Effects of Functional and Declarative Modeling Frameworks on System Simulation

John Morris
CEDAR Laboratory
Department of Mechanical Engineering,
Clemson University,
Clemson, SC USA
email: jhmrrs@clemson.edu

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

John Wagner, P.E.[1]
Department of Mechanical Engineering,
Clemson University,
Clemson, SC USA
email: jwagner@clemson.edu

System modeling frameworks can be categorized into imperative and declarative paradigms. Which paradigm a model is expressed in effects how the model may be utilized; 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.

Keywords: declarative modeling, functional programming, system simulation, model-based systems engineering

1 Introduction

Modelers, trying to represent the behavior of a complex system, are limited by the expressiveness of the available frameworks [1]. 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 [2]. 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.

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 [3]. 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, with a key observation being that heterogeneous systems models may benefit from being represented in functional, declarative frameworks rather than with procedural or procedurally-coupled object-oriented approaches. This is based on 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 shown to be 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 the functional modeling paradigm discussed in this paper is aligned with practices of computer science, such as described by [4] and [5], 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 5). Functional modeling as used here does not refer to the deconstruction of a system into its *functionalities* [6], a common method espoused by Pahl and Beitz [7].

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 [8].

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 Fig. 1(*a*) or the SysML block definition diagram in Fig. 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 [9]. Ross Ashby was the first to conceive as a system as a list of variables, whose values are generally considered the state of a system [10]. Simulation is the prediction of a system's state–or at least a part of it–through reasoning on the behavior of the system [11]. 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 [12]. Decisions about a system require simulation, especially during the design phase when the systems of interest are not yet physically realized [13].

[1]Corresponding Author.
Version 1.0, March 7, 2025

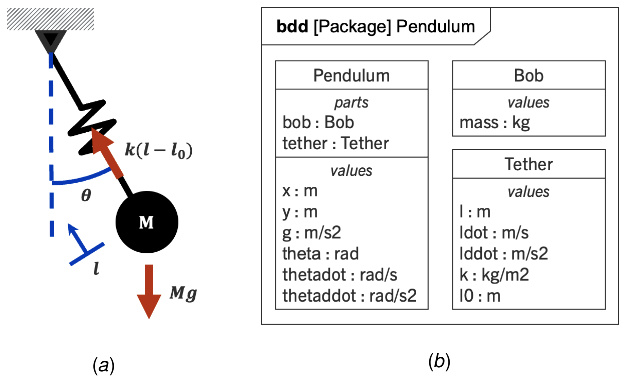


Fig. 1 Two descriptive models of an elastic pendulum: *(a)* a free body diagram showing the forces of the system; and *(b)* a SysML block definition diagram showing the system elements.

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 [14,15]. The morphism for mapping between sets of values is an algebraic function [16], 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 [17]. 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 F from the underlying model [18]. Some models express F explicitly, while others provide F by the composition of sub-functions $F = f_n \circ f_{n-1} \circ \ldots \circ f_2 \circ f_1$, 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.

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 [19]. Models are developed to capture the combinatorial interactions between variables [20]. A model is considered underconstrained 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 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 [21].

3 Modeling Paradigms

The variety of ways to represent and compose functions results in a plurality of modeling languages [22], such as the examples given in Table 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 [23]. The rules, which describe how system components may be combined [24], are collectively referred to as a framework [4] or a formalism [9]. 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

Table 1 Examples of system modeling frameworks, adapted from [26]

Framework	Domain	Objective	Source
Bond graphs	Dynamic	Description	[27,28]
Block diagrams	Dynamic	Simulation	[29,30]
Stock and Flow	Dynamic	Simulation	[31,32]
Entity-Relationship	Knowledge	Simulation	[33]
Circuit diagrams	Electronics	Description	[15]
Flow charts	Processes	Description	[34]
Petri nets	Processes	Description	[14]
State machines	Processes	Simulation	[35]
Markov chains	Processes	Simulation	[36]
Gantt charts	Scheduling	Description	[37]
PERT diagrams	Scheduling	Simulation	[38]
Bayesian networks	Stochastic	Simulation	[39,40]
Causal models	Multi-domain	Description	[41]
SysML diagrams	Multi-domain	Description	[42]
EXPRESS	Multi-domain	Description	[43]
Algebraic Expressions	Multi-domain	Description	
Constraint graphs	Multi-domain	Simulation	[26,44–46]

Table 2 Informal categorization of modeling frameworks as imperative or declarative, and further distinguished by typical base data representation.

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

replaced without affecting the underlying interpretation. Indeed, it can be said that the framework rules provide the interpretation of a model [25]. 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 simulatability. Two primary paradigms of providing a simulation function F are considered: imperative and declarative frameworks in Sections 3.1 and 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 3.4), while models that express each relationship as a function are functional (Section 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 2. Note that the categorizations of modeling frameworks as imperative of declarative, as well as the degree of objectification, are provided for demonstrative purposes as such designations are fairly subjective.

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 Fig. 2(*a*). 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 $F := \{\text{LA}\} \to \{\text{Seattle}\}$ is explicitly given in the model as $f_{\text{LA}\to\text{SEA}}$, however $F := \{\text{LA}\} \to \{\text{NYC}\}$ must be composed from other functions in the model: $f_{\text{CHI}\to\text{NYC}} \circ f_{\text{DEN}\to\text{CHI}} \circ f_{\text{LA}\to\text{DEN}}$.

