Declarative, Multi-physics Simulation Between Applications via Constraint Hypergraphs

John Morris[1]
Department of Mechanical Engineering,
Clemson University,
Clemson, SC USA
email: jhmrrs@clemson.edu

Abhishek Indupally
Department of Mechanical Engineering,
Clemson University,
email: aindupa@clemson.edu

Satchit Ramnath
Department of Mechanical Engineering,
Clemson University,
email: sramnat@clemson.edu

Gregory Mocko
Department of Mechanical Engineering,
Clemson University,
email: gmocko@clemson.edu

John Wagner
Department of Mechanical Engineering,
Clemson University,
email: jwagner@clemson.edu

To avoid the blind spots and brittleness of imperatively simulating a physical system, modelers often turn to declarative methods that can autonomously execute a model. Such solvers rely on a model structure that allows procedures for transforming inputs to outputs to be automatically discovered. However, the model structures used for traditional declarative solvers are insular, in that they are often isolated to specific modeling domain. Furthermore, these specific model types are unable to be understood by the advanced computing applications required to simulate complex systems. Here we provide a general purpose modeling framework that can integrate software functionalities into the declarative simulation of a model, offering for the first time the ability to define a multi-physics, multi-scale system independent of its eventual simulation. This is accomplished by representing a system as a constraint hypergraph. System models are deconstructed into state variables and relationships in the graph. The APIs of external tools are integrated into the model as inter-variable functions. By encoding these functionalities in the graph, a solver is able to autonomously arrange these relations into executable simulation processes, enabling fully declarative simulation. This is demonstrated by integrating the capabilities of three software platforms together into a single model of a crankshaft from a piston engine: solid geometry (Onshape), structural mechanics (Ansys Mechanical), and kinematic analysis (MATLAB). 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.

Keywords: Simulation/Physics Based Modeling, Computer-Aided Design (CAD), CAD/Features Technology, Design Automation, Systems Engineering

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 [1]. 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.

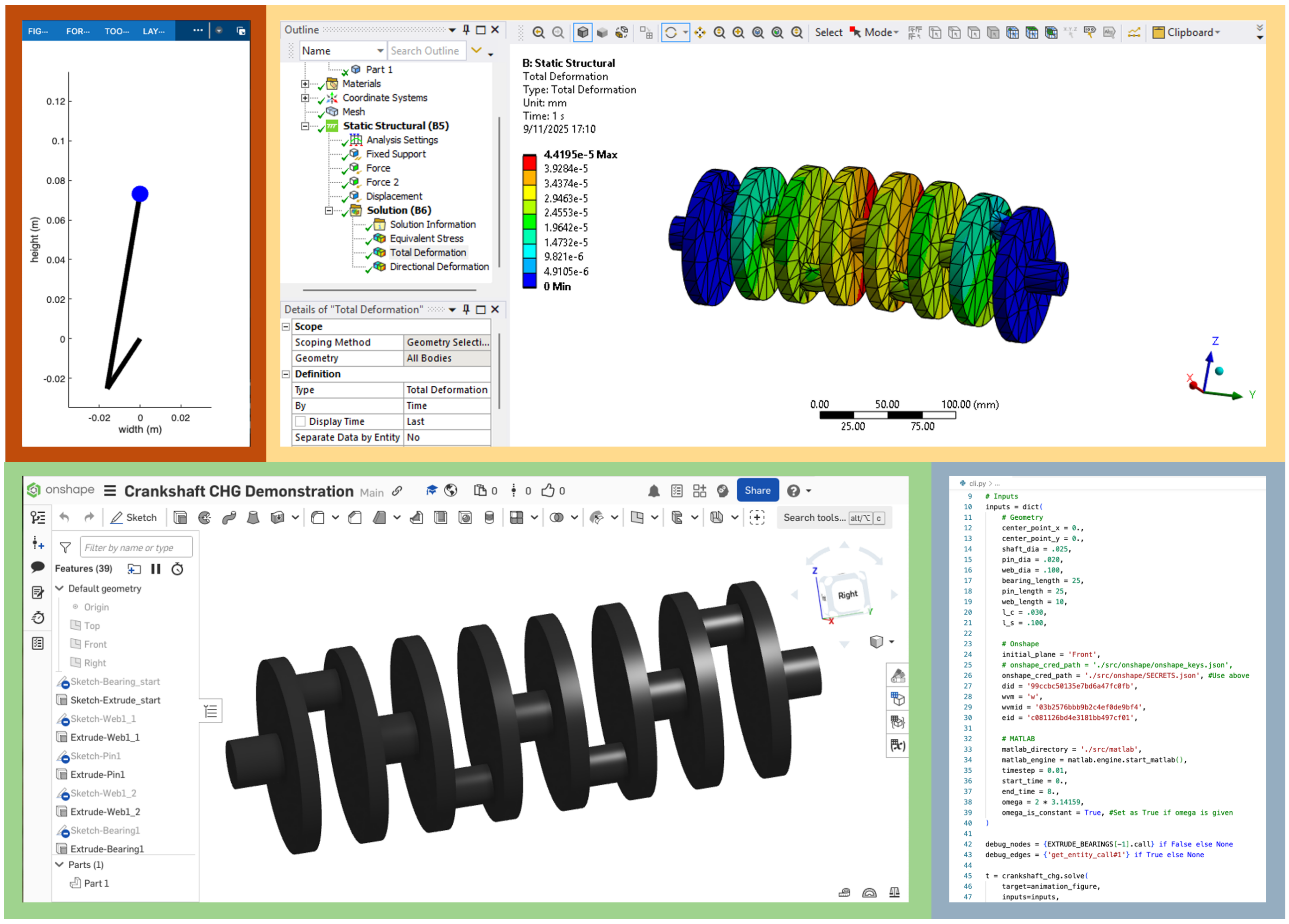
In contrast to the inflexibility of procedural simulation, this paper discusses a declarative framework for systems modeling and simulation, previously introduced in [2]. Termed a constraint hypergraph (CHG), this framework reduces a system to a set of state variables related by mathematical functions. 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) [3]. 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.

