed
supply X	kW	Current energy that the X can supply
cost X	$/kWh	Cost of generating X's supply
is cost per unit X	*bool*	True if supply cost is per unit vs. lump
supply tuple X	*tuple*	Values for calculating X supply
req demand X	kW	Minimum power required for X
max demand X	kW	Maximum power X can receive
benefit X	$/kWh	Benefit of meeting X's demand
demand tuple X	*tuple*	Values for calculating X's demand

Node	Units	Description
States Associated with Photovoltaic Arrays (P)		
sunlight directory	*string*	Directory for solar data
sunlight filename	*string*	Filename for sunlight CSV file
sunlight data	*list*	Yearly sunlight values by hour
sunlight data label	*string*	Name of column for sunlight data
sunlight	Wh/m^2	Energy from sun for a specific hour
area P	m^2	Area of P
efficiency P		Efficiency of P
States Associated with Batteries (B)		
charge level B	kWh	Amount of charge in B
charge capacity B	kWh	Max charge B can hold
charge efficiency B		Efficiency of converting power to SOC
max output B	kW	Maximum power B can output
max charge rate B	kW	Maximum power B can receive
scarcity factor B		Cost gain of using depleted B
trickle prop B		SOC to reduce to trickle charge
is charging B	*bool*	True if B is charging
soc B		State of charge for the B
States Associated with Load Actors (L)		
normal load L	kW	Normal load (demand) of L
critical load L	kW	Critical load (demand) of L
States Associated with Buildings (B)		
type B	*enum*	Type of B
lights load B	kW	Lights load of B
equipment load B	kW	equipment load of B
load data B	*list*	List of dictionaries for B load data
building filename B	*string*	Filename for B load CSV data
load directory	*string*	Directory for B load CSV data
normal key B	*string*	Normal load column in CSV data
lights key B	*string*	Lights load column in CSV data
equipment key B	*string*	Equipment load column in CSV data
States Associated with Generators (G)		
next refuel hour G	hr	Next hour G will be refueled
prob refueling G		Probability of refueling G during the day
fuel level G	Liters	Amount of fuel in G
out of fuel G	*bool*	True if G is out of fuel
starting fuel level G	Liters	Starting fuel level in G
fuel capacity G	Liters	Max amount of fuel in G
max output G	kW	Max power G can output
consumption G	Liters	Fuel consumption for G
max consumption G	Liters	Max fuel consumption for G

Table D.4: List of nodes (variables) described in the CHG of a microgrid given in Chapter 6, along with its units and description. Note that the use of a standalone capital letter indicates that a node is given for each actor $X \in \mathbf{X}$, where $\mathbf{X}$ is the set of all actors of the type indicated in the section title (such as Generators, or all actors in general).

Appendix E. Diagram of Microgrid CHG

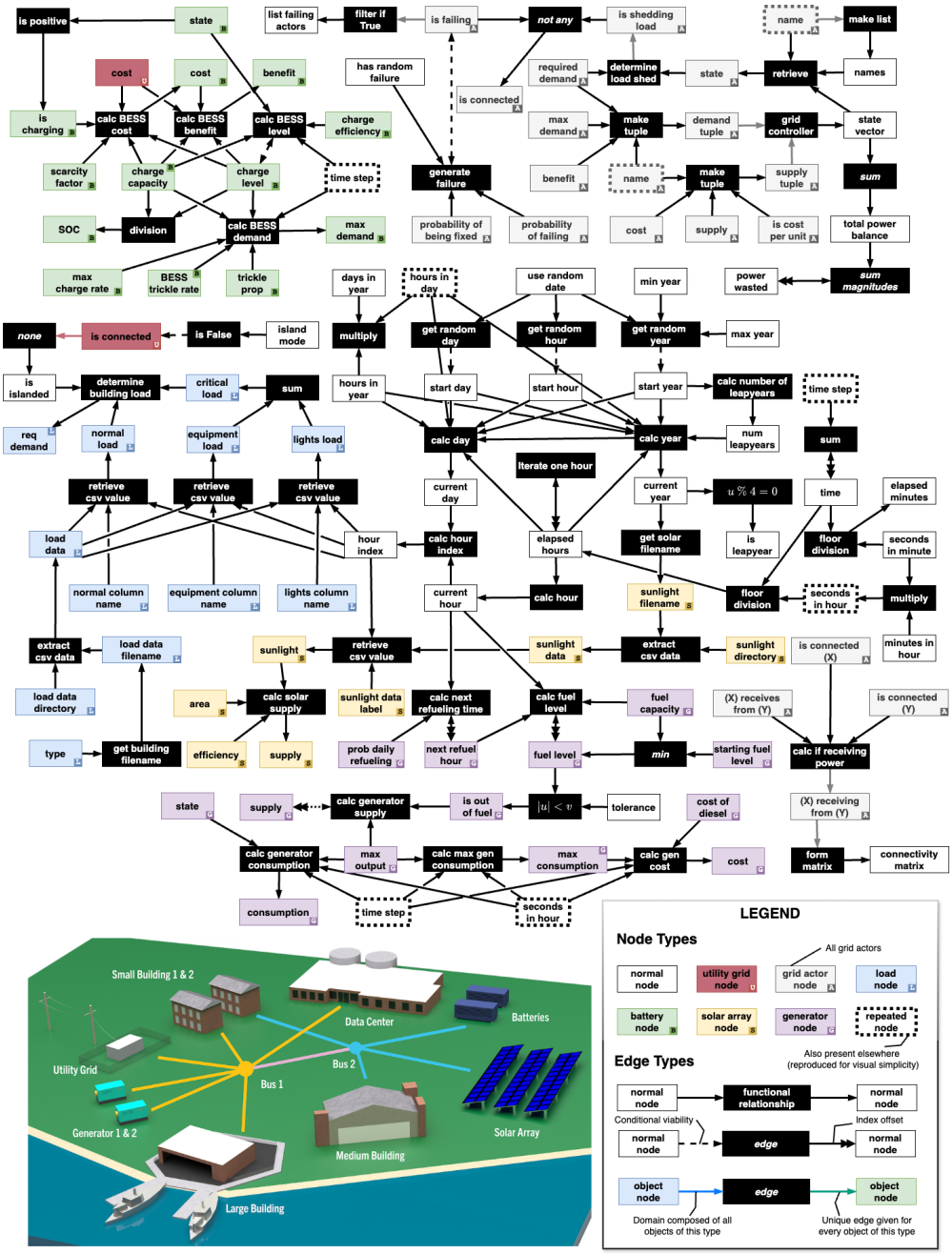


Figure E.2: Diagram of microgrid CHG showing 5 different types of connected actors either supplying or receiving energy, as described in Chapter 6. Each actor class adheres to a similar pattern shown once for simplicity. This means that all nodes that are not white or black are given once for each object; e.g. there are two nodes for charge capacity, one for each battery on the grid (green) connected in the same way.