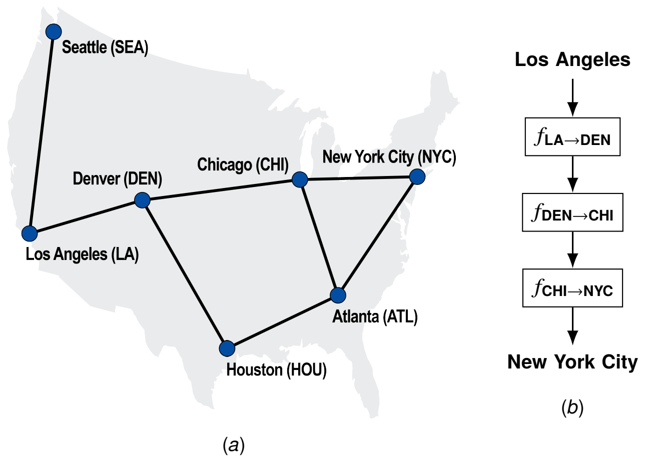


Fig. 2 Two analogies of modeling: (*a*), a declarative model expressing every possible route (simulation) that can be connected between the cities (system variables); and (*b*), an imperative model showing the steps necessary to travel from Los Angeles (the input) to New York City (the output).

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 Fig. 2(*b*). 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 system.

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 [24]. Flowcharts are an example of an imperative model where the executor is not a computer, such as the flowchart in Fig. 3 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 [47].

In prescribing the steps of a simulation, these models are immediately executable. An imperative model defines how a simulation should be calculated [48], a necessary provision before executing a process whether by a computer [49] or biological engine [50]. 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 behavior for the specific fact they are purposed to simulate. Their simplicity, however, belies the difficulty in understanding the systems they communicate [51]. A process has no intrinsic points where it can be broken or reconfigured. Imperative models are consequently more difficult to connect and adapt [30].

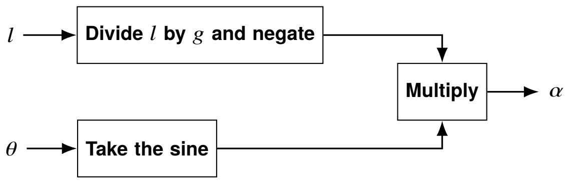


Fig. 3 A flowchart showing how to calculate the angular acceleration α of a pendulum based on inputs of tether length l and angular position θ.

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 [52].

3.2 Declarative Modeling. While an imperative model describes the steps needed to calculate a single output, declarative models describe all possible values that could be obtained from a simulation. Though any process that abstracts lower-level instructions may be thought of as declarative [49], the true test for a declarative language is whether the values of state variables remain the same throughout the model [48]. This is because a declarative model 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 Fig. 2(*a*) 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 [53], leaving the work of routing the lower-level process to an execution agent [47]. This is evident in SysML, where constraints in a parametric diagram must be solved by an external solver [54,55].

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 [48].

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 [24]) or proprietary compilers (such as with Modelica [56]). Functional languages expose information by expressing each declared facts as a function, so that a compiler can chain functions together using function composition [57,58]. The rigor and power of this method has made functional programming nearly synonymous with declarative modeling, though the two are decidedly distinct [48].

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 the those within the functional modeling paradigm. Functional models represent a system as a set of basic functions with behavior defined by composing functions [59]. 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 [57]. Such functionality is sometimes referred to as composability [13,60,61], modularity [57,62], extensibility [53,63], or interoperability [64]. The rigor of function composition, supported by Church's lambda calculi [65], drove the development of functional languages such as LISP, Haskell, and Miranda [66].

Using function composition as the ”glue” for bringing elements together brings significant benefits for modeling systems [57]. It was shown in Section 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 [67]. In other words, a function describes the affect of the system on a single variable, such as an output desired in a simulation [5].

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 [24]. Every object represents an independent system composed of elements that exhibit collective behavior. Objects in a model consequently contain elements (often variables) and manifest behaviors encoded in a set of processes (usually 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 [48]. The tools for defining a system are provided by functional programming, consequently most object definitions adhere to the functional modeling paradigm [68]. The difference between the two paradigms is in consideration of how subsystems interact. While a functional model treats all elements as available to be connected (a flat representation), an object-oriented paradigm encapsulates certain portions of the system. Encapsulated objects (subsystems) have their own state distinct from the state of the global system [68]. The global system behavior is then described by coupling subsystems along predefined ports [13]. The resulting inter-subsystem coupling is often (though not always [17]) conducted imperatively [69], resulting in sequences of tasks sent between objects [70]. The SysML model shown in Fig. 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 [55].

4 Effects on Simulation

Though simulation can be performed with models expressed in any of the discussed paradigms, there is a clear advantage for frameworks that are both functional and declarative. The authors claim that these models are more connective, like the systems they describe, allowing them to better capture system behavior. This claim is supported by answering why connective frameworks are necessary for system simulation. The remainder of the section investigates why imperative and non-functional paradigms (including object-oriented frameworks) are limited in their interoperability. These takeaways of this section are summarized in Table 3.

4.1 Need for Interoperability in Simulation. All systems change [71]. The philosophical argument that the state of reality, and our interpretation of it, is constantly changing is at least as old as Heraclitus. The evolutions of a system of interest is ultimately due to reality's reluctance to be confined by a closed-world definition. In contrast with a virtual pendulum existing in a vacuum, a real pendulum might be influenced by the day's weather, vibrations on the floor above, the material of its tether, etc. To represent the pendulum, a modeler selects a subset of factors based by the model's intended use [10]. Every combination of phenomena may yield a unique system representation expressing different aspects of the modeled system–perhaps one scope associated with the dynamics of the pendulum, while another considers the bacterial on the bob. Because no finite model can every claim to fully represent a system, the best case for a general model is one that is adaptable to changes in its scope. Here the modeler can retain the behavior between various scope definitions, such as using the equations of motion to see how the bob's speed effects bacterial growth.

The evolving nature of a model's scope constitutes the primary challenge of modeling systems: model interoperability [72,73]. The traditional definition of interoperability is a system's ability to exchange information with an interfacing agent without loss of meaning [74]. When considering simulation, the exchanged information becomes the facts–or state variables–considered in the system model, while the meaning is given by the inter-variable relationships. This deconstructionist interpretation of interoperability is restated by the authors as the consistent expression of system facts despite modification of a model's scope.