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 Fig. 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

[1]Corresponding Author.
Preprint, September 13, 2025



B: Static Structural
Total Deformation
Type: Total Deformation
Unit: mm
Time: 1 s
9/11/2025 17:10

🟥	4.4195e-5 Max
🟧	3.9284e-5
🟨	3.4374e-5
🟩	2.9463e-5
🟩	2.4553e-5
🟩	1.9642e-5
🟦	1.4732e-5
🟦	9.821e-6
🟦	4.9105e-6
🟦	0 Min

Details of "Total Deformation"

Scope	
Scoping Method	Geometry Selecti...
Geometry	All Bodies
Definition	
Type	Total Deformation
By	Time
Display Time	Last
Separate Data by Entity	No

```python
# Inputs
inputs = dict(
    # Geometry
    center_point_x = 0.,
    center_point_y = 0.,
    shaft_dia = .025,
    pin_dia = .020,
    web_dia = .100,
    bearing_length = 25,
    pin_length = 25,
    web_length = 10,
    l_c = .030,
    l_s = .100,

    # Onshape
    initial_plane = 'Front',
    # onshape_cred_path = './src/onshape/onshape_keys.json',
    onshape_cred_path = './src/onshape/SECRETS.json', #Use above
    did = '99ccbc50135e7bd6a47fc0fb',
    wvm = 'w',
    wvmid = '03b2576bbb9b2c4ef0de9bf4',
    eid = 'c081126bd4e3181bb497cf01',

    # MATLAB
    matlab_directory = './src/matlab',
    matlab_engine = matlab.engine.start_matlab(),
    timestep = 0.01,
    start_time = 0.,
    end_time = 8.,
    omega = 2 * 3.14159,
    omega_is_constant = True, #Set as True if omega is given
)

debug_nodes = {EXTRUDE_BEARINGS[-1].call} if False else None
debug_edges = {'get_entity_call#1'} if True else None

t = crankshaft_chg.solve(
    target=animation_figure,
    inputs=inputs,
```

Fig. 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).

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.

2 Review of System Modeling and Simulation

A system is an arrangement of things, such that the things exhibit some specific behavior [4]. The work of an engineer or decision-maker in any field is to provide a system whose behavior achieves a specific value-adding objective [5]. Similarly, the goal of a scientist is to characterize the behavior of the complex, interconnected system that is the universe [6]. From both cases it can be seen that nearly all human operations require some method for understanding the behavior of a system [7], 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 [8].

2.1 System Modeling. What makes a plurality of things a system is their interactions [9]. 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 [10]. 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 [11]. 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 [12]. 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 [13]. 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 [14] (the word function here is denotes a mathematical morphism [15] rather than a role of a system as used in design theory [16]). 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.

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 [7]. 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 [17]. 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* [18]) connecting some known inputs to the unknown outputs [19,20].

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

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

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

$$\omega = \frac{d\theta}{dt} \tag{3}$$

$$\dot{y} = \frac{dy}{dt} \tag{4}$$

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 Fig. 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:

$$f(\omega) \rightarrow \theta \coloneqq \int \omega dt \tag{5}$$

$$g(\theta, l_c, l_s) \rightarrow y \coloneqq l_c \cos\theta + \sqrt{l_s^2 - l_c^2 \sin^2\theta} \tag{6}$$

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 [21].

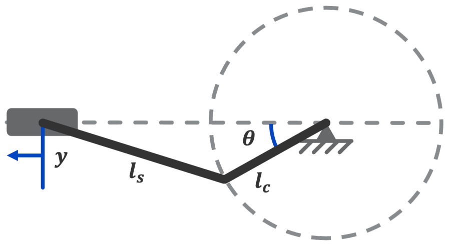


Fig. 2 Kinematic diagram of a slider-crank mechanism.

