Interactive Programmatic Modeling

(Invited Paper)

DAVID BROMAN*, KTH Royal Institute of Technology, Sweden

Modeling and computational analyses are fundamental activities both within science and engineering. Analysis activities can take various forms, such as simulation of executable models, formal verification of model properties, or inference of hidden model variables. Traditionally, tools for modeling and analysis have similar workflows: (i) a user designs a textual or graphical model or the model is inferred from data, (ii) a tool performs computational analyses on the model, and (iii) a visualization tool displays the resulting data. This paper identifies three inherent problems with the traditional approach: the recomputation problem, the variable inspection problem, and the model expressiveness problem. As a solution, we propose a conceptual framework called Interactive Programmatic Modeling (IPM). We formalize the interface of the framework and illustrate how it can be used in two different domains: equation-based modeling and probabilistic programming.

CCS Concepts: • **Software and its engineering** → **Development frameworks and environments**.

Additional Key Words and Phrases: Modeling languages, Cyber-Physical Systems, Probabilistic Programming

ACM Reference Format:
David Broman. 2021. Interactive Programmatic Modeling: (Invited Paper). *ACM Trans. Embedd. Comput. Syst.* 20, 4, Article 33 (June 2021), 25 pages. https://doi.org/10.1145/3431387

1 INTRODUCTION

A model describes an abstraction of another entity—such as a system or a process—by removing details. The abstraction makes it possible to analyze the model using computational methods, such as simulation (to get a timed trace from the model), formal verification (to formally verify some logical properties of the model), and inference (by inferring hidden variables within the model). Both scientists and engineers make use of models, but for different reasons [41]. Engineers *design* models to reason about properties, and to facilitate the *construction* of systems, using the model as a blueprint. By contrast, scientists typically construct models *to understand* the underlying system, such as a biological or physical system.

Regardless of whether models are created and analyzed for scientific or engineering purposes, it is vital to have good tool support for the analysis task. Fig. 1(a) and Fig. 1(b) show two traditional tool workflows for performing modeling and analysis. Fig. 1(a) depicts the workflow where a graphical or textual model is analyzed and visualized. In this case, the model itself contains explicit probes and scopes that indicate what data should

*This paper defines and extends that idea and concept of interactive programmatic modeling, which was first mentioned in the following conference paper: David Broman. 2019. A Vision of Miking: Interactive Programmatic Modeling, Sound Language Composition, and Self-Learning Compilation. In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering (SLE '19)*, October 20–22, 2019, Athens, Greece. ACM, New York, NY, USA. Besides the basic idea of the concept, all contributions, including the formalization, are new in this paper.

Author's address: David Broman, dbro@kth.se, EECS and Digital Futures, KTH Royal Institute of Technology, Sweden.

1539-9087/2021/6-ART33 $15.00
https://doi.org/10.1145/3431387

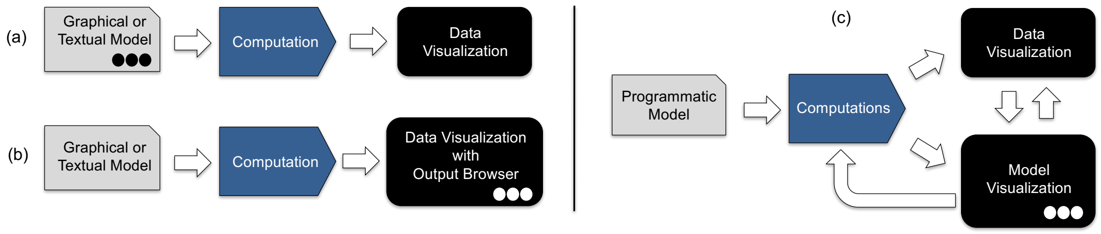

Fig. 1. Figures (a) and (b) depict the data flow of traditional approaches to modeling, computation, and visualization. The gray boxes denote the model (artifact), either as textual or graphical models. The dots (black or white) illustrate probes or selection boxes that determine which model variables to visualize. The blue arrow boxes capture arbitrary computation (e.g. simulation, verification, or inference). Finally, the black rounded boxes illustrate the components that perform the actual visualization. Figure (c) illustrates a simplified version of the interactive programmatic modeling approach advocated in this paper. We describe the details of this conceptual framework in Section 2.

be visualized (the filled dots in the figure). For instance, well-known graphical modeling environments, such as Ptolemy II [20, 52] and Simulink [44] make use of such an approach.

By contrast, Fig. 1(b) illustrates a flow where the model itself does not contain any probes. Instead, an output browser—usually in the form of a tree mirroring the original graphical model hierarchy—selects which variables that are visualized. Several tools based on the Modelica language [46], for instance OpenModelica [23] and Dymola [19], make use of such output browsers.

We have identified three general problems with the two traditional approaches in Fig. 1(a) and (b).

- **The recomputation problem:** In case (a), if a parameter is changed or a different set of variables are selected for observation (for instance, by adding new probes in the model), typically all data need to be recomputed[1]. In case (b) only some parameter changes need recomputations. However, computing and then displaying only a selection of the output data can be both computationally demanding and require large amount of storage.
- **The variable inspection problem:** In case (a), the model includes the variables selected for inspection. This makes it intuitive when selecting the variables (same abstraction as the model), but results in the problem that variables are hard-coded into the model. In case of (b), variable selection is not hard-coded into the model, but a tree-based browser selector has a different abstraction than the original model, making it harder to select the right variable.
- **The model expressiveness problem:** Graphical modeling is common in various engineering disciplines, whereas visual programming [1] (using graphical notation) has not gained the same attention. Although simple graphical models are typically quite intuitive and easy to work with, they become less intuitive when expressing a more complex program or model.

To mitigate these problems, we propose a new conceptual framework called *interactive programmatic modeling*. Fig. 1(c) depicts a high-level view of the framework. A *programmatic model* is a textual model, which has programming components (such as loops, if-conditions, higher-order functions, sequences etc.) that make it possible to create very expressive models. A programmatic model can contain default values for parameters, but does not include probes. The selection is instead performed in a visual representation that is *automatically generated using automatic layout* [56], based on the programmatic model. If a user changes a parameter value in the visual model, only the necessary parts of the output data are recomputed.

[1]Note that for commercial closed source environments, it is difficult to know to what extent caching mechanisms are implemented internally.

We contend that the proposed approach addresses these problems in the following way: (i) recomputation is done only when needed (lazily), (ii) variables for inspection are selected in the graphical visualization view, without hard-coding variables for selection in the original model, and (iii) the modeling language is kept expressive by using textual modeling, but still gives high-level intuition by using automatic layout and model visualization.

The proposed approach has several new elements, but the ideas are also based on various previous approaches. The three closest related works are (i) the approach of pragmatics in the KIELER project [24], (ii) the concept of live programming in the programming language community [38, 58], and (iii) earlier ideas of illustrative programming [22]. A more detailed comparison can be found in the related work section (Section 6). To be more specific, the key contributions presented in this paper are as follows:

- We define a conceptual framework called *interactive programmatic programming (IPM).* The framework supports both static systems and time-series systems. We describe and formalize the interface of the new framework (Section 2).
- We illustrate and discuss how the framework can be instantiated and used in the domain of equation-based modeling of physical systems. Specifically, we exemplify how the approach can be used for continuous-time models, formalized using differential-algebraic equations. Also, we give an overview of how such a system can be implemented and further extended (Section 3). Moreover, we discuss the domain of probabilistic programming. We encode static Bayesian networks and discuss how state-space models can interact with the IPM environment (Section 4).

Finally, we discuss potential threats of validity and how the overall approach relates to the problems stated in the introduction (Section 5).

2 THE IPM FRAMEWORK

This section describes the conceptual framework of interactive programmatic modeling (IPM). First, we define the meaning of a programmatic model, and how it relates to analysis techniques and systems. This is followed by an informal overview of the conceptual framework, where we describe how different components in the framework are connected. Finally, we formalize the interfaces of the computation components, and discuss how the framework mitigates the problems with the traditional approach to modeling and analysis, as described in the introduction.

2.1 Programmatic Models

The term *programmatic model* has recently been used in the context of probabilistic programming [48]. In this paper, we generalize its meaning by extending Marvin Minsky's definition from 1965 on the relationship between models, systems, and experiments [16]. We define a programmatic model as follows:

Definition 2.1 (Programmatic Model). A programmatic model (M) is a computer program where its behavior describes an abstraction of a system (S), such that an analysis technique (A) can be performed on M to answer questions about S.

We can observe a few key points in this definition:

- *A programmatic model is a computer program.* There must exist well-defined syntax and semantics for the language that the programmatic model is written in. In our context, the program's concrete syntax is assumed to be textual. A programmatic model may then include any kind of abstractions that we usually find in programming languages, such as records, loops, recursion, higher-order functions, objects and more.