Appendix F. Core Module for ConstraintHg Package (v0.2.3)

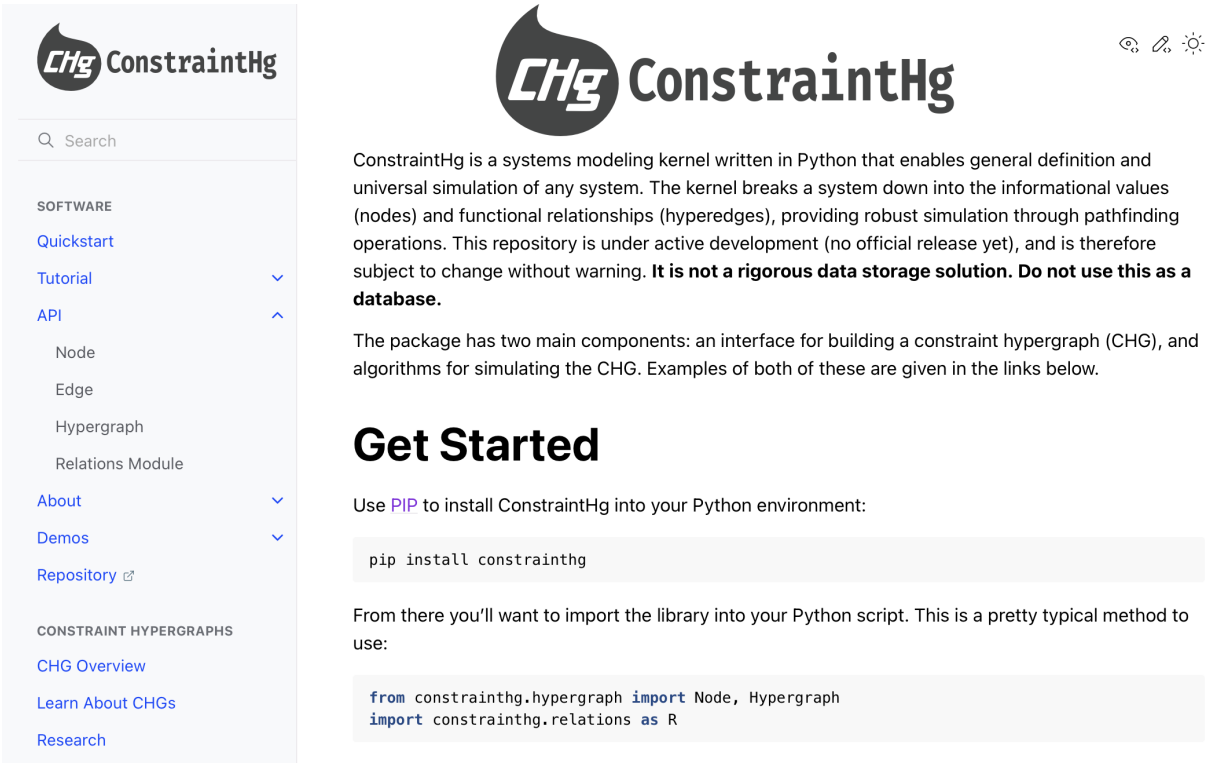


Figure F.3: Landing page for ConstraintHg documentation, available at constrainthg.readthedocs.io/en/latest/

```
"""
Copyright 2025 John Morris

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.

| File: hypergraph.py
| Author: John Morris
|   - jhmrrs@clemson.edu
|   - https://orcid.org/0009-0005-6571-1959
| Purpose: Classes for storing and traversing a constraint hypergraph.
"""
```

```
from typing import Callable, List
from inspect import signature
import logging
import itertools
from enum import Enum

logger = logging.getLogger('constrainthg')


# Helper functions
def append_to_dict_list(d: dict, key, val):
    """Appends the value to a dictionary where the dict.values are
    lists."""
    if key not in d:
        d[key] = []
    d[key].append(val)
    return d


def make_list(val) -> list:
    """Ensures that the value is a list, or else a list containing the
    value."""
    if isinstance(val, list):
        return val
    if isinstance(val, str):
        return [val]
    try:
        return list(val)
    except TypeError:
        return [val]


def make_set(val) -> list:
    """Ensures that the value is a set, or else a set containing the
    value."""
    if isinstance(val, set):
        return val
    if isinstance(val, str):
        return {val}
    try:
        return set(val)
    except TypeError:
        return {val}


class TNode:
    """A basic tree node for printing tree structures."""
    class conn:
        """A class of connectors used for indicating child nodes."""
        elbow = "└──"
        pipe = "│  "
        tee = "├──"
        blank = "   "
        elbow_join = "└─>"
        tee_join = "├─>"
```

```
        elbow_stop = "└─●─"
        tee_stop = "├─●─"

    def __init__(self, label: str, node_label: str, value=None,
                 children: list=None, cost: float=None, trace: list=None,
                 gen_edge_label: str=None, gen_edge_cost: float=0.0,
                 join_status: str='None', max_display_length: int=12):
        """
        Creates the root of a search tree.

        Parameters
        ----------
        label : str
            A unique identifier for the TNode, necessary for
            pathfinding.
        node_label : str
            A string identifying the node represented by the TNode.
        value : Any, optional
            The value of the tree solved to the TNode.
        children : list, optional
            TNodes that form the source nodes of an edge leading to the
            TNode.
        cost : float, optional
            Value indicating the cost of solving the tree rooted at the
            TNode.
        trace : list, optional
            Top down trace of how the TNode could be resolved, used for
            path exploration.
        gen_edge_label : str, optional
            A unique label for the edge generating the TNode (of which
            `children` are source nodes).
        gen_edge_cost : float, default=0.
            Value for weight (cost) of the generating edge, default is
            0.0.
        join_status : str, optional
            Indicates if the TNode is the last of a set of children,
            used for printing.
        max_display_length : int, default=12
            The maximum characters to display for the value of the node.


        Properties
        ----------
        index : int
            The maximum times the TNode or any child TNodes are repeated
            in the tree.
        values : dict
            The values of all the child TNodes of the form
            {label : [Any,]}.
        """
        self.node_label = node_label
        self.label = label
        self.value = value
        self.children = [] if children is None else children
        self.trace = [] if trace is None else trace
        self.gen_edge_label = gen_edge_label
```

```
        self.gen_edge_cost = gen_edge_cost
        self.values = {node_label: [value,]}
        self.join_status = join_status
        self.index = max([1] + [c.index for c in self.children])
        self.max_display_length = max_display_length
        self.cost = cost

    def print_conn(self, last=True) -> str:
        """Selecter function for the connector string on the tree
        print."""
        if last:
            if self.join_status == 'join':
                return self.conn.elbow_join
            if self.join_status == 'join_stop':
                return self.conn.elbow_stop
            return self.conn.elbow
        if self.join_status == 'join':
            return self.conn.tee_join
        if self.join_status == 'join_stop':
            return self.conn.tee_stop
        return self.conn.tee

    def print_tree(self, last=True, header='',
                   checked_edges: list=None) -> str:
        """Prints the tree centered at the TNode

        Adapted from https://stackoverflow.com/a/76691030/15496939,
        PierreGtch, under CC BY-SA 4.0.