Though there are many mechanisms for constructing simulation processes [22], several modeling frameworks (or formalisms [23,24]) 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 [25]. 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 [26].

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 [27,28]. Equations (1–4) are declarative, with the simulation functions specified in Eqs. (5) and (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 1:

Block 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);
animatePiston(y, theta, l_c, timestep);

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 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 2, where the model connects an input of y to solve for θ:

Block 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);
fig = animatePiston(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 1 and 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:

$$\sum_{i=1}^{n-1}(n-i)\binom{n}{i} \tag{7}$$

The exponential growth rate of Eq. 7 relative to n, as shown in Fig. 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 [25].

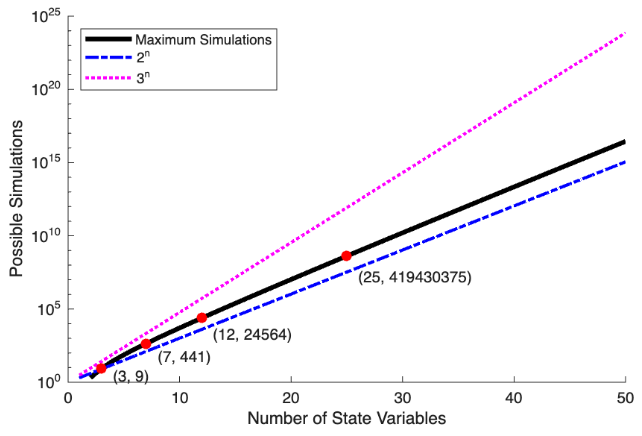


Fig. 3 Exponential growth of maximum input/output pairings in a system as a function of the number of system variables.

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 [29]. This paper builds upon this review by demonstrating CHGs, a specific declarative formalism introduced in [2].

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 [30,31], or categorical sheaves [32,33]. 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 [34]. A CHG for the kinematic model of the slider-crank given in Blocks 1 and 2 is shown in Fig. 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 representing 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 Fig. 5,

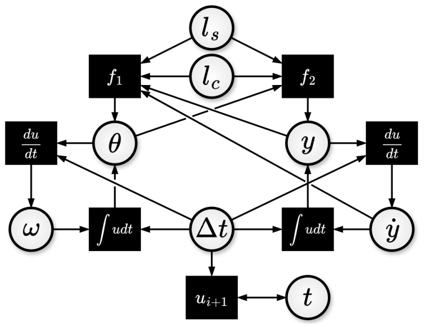


Fig. 4 CHG model of slider-crank kinematics, with f_1 and f_2 given by Eqs. (1) and (2).

depicting two different paths drawn through the CHG from Fig. 4 representing the simulation process given in Blocks 1 and 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 Fig. 6.

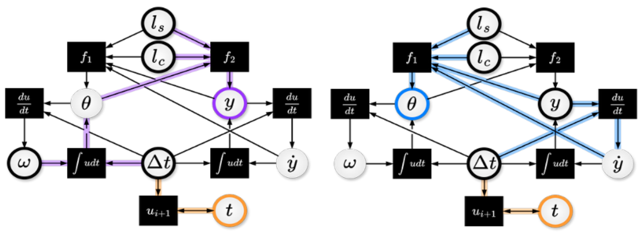


Fig. 5 CHG from Fig. 4 showing the two simulation processes imperatively given by Blocks 1 (left) and 2 (right) as paths through the graph connecting inputs (black, bolded outline) with outputs (colored, bolded outline).

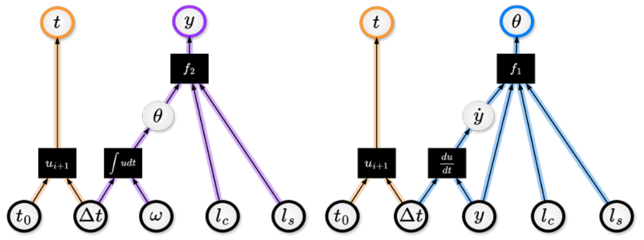


Fig. 6 Paths through the CHGs shown in Fig. 5 shown as trees, with a unique tree for each output.

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 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 [35], but cannot query a relational database management system (RDBMS). Modelica's solver similarly solves differential equations [36], but is not used to construct geometric models despite its goal of multi-domain system modeling [37]. SysML was intended as a general modeling language for all systems, but has struggled to be adopted for hybrid simulation [38,39]. 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 [40]. This is contrasted with CHGs, which are not specialized for any particular domain, but rather system simulation 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 [41]. 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.

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. 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 $A\colon B$ and $B\colon A$, and three hyperedges, e.g. $\{A, B\}\colon C$. All nine of these pairings can be discovered on a CHG with only three edges: $A \longrightarrow B$, $B \longrightarrow C$, and $C \longrightarrow A$. This is because of composition; for example, the simulation for $A\colon C$ can be computed as the path $A \longrightarrow B$ composed with $B \longrightarrow C$.