- *The programmatic model is an abstraction of a system.* This means that the model does not, intentionally, contain all the details of the modeled system. Specifically, by analyzing the model, we can answer questions about the system. In this paper, we use the term *analysis* for any kind of experiment (e.g. using simulation), formal verification, or inference that can be performed on a model.

In the rest of the paper, we will see how programmatic models can be used in different context, and for different modeling tasks.

2.2 Framework Overview

Consider Fig. 2 that gives an overview of the proposed framework. The framework is divided into three parts: (i) model and data, (ii) computations, and (iii) views and control.

The input to the framework is a programmatic model, together with related input data (gray rectangular boxes depicting artifacts). The computation part of the framework consists of four separate computational components (the blue arrow boxes). The *model compiler* translates a programmatic model into a new artifact, called the *execution engine*. This translation (compilation) enables more efficient analysis, since the analysis task, such as numerical simulation, is then performed using compiled code. If the analysis is not compiled, but instead interpreted, the model compiler step is not needed. Note also how the execution engine is embedded as an artifact (gray box) inside a processing component. The reason is that the generated artifact (the execution engine) is executable and performs the analysis by running the execution engine. When the execution engine runs, it takes

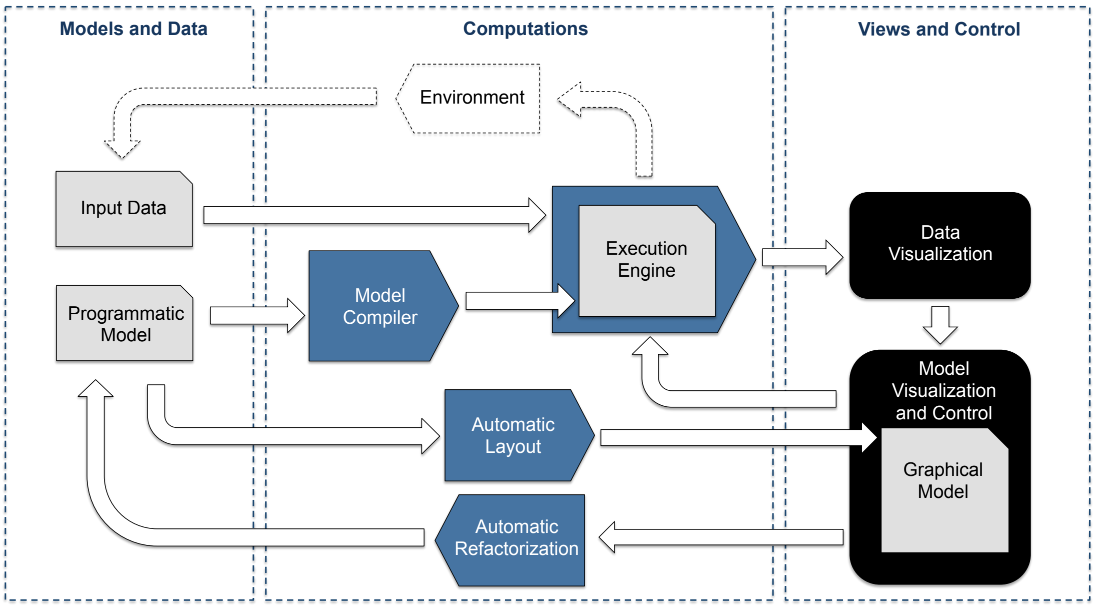

Fig. 2. An overview of the IPM framework. The figure is divided into three parts: (i) models and data, (ii) computations, and (iii) views and control. The gray rectangular boxes depict data artifacts, the blue arrow boxes represent computational components, and the black rounded boxes depict visualization components. The dashed feedback loop at the top of the figure depicts an optional external environment that computes input data.

as input the *input data*, which can either be a static data set or a dynamic stream of data. The generated data is finally visualized and displayed to the user (see the black *data visualization* component).

At the top of the figure, there is a feedback loop representing an environment. This feedback loop is optional (dashed lines) in the framework, meaning that either the input data is statically precomputed (without the feedback loop) or produced by the environment in each time step as a result of the output from the execution engine. Examples of scenarios where environment feedback loops are relevant are for instance hardware-in-the-loop (HIL) simulations or co-simulation. We will discuss such examples in Section 3.3.

The programmatic models are computer programs, written using a concrete textual syntax. Although certain models are quite easy to model in text, a graphical visualization can be very valuable to better understand the structure of the model. For instance, if a state machine or an electrical circuit is modeled using text, a graphical visualization can make it easier to see how electrical components or states in the state machine are connected. To enable such an automatic visualization, a graphical representation can be generated using automatic layout [56] algorithms (see the *automatic layout* component). The generated *graphical model* is in itself an artifact generated by the *automatic layout* component.

From the *model visualization and control* component, there are two outgoing arrows. If a user changes the parameters of the graphical model, the change can in some circumstances result in a *structural change* of the model, something that may need to be updated and reflected in the original programmatic model. This means that the *automatic refactorization* component executes and updates the programmatic model. Moreover, if certain variables are selected for visualization, or non-structural parameters are changed, the *execution engine* needs to recompute and generate new output data for visualization.

We can also see that the component for *data visualization* can output data to the *model visualization and control* component. This may happen in an environment where the analysis data, such as simulation data, can be visualized inside the model itself. This is for instance done in LabVIEW [37], where you can step through a simulation and see visually how the model changes.

The observant reader may see that this last connecting loop between the *model visualization and control* component, the *execution engine* component, and the *data visualization* component is an instantiation of the Model-View-Control (MVC) design pattern [53]. In this case, the graphical visualization is the *controller* (selection of variables and configuration of parameters), the execution engine the *model* (using the somewhat confusing terminology of a model in the MVC pattern), and the data visualization the *view*. We will not discuss the MVC pattern further in this paper, but it naturally fits the conceptual framework presented in this paper.

2.3 Interface Formalization

In this section, we formalize the interfaces for computational components in Fig. 2. Although this section gradually introduces the different notations and functions, for convenience, a summary of the formalization is given in Fig. 3.

The heart of the framework is the execution engine that performs the actual analysis of the programmatic model. A requirement of the overall framework is to be able to handle both static systems (without the notion of time) and dynamic transition systems (where the state evolve over time). Let us start with the formalization of the execution engine.

Let M be the set of programmatic models and S be the set of states, where $S_m \subseteq S$ is the set of states for a specific model $m \in M$. Before the computation starts, the computation engine needs to initialize the model instance, that is, to compute the initial state $s_0 \in S$. Let U be the set of input identifiers and Y be the set of output identifiers. Note that we do not use the term input or output variables here because an identifier cannot be assigned a value, only uniquely identify a specific input or output. Moreover, let $\mathbb{V}$ be the set of potential values that can be associated with an input or output identifier. Hence, a tuple $(u, v) \in U \times \mathbb{V}$ represents an input pair,

Models and Values

Set of programmatic models	M
Set of state valuations	S
Set of input identifiers	U
Set of output identifiers	Y
Set of values a variable can take	$\mathbb{V}$
Set of graphical models	G
Set of environment states	E

Parameters

Set of computation parameters	P_c
Set of output parameters	P_o
Set of structural parameters	P_s
Set of all parameters	$P = P_c \cup P_o \cup P_s$

Input alternatives

$\mathbf{U} = \overline{U}$

$\mathbf{U} = E \times (E \rightarrow \overline{U}) \times (\overline{Y} \times E \rightarrow E)$

Functions

$$\begin{aligned}
\texttt{init}_m &: \mathbf{U} \times \overline{P}_c \rightarrow S \\
\texttt{trans}_m &: S \times \mathbf{U} \times \overline{P}_c \rightarrow S \\
\texttt{output}_m &: S \times \mathbf{U} \times \overline{P}_o \times \mathcal{P}(Y) \rightarrow \overline{Y} \\
\texttt{compile} &: M \times \overline{P}_s \rightarrow (\mathbf{U} \times \overline{P}_c \rightarrow S) \times (S \times \mathbf{U} \times \overline{P}_c \rightarrow S) \times (S \times \mathbf{U} \times \overline{P}_o \times \mathcal{P}(Y) \rightarrow \overline{Y}) \\
\texttt{layout} &: \overline{P}_s \times M \rightarrow G \\
\texttt{refactor} &: \overline{P} \times G \times M \rightarrow M
\end{aligned}$$

Fig. 3. A formalization of the IPM framework interfaces.

associating a specific identifier $u \in U$ with a specific input value $v \in \mathbb{V}$. To simplify the formalization, we define the following

$$\overline{U} \overset{\text{def}}{=} \mathcal{P}(U \times \mathbb{V}) \tag{1}$$

$$\overline{Y} \overset{\text{def}}{=} \mathcal{P}(Y \times \mathbb{V}) \tag{2}$$

where $\mathcal{P}(A)$ is the powerset notation. For instance, suppose the set of input identifiers is $U = \{u_1, u_2\}$. For a specific input point, we associate value 7 with u_1, and 11 with u_2. In such a case, $\{(u_1, 7), (u_2, 11)\} \in \overline{U}$ is one specific input set.