        """
        out = str()
        out += header + self.print_conn(last) + str(self)
        if checked_edges is None:
            checked_edges = []
        if self.gen_edge_label in checked_edges:
            out += ' (derivative)\n' if len(self.children) != 0 else '\n'
            return out
        out += '\n'
        if self.gen_edge_label is not None:
            checked_edges.append(self.gen_edge_label)
        for i, child in enumerate(self.children):
            c_header = header + (self.conn.blank if last else self.conn.pipe)
            c_last = i == len(self.children) - 1
            out += child.print_tree(header=c_header,
                                    last=c_last,
                                    checked_edges=checked_edges)
        return out

    def get_descendents(self) -> list:
        """Returns a list of child nodes on all depths (includes
        self)."""
        out = [self]
        for c in self.children:
            out += c.get_descendents()
        return out

    @property
```

```
    def cost(self) -> float:
        """The total sum of the weights of each edge in the tree."""
        if self.calc_cost is None:
            self.calc_cost = self.get_tree_cost()
        return self.calc_cost

    @cost.setter
    def cost(self, val: float):
        """Sets the cost property of the TNode."""
        if val is None:
            self.calc_cost = self.get_tree_cost()
        elif not (isinstance(val, float) or isinstance(val, int)):
            raise TypeError("Input to cost must be float type.")
        else:
            self.calc_cost = float(val)

    @cost.deleter
    def cost(self):
        """Deletes the cost property of the TNode."""
        self.calc_cost = None

    def get_tree_cost(self, root=None, checked_edges: set=None) -> float:
        """Returns the cost of solving to the leaves of the tree."""
        if root is None:
            root = self
        if checked_edges is None:
            checked_edges = set()
        total_cost = 0
        if root.gen_edge_label not in checked_edges:
            total_cost += root.gen_edge_cost
            checked_edges.add(root.gen_edge_label)
            for st in root.children:
                total_cost += self.get_tree_cost(st, checked_edges)
        return total_cost

    def __str__(self) -> str:
        out = self.node_label
        if self.value is not None:
            if isinstance(self.value, float):
                out += f'={self.value:.4g}'[:self.max_display_length]
            else:
                out += f'={self.value}'[:self.max_display_length]
        out += f', index={self.index}'
        if self.cost is not None:
            out += f', cost={self.cost:.4g}'
        return out


class Node:
    """A value in the hypergraph, equivalent to a wired connection."""
    def __init__(self, label: str, static_value=None,
                 generating_edges: set=None,
                 leading_edges: set=None, super_nodes: set=None,
                 sub_nodes: set=None,
                 description: str=None, units: str=None):
        """Creates a new `Node` object.
```

```
        Parameters
        ----------
        label : str
            A unique identifier for the node.
        static_value : Any, optional
            The constant value of the node, set as an input.
        generating_edges : set, optional
            A set of edges that have the node as their target.
        leading_edges : set, optional
            A set of edges that have the node as one their sources.
        super_nodes : Set[Node], optional
            A set of nodes that have this node as a subset, see
            note [1].
        sub_nodes : Set[Node], optional
            A set of nodes that that have this node as a super node, see
            note [1].
        description : str, optional
            A description of the node useful for debugging.
        units : str, optional
            Units of value.
        starting_index : int, default=1
            The starting index of the node.


        Properties
        ----------
        is_constant : bool
            Describes whether the node should be reset in between
            simulations.


        Notes
        -----
        1. The subsetting accomplished by `super_nodes` is best
        conducted using `via` functions on the edge, as these will be
        executed for every node value. One case where this is impossible
        is when the node has leading edges when generated by a certain
        generating edge. In this case the `via` function cannot be used
        as the viability is *edge* dependent, not *value* dependent.
        Super nodes are provided for this purpose, though do not provide
        full functionality. When searching, the leading edges of each
        super node are added to the search queue as a valid path away
        from the node.
        """
        self.label = label
        self.static_value = static_value
        self.generating_edges = set() if generating_edges is None else generating_edges
        self.leading_edges = set() if leading_edges is None else leading_edges
        self.description = description
        self.units = units
        self.is_constant = static_value is not None
        self.super_nodes = set() if super_nodes is None else make_set(super_nodes)
        self.sub_nodes = set() if sub_nodes is None else make_set(sub_nodes)
        for sup_node in self.super_nodes:
            if not isinstance(sup_node, tuple):
```

```
                sup_node.sub_nodes.add(self)
            for sub_node in self.sub_nodes:
                if not isinstance(sub_node, tuple):
                    sub_node.super_nodes.add(self)

    def __str__(self) -> str:
        out = self.label
        if self.description is not None:
            out += ': ' + self.description
        return out

    def __iadd__(self, o):
        return self.union(self, o)

    @staticmethod
    def union(a, *args):
        """Performs a deep union of the two nodes, replacing values of
        `a` with those of `b` where necessary."""
        for b in args:
            if not isinstance(a, Node) or not isinstance(b, Node):
                raise TypeError("Inputs must be of type Node.")
            if b.label is not None:
                a.label = b.label
            if b.static_value is not None:
                a.static_value = b.static_value
                a.is_constant = b.is_constant
            if b.description is not None:
                a.description = b.description
            a.generating_edges = a.generating_edges.union(b.generating_edges)
            a.leading_edges = a.leading_edges.union(b.leading_edges)
            a.super_nodes = a.super_nodes.union(b.super_nodes)
            a.sub_nodes = a.sub_nodes.union(b.sub_nodes)
        return a


class EdgeProperty(Enum):
    """Enumerated object describing various configurations of an Edge
    that can be passed during setup. Used as shorthand for common
    configurations.

    Members
    -------
    LEVEL : 1
        Every source node in the edge must have the same index for the
        edge to be viable.
    DISPOSE_ALL : 2
        Every source node can only be used once per execution.
    """
    LEVEL = 1
    DISPOSE_ALL = 2


class Edge:
    """A relationship along a set of nodes (the source) that produces a
    single value."""
    def __init__(self, label: str, source_nodes: dict, target: Node,
```

```
            rel: Callable, via: Callable=None, index_via: Callable=None,
            weight: float=1.0, index_offset: int=0, disposable: list=None,
            edge_props: EdgeProperty=None):
        """Creates a new `Edge` object. This should generally be called
        from a Hypergraph object using the Hypergraph.addEdge method.

        Parameters
        ----------
        label : str
            A unique string identifier for the edge.
        source_nodes : dict{str : Node | Tuple(str, str)} | list[Node |
                    Tuple(str, str)] | Tuple(str, str) | Node
            A dictionary of `Node` objects forming the source nodes of
            the edge, where the key is the identifiable label for each
            source used in rel processing. The Node object may be a Node,
            or a length-2 Tuple (identifier : attribute) with the first
            element an identifier in the edge and the second element a
            string referencing an attribute of the identified Node to
            use as the value (a pseudo node).
        target : Node
            Node that the edge maps to.
        rel : Callable
            A function taking the values of the source nodes and
            returning a single value (the target).
        via : Callable, optional
            A function that must be true for the edge to be traversable
            (viable). Defaults to unconditionally true if not set.
        index_via : Callable, optional
            A function that takes in handles of source nodes as inputs
            in reference to the *index* of each referenced source node,
            returns a boolean condition relating the indices of each.
            Defaults to unconditionally true if not set, meaning any
            index of source node is valid.
        weight : float > 0.0, default=1.0
            The quanitified cost of traversing the edge. Must be
            positive, akin to a distance measurement.
        index_offset : int, default=0
            Offset to apply to the target once solved for. Akin to
            iterating to the next level of a cycle.
        disposable : list, optional
            A list of source node handles that should not be evaluated
            for future cyclic executions of the edge. That is, each
            TNode that corresponds to a handle in `disposable` is
            removed from `found_tnodes` after a successful edge calculation.
        edge_props : List(EdgeProperty) | EdgeProperty | str | int, optional
            A list of enumerated types that are used to configure the
            edge.