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 defined. 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.

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 [42]. Methods of breaking down silos and integrating models are often imperative [3,43,44], often indicated when coupling is described by a flowchart or workflow. Imperatively coupled models establish processes by which different software subsystems may exchange messages [45]. This is a hallmark of encapsulation, where a subsystem has a local state that is distinct and unaffected by the global program [46]. Encapsulated, imperatively-coupled simulations can still be highly useful–as evidenced by recent usage of the Functional-Mockup Interface [47,48]–yet lack the ability to fully simulate a system, as demonstrated in Section 2.2 and discussed further in [29].

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 [49]. Instead, each model is fitted to Modelica's interface, so that system inputs can be automatically passed to the declarative backend solver [7]. 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 Fig. 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 Fig. 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 Fig. 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 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 [50]) is one of the most challenging aspects of model-based engineering [51–53]. While CHGs do not answer every issue for interoperability, for instance, they do not provide mechanisms for syntactic interoperability [54], 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 Multi-Domain Modeling of Crankshaft

These claims are demonstrated through simulating a multi-physics model of a crankshaft with form as given in Figures 7 and 8, with parameters explained in Table 1. This model extends the simple kinematic system shown in Fig. 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. (1)–(4) still hold true, the addition of mass requires a more detailed model of the crankshaft.

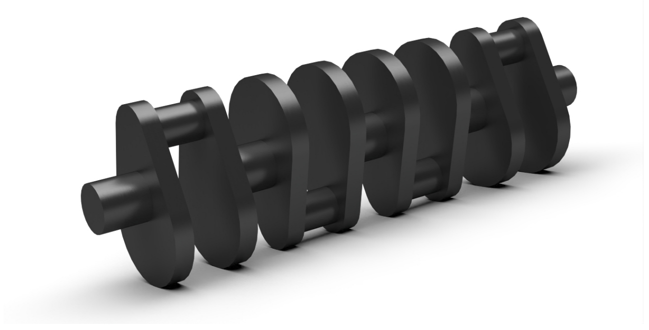


Fig. 7 Image of the modeled crankshaft.

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 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 platform. 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 Fig. 10, though only covering a portion of the graph due to the scale of the system. The development process follows that given in Fig. 4: represent each system variable as a node, then relate nodes to each other through multidimensional edges representing functions.

Table 1 Input parameters for crankshaft

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	kg 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

In addition to the diagrams shown in the section, the actual model is written in the Python programming language.[2] The declarative engine that solves the model is the open-source package ConstraintHg [55], 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.

4.2 Process. The process of forming a CHG, as described in Fig. 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 1. 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 Re-

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

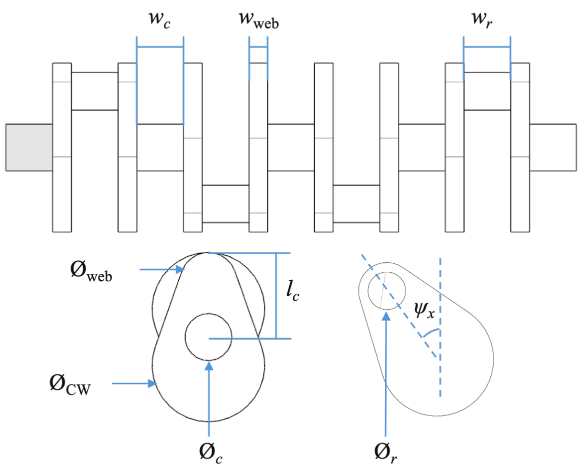


Fig. 8 Geometric parameters for the crankshaft, as described in Table 1, with the initial main bearing surface built in Fig. 10 shaded in gray.

source 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 [56]. A partial CHG with these nodes included is shown in Fig. 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 application offering an established API [57]. 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.