Next, let P be the set of all input parameter identifiers. In contrast to input data, a parameter value can only be changed before the computational analysis starts. For brevity, we use the word *parameter* as a synonym for *parameter identifier*. We divide the set of parameters into three subsets: P_c for computation parameters, P_o for output parameters, and P_s for structural parameters. The set of all parameters is $P = P_c \cup P_o \cup P_s$. All sets are distinct, that is, $P_c \cap P_o = P_c \cap P_s = P_o \cap P_s = \emptyset$. The meanings of the different types of parameters are central to the framework, and we introduce each of them in turn in this section. We use the same notation with an overline as in definitions (1) and (2) for associating a set of parameters with parameter values.

The abstract behavior of a computation engine is formalized using three mathematical functions. The first function

$$\texttt{init}_m \colon \mathbf{U} \times \overline{P}_c \to S \tag{3}$$

initializes the engine. That is, given input data $u \in \mathbf{U}$ and a set of computation parameter values $p_c \in \overline{P}_c$, the function returns an initial state $s_0 \in S$. The subscript m of function $\texttt{init}_m$ means that the function is specialized specifically for model $m \in M$. That is, the functions of the execution engine are specialized for a specific model. Note also that the parameters belong to a computation parameter set p_c. The meaning of p_c can be interpreted as the parameters that may affect the initialization.

We use symbol $\mathbf{U}$ to show two alternative definitions for how input data is handled (see Figure 3). That is, either $\mathbf{U} = \overline{U}$ which means that the framework uses static input data, or that the framework uses a feedback loop with an environment for computing input data (see the top part of Figure 2). We will come back later in this section with details about the environment alternative, but for now, assume that $\mathbf{U} = \overline{U}$.

We define the rest of the computation functions as a transition system that is based on the idea of a Mealy machine. More precisely, a Mealy machine is a deterministic finite automaton (DFA) together with an output function. In each time step, the state is updated using the following function (called the state update and transition function):

$$\texttt{trans}_m \colon S \times \mathbf{U} \times \overline{P}_c \to S. \tag{4}$$

Each state transition takes the previous state as input and returns a new updated state. Besides the previous state, the function also takes the input $u \in \mathbf{U}$ for this particular transition, and the computation parameter values $p \in \overline{P}_c$. For any state in the transition system, including the computed initial state, the system can compute the set of output values $y \in \overline{Y}$ using the output function

$$\texttt{output}_m \colon S \times \mathbf{U} \times \overline{P}_o \times \mathcal{P}(Y) \to \overline{Y}. \tag{5}$$

Note that the signature $\mathcal{P}(Y)$ for the last parameter. When the output function is called, an argument $y \in \mathcal{P}(Y)$ for this parameter states the set of output identifiers that should be computed as output. This means that the framework can select which outputs that are computed, depending on a user configuration.

The reason for separating the parameters into the two sets P_c and P_o relates to the recomputation problem. For instance, if a user changes an output parameter, the sequence of states do not have to be recomputed, only the output function needs to be executed for each step. Note that such an optimization requires that all states or state checkpoints need to be saved. Hence, computation time has a direct tradeoff to storage space.

Consider Fig. 2 again. Note how the *model compiler* takes a programmatic model as input and generates the *execution engine* component (the arrow points directly to the gray box). That is, the model compiler *compiles* the programmatic model and generates the execution engine, which consists of the three functions $\texttt{init}_m$, $\texttt{trans}_m$, and $\texttt{output}_m$. Hence, the *model compiler* itself consists of a function

$$\texttt{compile} \colon M \times \overline{P}_s \to (\mathbf{U} \times \overline{P}_c \to S) \times (S \times \mathbf{U} \times \overline{P}_c \to S) \times (S \times \mathbf{U} \times \overline{P}_o \times \mathcal{P}(Y) \to \overline{Y}) \tag{6}$$

that takes as input a programmatic model $m \in M$ and a set of structural parameters $p \in P_s$, and outputs the three functions $\texttt{init}_m$, $\texttt{trans}_m$, and $\texttt{output}_m$. Note how the signatures for each of these functions correspond to the three Cartesian products that are the output of the `compile` function. We can make two observations concerning the `compile` function.

Firstly, note how the `compile` function takes a programmatic model $m \in M$ as input, but the model is not part of the output. The reason is that the model is implicitly part of the generated functions. This is why the subscript

m is given for the output functions. That is, functions $\texttt{init}_m$, $\texttt{trans}_m$ and $\texttt{output}_m$ are generated when function `compile` is invoked with a model $m \in M$.

Secondly, the function takes a set of structural parameters $p \in P_s$ as input. In contrast to the other parameters, the structural parameters affect the structure of the generated model. For instance, in a mechanical model, a structural parameter can be the number of discretizations that are made over a mechanical component. Such discretization typically affects the number of generated differential equations. As a consequence, different compiled versions of the state transition function need to be generated.

If we consider Fig. 2 again, there are two more computational components: *automatic layout* and *automatic refactorization*, each corresponding to the functions `layout` and `refactor`, respectively. The automatic layout function

$$\texttt{layout}\colon \overline{P}_s \times M \to G \tag{7}$$

generates a graphical visualization $g \in G$ from a set of structural parameters $p \in P_s$ and a programmatic model $m \in M$. Note that only structural parameters affect the basic graphical visualization. By contrast, only the computation parameters P_c and the output parameters P_o affect the computation of the output data (see the functions in the execution engine). As a consequence, if parameters are updated in the graphical user interface in the *model visualization and control* component, different actions are taken depending on which parameters that are changed. For instance, if a structural parameter $p \in P_s$ is changed, the automatic layout needs to be recomputed and the model compiler needs to be activated to generate a new execution engine. Hence, in the interactive environment, only the required components are activated, in a demand-driven (lazy) fashion.

We should also note that the graphical model itself may be parameterized on the output data (see the arrow between the data visualization component and the model visualization component in Figure 2). For instance, suppose that the visualization is a 3D visualization of a mechanical system. In such a case, the output data for the initial state of the system are needed to visualize the initial state of the 3D model. A simulation sequence can in the same way be used to visualize a 3D movement. We will, however, not elaborate further on how graphical models are parameterized or visualized.

The last function is the automatic refactorization function

$$\texttt{refactor}\colon \overline{P} \times G \times M \to M \tag{8}$$

that takes a set of parameter configurations $p \in P$, a graphical model $g \in G$, and a programmatic model $m \in M$ as input, and generates a new programmatic model $m' \in M$ as output. Such a refactorization function is optional in the framework. The purpose of the refactorization is to propagate changes or configuration settings in the graphical model back to the textual programmatic model. In the simplest case, this can be default settings for the different parameters, where the default parameters are encoded into the model. In a more complex scenario, graphical changes of the graphical model might be propagated back to the programmatic model. If changes can be done in both the textual and in the graphical model, and changes are propagated in both directions, it is typically referred to as *round-trip engineering*. We do not advocate such an approach in this work, but limited refactorization can be useful, especially configuration of default parameters.

Finally, as explained earlier, an alternative of using explicit static inputs using definition $\mathbf{U} = \overline{U}$, we can compute the input using an external environment, using the following definition:

$$\mathbf{U} = E \times (E \to \overline{U}) \times (\overline{Y} \times E \to E) \tag{9}$$

In this case $(e_0, f_o, f_t) \in \mathbf{U}$ is a triple, where e_0 is an initial environment state, f_o an environment output function, and f_t an environment transition function. As part of the implementation of the $\texttt{init}_m$ function, the input data $u \in \overline{U}$ is generated by calling $f_o(e_0)$. Then, in the implementation of $\texttt{trans}_m$, the environment state is updated using $f_t(y, e)$, where $y \in \overline{Y}$ is output data sent as input to the environment, and e is the previous environment state. Finally, within the implementation of $\texttt{output}_m$, the input data is computed in each time step using $f_o(e)$ for some environment state e. Note the feedback loop, where output data from the computation engine become input to the environment, and where the output data from the environment become input data to the compute engine. To break the circular dependency between these components, there needs to be a delay between the components. We will elaborate further on this with a concrete example in Section 3.3.