        Properties
        ----------
        found_tnodes : dict
            A dict of lists of source_tnodes that are viable trees to a
            source node, with each sub_dict referenced by index. format:
            {node_label : list[TNode,]}
        subset_alt_labels : dict
```

```
            A dictionary of alternate node labels if a source node is a
            super set, format: {node_label : List[alt_node_label,]}
        """
        self.rel = rel
        self.via = self.via_true if via is None else via
        self.index_via = self.via_true if index_via is None else index_via
        self.source_nodes = self.identify_source_nodes(source_nodes, self.rel, self.via)
        self.create_found_tnodes_dict()
        self.target = target
        self.weight = abs(weight)
        self.label = label
        self.index_offset = index_offset
        self.disposable = disposable
        self.edge_props = self.setup_edge_properties(edge_props)

    def create_found_tnodes_dict(self):
        """Creates the found_tnodes dictionary, accounting for super
        nodes."""
        self.subset_alt_labels = {}
        self.found_tnodes = {}
        for sn in self.source_nodes.values():
            if not isinstance(sn, tuple):
                self.subset_alt_labels[sn.label] = []
                self.found_tnodes[sn.label] = []
                for sub_sn in sn.sub_nodes:
                    self.subset_alt_labels[sn.label].append(sub_sn.label)

    def add_source_node(self, sn):
        """Adds a source node to an initialized edge.

        Parameters
        ----------
        sn : dict | Node | Tuple(str, str)
            The source node to be added to the edge.
        """
        if isinstance(sn, dict):
            key, sn = list(sn.items())[0]
        else:
            key = self.get_source_node_identifier()
        if not isinstance(sn, tuple):
            sn.leading_edges.add(self)
            self.found_tnodes[sn.label] = []

        source_nodes = self.source_nodes | {key: sn}
        if hasattr(self, 'og_source_nodes'):
            self.og_source_nodes[key] = sn
        self.source_nodes = self.identify_source_nodes(source_nodes)
        self.edge_props = self.setup_edge_properties(self.edge_props)

    def setup_edge_properties(self, inputs: None) -> list:
        """Parses the edge properties."""
        eps = []
        if inputs is None:
            return eps
        inputs = make_list(inputs)
        for ep in inputs:
```

```
            if isinstance(ep, EdgeProperty):
                eps.append(ep)
            elif ep in EdgeProperty.__members__:
                eps.append(EdgeProperty[ep])
            elif ep in [item.value for item in EdgeProperty]:
                eps.append(EdgeProperty(ep))
            else:
                logger.warning(f"Unrecognized edge property: {ep}")
        for ep in eps:
            self.handle_edge_property(ep)
        return eps

    def get_source_node_identifier(self, offset: int=0):
        """Returns a generic label for a source node."""
        return f's{len(self.source_nodes) + offset + 1}'

    def handle_edge_property(self, edge_prop: EdgeProperty):
        """Perform macro functions defined by the EdgeProperty."""
        if edge_prop is EdgeProperty.LEVEL:
            self.make_edge_level()
        elif edge_prop is EdgeProperty.DISPOSE_ALL:
            self.disposable = [key for key in self.source_nodes]

    def make_edge_level(self):
        """Adds a condition to the via function forcing all node indices
        to be equivalent."""
        if not hasattr(self, 'og_source_nodes'):
            self.og_source_nodes = dict(self.source_nodes.items())
            self.og_rel = self.rel
            self.og_via = self.via
        sns = dict(self.source_nodes.items())
        tuple_idxs = {label: el[0] for label, el in sns.items()
                      if isinstance(el, tuple)}
        for label, sn in sns.items():
            if isinstance(sn, tuple) or label in tuple_idxs.values():
                continue
            next_id = self.get_source_node_identifier()
            self.source_nodes[next_id] = (label, 'index')
            tuple_idxs[next_id] = label

        def og_kwargs(**kwargs):
            """Returns the original keywords specified when the edge was
            created."""
            return {key: kwargs[key] for key in kwargs
                    if key in self.og_source_nodes}

        def level_check(*args, **kwargs):
            """Returns true if all passed indices are equivalent."""
            if not self.og_via(*args, **kwargs):
                return False
            idxs = {val for key, val in kwargs.items() if key in tuple_idxs}
            return len(idxs) == 1

        self.via = level_check
        self.rel = lambda *args, **kwargs: self.og_rel(*args, **og_kwargs(**kwargs))
```

```
    @staticmethod
    def get_named_arguments(methods: List[Callable]) -> set:
        """Returns keywords for any keyed, required arguments
        (non-default)."""
        out = set()
        for method in methods:
            for p in signature(method).parameters.values():
                if p.kind == p.POSITIONAL_OR_KEYWORD and p.default is p.empty:
                    out.add(p.name)
        return out

    def identify_source_nodes(self, source_nodes, rel: Callable=None,
                              via: Callable=None) -> dict:
        """Returns a {str: node} dictionary where each string is the
        keyword label used in the rel and via methods."""
        if rel is None:
            rel = self.rel
        if via is None:
            via = self.via
        if isinstance(source_nodes, dict):
            return self.identify_labeled_source_nodes(source_nodes, rel, via)
        source_nodes = make_list(source_nodes)
        return self.identify_unlabeled_source_nodes(source_nodes, rel, via)

    def identify_unlabeled_source_nodes(self, source_nodes: list,
                                        rel: Callable, via: Callable) -> dict:
        """Returns a {str: node} dictionary where each string is the
        keyword label used in the rel and via methods."""
        arg_keys = self.get_named_arguments([via, rel])
        arg_keys = arg_keys.union({f's{i+1}' for i in range(len(source_nodes)
                                                            - len(arg_keys))})

        out = dict(zip(arg_keys, source_nodes))
        return out

    def identify_labeled_source_nodes(self, source_nodes: dict, rel: Callable,
                                      via: Callable) -> dict:
        """Returns a {str: node} dictionary where each string is the
        keyword label used in the rel and via methods."""
        out = {}
        arg_keys = self.get_named_arguments([rel, via])
        arg_keys = arg_keys.union({str(key) for key in source_nodes})

        for arg_key in arg_keys:
            if len(source_nodes) == 0:
                return out
            if arg_key in source_nodes:
                sn_key = arg_key
            else:
                sn_key = list(source_nodes.keys())[0]
            out[arg_key] = source_nodes[sn_key]
            del source_nodes[sn_key]

        return out

    def process(self, source_tnodes: list):
```

```
        """Processes the tnodes to get the value of the target."""
        source_vals, sourcs_idxs = self.get_source_vals_and_idxs(source_tnodes)
        target_val = self.process_values(source_vals, sourcs_idxs)
        if target_val is not None:
            self.dispose_solved_tnodes(source_tnodes)
        return target_val

    def get_source_vals_and_idxs(self, source_tnodes: list) -> tuple:
        """Returns two dictionaries mapping a source identifier with a
        value (1) or its index (2).
        """
        source_values, source_indices = {}, {}

        tuple_keys = filter(lambda key: isinstance(self.source_nodes[key], tuple),
                            self.source_nodes)
        for key in tuple_keys:
            value = self.get_psuedo_node_value(source_tnodes,
                                               *self.source_nodes[key])
            source_values[key] = value

        for st in source_tnodes:
            for key, sn in self.source_nodes.items():
                if not isinstance(sn, tuple) and st.node_label == sn.label:
                    source_values[key] = st.value
                    source_indices[key] = st.index
                    break

        return source_values, source_indices

    def get_psuedo_node_value(self, source_tnodes: list,
                              pseudo_identifier: str,
                              pseudo_attribute: str):
        """Identifies the source node and returns its attribute given by
        the pseudo-node notation.