A change in scope may involve modifying the state variables, their relationships, or both. The first case is often brought about when a modeler considers additional phenomena, such as expanding the model of the simple pendulum to include friction. Accounting for a change in state might result in the development of an entirely new model, a reaction that views the modified system as distinct from the original. The second case involves when the interpretation of the system's behavior changes, such as considering the motion of a pendulum to be linear (the classic small-angle interpretation). Behavioral changes might also occur when discovering new interactions between parameters, such as updating a model of a Foucault pendulum to consider the rotation of the earth. If a system interfaces with another system (the traditional view of interoperability), then it is most likely that both of these cases occur, in that the states of the interfacing system are coupled in some way with the states of the original.

A model that is not interoperable is useful in only a single use case. It is one of the primary objectives of a modeler to create models that may adapt to multiple use cases [57]. Consequently, for system modelers, model frameworks should be principally evaluated across their ability to accurately demonstrate system facts even when their scopes are modified in the previously discussed ways. While the author's leave discussions of accuracy and fidelity to other works, considering how scope evolutions are handled motivates this paper's example on the four modeling paradigms in Table 2.

4.2 Limitations of Imperative Models. The least interoperable models are those constructed in an imperative framework, such as the flowchart shown in Fig. 3. 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 for 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, nor what the effect of breaking them to extract system information. 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 [14].

Functional imperative models, such as the block diagram shown in Fig. 6, 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 [30]. 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.

4.3 Limitations of 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 [75]. This allows (indeed, requires) interfaces to be built by which objects can interact with each other [70]. Object interfaces enable greater composability in the same way a handle simplifies interactions with a door.

The pitfall of encapsulated models is when the system does not adhere to the object-oriented worldview. While it is natural to decompose a system into distinct entities [76], such distinctions are inevitably arbitrary. While simple systems are readily classified, it is often impossible to abstract systems that interact with other systems across many dimensions (often termed reactive systems)[77]. 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 [78], what Nielsen et al. call the ”synergistic collaboration of constituents” [64].

Objects that are not required to maintain a separate sense of state are an exception to this, allowing 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 [24,48,68–70]. 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 Fig. 4. 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 [9,17].

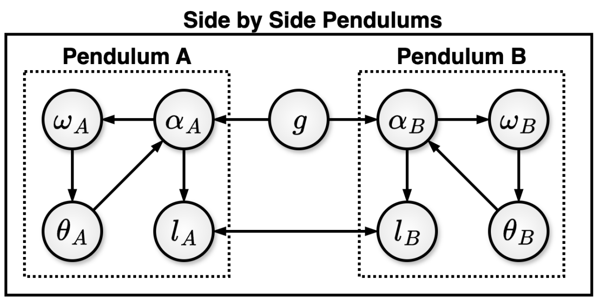


Fig. 4 A system representation of two pendulums hung side by side (but not coupled). The variables associated with each pendulum are shown in the dashed boxes, while inter-variable algebraic relationships are shown with solid arrows.

To be truly object-oriented, subsystems must 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 an object-oriented paradigm (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 this limits the expressiveness of these languages such as inheritance schemes.

Encapsulated objects are not without their benefits in modeling. The rigid, hierarchical structures [79] 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) [80], 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 [81,82].

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 [3]. 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” [83], which showed the limits of procedurally connected objects to reflect changes in behavior [84]. Though solutions to this problem exist [85], the underlying philosophy of independent subsystems clearly is at odds with the idea of emergent, holistic behaviors prevalent in systems theory.

5 Study of Simulation Methods by Paradigm

To demonstrate these differences between different paradigms, two simulation case studies are conducted in five modeling frameworks. The system for each case is a double pendulum 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 Fig. 5. This scope of this system consists of the following variables and functions:

Variables, with values given for constants:

- $g = 9.81\ \mathrm{m/s^2}$, the gravitational acceleration;
- l_A, $l_B = 1$ m, the length of the pendulums' tethers;
- 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 (s), the time of the current frame of consideration.

Relationships:

$$f_{\alpha_X} : \{\omega_{X,i}, \omega_{X,i-1}, \Delta t\} \rightarrow \alpha_A = \frac{\omega_{X,i} - \omega_{X,i-1}}{\Delta t} \tag{1}$$

$$\begin{aligned} f'_{\alpha_B} &: \{\theta_A, \theta_B, \omega_A, \alpha_A, l_A, l_B, g\} \rightarrow \alpha_B = \\ &-\frac{1}{l_B}\left(\ddot{x}\cos\theta_B + \sin\theta_B\,(\ddot{y} + g)\right) \\ &\text{where:} \\ &\ddot{x} = l_A\left(\alpha_A\cos\theta_A - \omega_A^2\sin\theta_A\right) \text{ and} \\ &\ddot{y} = l_A\left(\alpha_A\sin\theta_A + \omega_A^2\cos\theta_A\right) \end{aligned} \tag{2}$$

$$f_{\omega_X} : \{\alpha_X, \omega_X, \Delta t\} \rightarrow \omega_X = \alpha_X * \Delta t + \omega_X \tag{3}$$

Table 3 Overview of simulation in various modeling paradigms.

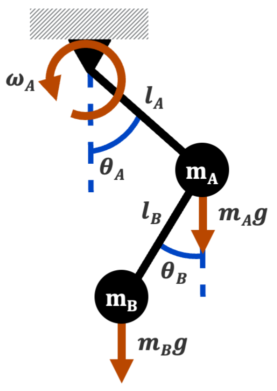
Paradigm	Definition	Strengths	Weaknesses
Imperative	Defines the specific process transforming inputs to outputs	Simple to develop	Manual simulation processes for each input/output
Declarative	Defines the relationships which are processed by a solver to compose the simulation	General simulation of a system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

$$f'_{\omega_A} : \{t\} \to \omega_A = \begin{cases} -\frac{\pi}{4} & \text{if } t \ \% \ 4 < 2 \\ \frac{\pi}{4} & \text{otherwise} \end{cases} \tag{4}$$