3 PROGRAMMATIC MODELING IN THE DOMAIN OF EQUATION-BASED PHYSICAL MODELING

In this section, we illustrate how the proposed conceptual framework can be applied to the domain of equation-based physical modeling. Specifically, we discuss how the approach can be implemented in the context of continuous-time modeling languages supporting differential-algebraic equations. This section ends with a brief discussion on aspects of extending the approach to target mixed continuous/discrete equation-based modeling, structurally dynamic systems, and using the Functional Mock-up Interface (FMI) standard [47].

3.1 Equation-Based Programmatic Modeling

Equation-based object-oriented modeling [7] is a modeling paradigm especially targeted for modeling of complex physical systems. There are many research languages in this domain [13, 27, 64], as well as professional commercial tools, such as Dymola [19] and Simscape [43]. One of the prominent languages within this category is Modelica [46], which can be used for equation-based modeling. The key property of these languages is that they are *acausal*, meaning that the flow of information between different components is bidirectional. This is the major difference to standard block-based modeling environments, such as Simulink [44], where blocks have inputs and outputs, and where the ports cannot be bidirectional.

Modelica has both a graphical view and a textual view, where the challenge for the toolchain is to keep these two views in sync. The Modelica tools do not make use of automatic layout algorithms. Instead, the modeler is responsible for the layout of the graphical model. The coordinates and the layout are then saved as annotations in the textual source code.

In this section, we illustrate the IPM framework by using an embedded domain-specific language encoded in the meta-programming language Modelyze [13]. As an application example, we use a simple control system of a mechatronic system. The example was previously given in our earlier paper [13] and technical report [12], but not in the context of the new conceptual framework.

Consider the following programmatic model, written in Modelyze.

```
def ControlSys() = {
    def s1, s2, s3, s4:Signal;
    def r1, r2, r3, r4:Rotational;
    ConstantSource(1, s1);
    Feedback(s1, s4, s2);
    PID(3, 0.7, 0.1, 10, s2, s3);
    DCMotor(s3, r1);
    IdealGear(4, r1, r2);
    serialize(5, r2, r3, ShaftElement);
    Inertia(0.3, r3, r4);
    SpeedSensor(r4, s4);
}
```

The model is arguably quite compact, and it contains a number of instantiations of subcomponents. Each of the subcomponents (such as `PID`, `DCMotor`, `IdealGear` etc.) are wired together using connection nodes, defined using the `def` keyword. For instance, the `Feedback` component is connected to the `PID` controller via node `s2`. Likewise, the `DCMotor` and the `IdealGear` are connected using node `r1`. Note that the different nodes have different types: `s1` is a Signal (one direction), whereas `r1` is an acausal connection in the 1-dimensional mechanical rotational domain.

Although it is possible to follow the connections in the textual programmatic model, it would certainly be valuable to graphically visualize the topology of the model. Fig. 4 depicts a graphical visualization of the textual model. The top part shows how the different components are connected together, and the lower part visualizes the internal topology of some of the subcomponents.

The equation-based model shown in this section consists of a hierarchy of components, which during a so called *elaboration phase* [11] translates the whole model down to a system of differential-algebraic equations (DAEs). The general form of a DAE can be written as

$$F(t, \dot{x}, x, y, u) = 0 \tag{10}$$

where t is the independent variable (typically representing time), x is a vector of dependent variables that appear differentiated, y is a vector of dependent variables without any derivatives (called algebraic variables), and u is a vector of input variables. By using index-reduction techniques [45, 50], a DAE may be translated into an index-1 semi-explicit form

$$\dot{x} = f(t, x, y, u), \tag{11}$$

$$0 = g(t, x, y, u). \tag{12}$$

DAE problems can either be solved directly by a numerical DAE solver, such as the SUNDIALS solver IDA [34], or explicitly using a combination of an ODE solver (to solve (11) in each time step) and an algebraic solver (to solve (12)).

If we consider the `ControlSys` model again, we can see that there are different kinds of parameters in the model. For instance, the value 1 of the `ConstantSource` component states the reference signal. This parameter is a computation parameter and affects both the initialization of the DAE and the approximation in each time step. Note, however, that the model does not have to be recompiled if this computation parameter is changed.

The graphical representation is good for understanding the topology, but it is not straight forward to encode more complex models graphically, such as the use of higher-order function. In this example of a programmatic model, there is a function call to a function named `serialize`, which takes a higher-order model [10] and composes n number of instances of that model in series (in this case 5 shaft elements). In this case, the number of instances in series can be defined as a structural parameter $p \in P_s$. If this parameter changes, the framework needs to recompile the model because it changes the structure of the differential-algebraic equation system. Note that this kind of programmatic model can make use of arbitrary higher-order function and combinator function, to make modeling very *expressive*.

Finally, a central problem for this kind of models is what we called *the variable inspection problem*. Modelyze handles this by making it possible for the user to add explicit probes in the programmatic model. This is, however, not a good solution because the user might want to inspect other variables in the future. Modelica instead uses a tree-based browser selector, where the user can select the states to inspect. However, this solution is not always intuitive because the tree-based view differs from the original view, and a large set of output data needs to be generated, even if the corresponding parameters are not selected and displayed. For instance, assume that the user wants to know the potential between the left hand side of the resistor and the right hand side of the inductor, inside the `DCMotor`. In the original text probe approach, the model needs to be edited, and a new measurement component needs to be added. In the Modelica approach, the data can be displayed *if* the output variable exists in the tree selector. If not, then the model also needs to be modified. In the proposed approach, only the $\texttt{output}_m$ function needs to be updated, if this selection is done in the graphical model. This means that the simulation

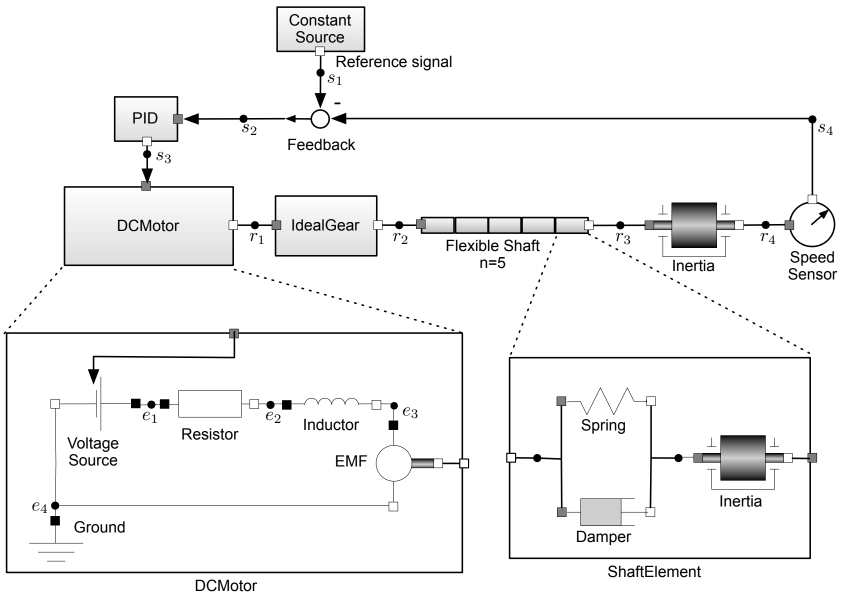

Fig. 4. A graphical visualization of the control-system model [12]. The figure is only a graphical illustration and has not been constructed using automatic layout.

(transition of state values) do not need to be recomputed, only the output function for this newly added output variable.

3.2 Implementation Aspects

In this subsection, we discuss various implementation aspects of implementing an IPM framework for a continuous-time equation-based modeling language. The code described in this section is given as high-level pseudo code, giving perspectives of how the suggested approach can be implemented using the formalization described in Section 2. The approach is based on the author's experience of being the main developer and designer of the Modelyze[2] framework [13], and being involved in the design of the Modelica[3] language. However, implementing a complete IPM system for an equation-based object-oriented (EOO) language is a major undertaking, and outside the scope of this article. As a consequence, this section illustrates the key aspects and insights of adjusting an EOO language to fit the conceptual IPM framework.

Figure 5 lists the pseudo code for five functions of an execution engine. The lower part of the figure lists functions `init`, `trans`, `output`, and `compile`, which are all implementations of the formal signatures described in Section 2. Three of these functions are marked as `generated` and shown in dashed boxes. The purpose of this special treatment is to highlight that these conceptual functions are compiled/generated by the `compile` function.