5 Results

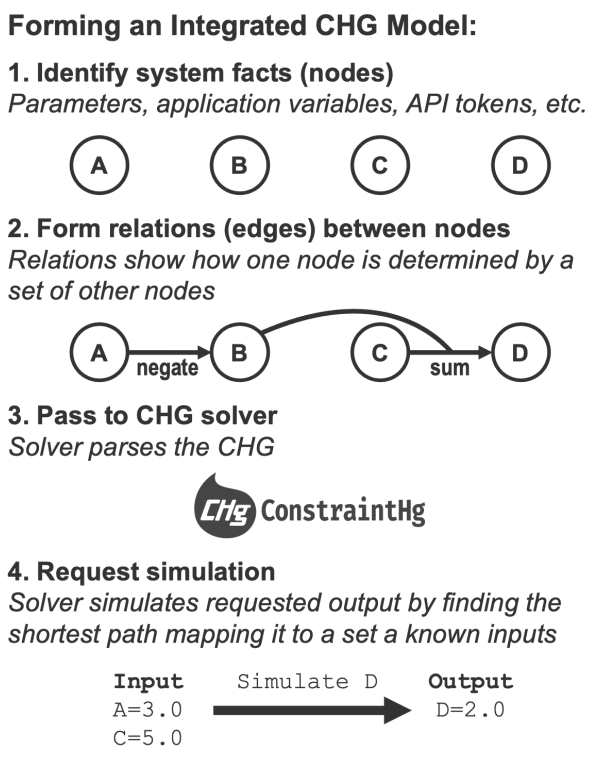
Once prepared, the CHG contains a full model of the system that can be immediately simulated. The user prepares a caller file such as the one shown in Block 3. This file does three primary things: first, it initiates the API clients of the used software applications; second, it declares the list of known 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 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
```



Forming an Integrated CHG Model:
1. Identify system facts (nodes)
Parameters, application variables, API tokens, etc.
A
B
C
D
2. Form relations (edges) between nodes
Relations show how one node is determined by a set of other nodes
A
negate
B
C
sum
D
3. Pass to CHG solver
Solver parses the CHG
CHg ConstraintHg
4. Request simulation
Solver simulates requested output by finding the shortest path mapping it to a set a known inputs
Input
A=3.0
C=5.0
Simulate D
Output
D=2.0

Fig. 9 Process of forming and simulating a CHG, incorporating the demonstration CHG solver *ConstraintHg* [55].

```
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 Fig. 10. The result of the execution call is saved to the target node of the pro-