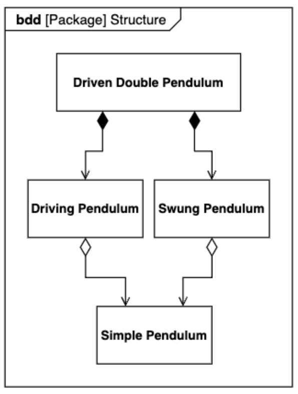
$$f_{\theta_X} : \{\omega_X, \theta_X, \Delta t\} \to \theta_X = \omega_X * \Delta t + \theta_X \tag{5}$$

$$f_t : \{t, \Delta t\} \to t = \Delta t + t \tag{6}$$

Equations 3 and 5 are a first-order Eulerian integration of α and ω to yield ω and θ respectively for each pendulum, while Eq. 1 is a first-order differentiation of α in terms of ω. While none of these are very robust, there simple relationships help show the functional nature of simulation without relying on proprietary numerical recipes. In addition to these differential terms, there are two other functions that result from the interaction of the driving pendulum with the freely swinging one: Eq. 2 and 4.



(a)



(b)

Fig. 5 Free-body diagram of a driven double pendulum, where the upper bob rotates around the fixed point at a fixed angular velocity shown as (a) a free body diagram; and (b) a SysML block definition diagram showing the class deconstruction.

The two different cases each simulate a different fact for the pendulum system:

1. θ_B at $t = 2\Delta t$;
2. $\theta_A + \theta_B$ at $t = 4.0$;

Each of these have the same initial inputs of with initial inputs of $\theta_A, \theta_B = \frac{\pi}{4}$, and $\alpha_A, \omega_A, \omega_B = 0$, in addition to the constants defined above. Models have been prepared in five different languages for the two cases. Although only featured snippets of the models are included here, all programs are provided in full in an open-source repository.[2]

Imperative, Non-functional: MATLAB
Imperative, Functional: Block diagrams (prepared in Simulink)
Imperative, Object-Oriented: C++
Declarative, Object-Oriented: Modelica
Declarative, Functional: Constraint hypergraphs

[2]Full scripts for all simulations are available on GitHub

The latter of these is a graphical method for representing and simulating systems detailed in [26]. It's use here is justified by its descriptive way of detailing system behavior. An abbreviated overview of the framework is that each state variable is represented as a node connected to each other by multidimensional hyperedges. Hyperedges represent the functions (shown as black boxes) in the system, connecting each variable in the function's domain to a single variable in the function's codomain. The declarative solver for the hypergraph is a pathfinding algorithm that constructs a path from nodes whose value is known (inputs) to another node whose value is unknown (the output). The resulting route through the graph is a chain of composed functions that, when calculated, simulates the value of the output. This can be more intuitively thought of as a graph of all possible block diagrams expressible in a system. The constraint hypergraphs are depicted visually, with simulations conducted using the Python package ConstraintHg [86], which serves as the underlying declarative solver for the framework.

The first case (1) is a simple demonstration of how simulation is based on function composition regardless of the implementing framework. Algebraically, the simulation of the θ_B (for the value following the initial input at $t_1 = \Delta t$) must be given by the following functions:

$$\theta_B = f_{\theta_B} \circ f_{\omega_B} \circ f'_{\alpha B} \tag{7}$$

To be simulatable, each framework must have some mechanism for exposing the composed functions. The most basic way to do this is by providing the function chain as an explicit process, such as in the imperative script (written in MATLAB):

Block 1: Imperative model simulating case 1 in MATLAB.

```
% ... Initialized variables

time = time + time_step;
xddot = lA * (alphaA * cos(thetaA) - omegaA^2 * sin(thetaA));
yddot = lA * (alphaA * sin(thetaA) + omegaA^2 * cos(thetaA));
alphaB = (xddot * cos(thetaB) + sin(thetaB) * (yddot + g)) / -lB;
omegaB(2) = alphaB * time_step + omegaB;
thetaB(2) = omegaB(2) * time_step + thetaB;

disp(thetaB(2));
```

This script is not easy to rearrange due to the constant state mutations. Is it possible to put the third line before the first? Can `thetaB` 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. A functional paradigm, such as the block diagram shown in Fig. 6, remedies this by expressing the functions generating the process. The block diagram 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.

Both 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. 7. Each unique process requires a corresponding imperative model to be configured. These procedural approaches are contrasted with the declarative constraint hypergraph in Fig. 7. The figure shows the model of the entire system, though it

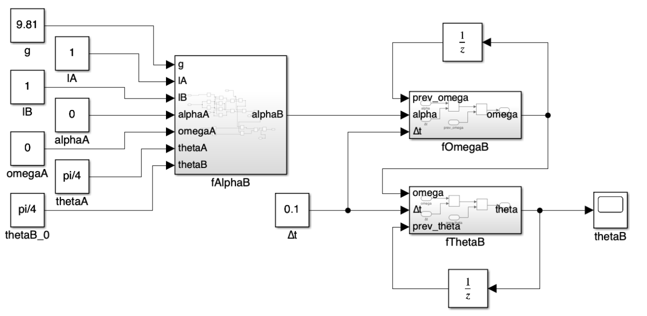


Fig. 6 Imperative, functional model simulating case 1 in a block diagram, implemented in Simulink.

would still be solvable if only the functions of Eq. 7 were included. Rather than specifying an explicit process, the hypergraph represents all possible function composition chains. The actual simulation is prepared by a pathfinding algorithm that searches for the shortest possible route connecting the provided pairing of inputs and outputs for case 1. This automatically composes the functions of Eq. 7, allowing the simulation to be executed automatically.

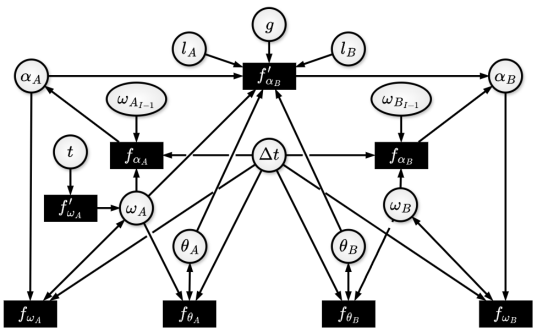


Fig. 7 Declarative, functional model of the driven double pendulum in a constraint hypergraph.