[2]http://www.modelyze.org
[3]http://www.modelica.org

```
function compute(model, inputData, pvc, pvo, pvs, outIdents, maxTime){
  if(model || pvs)
      (_init, _trans, _output) = compile(model, pvs);

  if(model || inputData || pvc || pvs)
    _state[0] = _init(inputData[0], pvc);

  i = 0;
  while(i * STEPSIZE <= maxTime){
    if(i * STEPSIZE > _oldMaxTime || model || inputData || pvc || pvo || pvs || outIdents)
      _outputData[i] = _output(_state[i], inputData[i], pvo, outIdents);

    if(i * STEPSIZE >= _oldMaxTime || model || inputData || pvc || pvs)
      _state[i+1] = _trans(_state[i], inputData[i], pvc);
    i++;
  }
  _oldMaxTime = maxTime;
  return _outputData;
}
```

```
function compile(model, pvs){
  (us, ys, pc, po) = extractIdents(model);
  eqsys = elaborateHierarchy(model, pvs);
  eqsys2 = elaborateDerivatives(eqsys);
  eqsys3 = indexReduction(eqsys2);
  init = codegenInit(eqsys3, us, pc);
  trans = codegenTrans(eqsys3, us, pc);
  output = codegenOutput(eqsys3, us, ys, po);
  return (init, trans, output);
}
```

```
generated ≈ init(inputVals, pvc){
  (fixedVals, guessVals) = extractVals(pvc, _eqsys);
  eqsys = applyEqSys(_eqsys, inputVals, pvc);
  (sv,svd) = initialValueSolver(eqsys, inputVals,
                        pvc, fixedVals, guessVals);
  solverState = initDAESolver(sv, svd);
  return solverState;
}
```

```
generated ≈ output(state, inputVals, pvo, outIdents){
  sv = getStateVarVals(state);
  paramVals = computeOutputVals(sv, pvo, inputVals
                                  outIdents);
  return paramVals;
}
```

```
generated ≈ trans(solverState, inputVals, pvc){
  residual = applyResidual(_residual, inputVals,pvc);
  jacobian = applyJacobian(_jacobian, inputVals,pvc);
  solverState2 = stepDAESolver(solverState, STEPSIZE,
                               residual, jacobian);
  return solverState2;
}
```

Fig. 5. The figure shows pseudo code for the five main functions that may be involved in the implementation of a computation engine for an equation-based modeling language. Function compute demonstrates the central part of the execution engine and is invoked every time a change occurs in the model, in the input, or in some parameters. The compute function invokes the compile function, which is defined in the lower left part of the figure. The compile function is responsible for compiling and generating code for the three functions: init, trans, and output. Note that these functions are shown in dashed boxes and with the keyword generated because they are not implemented directly, but rather generated (staged) by the compile function. The implementations for these three functions approximately demonstrate what is supposed to be generated by the compile function.

The actual implementation of init, trans and output should be seen as an illustration for how they might be generated.

Function compute—listed at the top of the figure—shows how these functions are invoked in a lazy fashion, such that recomputations are avoided. In the rest of this section, we discuss each of these functions in turn.

Recall the signature of function compile from Section 2:

$$\texttt{compile:}\ M \times \overline{P}_s \to (\mathbf{U} \times \overline{P}_c \to S) \times (S \times \mathbf{U} \times \overline{P}_c \to S) \times (S \times \mathbf{U} \times \overline{P}_o \times \mathcal{P}(Y) \to \overline{Y}) \tag{13}$$

The compile function takes a model and a parameter value set of structural parameter with name pvs as input, and returns a triple (init, trans, output) as output. Note that the compile function shows how a model is compiled by returning the compiled functions as elements of the triple. In a real system, this is of course practically more complicated. It can be done by generating C code of the equation system and runtime system, as done in e.g., OpenModelica or Dymola. Alternatively, it can be just-in-time (JIT) compiled, as done in the research compiler Hydra [29]. If the equation system is interpreted, as done in e.g., Sol [64] or Modelyze [13], the procedure described in this compile function still needs to be executed to some extent, but the overhead of the compilation is eliminated and the runtime simulation is instead more expensive. There are also techniques, such as partial evaluation, which is for instance used in the Modelyze runtime system, but this is outside the scope of this description.

On the first line of function `compile`, identifiers are extracted and categorized. This is followed by elaborating the hierarchy (line 2). Although this is shown here in just one line of code, this is the major part of an EOO compiler. The elaboration includes—among many things—inheritance, model redeclarations, and equation flattening in object-oriented languages such as Modelica, and higher-order function evaluation and connection semantics elaboration [11] in functional modeling languages. The end result of the elaboration phase (sometimes called the flattening phase) is a system of differential-algebraic equations, here denoted `eqsys`. This system of equations is further elaborated and transformed in several steps. Here we show two major phases, although EOO compilers may have additional phases. In the `elaborateDerivatives` function (line 4), higher-order derivatives (e.g. x'') are removed, and translated into first-order derivatives by expanding the equation system. Differential-algebraic equation systems of higher index [39] cannot be solved directly by a numerical solver. Instead, an index reduction procedure (line 5) needs to reduce it to index 1, such that it can be solved by a DAE solver, or a combination of an ODE solver and a non-linear algebraic solvers. This is another major part of an EOO compiler and is discussed extensively in the literature [45, 50, 51]. Lines 6-10 illustrates how each of the three functions are compiled into the three functions `init`, `trans`, and `output`.

The code generated function `init` has two parameters: `inputVals`, which represents a set of identifier value pairs, and `pvc`, the parameter value set of computation parameters. The main task for an init function in an EOO context is to initialize the DAE system. In contrast to the standard initial value problem for ODEs, a DAE needs to fulfill all algebraic constraints (and hidden constraints [50]). Hence, a model compiler typically makes a difference between fixed initialization values that must hold, and guess values which do not have to hold, and can be used as start values when searching for an initial value solution. These values are first extracted (line 2), and then used during the initial value solver (lines 4-5). Note how we use the underscore notation (e.g. `_eqsys`) to denote that a variable is global. In the case of the generated function, some of the global variables are also marked in italic font, which means that they are *generated* by the compiler. For instance, in this case, `_eqsys` has been generated by the compile function and is assumed to be available in the generate `init` function. The initial value solver in itself can be implemented in different ways, for instance by using a standard numerical solver suite such as SUNDIALS [34] or by solving an over-determined equation system. In the example, the initial value solver returns two vectors: `sv`, the state variable vector, and `svd`, the state variables in differentiated form. On line 6, we initialize the numerical solver engine. It is common to use implicit DAE solvers, such as IDA from the SUNDIALS suite, but ordinary explicit solver methods, such Runge-Kutta can also be used. Regardless of the procedure, the solver needs to keep a state, and this what the `init` function returns.

The generated `trans` function computes the transition of one step. For a numerical implicit solver such as IDA, the step size is given as a floating-point number (line 4). Such a step size only indicates how much further the DAE solver should proceed: the numerical solver typically performs variable sized steps internally. Lines 2 and 3 compute the *residual* (the vector of expressions $\mathbf{x}$ in a DAE $\mathbf{x} = 0$) and the *jacobian* of the equation system for specific input and parameter values. Note how the function assumes the existence of a pre-compiled version of the residual and the jacobian (denoted using an underscore, e.g., `_residual`). For simplicity, we assume that the input values are fixed in each time step, although various interpolation methods can also be used. Finally, the `trans` function returns the current solver state.

The generated function output produces the output data that can later be visualized. To be able to compute output data based on input values and output parameters (lines 3-4), the state value vector sv needs to be extracted from the DAE solver state (line 2). Typically, the state variables are stored as an array, and is the main state of the DAE solver. The function returns the chosen output parameter values.

The compute function (top of the figure) is not an explicit part of the formalization in Section 2. It is an example of the principles of how the different functions can be used together. In this scenario, we assume that the compute function is part of the compute engine and is invoked (triggered as an event) if any relevant data changes. For instance, if the model is changed (someone edited the text code), the model parameter carries information that the model has been changed. If, instead, for instance some structural parameter values are updated in the GUI, parameter pvs is changed. Note also how we use the parameters both as a value and as a boolean for checking if it has been changed. For instance, on line 2, the if statement checks if either the model or the structural parameters pvs has been changed. If either of them has been updated, the compile function is invoked, and the three functions _init, _trans, and _output are recompiled. Note how these three compiled/generated functions are written with an underscore. This means, as mentioned earlier, that they are global. That is, if the compute function is invoked a second time where the model has not been changed, these functions are already available, and do not have to be recomputed. Note also that we assume that the model is marked as updated the first time the compute function is called. A concrete implementation would of course implement this in a different way depending on the source language. The purpose of this approach is to give a compact way of illustrating the key ideas of the execution engine.