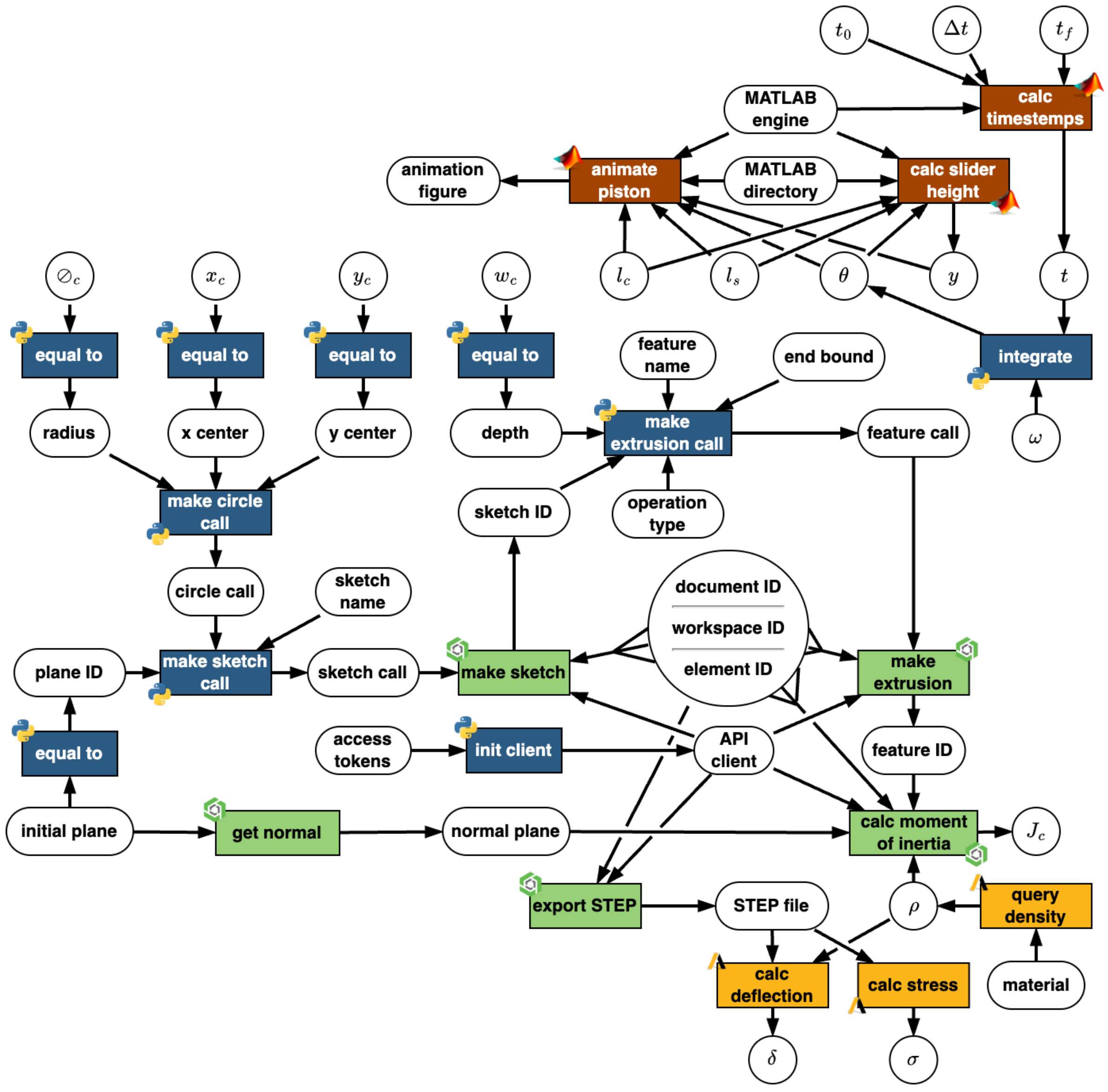


Fig. 10 A subset of the full CHG (reduced for brevity) showing FEA simulation in Ansys, kinematic animation in MATLAB, and solid modeling of the crankshaft's primary bearing(shown as shaded in Fig. 8) in Onshape. Basic connections and algebraic relationships are calculated in Python.

cessed 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.

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 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

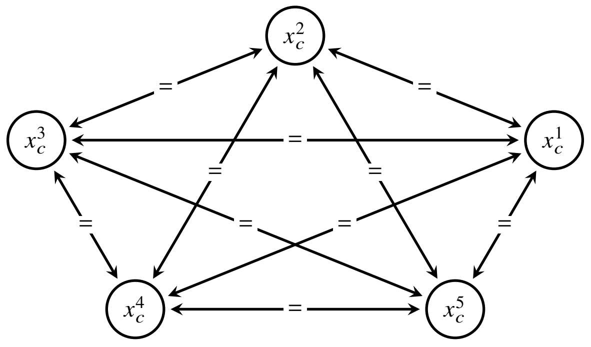


Fig. 11 Declarative representation for horizontal location of main journals on crankshaft as a CHG where all edges indicate equivalency.

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 Fig. 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 [58], 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 operations 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 [3], 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 [59]. The mechanisms for model composability can also drive distributed simulation, yielding benefits for execution time and resource management [18,59].

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.

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 [55]. 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.

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 [2] and how they engender declarative, functional modeling [29]. 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] Van Bossuyt, Douglas L., Allaire, Douglas, Bickford, Jason, Bozada, Thomas A., Chen, Wei, Cutitta, Roger P., Cuzner, Robert, Fletcher, Kristen, Giachetti, Ronald, Hale, Britta, Huang, H. Howie, Keidar, Michael, Layton, Astrid, Ledford, Allison, Lesse, Marina, Lussier, Jonathan, Malak, Richard, Mesmer, Bryan, Mocko, Gregory, Oriti, Giovanna, Selva, Daniel, Turner, Cameron, Watson, Michael, Wooley, Ana and Zeng, Zhen. "The Future of Digital Twin Research and Development." Vol. 25 No. 8 : p. 080801. doi: 10.1115/1.4068082.

[2] Morris, John, Mocko, Gregory and Wagner, John. "Unified System Modeling and Simulation via Constraint Hypergraphs." Vol. 25 No. 6 : p. 061005. doi: 10.1115/1.4068375.

[3] Montalbano, Andrew, Mocko, Gregory and Li, Gang. "Integration of Vehicle Dynamics with Finite Element Composite Structure Simulation." *Proceedings of the 2024 Ground Vehicle Systems Engineering and Technology Symposium (GVSETS)*. NDIA.

[4] International Organization for Standardization. "Systems and Software Engineering - System Life Cycle Processes." Accessed 2024-09-21, URL https://www.iso.org/standard/81702.html.

[5] Lee, Edward A. "Determinism." Vol. 20 No. 5 : pp. 38:1–38:34. doi: 10.1145/3453652.

[6] Boulding, Kenneth. "General Systems Theory: The Skeleton of Science." Vol. 2 No. 3 (Ap 1956): pp. 197–208. Accessed 2025-01-08, URL http://www.panarchy.org/boulding/systems.1956.html.

[7] Cellier, François E. *Continuous System Modeling*. Springer. doi: 10.1007/978-1-4757-3922-0.

[8] Zeigler, Bernard P. *Multifacetted Modelling and Discrete Event Simulation*. Academic Press.

[9] Blanchard, Benjamin S. and Fabrycky, W. J. *Systems Engineering and Analysis*, 3rd ed. Prentice Hall.

[10] Floridi, Luciano. *Information: A Very Short Introduction*. Very Short Introductions, Oxford University Press.

[11] Ashby, W. Ross. *An Introduction to Cybernetics*, internet ed. Chapman & Hall. Accessed 2025-02-26, URL http://pcp.vub.ac.be/books/IntroCyb.pdf.

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

[13] Willems, Jan C. "The Behavioral Approach to Open and Interconnected Systems." Vol. 27 No. 6 : pp. 46–99. doi: 10.1109/MCS.2007.906923.

[14] Spivak, David I. "Category Theory for Scientists." URL 1302.6946.

[15] Herstein, Israel N. *Topics in Algebra*, 1st ed. Blaisdell Publishing Company.