        """
        sn_label = self.source_nodes.get(pseudo_identifier, None)
        if sn_label is None:
            return None
        sn_label = self.source_nodes[pseudo_identifier].label
        for st in source_tnodes:
            if st.node_label == sn_label:
                value = getattr(st, pseudo_attribute)
                return value
        return None

    def process_values(self, source_vals: dict,
                       source_indices: dict=None):
        """Finds the target value based on the source values and
        indices."""
        if None in source_vals:
            return None
        if source_indices is not None and not self.index_via(**source_indices):
            return None
        if self.via(**source_vals):
            return self.rel(**source_vals)
        return None
```

```
    def dispose_solved_tnodes(self, source_tnodes: list):
        """Once a TNode has been processed, it is removed from the
640     `found_tnodes` list *only* if it has been marked for removal via
        inclusion in the `disposable` list.

        This ensures that nodes from previous cycles don't get
        revetted for future edges, greatly simplifying simulation.
645
        Process
        -------
        1. Get an identifier from the disposal list
        2. Get the label for the source node corresponding to the
650     identifier
        3. Find the tnode used in the solution (from source_tnodes) with
        the matching node_label
        4. Find the set of found_tnodes from the edge corresponding to
        the node label
655     5. Remove the tnode in found_tnodes with the same index as the
        solved tnode
        """
        if self.disposable is not None:
            count = 0
660         for identifier in self.disposable:
                sn = self.source_nodes.get(identifier, None)
                if sn is None or isinstance(sn, tuple):
                    continue
                node_label = sn.label
665
                for st in source_tnodes:
                    if st.node_label == node_label:
                        index = st.index
                        break
670             else:
                    continue

                count += self.dispose_of_tnodes_with_index(node_label, index)
            logger.debug(f'(Disposed of {count} nodes in {self.label})')
675
    def dispose_of_tnodes_with_index(self, node_label: str,
                                     index: int) -> int:
        """Removes each TNode from the edge property `found_tnodes` with
        a matching node_label and index. Returns the number of TNodes
680     succesfully removed.
        """
        matching_tnodes = self.found_tnodes.get(node_label, None)
        if matching_tnodes is None:
            return 0
685     matching_tnodes = [mt for mt in matching_tnodes]
        self.found_tnodes[node_label] = [t for t in matching_tnodes
                                         if t.index != index]
        return len(matching_tnodes) - len(self.found_tnodes[node_label])

690 def get_source_tnode_combinations(self, t: TNode, DEBUG: bool=False):
        """Returns all viable combinations of source nodes using the
        TNode `t`."""
```

```
        if not self.add_found_tnode(t):
            return []

        st_candidates = []
        if DEBUG:
            for st_label, sts in self.found_tnodes.items():
                val_idxs = [f'{str(st.value)[:4]}({st.index})' for st in sts]
                var_info = ', '.join(val_idxs)
                msg = f' - {st_label}: ' + var_info
                logger.log(logging.DEBUG + 2, msg)

        for st_label, sts in self.found_tnodes.items():
            if st_label == t.node_label:
                st_candidates.append([t])
            elif len(sts) == 0:
                return []
            else:
                st_candidates.append(sts)

        st_combos = itertools.product(*st_candidates)
        return st_combos

    def add_found_tnode(self, t: TNode) -> bool:
        """Returns true if `t` successfully added as a viable path to a
        source node."""
        node_label = self.get_relevant_node_label(t)
        if self.check_tnode_already_found(t, node_label):
            return False
        append_to_dict_list(self.found_tnodes, node_label, t)
        return True

    def get_relevant_node_label(self, t: TNode) -> str:
        """Returns the node label of `t` or of the super set of `t`, if
        present."""
        if t.node_label not in self.found_tnodes:
            for label, sub_labels in self.subset_alt_labels.items():
                if t.node_label in sub_labels:
                    return label
        return t.node_label

    def check_tnode_already_found(self, t: TNode,
                                  source_node_label: str) -> bool:
        """Returns True if `t` has already been found as a path to the
        source node."""
        ft_labels = [ft.label for ft in self.found_tnodes[source_node_label]]
        return t.label in ft_labels

    @staticmethod
    def via_true(*args, **kwargs):
        """Returns true for all inputs (unconditional edge)."""
        return True

    def __str__(self):
        return self.label
```

```
class Pathfinder:
    """Object for searching a path through the hypergraph from a
    collection of source nodes to a single target node. If the
    hypergraph is fully constrained and viable, then the result of the
    search is a singular value of the target node."""
    def __init__(self, target: Node, sources: list, nodes: dict,
                 no_weights: bool=False, memory_mode: bool=False):
        """Creates a new Pathfinder object.

        Parameters
        ----------
        target : Node
            The Node that the Pathfinder will attempt to solve for.
        sources : list
            A list of Node objects that have static values for the
            simulation.
        nodes : dict
            A dictionary of nodes taken from the hypergraph as
            {label : Node}.
        no_weights : bool, default=False
            Optional run mode where weights aren't considered. This
            speeds up the simulation but prevents model switching.
        memory_mode : bool, default=False
            Optional run mode where all encountered TNodes are stored to
            a list property. Increases memory usage.


        Properties
        ----------
        search_counter : int
            Number of nodes explored.
        explored_edges : dict
            Dict counting the number of times edges were processed
            {label : int}.
        explored_tnodes : list
            Dict containing the all TNodes explored during searching,
            if not running in memory mode.
        """
        self.nodes = nodes
        self.source_nodes = sources
        self.target_node = target
        self.no_weights = no_weights
        self.memory_mode = memory_mode
        self.search_roots = []
        self.search_counter = 0
        self.explored_edges = {}
        self.explored_nodes = []

    def search(self, min_index: int=0, debug_nodes: list=None,
               debug_edges: list=None, search_depth: int=10000):
        """Searches the hypergraph for a path from the source nodes to the
        target node. Returns the solved TNode for the target, with a dictionary
        of found values {label : [Any,]} given by the `target.values`.

        Parameters
        ----------
```

```
        min_index : int, default=0
            Minimum index of the target node.
        debug_nodes: list, optional
            List of nodes to log additional information for.
        debug_edges : list, optional
            List of edges to log additional information for.
        search_depth : int, default=10000
            Number of TNodes to explore before search is failed.
        """
        debug_nodes = [] if debug_nodes is None else debug_nodes
        debug_edges = [] if debug_edges is None else debug_edges
        self.explored_nodes, self.explored_edges = [], {}
        logger.info(f'Begin search for {self.target_node.label}')

        for sn in self.source_nodes:
            st = TNode(f'{sn.label}#0', sn.label, sn.static_value, cost=0.)
            self.search_roots.append(st)

        while len(self.search_roots) > 0:
            if self.search_counter > search_depth:
                self.log_debugging_report()
                raise Exception("Maximum search limit exceeded.")