At a high level, the compute function makes sure to only recompute parts of the simulation, if it is needed. For instance, if model, input data, computational parameters (pvc), or structural parameters (pvs) have been changed, the whole simulation needs to be recomputed, and the system initialized again (lines 5-6). Note how the states are stored in an array of states _state. Such storage can be optimized (e.g., only save the complete solver state for the last values in the time series), but for simplicity, we show a straightforward solution where all states are stored in the array.

The rest of the computations take place in a while loop (lines 9-16). Typically, equation-based simulations are performed up to a certain time value (here using parameter maxTime). The time is often represented as a floating-point number, although an integer value with a specific resolution is preferred for mixed continuous/discrete simulations [18]. In this example, we use an integer to represent the steps that are taken for generating the output trace (using a floating-point number value STEPSIZE for the conversion). Note, however, that the DAE solver in the trans function still can take smaller steps internally, as is standard in variable step-size solver.

Inside the while loop, the if statements check if either the _output function or the _trans function or both functions should be called. Note also how a variable _oldMaxTime is used to make sure that old states in the time series are not recomputed, unless certain parameters or the model has been updated. We assume that _oldMaxTime is initialized to a negative value from the beginning.

In the end, the computation engine outputs the output data, which may have been completely or partially recomputed. A careful reader can see that there is a clear pattern of what needs to be recomputed. If compile is triggered, all other functions need to be reinvoked. If, on the other hand, init is called, trans and output need to be computed for all time points up to maxTime. Note also that if the maxTime is extend, only trans and output are called for the extended time. Also, if a parameter triggers only output to be called, it will regenerate all data points without calling any of the other functions.

Out of the formalized interface functions in Section 2, there are two functions that have not yet been discussed: layout and refactor. These functions are highly dependent on the underlying language's syntax and semantics, and we will therefore not describe any concrete implementations. Instead, we will discuss some general principles.

First, the layout function layout(p_s, m) generates a graphical representation in some graphical environment. The structural parameters p_s may affect this layout, because it can change the structure of the graphical model.

For instance, if higher-order functions or loops (as in Modelica) are used, several repeated elements should potentially be visualized. An important aspect (and challenge) is to make this readable even if there is a large number of elements. Computation and output parameters are intentionally not part of the function signature. Instead, if the computed data goes via the visualization component. That is, G can be seen as being parameterized by the output time-series data.

Second, the refactor function `refactor`(p, g, m) is the reverse of the layout component. In our approach, we do not advocate a refactor component to the same extent as automatic layout. However, if it is implement for an EOO language, it can, for instance, be used for changing default parameter values of certain components (e.g. changing the voltage value of a voltage source).

3.3 Further Extensions to Equation-Based Modeling

There are many variants of equation-based languages for physical modeling that have different properties and challenges. In the previous section, we describe a rather simple EOO language which only contained continuous-time modeling. However, most EOO languages also support hybrid modeling, that is, models that combine continuous-time and discrete-time signals. To define robust semantics for such languages with acausal connections is an active research area, in particular since discrete changes can change the DAE index of the model. For instance, see the paper by Furic [25] that describes several problems that exist in the standardized Modelica language.

For basic hybrid acausal models where the number of equations stay the same, the IPM framework should be rather straightforward to extrapolate from what is described in the previous section. The main differences concern the event loop in the `compute` function that needs to alternate between a continuous-time mode and a discrete mode, where the transition takes place on state or time events (where the former is implemented using zero-crossing detection on state variables). This means that the output data is not a (discretized) continuous signal, but rather a piece-wise continuous signal, which can be represented using e.g., a superdense time [42].

However, there are also research languages supporting structurally dynamic systems. This means that discrete events can trigger mode changes of the equation system, that is, the underlying system of equations changes during simulation time. Examples of language that support some form of structurally dynamic acausal systems are Mosilab [49], Sol [64], the M-MC DSL in Modelyze [13], MCL [35], and Hydra [28]. The challenge for the IPM approach with this kind of languages is that there is not one static compile phase in the beginning of the process, since the system of equations changes over time. Within the IPM framework, a structurally dynamic system could be handled in one of the following ways, depending on the language's underlying semantics:

- *Fixed number of modes.* Resent research results [3, 14] show that it is possible to handle the index reduction problem for models with a fixed number of modes. This means that the IPM framework can be used in a similar way as described earlier, where all modes are compiled statically in the beginning of the process.
- *Dynamic number of modes.* In such approach, the equation system changes dynamically and it is not possible to statically determine a set of possible modes. With such a scenario, the elaboration and index reduction take place during simulation time. The evaluation can be done using interpretation (as in Sol and Modelyze) or using JIT compilation (as in Hydra). A tool can also potentially cache previous modes, thus avoiding recompilation/elaborations if the same mode is visited repeatedly. In this case, the underlying mechanism for JIT compilation is done in the transition function, and the avoidance of recomputations need to be done within this function (e.g., using caching of modes).

A structurally dynamic system also gives new interesting challenges when it comes to graphical visualization. For a fixed number of modes, the different modes can be visualized as a state machine (where the states are modes, as done in e.g., modal models in Ptolemy II [20]). However, for dynamic number of modes, the current IPM framework is limited to approximate a static representation of the system. Further investigations are needed to see what this would mean in a visualization context. It should, however, be noted that basically all of the above mentioned structurally dynamic languages are research languages, and that the theory for such system is still under development in the community.

Another extension aspect is to extend equation-based language simulations with an external environment, as depicted at the top of Figure 2. In the formalization, the input is defined as:

$$\mathbf{U} = E \times (E \to \overline{U}) \times (\overline{Y} \times E \to E) \tag{14}$$

As described before, the input $(e_0, f_o, f_t) \in \mathbf{U}$ is a triple, where e_0 is the initial environment, f_o the environment's output function, and f_t the environment's transition function. An example scenario where this can be useful is, for instance, hardware-in-the-loop simulations, where the environment is describing a real hardware platform of a controller (where the IPM simulation is seen as the plant for the controller). Another general scenario is co-simulation [30] where another simulation tool is coupled with the IPM environment via the environment. A standard for co-simulation is FMI (Functional Mock-up Interface) [47]. Coupling an FMU (Functional Mock-up Unit) with the IPM framework can be done as follows (using FMI functions, as defined in [9]):

(1) Within the `compute` function, initialize the FMU and create the initial environment state e_0.
(2) In the `init` function, compute the input values using $f_o(e_0)$, where function f_o calls FMI `get` on relevant variables of the FMU. Save this as IPM input data u in the IPM state.
(3) Within the `while` loop inside the compute function, use u as the input data, if it exists for this time step. Otherwise, compute a new u using $u = f_o(e)$.
(4) Call `output` using u as the input data and save the result in y.
(5) Call function `trans` using u.
(6) Use the output y from step (4) and call $f_t(y, e)$. This will both use FMI `set` functions, and then call FMI `doStep`, which advances the state of the FMU. The result is a new environment state e' which is stored in the IPM state as e.
(7) Go to step (3), as part of the `while` loop.

We can make a few observations from the above procedure. First, we note that we only interact with one FMU and not a set of FMUs. However, a master algorithm (a director of FMUs) can compose and encapsulate a set of FMUs and create one single FMU. Second, the latest version of the FMI standard (FMI 2.0.1 [47] page 106) explicitly states that it is not allowed to call FMI `get` directly after calling `set` without calling `doStep` in between. This requirement is fulfilled in step (6) above because the FMI `set` is followed by a `doStep`. This step in between `set` and `get` also breaks the circular loop between the IPM components and the FMU components.

4 PROBABILISTIC PROGRAMMATIC MODELING

In this section, we discuss other domains that can be suitable for the IPM approach. In particular, we discuss the domain of probabilistic programming and how such programmatic models can benefit from the proposed approach.

4.1 Probabilistic Programming with Finite Number of Variables

Probabilistic programming is an emerging paradigm, where computer programs are used for defining probabilistic models. There exists a large number of probabilistic programming languages (PPLs) [4, 48, 63], where most of them are research efforts. In this section, we exemplify programmatic modeling in PPLs by using the probabilistic programming language WebPPL [31], which is embedded into JavaScript.