[16] Eisenbart, Boris, Gericke, Kilian and Blessing, Luciënne. "An Analysis of Functional Modeling Approaches Across Disciplines." Vol. 27 No. 3 : pp. 281–289. doi: 10.1017/S0890060413000280.

[17] Cellier, François E. and Kofman, Ernesto. *Continuous System Simulation*. Springer. doi: 10.1007/0-387-30260-3.

[18] Gomes, Cláudio, Thule, Casper, Broman, David, Larsen, Peter Gorm and Vangheluwe, Hans. "Co-Simulation: A Survey." Vol. 51 No. 3 : pp. 49:1–49:33. doi: 10.1145/3179993.

[19] Patterson, Evan, Spivak, David I. and Vagner, Dmitry. "Wiring Diagrams as Normal Forms for Computing in Symmetric Monoidal Categories." *Proceedings 3rd Annual International Applied Category Theory Conference 2020*, Vol. 333: pp. 49–64. Open Publishing Association. doi: 10.4204/EPTCS.333.4. URL 2101.12046.

[20] Baez, John C. and Pollard, Blake S. "A Compositional Framework for Reaction Networks." Vol. 29 No. 09 : p. 1750028. doi: 10.1142/S0129055X17500283.

[21] Roy, Peter Van and Haridi, Seif. *Concepts, Techniques, and Models of Computer Programming*. MIT Press. URL _bmyEnUnfTsC.

[22] Sinha, Rajarishi, Paredis, Christiaan J. J., Liang, Vei-Chung and Khosla, Pradeep K. "Modeling and Simulation Methods for Design of Engineering Systems." Vol. 1 No. 1 : pp. 84–91. doi: 10.1115/1.1344877.

[23] Zeigler, Bernard P., Muzy, Alexandre and Kofman, Ernesto. *Theory of Modeling and Simulation*, 3rd ed. Elsevier. doi: 10.1016/B978-0-12-813370-5.00002-X.

[24] Tiller, Michael. *Introduction to Physical Modeling with Modelica*. No. SECS 615 in *The Kluwer International Series in Engineering and Computer Science*, Kluwer Academic Publishers.

[25] Abelson, Harold and Sussman, Gerald Jay. *Structure and Interpretation of Computer Programs*, 17th ed. The MIT Electrical Engineering and Computer Science Series, MIT Pr. [u.a.].

[26] Hudak, Paul. "Conception, Evolution, and Application of Functional Programming Languages." Vol. 21 No. 3 : pp. 359–411. doi: 10.1145/72551.72554.

[27] Reade, Chris. *Elements of Functional Programming*. International Computer Science Series, Addison-Wesley.

[28] Mitchell, John C. *Concepts in Programming Languages*. Cambridge University Press.

[29] Morris, John, Mocko, Gregory and Wagner, John. "Effects of Functional and Declarative Modeling Frameworks on System Simulation." .

[30] Friedman, George J. and Leondes, Cornelius T. “Constraint Theory, Part I: Fundamentals.” Vol. 5 No. 1 : pp. 48–56. doi: 10.1109/TSSC.1969.300244.

[31] Friedman, George J. and Phan, Phan. *Constraint Theory*. Vol. 23 of *IFSR International Series on Systems Science and Engineering*. Springer International Publishing. doi: 10.1007/978-3-319-54792-3.

[32] Schultz, Patrick, Spivak, David I. and Vasilakopoulou, Christina. “Dynamical Systems and Sheaves.” URL 1609.08086.

[33] Zardini, Gioele, Spivak, David I., Censi, Andrea and Frazzoli, Emilio. “A Compositional Sheaf-Theoretic Framework for Event-Based Systems (Extended Version).” Vol. 333 : pp. 139–153. doi: 10.4204/EPTCS.333.10. URL 2005.04715.

[34] Kapteyn, Michael G., Pretorius, Jacob V. R. and Willcox, Karen E. “A Probabilistic Graphical Model Foundation for Enabling Predictive Digital Twins at Scale.” Vol. 1 No. 5 : pp. 337–347. doi: 10.1038/s43588-021-00069-0.

[35] Breedveld, P.C. “Concept-Oriented Modeling of Dynamic Behavior.” Borutzky, Wolfgang (ed.). *Bond Graph Modelling of Engineering Systems: Theory, Applications and Software Support*. Springer. doi: 10.1007/978-1-4419-9368-7.

[36] Mattsson, Sven Erik, Elmqvist, Hilding and Otter, Martin. “Physical System Modeling with Modelica.” Vol. 6 No. 4 : pp. 501–510. doi: 10.1016/S0967-0661(98)00047-1.

[37] Otter, Martin and Elmqvist, Hilding. “Modelica.” Vol. 10 No. 29/30 : pp. 3–8. Accessed 2025-02-25, URL https://www.sne-journal.org/fileadmin/user_upload_sne/SNE_Issues_OA/SNE_10/sne.10.29-30.pdf.