The same is mostly true for an equation solver such as Modelica. Modelica, based originally on Dymola [87], 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 [88], providing a convenient platform for dynamic modeling. Models are generally provided in classes, which allow for convenient model reuse and modularization. The driven pendulum system has been broken down in Block 2 into three objects: one representing a simple pendulum, which is inherited by two pendulums representing the top (`DrivingPendulum`) and bottom (`SwungPendulum`) respectively. These latter two are eventually inherited in a super class (`DrivenPendulum`) that models the coupling between the two subsystems.

Block 2: 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;
```

As an equation solver, Modelica models must be fully constrained before a solution can be found (unlike the imperative process in Block 1, which only required the specific functions of Eq. 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 control the solution method, 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 2. 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 [17]. 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 that encapsulation makes in a model is shown in C++, an imperative language shown here simulating case 2. This case requires a full simulation of the system, for multiple time steps. The function composition for a single time step is given by the following equation:

$$f_{\theta_A} \circ f'_{\omega_A} + f_{\theta_B} \circ f_{\omega_B} \circ f'_{\alpha_B} \tag{8}$$

The system is first broken down into a set of objects similar to the deconstruction in Modelica given in Block 2 and visually described in Fig. 5(*b*), here printed as a class declaration (a header) to avoid redundancy:

Block 3: Object-oriented models for both cases in C++

```
class Pendulum {
  double theta, omega, alpha;
  double l, g, m, step;
  double f_alpha();
  double f_omega();
  double f_theta();
};

class SwungPendulum: public Pendulum {
  using Pendulum :: Pendulum;

  double f_alpha(double thetaA, double omegaA, double alphaA, double
        lA);
};

class DrivenPendulum: public Pendulum {
  double speed = PI / 4;

  using Pendulum :: Pendulum;
  double f_alpha(double prev_omega);
  double f_omega(double t);
};
```

Because C++ is a procedural language, the simulation of case 2 must be programmed explicitly. There are two ways to find these functions using objects in C++. The first is to connect the objects declaratively as was done in Modelica, merging them into a super class and preserving a single sense of state and avoiding encapsulation. This is shown in Block 4, again with most functions only instantiated rather than fully defined for brevity.

Block 4: Declarative coupling of objects simulating case 2 in C++

```
class DrivenPendulum {
  DrivingPendulum driver;
  SwungPendulum swung;
  double prev_omega;
  double omega;

  void driver_f_alpha();
  void driver_f_omega(double t);
  void swung_f_alpha();

  double sim_theta(double t) {
    driver_f_omega(t);
    driver_f_alpha();
    driver.f_theta();
    swung_f_alpha();
    swung.f_omega();
    swung.f_theta();
    return driver.theta + swung.theta;
  }
};

int main() {
  DrivenPendulum system(PI/4, PI/4);
  vector<double> time, theta_sum;
  time.push_back(0.0);
  double end_time = 4.0;

  while (time.back() < end_time) {
    time.push_back(time.back() + system.driver.step);
    theta_sum.push_back(system.sim_theta(time.back()));
  }

  cout << theta_sum.back();
}
```

The explicit simulation process in given in the function `sim_theta`. This function is in turn composed of other functions given in the `DrivenPendulum` class. Other simulations could be similar defined by composing the functions in alternative sequences, though this needs to be done manually by the modeler for each simulation process. Alternatively, the base classes `SwungPendulum` and `DrivingPendulum` can be connected procedurally by coupling them in the `main` function which acts as the global caller. By moving the coupling out of the class definition the objects become encapsulated, as shown in Block 5. The process becomes difficult to restructure, versus the rearrangable functions shown in the coupling of Block 4.

Block 5: Imperative coupling of objects simulating case 2 in C++

```
int main() {
  // Imperative connection
  DrivingPendulum driver(PI/4);
  SwungPendulum swung(PI/4);
  vector<double> thetaA, thetaB, omegaA, time;
  time.push_back(0.0);
  omegaA.push_back(0.0);
  double end_time = 4.0;

  while (time.back() < end_time) {
    time.push_back(time.back() + driver.step);
    omegaA.push_back(driver.f_omega(time.back()));
    driver.f_alpha(omegaA[omegaA.size() - 2]);
    thetaA.push_back(driver.f_theta());
    swung.f_alpha(driver.theta, driver.omega, driver.alpha,driver.l);
    swung.f_omega();
    thetaB.push_back(swung.f_theta());
  }