In this example, we create a Bayesian model, by following the example developed by George *et al.* [26], which is commonly given as an example in PPL online documentation [21] and in introductory texts [59]. The example

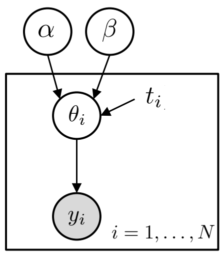

Fig. 6. A probabilistic model in the form of a Bayesian network using plate notation.

models 10 power plant pumps. Besides the model, the measured input data that is given for each pump contains the number of failures and time of operations. The model has two hidden hyperparameters, as well as one hidden variable for each pump, which states its failure rate. The model and its input data can be described in WebPPL as follows:

```
var t = [94.3, 15.7, 62.9, 126., 5.24, 31.4, 1.05, 1.05, 2.1, 10.5]
var y = [5, 1, 5, 14, 3, 19, 1, 1, 4, 22]
var N = t.length

var plate = function(alpha, beta, i){
  if(i < N){
    var theta = sample(Gamma({shape: alpha, scale: beta}))
    var delta = theta * t[i]
    observe(Poisson({mu:delta}), y[i])
    plate(alpha,beta,i+1)
  }
}

var pump = function(){
  var alpha = sample(Exponential({a:1.0}))
  var beta = sample(Gamma({shape:0.1, scale:1.0}))
  plate(alpha, beta, 0)
  return alpha
}
```

We can note that although this is a probabilistic model, it is encoded as a normal program, with functions and the use of recursion. The main model (function) is `pump`. There are two expressions `sample` that draw from the Exponential and the Gamma distributions, respectively. These two new hidden variables `alpha` and `beta` are inferred during the analysis.

The third line in the `pump` function calls the `plate` function, which is a recursive function. This function iterates over all 10 pumps, and observes the measured values of the failure rate. The hidden variables can then be *inferred* by using an approximative method, such as a Markov chain Monte Carlo (MCMC) method.

A graphical visualization of this model is given in Fig. 6. Since this probabilistic program has a finite number of variables (which is not always the case), the programmatic model can be visualized as a Bayesian network. In this kinds of Bayesian network, a standard *plate* notation is used to describe an array of variables, in this case N number of such indexed variables.

If this PPL problem is encoded into the IPM framework, only the $\texttt{init}_m$ and $\texttt{output}_m$ functions are used, not the $\texttt{trans}_m$ (transition) function. In this specific model, there is no notion of time, only an initialization phase (defining the priors) and inference of the hidden variables (the states). Note that array `t` states the *time of operation* for each of the pumps, which is used as part of the observation of the failure rate, but has nothing to do with a time series of data in the sense as described in Section 3. As a consequence, the whole inference engine is encoded into the init function $\texttt{init}_m \colon \overline{U} \times \overline{P}_c \to S$, assuming $\mathbf{U} = \overline{U}$. The `output` function is still used for generating the output data, but no transition is taking place. Note also that the output data variable values are not scalars, but distributions. In this case, the input $u \in \overline{U}$ is the input data that is observed (see arrays `t` and `y` in the programmatic model). The set of computation parameters $p \in \overline{P}_c$ is the parameter values to the prior distributions for `alpha` and `beta` (parameters to the exponential and gamma distributions). Note, for instance, that if these parameters are changed, the inference needs to be recomputed, but the model does not have to be recompiled. Likewise, if the actual observed data changes, the model is not recompiled. However, if the number of pumps changes (the value N), the model needs to be recompiled. This parameter can be viewed as a structural parameter.

4.2 Probabilistic Programming with Unbounded Number of Variables

The previous example is a PPL model with fixed number of latent variables (described using the `sample` statement) and a fixed number of observations. However, probabilistic programming can also be used online, where inferred states change over time. In such a scenario, the complete IPM framework is used in the same way as in Section 3.

As a second example, suppose a country is developing a new aircraft that must be able to operate and estimate accurate positions (correct geographic coordinates), without using a satellite system, such as GPS. Suppose the aircraft has an elevation map over the terrain that it is flying over and that it can measure the distance between the ground and the plane using an altimeter. A probabilistic program can then be encoded as follows. At a specific point in time t_i it samples the distance to the ground, and then, based on prior information, infers an estimate of the aircraft's position. Then, at the next time point t_{i+1}, another sample is taken from the altimeter (for instance 10 ms later) and a new position is estimated. By repeating this procedure, the inference machinery receives more information, and can make better estimates.

In this second example, all functions (`compile`, `init`, `trans`, and `output`) are involved. This is also a good example where it would be applicable to incorporate an environment to generate the input, similar to what is explained in Section 3.3. In the `init` function, the compiled PPL would sample the position initially using some prior distribution, e.g., a uniform distribution over the map. Then, in each transition step (in the `trans` function), the environment is observed (function $f_o(e)$) which in this example means that we sample the altimeter. Then in the transition, we sample a new position, by using the information about the speed of the aircraft. The next time `trans` is called, a new value is observed, and a new sample is generated for the next transition. The underlying inference algorithm suitable for this kind of model is Sequential Monte Carlo (SMC), where a so called resample step takes place at the observation (in this case in the `trans` function).

There are several challenges when encoding PPLs in the IPM framework. One aspect is determinism. A probabilistic program is probabilistic and not deterministic. However, given a prior distribution, some algorithms (including SMC) converge to the appropriate posterior given enough computation power. In the case of SMC, it converges when the number of particles goes towards infinity. Still, practically, this means that every time the IPM framework is executed, a slightly different solution is generated. However, this is not a fundamental problem for the framework as such. Recomputations are needed if a structural parameter or the input environment changes.

But, if only the output function or the time horizon is changing, prior state estimates can be used in the same way as in the EOO examples.

Another challenge is the graphical visualization. For finite Bayesian networks, there is a natural visualization, but how do we visualize arbitrary probabilistic programs? One possibility is to visualize the execution trace as a Bayesian network, as it evolves. In fact, this is actually how state-space and Hidden Markov models typically are visualized. However, there are other domains, such as PPLs for phylogenetics [54], where a domain-specific tree visualization would be more appropriate. Hence, a potential approach might be to focus on certain domain-specific probabilistic programming languages, and to design specific visualizations that are applicable in these domains.

5 DISCUSSION

In this section, we discuss to what extent the modeling problems in Section 1 can be solved with the proposed IPM framework. This is followed by a brief discussion of the generality of the approach and the treat of validity of the conclusions.

5.1 Modeling environment problems

In the introduction, we introduce three problems with the traditional approaches of modeling environments. A relevant question then concerns to what extent we can claim that the IPM framework actually solves these problems.

The recomputation problem: We have theoretically discussed and argued for how different parameters and input data can be used to trigger recomputations only for needed parts. To some extent, tools such as Dymola avoid recompilation if only parameters that affect the computations are changed. However, as stated in the original problem, these tools use a tree-based view, where all output variables are specified and always computed. In contrast, in the proposed approach only necessary output variables are computed in a lazy manner. This can have a large impact on computation performance, since models, such as Modelica models, can contain hundreds of thousands of unknown output variables. We contend that many of these existing tools can partially be extended with the IPM approach. For instance, even if a Modelica tool does not use automatic layout (the graphical annotations are part of the standard), there is no requirement of using a tree-based selection view and to compute all output values. Output variables could be selected directly in the graphical user interface, as described in the paper, and therefore overcome the current limitation.

The variable inspection problem: The proposal is to use the same abstraction that is used for the model to select parameters, not another abstraction, such as a tree. This becomes even clearer when someone wants to measure something between two components in a model. For instance, assume that a user wants to measure the voltage drop over two components in an electrical circuit. In the tree-based view, this not directly possible, since the potential is always given relative to some fixed reference point. By introducing the configurable `output` function, it is possible to compute different outputs, depending on what is requested. Note that there are a quadratic number of possible measurement combinations when measuring between two points. We can also observe that many modeling languages have a natural hierarchy of components, which is one of the reasons that some tools use a tree-based output view. However, the IPM framework only uses sets of identifiers to denote parameters and variables. Hence, the hierarchical mapping, if it exists, must be preserved during the graphical generation. We should also note that a pure hierarchical approach breaks down if the system is structurally dynamic, and certain variables can become present and absent during simulation time.

The model expressiveness problem: It is not possible to argue that a textual code is formally more expressive than a graphical model: a graphical model can also contain text, which means that it can be as expressive. Hence, expressiveness becomes a subjective matter that is hard to measure, and can probably only be judged empirically by conducting user studies. Hence, the reason for why textual modeling may be perceived as more expressive (or more intuitive for more complex modeling problems) is based on arguments, without scientific proofs. A common argument is that visual programming is good for simple tasks, but that it does not scale for more complex tasks. For instance, popular visual programming tools, such as Scratch[4], are good for simple tasks and for learning programming for beginners. However, graphical syntax does not necessarily make it easier to reason about a program. From a more philosophical perspective, we often say that a picture says more than a thousand words. This might be true, but can a picture give a meaning as precise and unambiguous as a sentence with words? During the past decades, there have been several attempts of developing various forms of visual programming and modeling languages, where one larger effort is Executable UML. Although it has received certain attraction in the domain of model-driven architecture, text-based programming languages have dominated since the early development of programming languages and compilers. It is possible to express complex tasks in a graphical modeling environment: environments and languages such as SCADE, UML, and Modelica have shown various ways to express complex models graphically. But it does not mean that it is intuitive when developing more complex models. As can be seen from this discussion, it is not that easy to draw any strong conclusions, more than that text based programming is without question dominating in the software industry, whereas graphical modeling is more accepted in modeling languages targeting more classic engineering disciplines, such as electrical and mechanical engineering.