            labels = [f'{s.node_label}' for s in self.search_roots]
            logger.debug('Search trees: ' + ', '.join(labels))

            root = self.select_root()

            logger.debug(f'Exploring <{root.label}>, index={root.index}:')
            if self.memory_mode:
                self.explored_nodes.append(root)

            if root.node_label is self.target_node.label and root.index >= min_index:
                logger.info(f'Finished search for {self.target_node.label} with value of {root.value}')
                self.log_debugging_report()
                return root

            self.explore(root, debug_nodes, debug_edges)

        logger.info('Finished search, no solutions found')
        self.log_debugging_report()
        return None

    def explore(self, t: TNode, debug_nodes: list=None, debug_edges: list=None):
        """Discovers all possible routes from the TNode."""
        leading_edges = self.get_edges_to_explore(t, debug_nodes)
        if t.node_label in debug_nodes:
            logger.log(logging.DEBUG + 2,
                       f'Exploring {t.node_label}, index: {t.index}, '
                       + 'leading edges: '
                       + ', '.join(str(le) for le in leading_edges)
                       + f'\n{t.print_tree()}')

        for i, edge in enumerate(leading_edges):
            if edge.label not in self.explored_edges:
                self.explored_edges[edge.label] = [0, 0, 0]
```

```
            self.explored_edges[edge.label][0] += 1

            DEBUG = edge.label in debug_edges
            level = logging.DEBUG + (2 if DEBUG else 0)
            logger.log(level, f"- Edge {i}=<{edge.label}>, target=<{edge.target.label}>:")

            combos = edge.get_source_tnode_combinations(t, DEBUG)
            for j, combo in enumerate(combos):
                pt = self.make_parent_tnode(combo, edge.target, edge)
                self.explored_edges[edge.label][1] += 1
                if pt is not None:
                    self.explored_edges[edge.label][2] += 1

                node_indices = ', '.join(f'{n.label} ({n.index})' for n in combo)
                logger.debug(f' - Combo {j}: ' + node_indices + f'-> <{str(pt)}>')

    def get_edges_to_explore(self, t: TNode, debug_nodes: list=None) -> list:
        """Finds and orders all edges leading from the node by label."""
        n = self.nodes[t.node_label]
        super_node_leading_edges = (sup_n.leading_edges for sup_n in n.super_nodes)
        leading_edges = list(n.leading_edges.union(*super_node_leading_edges))
        leading_edges.sort(key=lambda le: le.label)
        return leading_edges

    def make_parent_tnode(self, source_tnodes: list, node: Node, edge: Edge):
        """Creates a TNode for the next step along the edge."""
        parent_val = edge.process(source_tnodes)
        if parent_val is None:
            return None
        node_label = node.label
        children = source_tnodes
        gen_edge_label = edge.label + '#' + str(self.search_counter)
        label = f'{node_label}#{self.search_counter + 1}'
        cost = 0.0 if self.no_weights else None

        parent_t = TNode(label,
                         node_label,
                         parent_val,
                         children,
                         cost=cost,
                         gen_edge_label=gen_edge_label,
                         gen_edge_cost=edge.weight)
        parent_t.values = self.merge_found_values(parent_val,
                                                  node.label,
                                                  source_tnodes)
        parent_t.index += edge.index_offset

        if self.edge_resolves_input(parent_t):
            return None
        self.search_roots.append(parent_t)
        self.search_counter += 1
        return parent_t

    def edge_resolves_input(self, parent_t: TNode):
        """Returns True if the edge attempts to resolve the first index
```

```
        of an input.

        Note that inputs can be resolved as part of cycles, but only for
        later indices (2 or greater).
        """
        source_node_labels = [sn.label for sn in self.source_nodes]
        target_is_input = parent_t.node_label in source_node_labels
        resolves_input = parent_t.index == 1 and target_is_input
        return resolves_input

    def select_root(self) -> TNode:
        """Determines the most optimal path to explore."""
        if len(self.search_roots) == 0:
            return None

        min_idx = min(self.search_roots, key=lambda t: t.index).index
        lowest_idx_roots = filter(lambda t: t.index == min_idx, self.search_roots)
        root = min(lowest_idx_roots, key=lambda t: t.cost)

        self.search_roots.remove(root)
        return root

    def merge_found_values(self, parent_val, parent_label,
                           source_tnodes: list) -> dict:
        """Merges the values found in the source nodes with the parent node."""
        values = {parent_label: []}
        for st in source_tnodes:
            for label, st_values in st.values.items():
                if label not in values or len(st_values) > len(values[label]):
                    values[label] = st_values
        values[parent_label].append(parent_val)
        return values

    def log_debugging_report(self):
        """Prints a debugging report of the search."""
        out = f'\nDebugging Report for {self.target_node.label}:\n'
        out += f'\tFinal search counter: {self.search_counter}\n'
        out += '\tExplored edges (# explored | # processed | # valid solution):\n'
        sorted_edges = list(self.explored_edges.items())
        sorted_edges.sort(key=lambda a: max(a[1]), reverse=True)
        for e, vals in sorted_edges:
            out += f'\t\t<{e}>: ' + ' | '.join([str(v) for v in vals]) + '\n'
        logger.log(logging.DEBUG + 1, out)


class Hypergraph:
    """Builder class for a hypergraph. See demos for examples on how to
    use.

    Properties
    ----------
    nodes : dict
        Nodes in the hypergraph, {label : Node}
    edges : dict
        Edges in the hypergraph, {label : Edge}
```

```
        solved_tnodes : list
            List of solved TNodes from a simulation. Only set if run in
            `memory_mode`.
        no_weights : bool
            Indicates no weights have been given to the edges in the
            Hypergraph, speeding up processing (but preventing model
            switching).
        memory_mode : bool
            Indicates whether the TNodes in the Hypergraph should be saved
            between calls.
        """
        def __init__(self, no_weights: bool=False, setup_logger: bool=False,
                     logging_level=None, memory_mode: bool=False):
            """Initialize a Hypergraph.

            Parameters
            ----------
            no_weights : bool, default=False
                Optional run mode where weights aren't considered. This
                speeds up the simulation but prevents model switching.
            setup_logger : bool, default=False
                Sets up logging in the library (off by default). The logging
                level can be set by calling `Hypergraph.set_logging_level`.
            logging_level : int | str, optional
                Sets the logging level for the library. Setting the logging
                level requires an additional logging handler to be passed to
                `logger.getLogger('constrainthg')`. This can be done at the
                application level (in the calling script) or automatically
                by passing `setup_logger` as True.
            memory_mode : bool, default=False
                Store every solved for TNode to the Hypergraph.