[38] Wolny, Sabine, Mazak, Alexandra, Carpella, Christine, Geist, Verena and Wimmer, Manuel. “Thirteen Years of SysML: A Systematic Mapping Study.” Vol. 19 No. 1 : pp. 111–169. doi: 10.1007/s10270-019-00735-y.

[39] Xinquan, Wu, Xuefeng, Yan, Xingchan, Li and Yongzhen, Wang. “Simulating Hybrid SysML Models: A Model Transformation Approach under the DEVS Framework.” Vol. 79 No. 2 : pp. 2010–2030. doi: 10.1007/s11227-022-04654-6.

[40] Åström, Karl Johan, Elmqvist, Hilding and Mattsson, Sven Erik. “Evolution of Continuous-Time Modeling and Simulation.”

[41] Gaines, Brian R. “General Systems Research: Quo Vadis?” Vol. 24 : pp. 1–9.

[42] Quek, Hou Yee, Hofmeister, Markus, Rihm, Simon D., Yan, Jingya, Lai, Jiawei, Brownbridge, George, Hillman, Michael, Mosbach, Sebastian, Ang, Wilson, Tsai, Yi-Kai, Tran, Dan N., Tan, Soon Kang, William and Kraft, Markus. “Dynamic Knowledge Graph Applications for Augmented Built Environments Through “The World Avatar”.” Vol. 91 : p. 109507. doi: 10.1016/j.jobe.2024.109507.

[43] Margara, Alessandro, Pezzè, Mauro, Pivkin, Igor V. and Santoro, Mauro. “Towards an Engineering Methodology for Multi-Model Scientific Simulations.” *2015 IEEE/ACM 1st International Workshop on Software Engineering for High Performance Computing in Science*: pp. 51–55. doi: 10.1109/SE4HPCS.2015.15.

[44] Cantot, Pascal. *Simulation and Modeling of Systems of Systems*. John Wiley & Sons.

[45] Pinson, Lewis J. and Wiener, Richard S. *An Introduction to Object-Oriented Programming and Smalltalk*. Addison-Wesley.

[46] Wegner, Peter. “Concepts and Paradigms of Object-Oriented Programming.” Vol. 1 No. 1 : pp. 7–87. doi: 10.1145/382192.383004.

[47] Blochwitz, T, Otter, M, Arnold, M, Bausch, C, Clauß, C, Elmqvist, H, Junghanns, A, Mauss, J, Monteiro, M, Neidhold, T, Neumerkel, D, Olsson, H, Peetz, J-v and Wolf, S. “The Functional Mockup Interface for Tool Independent Exchange of Simulation Models.” *Proceedings 8th Modelica Conference*. Accessed 2022-01-13, URL http://www.functional-mockup-interface.org.

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

[49] Fritzson, Peter. *Introduction to Modeling and Simulation of Technical and Physical Systems with Modelica*. IEEE Press. doi: 10.1002/9781118094259.

[50] Wagg, D. J., Worden, K., Barthorpe, R. J. and Gardner, P. “Digital Twins: State-of-the-Art and Future Directions for Modeling and Simulation in Engineering Dynamics Applications [Special Section].” Vol. 6 No. 3 : pp. 030901:1–030901:18. doi: 10.1115/1.4046739.

[51] Dolk, Daniel R. and Kottemann, Jeffrey E. “Model Integration and a Theory of Models.” Vol. 9 No. 1 : pp. 51–63. doi: 10.1016/0167-9236(93)90022-U.

[52] National Academies of Sciences, Engineering, and Medicine. *Foundational Research Gaps and Future Directions for Digital Twins*. The National Academies Press. doi: 10.17226/26894.

[53] INCOSE. “Systems Engineering Vision 2035.” Accessed 2025-02-18, URL https://www.incose.org/2021-redesign/load-test/systems-engineering-vision-2035.

[54] for Standardization, International Organization. “Information Technology-Cloud Computing-Interoperability and Portability.” doi: 10.3403/30313036U.

[55] Morris, John. “ConstraintHg.” doi: 10.5281/zenodo.15278018.

[56] Gordon, Andrew D. *Functional Programming and Input/Output*, [repr. der ausg.] 1994 ed. Distinguished Dissertations in Computer Science, Cambridge Univ. Press.

[57] “Glassworks API.” Accessed 2025-03-15, URL https://cad.onshape.com/glassworks/explorer.

[58] Nandi, Chandrakana, Wilcox, James R., Panchekha, Pavel, Blau, Taylor, Grossman, Dan and Tatlock, Zachary. “Functional Programming for Compiling and Decompiling Computer-Aided Design.” Vol. 2 : pp. 1–31. doi: 10.1145/3236794.

[59] Taylor, Simon J. E. “Distributed Simulation: State-of-the-Art and Potential for Operational Research.” Vol. 273 No. 1 : pp. 1–19. doi: 10.1016/j.ejor.2018.04.032.