  double theta_sum = thetaB.back() + thetaA.back();
  cout << theta_sum;
}
```

6 Discussion

The pendulum example above was specifically tailored to highlight the differences with simulation between paradigms, as tabulated in Table 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.

Object-oriented frameworks have been lauded as the cure to system interoperability [72], and certainly the use of encapsulated objects increases program modularity [69]. However, it is the opinion of the authors that the types of systems being modeled can significantly impact this. Software systems, which must exist on a variety of different platforms, benefit from static, modular models that can be used in many unique environments. As the most general way to connect objects, imperative couplings have become the default method for structuring object-oriented models. However, a scientist or engineer attempting to capture the behavior of a system should be wary of attempting to define static, independent subsystems, as they will likely fail to observe the collective behavior emerging from the interactions between system entities. This becomes especially apparent with data-hiding, which is promoted in software development to help prevent inadvertent tampering with a system model [89]. However, for more general systems, data hiding should be discouraged so that all interactions between data can be modeled.

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.

6.1 Future Work.

The expression problem discussed in Section 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 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 [73,90] and model-based systems engineering [2].

7 Conclusion

This paper is exploratory in nature, laying the groundwork for how the framework a model is developed under influences its utilization, especially regarding 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] Beeson, M. J., 1988, "Towards a Computation System Based on Set Theory," Theoretical Computer Science, **60**(3), pp. 297–340.

[2] INCOSE, 2021, "Systems Engineering Vision 2035," accessed 2025-02-18, https://www.incose.org/2021-redesign/load-test/systems-engineering-vision-2035

[3] Baker, G. L. and Blackburn, J. A., eds., 2010, *The Pendulum: A Case Study in Physics*, Oxford University Press, Oxford.

[4] MacLennan, B. J., 1990, *Functional Programming: Practice and Theory*, Addison-Wesley, Reading, Mass.

[5] Bird, R. and Wadler, P., 1988, *Introduction to Functional Programming*, Prentice Hall International Series in Computer Science, Prentice Hall, New York.

[6] Eisenbart, B., Gericke, K., and Blessing, L., 2013, "An Analysis of Functional Modeling Approaches across Disciplines," Artificial Intelligence for Engineering Design, Analysis and Manufacturing, **27**(3), pp. 281–289.

[7] Pahl, G., Beitz, W., Feldhusen, J., and Grote, K. H., 2007, *Engineering Design: A Systematic Approach*, 3rd ed., Springer, London.

[8] Wymore, A. W., 1993, *Model-Based Systems Engineering: An Introduction to the Mathematical Theory of Discrete Systems and to the Tricotyledon Theory of System Design*, Systems Engineering Series, CRC Press, Boca Raton, Fla.

[9] Zeigler, B. P., Muzy, A., and Kofman, E., 1976, *Theory of Modeling and Simulation*, 3rd ed., Elsevier.

[10] Ashby, W. R., 1956, *An Introduction to Cybernetics*, internet ed., Chapman & Hall, London.

[11] Gomes, C., Thule, C., Broman, D., Larsen, P. G., and Vangheluwe, H., 2018, "Co-Simulation: A Survey," ACM Computing Surveys, **51**(3), pp. 49:1–49:33.

[12] Banks, J., 1999, "Introduction to Simulation," *Proceedings of the 31st Conference on Winter Simulation: Simulation—a Bridge to the Future - Volume 1*, Association for Computing Machinery, New York, NY, USA, December 1, 1999, pp. 7–13, doi: 10.1145/324138.324142.

[13] Paredis, C., Diaz-Calderon, A., Sinha, R., and Khosla, P., 2001, "Composable Models for Simulation-Based Design," Engineering with Computers, **17**(2), pp. 112–128.

[14] Baez, J. C. and Pollard, B. S., 2017, "A Compositional Framework for Reaction Networks," Reviews in Mathematical Physics, **29**(09), p. 1750028.

[15] Fong, B., 2016, "The Algebra of Open and Interconnected Systems," Ph.D. thesis, arXiv, Oxford University, doi: 10.48550/arXiv.1609.05382, 1609.05382

[16] Herstein, I. N., 1964, *Topics in Algebra*, 1st ed., Blaisdell Publishing Company, Waltham, MA.

[17] Fritzson, P., 2011, *Introduction to Modeling and Simulation of Technical and Physical Systems with Modelica*, IEEE Press, Piscataway, NJ.

[18] Strachey, C., 2000, "Fundamental Concepts in Programming Languages," Higher-Order and Symbolic Computation, **13**(1), pp. 11–49.

[19] Willems, J. C., 2007, "The Behavioral Approach to Open and Interconnected Systems," IEEE Control Systems Magazine, **27**(6), pp. 46–99.

[20] Breiner, S., Denno, P. O., and Subrahmanian, E., 2020, "Categories for Planning and Scheduling," NIST, **67**(11).

[21] Polderman, J. W. and Willems, J. C., 1998, "Dynamical Systems," *Introduction to Mathematical Systems Theory: A Behavioral Approach*, Vol. 26 of Texts in Applied Mathematics, Springer, New York, NY, pp. 1–25.

[22] Sinha, R., Paredis, C. J. J., Liang, V.-C., and Khosla, P. K., 2001, "Modeling and Simulation Methods for Design of Engineering Systems," Journal of Computing and Information Science in Engineering, **1**(1), pp. 84–91.

[23] Pichler, P., Weber, B., Zugal, S., Pinggera, J., Mendling, J., and Reijers, H. A., 2012, "Imperative versus Declarative Process Modeling Languages: An Empirical Investigation," *Business Process Management Workshops*, F. Daniel, K. Barkaoui, and S. Dustdar, eds., Vol. 99, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 383–394.