            """
            self.nodes = {}
            self.edges = {}
            self.no_weights = no_weights
            self.logging_is_setup = self.check_if_logger_setup()
            if setup_logger:
                self.setup_logger()
            if logging_level is not None:
                self.set_logging_level(logging_level)
            self.memory_mode = memory_mode
            self.solved_tnodes = []

        def __iadd__(self, o):
            """Merges the passed Hypergraph to self via a union operation."""
            return self.union(self, o)

        def __add__(self, o):
            """Creates a shallow copy of self and joins that to ``o`` via a
            union operation."""
            if not isinstance(o, Hypergraph):
                raise Exception("Input must be of type Hypergraph.")
            new_hg = self.__copy__()
            new_hg = self.union(new_hg, o)
            return new_hg
```

```
    def __copy__(self):
        """Returns a shallow copy of the Hypergraph."""
        new_hg = Hypergraph(
            no_weights=self.no_weights,
            memory_mode=self.memory_mode,
        )
        self.union(new_hg, self)
        return new_hg

    def check_if_logger_setup(self) -> bool:
        """Checks if a Handler beyond the NullHandler was created for
        the logger."""
        if not logger.hasHandlers():
            return False
        non_null = [h for h in logger.handlers
                    if not isinstance(h, logging.NullHandler)]
        return len(non_null) > 0

    def setup_logger(self) -> logging.Logger:
        """An optional call to setup logging."""
        fh = logging.FileHandler("constrainthg.log")
        log_formatter = logging.Formatter(
            fmt="[{asctime} | {levelname}]: {message}",
            style="{",
            datefmt="%Y-%m-%d %H:%M",
        )
        fh.setFormatter(log_formatter)
        logger.addHandler(fh)
        self.logging_is_setup = True
        return logger

    def set_logging_level(self, logging_level=logging.INFO):
        """Sets the logging level.

        Parameters
        ----------
        logging_level : int | str, default=logging.INFO (20)
            The level to set logging to, based on the Python logging
            library. More information is available at
            https://docs.python.org/3/howto/logging.html#logging-levels

        Notes
        -----
        The logging approach is the following, with higher levels
        include all items logged on lower ones:
        - logging.DEBUG (10): all edges and found combinations are
        listed, as well as search trees at each explored node.
        - logging.DEBUG+1 (11): debugging report is logged after a
        search is complete.
        - logging.DEBUG+2 (12): edges passed to `debug_edges` and nodes
        passed to `debug_nodes` as arguments to `Hypergraph.solve` are
        logged, as well as search trees at each explored node.
        - logging.INFO (20): start and end of a search are logged.
        - Warnings and errors are handled by the logging package
        (logging.WARNING and logging.ERROR). Note that these will *not*
        print to `sys.stderr`, though they will normally get raised and
```

```
        returned by the library.
        """
        if not self.logging_is_setup:
            self.setup_logger()
        logger.setLevel(logging_level)

    def get_node(self, node_key) -> Node:
        """Caller function for finding a node in the hypergraph."""
        if isinstance(node_key, Node):
            node_key = node_key.label
        try:
            return self.nodes[node_key]
        except KeyError:
            msg = f'No node with label <{node_key}> found in Hypergraph.'
            raise KeyError(msg)

    def get_edge(self, edge_key) -> Node:
        """Caller function for finding a node in the hypergraph."""
        if isinstance(edge_key, Edge):
            edge_key = edge_key.label
        try:
            return self.edges[edge_key]
        except KeyError:
            return None

    def reset(self):
        """Clears all values in the hypergraph."""
        for node in self.nodes.values():
            if not node.is_constant:
                node.static_value = None
        for edge in self.edges.values():
            edge.create_found_tnodes_dict()
        self.solved_tnodes = []

    def request_node_label(self, requested_label=None) -> str:
        """Generates a unique label for a node in the hypergraph"""
        label = 'n'
        if requested_label is not None:
            label = requested_label
        i = 0
        check_label = label
        while check_label in self.nodes:
            check_label = label + str(i := i + 1)
        return check_label

    def request_edge_label(self, requested_label: str=None,
                           source_nodes: list=None) -> str:
        """Generates a unique label for an edge in the hypergraph."""
        label = 'e'
        if requested_label is not None:
            label = requested_label
        elif source_nodes is not None:
            label_names = [s.label[:4] for s in source_nodes[:-1]]
            label = '(' + ','.join(label_names) + ')'
            label += '->' + source_nodes[-1].label[:8]
        i = 0
```

```
            check_label = label
            while check_label in self.edges:
                check_label = label + '#' + str(i := i + 1)
            return check_label

        def add_node(self, node=None, *args, **kwargs) -> Node:
            """Creates (if necessary) a Node and inserts into the hypergraph.

            Wraps ``Hypergraph.insert_node`` and ``Node.__init__``.

            Parameters
            ----------
            node : Node | str, optional
                The node (or node label) to add to the hypergraph. If not
                provided, then args and kwargs are passed to Node.__init__.
            """
            if node is None:
                try:
                    node = Node(*args, **kwargs)
                except Exception:
                    return None
            return self.insert_node(node)

        def insert_node(self, node: Node, value=None) -> Node:
            """Adds a node to the hypergraph via a union operation."""
            if isinstance(node, tuple):
                return None
            if isinstance(node, Node):
                if node.label in self.nodes:
                    label = node.label
                    self.nodes[label] += node
                else:
                    label = self.request_node_label(node.label)
                    self.nodes[label] = node
            else:
                if node in self.nodes:
                    label = node
                else:
                    label = self.request_node_label(node)
                    self.nodes[label] = Node(label, value)
            return self.nodes[label]

        def add_edge(self, sources: dict, target, rel, via=None, index_via=None,
                     weight: float=1.0, label: str=None, index_offset: int=0,
                     disposable=None, edge_props=None):
            """Adds an edge to the hypergraph.

            Parameters
            ----------
            sources : dict{str : Node | Tuple(Node, str)} | list[Node |
                          Tuple(Node, str)] | Tuple(Node, str) | Node
                A dictionary of `Node` objects forming the source nodes of
                the edge, where the key is the identifiable label for each
                source used in rel processing. The Node object may be a Node,
                or a length-2 Tuple with the second element a string
                referencing an attribute of the Node to use as the value (a
```

```
            pseudo node).
        targets : list | str | Node
            A list of nodes that are the target of the given edge, with
            the same type as sources. Since each edge can only have one
            target, this makes a unique edge for each target.
        rel : Callable
            A function taking in a value for each source node that
            returns a single value for the target.
        via : Callable, optional
            A function that must be true for the edge to be traversable
            (viable). Defaults to unconditionally true if not set.
        index_via : Callable, optional
            A function that takes in handles of source nodes as inputs
            in reference to the *index* of each referenced source node,
            returns a boolean condition relating the indices of each.
            Defaults to unconditionally true if not set, meaning any
            index of source node is valid.
        weight : float, default=1.0
            The cost of traversing the edge. Must be positive.
        label : str, optional
            A unique identifier for the edge.
        index_offset : int, default=0
            Offset to apply to the target once solved for. Akin to
            iterating to the next level of a cycle.
        disposable : list, optional
            A list of source node handles that should not be evaluated
            for future cyclic executions of the edge. That is, each
            tnode that corresponds to a handle in `disposable` is
            removed from `found_tnodes` after a successful edge
            calculation.
        edge_props : List(EdgeProperty) | EdgeProperty | str | int, optional
            A list of enumerated types that are used to configure the
            edge.
        """
        source_nodes, source_inputs = self.get_nodes_and_identifiers(sources)
        target_nodes, target_inputs = self.get_nodes_and_identifiers([target])
        label = self.request_edge_label(label, source_nodes + target_nodes)
        edge = Edge(label, source_inputs, target_nodes[0],
                    rel, via, index_via, weight,
                    index_offset=index_offset, disposable=disposable,
                    edge_props=edge_props)
        self.edges[label] = edge
        for sn in source_nodes:
            sn.leading_edges.add(edge)
        for tn in target_nodes:
            tn.generating_edges.add(edge)
        return edge

    def insert_edge(self, edge: Edge):
        """Inserts a fully formed edge into the hypergraph."""
        if not isinstance(edge, Edge):
            raise TypeError('edge must be of type `Edge`')
        self.edges[edge.label] = edge
        tn = self.insert_node(edge.target)
        tn.generating_edges.add(edge)
```

```
    @staticmethod
    def union(a, *args):
        """Merges with another Hypergraph via a union operation,
        preserving all nodes and edges in the two graphs.