[24] Abelson, H. and Sussman, G. J., 1993, *Structure and Interpretation of Computer Programs*, 17th ed., The MIT Electrical Engineering and Computer Science Series, MIT Pr. [u.a.], Cambridge.

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

[26] Morris, J., Mocko, G., and Wagner, J., 2025, "Unified System Modeling and Simulation via Constraint Hypergraphs," [Manuscript submitted for publication in the Journal of Computing and Information Science in Engineering], (Special Issue on Networks and Graphs for Engineering Systems and Design).

[27] Borutzky, W., ed., 2011, *Bond Graph Modelling of Engineering Systems: Theory, Applications and Software Support*, Springer New York, New York, NY.

[28] Paytner, H. M., 2000, "The Gestation and Birth of Bond Graphs," accessed 2024-06-17, https://sites.utexas.edu/longoria/files/2020/10/Birth_of_-Bond_Graphs.pdf

[29] Palm, W. J., 2014, "Block Diagrams, State-Variable Models, and Simulation Methods," *System Dynamics*, Third edition ed., McGraw-Hill Science, New York, NY, pp. 250–318.

[30] Tiller, M., 2001, "Block Diagrams vs. Acausal Modeling," *Introduction to Physical Modeling with Modelica*, M. Tiller, ed., Springer US, Boston, MA, pp. 255–264.

[31] Fisher, I., 1896, "What Is Capital?" The Economic Journal, **6**(24), pp. 509–534.

[32] Baez, J., Li, X., Libkind, S., Osgood, N. D., and Patterson, E., 2023, "Compositional Modeling with Stock and Flow Diagrams," Electronic Proceedings in Theoretical Computer Science, **380**, pp. 77–96.

[33] Chen, P. P.-S., 1976, "The Entity-Relationship Model—toward a Unified View of Data," ACM Trans. Database Syst., **1**(1), pp. 9–36.

[34] Gilbreth, F. B. and Gilbreth, L. M., 1921, "Process Charts," *Annual Meeting of The American Society of Mechanical Engineers*, American Society of Mechanical Engineers, New York, 5 Dec 1921, accessed 2024-09-25, https://web.archive.org/web/20150509222833/https://engineering.purdue.edu/IE/GilbrethLibrary/gilbrethproject/processcharts.pdf

[35] Wach, P. and Salado, A., 2019, "Can Wymore's Mathematical Framework Underpin SysML? An Initial Investigation of State Machines," Procedia Computer Science, **153**, pp. 242–249.

[36] Ibe, O. C., 2013, *Markov Processes for Stochastic Modeling*, 2nd ed., Elsevier Insights, Elsevier, London.

[37] Gantt, H. L., 1913, *Work, Wages, and Profits*, Engineering Magazine.

[38] US Department of the Navy, 1958, "Program Evaluation Research Task Summary Report Phase 1," Government Printing Office, Washington DC, accessed 2023-06-17, https://web.archive.org/web/20151112203807/http://www.dtic.mil/dtic/tr/fulltext/u2/735902.pdf

[39] Daly, R., Shen, Q., and Aitken, S., 2011, "Learning Bayesian Networks: Approaches and Issues," The Knowledge Engineering Review, **26**(2), pp. 99–157.

[40] Kapteyn, M. G., Pretorius, J. V. R., and Willcox, K. E., 2021, "A Probabilistic Graphical Model Foundation for Enabling Predictive Digital Twins at Scale," Nature Computational Science, **1**(5), pp. 337–347.

[41] Pearl, J., 2009, *Causality*, Cambridge University Press.

[42] Object Modeling Group, 2007, "OMG Systems Modeling Language (OMG SysML)," accessed 2024-09-21, https://www.omg.org/spec/SysML/1.0/PDF

[43] International Organization for Standardization, 2004, "ISO 10303-11," accessed 2025-03-05, https://www.iso.org/standard/38047.html

[44] Friedman, G. J. and Leondes, C. T., 1969, "Constraint Theory, Part I: Fundamentals," IEEE Transactions on Systems Science and Cybernetics, **5**(1), pp. 48–56.

[45] Dechter, R., 1992, "Constraint Networks," *Encyclopedia of Artificial Intelligence*, 2nd ed., S. C. Shapiro, ed., Vol. 1, John Wiley & Sons Inc, New York, pp. 276–285.

[46] Lecoutre, C., 2013, *Constraint Networks: Targeting Simplicity for Techniques and Algorithms*, John Wiley & Sons.

[47] Hudak, P., 1989, "Conception, Evolution, and Application of Functional Programming Languages," ACM Comput. Surv., **21**(3), pp. 359–411.

[48] Mitchell, J. C., 2003, *Concepts in Programming Languages*, Cambridge University Press, Cambridge.

[49] Roy, P. V. and Haridi, S., 2004, *Concepts, Techniques, and Models of Computer Programming*, MIT Press.

[50] Floridi, L., 2010, *Information: A Very Short Introduction*, Very Short Introductions, Oxford University Press, Oxford.

[51] O'Donnell, M. J., 1977, *Computing in Systems Described by Equations*, No. 58 in Lecture Notes in Computer Science, Springer-Verlag, Berlin ; New York.

[52] Pešić, M., 2008, "Constraint-Based Workflow Management Systems: Shifting Control to Users," Ph.D. thesis, Technische Universiteit Eindhoven, Eindhoven, Netherlands, doi: 10.6100/IR638413.

[53] Reade, C., 1989, *Elements of Functional Programming*, International Computer Science Series, Addison-Wesley, Wokingham, England ; Reading, Mass.

[54] Peak, R. S., Burkhart, R. M., Friedenthal, S. A., Wilson, M. W., Bajaj, M., and Kim, I., 2007, "Simulation-Based Design Using SysML Part 1: A Parametrics Primer," INCOSE International Symposium, **17**(1), pp. 1516–1535.

[55] Friedenthal, S., Moore, A., and Steiner, R., 2015, *A Practical Guide to SysML : The Systems Modeling Language*, 3rd ed., Elsevier, Waltham.

[56] Otter, M. and Elmqvist, H., 2000, "Modelica," Simulation News Europe, **10**(29/30), pp. 3–8.

[57] Hughes, J., 1989, "Why Functional Programming Matters," The Computer Journal, **32**(2), pp. 98–107.

[58] Révész, G. E., 1988, *Lambda-Calculus Combinators, and Functional Programming*, No. 4 in Cambridge Tracts in Theoretical Science, Cambridge University Press, Cambridge.

[59] Huet, G., ed., 1990, *Logical Foundations of Functional Programming*, The UT Year of Programming Series, Addison-Wesley, Reading, Mass.

[60] Brown, K., Hanks, T., and Fairbanks, J., 2022, "Compositional Exploration of Combinatorial Scientific Models," doi: 10.48550/arXiv.2206.08755, 2206.08755

[61] Hedges, J., 2018, "Towards Compositional Game Theory," Ph.D. thesis, Queen Mary University of London, London, accessed 2024-09-10, http://qmro.qmul.ac.uk/xmlui/handle/123456789/23259

[62] Spivak, D. I., 2015, "The Steady States of Coupled Dynamical Systems Compose According to Matrix Arithmetic," doi: 10.48550/arXiv.1512.00802, 1512.00802

[63] Zenger, M., 2004, "Programming Language Abstractions for Extensible Software Components," Ph.D. thesis, EPFL, Lausanne, Switzerland, doi: 10.5075/epfl-thesis-2930.

[64] Nielsen, C. B., Larsen, P. G., Fitzgerald, J., Woodcock, J., and Peleska, J., 2015, "Systems of Systems Engineering: Basic Concepts, Model-Based Techniques, and Research Directions," ACM Comput. Surv., **48**(2), pp. 18:1–18:41.

[65] Church, A., 1941, *The Calculi of Lambda Conversion*, No. 6 in Annals of Mathematics Studies, Princeton University Press, Princeton, NJ.

[66] Barendregt, H. P., 1981, *The Lambda Calculus: Its Syntax and Semantics*, No. 103 in Studies in Logic and the Foundations of Mathematics, North-Holland, Amsterdam New York Oxford.

[67] Spivak, D. I., 2013, "Category Theory for Scientists," doi: 10.48550/arXiv.1302.6946, 1302.6946

[68] Wegner, P., 1990, "Concepts and Paradigms of Object-Oriented Programming," SIGPLAN OOPS Mess., **1**(1), pp. 7–87.

[69] Cook, W. R., 1991, "Object-Oriented Programming versus Abstract Data Types," *Foundations of Object-Oriented Languages*, J. W. de Bakker, W. P. de Roever, and G. Rozenberg, eds., Lecture Notes in Computer Science, Vol. 489, Springer, Berlin, Heidelberg, 1991, pp. 151–178, doi: 10.1007/BFb0019443.

[70] Pinson, L. J. and Wiener, R. S., 1988, *An Introduction to Object-Oriented Programming and Smalltalk*, Addison-Wesley, Reading, Mass.

[71] Wagg, D. J., Worden, K., Barthorpe, R. J., and Gardner, P., 2020, "Digital Twins: State-of-the-Art and Future Directions for Modeling and Simulation in Engineering Dynamics Applications [Special Section]," ASME J Risk and Uncert in Engrg Sys Part B Mech Engrg, **6**(3), pp. 030901:1–030901:18.

[72] Dolk, D. R. and Kottemann, J. E., 1993, "Model Integration and a Theory of Models," Decision Support Systems, **9**(1), pp. 51–63.

[73] Budiardjo, A. and Migliori, D., 2021, "Digital Twin System Interoperability Framework," Digital Twin Consortium, accessed 2021-12-07, https://www.digitaltwinconsortium.org/wp-content/uploads/sites/3/2022/06/Digital-Twin-System-Interoperability-Framework-12072021.pdf

[74] International Organization for Standardization, 2017, "Information Technology-Cloud Computing-Interoperability and Portability," doi: 10.3403/30313036U.

[75] Parnas, D. L., 1972, "On the Criteria to Be Used in Decomposing Systems into Modules," Commun. ACM, **15**(12), pp. 1053–1058.

[76] Rentsch, T., 1982, "Object Oriented Programming," ACM SIGPLAN Notices, **17**(9), pp. 51–57.

[77] Harel, D. and Pnueli, A., 1985, "Reactive Systems," *Logics and Models of Concurrent Systems*, K. Apt, ed., NATO ASI Series, Vol. 13, Springer, Berlin, Heidelberg, Jan 1985, accessed 2024-09-16, https://link.springer.com/chapter/10.1007/978-3-642-82453-1_17

[78] Lorenz, E., 1972, "Predictability: Does the Flap of a Butterfly's Wings in Brazil Set Off a Tornado in Texas?" accessed 2025-02-27, https://mathsciencehistory.com/wp-content/uploads/2020/03/132_kap6_lorenz_artikel_the_butterfly_effect.pdf

[79] Gomes, C., Meyers, B., Denil, J., Thule, C., Lausdahl, K., Vangheluwe, H., and De Meulenaere, P., 2019, "Semantic Adaptation for FMI Co-Simulation with Hierarchical Simulators," SIMULATION, **95**(3), pp. 241–269.

[80] Bertsch, C., Ahle, E., and Schulmeister, U., 2014, "The Functional Mockup Interface - Seen from an Industrial Perspective," *The 10th International Modelica Conference, March 10-12, 2014, Lund, Sweden*, 2014-03-10, pp. 27–33, doi: 10.3384/ecp1409627.

[81] Andersson, C., 2016, "Methods and Tools for Co-Simulation of Dynamic Systems with the Functional Mock-up Interface," Doctoral Dissertation in Mathematical Sciences, Lund University, Lund, Sweden, accessed 2024-03-12, https://www.maths.lth.se/na/staff/chria/phdthesis.pdf

[82] Wiens, M., Meyer, T., and Thomas, P., 2021, "The Potential of FMI for the Development of Digital Twins for Large Modular Multi-Domain Systems," Modelica Conferences, pp. 235–240.

[83] Krishnamurthi, S., Felleisen, M., and Friedman, D. P., 1998, "Synthesizing Object-Oriented and Functional Design to Promote Re-Use," *Proceedings of the 12th European Conference on Object-Oriented Programming*, ECCOP '98, Vol. 1445, Springer-Verlag, Berlin, Heidelberg, July 20, 1998, pp. 91–113, doi: 10.1007/BFb0054088.

[84] Reynolds, J. C., 1978, "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*, D. Gries, ed., Springer, New York, NY, pp. 309–317.

[85] Wadler, P., 1998, "The Expression Problem," accessed 2025-01-27, https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt

[86] Morris, J., 2024, "ConstraintHg," https://github.com/jmorris335/ConstraintHg

[87] Cellier, F. E., 1991, *Continuous System Modeling*, Springer, New York, NY.

[88] Cellier, F. E. and Kofman, E., 2006, *Continuous System Simulation*, Springer, New York.

[89] Larman, C., 2001, "Protected Variation: The Importance of Being Closed," IEEE Software.

[90] National Academies of Sciences, Engineering, and Medicine, 2024, *Foundational Research Gaps and Future Directions for Digital Twins*, The National Academies Press, Washington, D.C.