        """
        if not isinstance(a, Hypergraph):
            raise Exception('Input must by of type Hypergraph.')
        for b in args:
            if not isinstance(b, Hypergraph):
                raise Exception('Parameters are not of type Hypergraph.')
            for node in b.nodes.values():
                a.insert_node(node)
            for edge in b.edges.values():
                a.insert_edge(edge)
            a_tns = set(a.solved_tnodes).union(set(b.solved_tnodes))
            a.solved_tnodes = list(a_tns)
        return a

    def get_nodes_and_identifiers(self, nodes):
        """Helper function for getting a list of nodes and their
        identified argument format for various input types."""
        if isinstance(nodes, dict):
            node_list, inputs = [], {}
            for key, node in nodes.items():
                if isinstance(node, tuple):
                    if node[0] not in nodes:
                        raise Exception(f"Pseudo node identifier for '{node[0]}' not included in Edge.")
                else:
                    node = self.insert_node(node)
                    node_list.append(node)
                inputs[key] = node
            return node_list, inputs

        nodes = make_list(nodes)
        node_list = [self.insert_node(n) for n in nodes]
        inputs = [self.get_node(node) for node in nodes
                  if not isinstance(node, tuple)]
        return node_list, inputs

    def set_node_values(self, node_values: dict):
        """Sets the values of the given nodes.

        Creates a new node in the hypergraph if the given label is not
        found.
        """
        for key, value in node_values.items():
            try:
                node = self.get_node(key)
            except KeyError:
                node = self.insert_node(key, value)
            node.static_value = value

    def solve(self, target, inputs: dict=None, to_print: bool=False,
              min_index: int=0, debug_nodes: list=None, debug_edges: list=None,
              search_depth: int=100000, memory_mode: bool=False,
              logging_level=None, to_reset: bool=True):
```

```
        """Runs a BFS search to identify the first valid solution for
1310    `target`.

        Parameters
        ----------
        target : Node | str
1315        The node or label of the node to solve for.
        inputs : dict, optional
            A dictionary {label : value} of input values.
        to_print : bool, default=False
            Prints the search tree if set to true.
1320    min_index : int, default=0
            The minumum index of the node to solve for.
        debug_nodes : List[label,], optional
            A list of node labels to log debugging information for.
        debug_edges : List[label,], optional
1325        A list of edge labels to log debugging information for.
        search_depth : int, default=100000
            Number of nodes to explore before concluding no valid path.
        memory_mode : bool, default=False
            Found TNodes in the path are saved to the Hypergraph.
1330    logging_level : int | str, optional
            The logging level to use for the simulation. Configures
            logging if not already configured. `logging.DEBUG` or
            `logging.INFO` are informative levels. See
            `Hypergraph.set_logging_level` for more information.
1335    to_reset : bool, default=True
            Resets the Hypergraph so that only nodes with static values
            are preseeded. Should be `True` for independent simulations,
            `False` for repeated simulations of different values from
            the same scenario.
1340
        Returns
        -------
        TNode | None
            the TNode for the minimum-cost path found
1345    dict | None
            a dictionary of values found for each node in the search
            path, as {label : List[value,]}
        """
        if logging_level is not None:
1350        prev_logging_level = logger.getEffectiveLevel()
            self.set_logging_level(logging_level)
        if to_reset:
            self.reset()

1355    inputs = {} if inputs is None else inputs
        self.set_node_values(inputs)
        source_nodes = self.process_source_nodes(inputs)

        try:
1360        target_node = self.get_node(target)
        except KeyError:
            msg = f'Target node {str(target)} not found in Hypergraph.'
            raise KeyError(msg)
```

```
        pf = Pathfinder(
            target=target_node,
            sources=source_nodes,
            nodes=self.nodes,
            no_weights=self.no_weights,
            memory_mode=self.memory_mode or memory_mode,
        )
        try:
            t = pf.search(
                min_index=min_index,
                debug_nodes=debug_nodes,
                debug_edges=debug_edges,
                search_depth=search_depth,
            )
            if self.memory_mode or memory_mode:
                self.solved_tnodes = pf.explored_nodes
        except Exception as e:
            logger.error(str(e))
            raise e
        finally:
            if logging_level is not None:
                self.set_logging_level(prev_logging_level)
        if to_print:
            print("No solutions found" if t is None else t.print_tree())
        return t

    def process_source_nodes(self, inputs):
        """Processes source nodes for the simulation."""
        source_nodes = []
        for label in inputs:
            try:
                source_nodes.append(self.get_node(label))
            except KeyError:
                msg = f'Input node <{label}> not found in Hypergraph.'
                raise KeyError(msg)
        source_nodes += self.get_constant_nodes(inputs)
        return source_nodes

    def get_constant_nodes(self, inputs: list=None):
        """Returns all the constant nodes in the Hypergraph, optionally
        filtered by nodes not in `inputs`."""
        if inputs is None:
            inputs = []
        constant_nodes = [node for node in self.nodes.values()
                          if node.is_constant and node.label not in inputs]
        return constant_nodes

    def print_paths(self, target, to_print: bool=False) -> str:
        """Prints the hypertree of all paths to the target node."""
        try:
            target_node = self.get_node(target)
        except KeyError:
            msg = f'Target node {str(target)} not found in Hypergraph.'
            raise KeyError(msg)
        target_tnode = self.print_paths_helper(target_node)
        out = target_tnode.print_tree()
```

```
        if to_print:
            print(out)
        return out

    def print_paths_helper(self, node: Node, join_status='none',
                           trace: list=None) -> TNode:
        """Recursive helper to print all paths to the target node."""
        if isinstance(node, tuple):
            return None
        label = f'{node.label}#{0 if trace is None else len(trace)}'
        t = TNode(label, node.label, node.static_value,
                  join_status=join_status, trace=trace)
        branch_costs = []
        for edge in node.generating_edges:
            if self.edge_in_cycle(edge, t):
                t.node_label += '[CYCLE]'
                return t

            child_cost = 0
            for i, child in enumerate(edge.source_nodes.values()):
                c_join_status = self.get_join_status(i, len(edge.source_nodes))
                c_trace = t.trace + [(t, edge)]
                c_tnode = self.print_paths_helper(child, c_join_status, c_trace)
                if c_tnode is None:
                    continue
                child_cost += c_tnode.cost if c_tnode.cost is not None else 0.0
                t.children.append(c_tnode)
            branch_costs.append(child_cost + edge.weight)

        t.cost = min(branch_costs) if len(branch_costs) > 0 else 0.
        return t

    def edge_in_cycle(self, edge: Edge, t: TNode):
        """Returns true if the edge is part of a cycle in the tree rooted at
        the TNode."""
        return edge.label in [e.label for tt, e in t.trace]

    def get_join_status(self, index, num_children):
        """Returns whether or not the node at the given index is part of a
        hyperedge (`join`) or specifically the last node in a hyperedge
        (`join_stop`) or a singular edge (`none`)"""
        if num_children > 1:
            return 'join_stop' if index == num_children - 1 else 'join'
        return 'none'
```