5.2 Generality of the Approach

Another important question is how general the approach is. That is, which other domains and languages can potentially be encoded in the conceptual framework? Because the interfaces and the approach are quite general, we contend that the approach is quite flexible and can be used in many settings. For instance, within the Miking project [8], we have recently been prototyping an IPM framework for simple domain-specific languages specifying deterministic finite automata (DFA). In this case, the main motivation is to develop a simple learning tool when teaching the foundation of regular expressions and DFAs in compiler courses. In this specific case, the visualization could easily be implemented using the Graphviz tool.

Other domains where the approach may be applicable is verification and simulation of formal timed systems. For instance, a timed automata is a finite automata that also has clock variables. It is a subset of the more general form of hybrid automata [2], where a timed automata is limited to the case where clocks advance with the same rate. In environments such as Uppaal [40], modeling is done graphically, where both properties of safety and liveness can be checked on the automata, and where the automata can be simulated to study its behavior. Timed automata ought to be another domain that fits well with the proposed framework. The framework does not provide anything new in terms of the formal verification procedure. However, textual modeling, graphical visualization of the automata, as well as simulation of the time series data (simulating one potential trace) fit the approach potentially well.

Another relevant question is if the approach is also applicable to modeling of software systems, as a way to visualize and debug programs. An interesting direction to explore could be to combine this approach with RR debugging [55], that is, debugging where the whole trace of an execution of a program is recorded. Given the recorded program as input, an interesting direction would be to use a variant of this framework to visualize the call graph of a program, and to show within the graph the internal states of the different functions, at different points in time. Although speculative, it could be another interesting direction to explore in the future.

5.3 Threat of Validity

This paper is an invited positioning paper, rather than a traditional research paper. This means that the theory, explanations, and conclusions are based on experiences rather than concrete proofs or implementations. There

[4]https://scratch.mit.edu/

is of course a risk that parts of the proposed framework are more complicated than expected, or that a user of the framework needs to side step because the framework is too rigid. For these reasons, it is important to remember that the aim of the framework is to structure the thoughts for someone implementing an IPM system or something similar. For instance, the aim is not that a developer implements the functions with the signatures exactly as described in the paper. Instead, the conceptual frameworks should be seen as a guideline for structuring thoughts and to highlight main design principles.

The risk of incorrect assumptions regarding the interface semantics for the `init`, `trans`, and `output` functions are rather small, since the mapping to EOO and PPL languages are quite straightforward. However, the graphical visualization aspects are far from trivial or clear. In particular, it is not obvious how to visualize large models using structural dynamic semantics or larger models. As discussed earlier, a potential direction is to focus on domain-specific languages, where the graphical representation becomes domain specific as well. Since the claims regarding visualization are very low, the threat of validity is also low.

Finally, it can be questioned if it is fair to compare a conceptual framework with available software tools, such as various Modelica tools. However, there is not a clear border line between the IPM approach and other existing approaches. As discussed earlier, some of the ideas proposed in this paper can potentially be incorporated in existing tools, to improve the user experience.

6 RELATED WORK

The proposed framework and its formalization is new, but many of the ideas for this work are based on previous work. This section gives a brief summary of the closest related work.

The work about *pragmatics* for model-based design by Furhmann and von Hanxleden [24] is without question the closest related work. In their paper, and following up work on the KIELER project[5], they emphasize the role of the MVC pattern, and how the intersection applies to the pragmatics of graphical model-based design. As discussed earlier in our paper, the MVC pattern also naturally fits in our framework, although the use of the model differs (in Furhmann and von Hanxleden's paper, the simulation engine is placed in the controller). Likewise, and inspired by the KIELER project, automatic layout algorithms are used to visualize and filter the graphical model. The same group has also developed a synchronous graphical formalism called SCCharts [61], which makes use of pragmatics and automatic layout. Other work in this direction envision multi-view modeling in general, and usage-specific views in particular [62]. Although there are many similarities in the fundamental ideas, there are a number of differences. Firstly, their work has focused on synchronous programs and variants of Statecharts [33], whereas our conceptual framework is general, and discussed in the context of two completely different domains: equation-based object-oriented modeling (acausal) and probabilistic programming. Moreover, the main contribution of this work is a formal framework that puts the different aspects together, with an emphasize on different sets of parameters, and how it affects recomputations. Also, the discussion on how and where variable selection should be done is new in our work.

There is a rather large community working on live programming [58]. Live programming means that the user gets instant feedback when the software is changed. Many of the ideas date back to Squeak/Smalltalk [36], and recent studies [38] of live programming has been performed using Pharo, a language and environment derived from Squeak/Smalltalk. The concepts advocated in this article can without questions be categorized as a form of live programming environment, where the focus is on programmatic models, and not on normal software programs, which is mostly discussed in the live programming literature. Recently, there has been work on multi-paradigm modeling in relation to live programming [60]. The focus of this work has been on finite state machines and block diagrams, and not on equation-based or probabilistic modeling, as discussed in our paper.

[5]https://www.rtsys.informatik.uni-kiel.de/en/research/kieler/

Tanimoto has categorized live programming into several different levels, first into four distinct levels [57], and then later extended into seven levels [58]. Based on these two categorizations, the aim for the IPM framework is to be at level 3, which means that feedback is incrementally given automatically if any part of the model (or the parameters) are changed. Higher levels in Tanimoto's categories mean that programs can change during execution, which is outside the scope of the IPM framework.

There are many previous ideas about interactive programming, and making programming and modeling more interactive. One early concept called *Illustrative Programming* was introduced and discussed by Fowler [22]. Specifically, he emphasizes the idea of combining the execution of the program with its definition, and gives a spreadsheet program as an example. The main difference between illustrative programming and the IPM framework is that we emphasize different views (textual modeling, graphical visualization), whereas Fowler advocates the same view for both the execution and the definition of the program.

Within the object-oriented programming paradigm, a very large body of work exists concerning graphical modeling of software components, where the Unified Modeling Language (UML) is the dominating standard. Within the functional programming paradigm, significantly less has been done in regards to graphical modeling. The most comprehensive work so far is likely the work on drawing programs by Addis and Addis [1]. All these previous works focus on software modeling and visual programming, whereas our framework focuses on modeling languages for engineering and scientific applications in general, and mathematical modeling in particular.

Many languages, tools, and environments for model-based design make use of both graphical and textual modeling. However, either only textual modeling is advocated, as done in hardware description languages such as VHDL and Verilog, in system-level modeling languages such as SystemC [5] that is based on C++, or variants of synchronous languages [6, 32]). Alternatively, graphical modeling is the central paradigm, for instance in Ptolomy II [20, 52], Simulink [44], and LabVIEW [37]. Some synchronous languages and environments can be used both for graphical and textual modeling/programming. For instance, the SCADE 6 [17] environment allows both textual and graphical models for both block diagrams and state machines. Some languages and environments, such as Modelica environments (e.g. OpenModelica [23] and Dymola [19]) use a combination of graphical and textual modeling. In these tools, the user can edit the model both using text or via the graphical user interface. All editing is done by the user, and the graphical layout is stored as annotations in the textual model. The currently available tools may of course in the future be extended with automatic layout functionality, thus getting closer to the IPM approach proposed in this paper. Most probabilistic programming languages, such as Pyro [4], Anglican [63], WebPPL [31], Stan [15], and Birch [48] are textual languages, where the programmer explicitly states the input data and what the output should be. We contend that these kinds of languages can benefit from development environments that have the properties discussed in this paper.

The ideas for this work has evolved over time, especially during the development of the Modelyze [13] meta-programming environment. The IPM framework is one of the central components of the new initiative of Miking [8], a framework for developing efficient compilers for domain-specific languages.

7 CONCLUSIONS

This paper introduces a conceptual framework called interactive programmatic modeling. The key idea of the framework is to advocate textual modeling (to enable conciseness and expressiveness), but at the same time automatically generate graphical views of the same model. We say that the model is programmatic because its meaning is described as a computer program, which may include variables, higher-order functions, recursion, if-statements etc. Several aspects of the proposed framework have been discussed and introduced in previous work. The main contribution of this paper is to put several of these aspects together in a simple, yet formal, conceptual framework.

ACKNOWLEDGMENTS

This work was partially financially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, by the Swedish Foundation for Strategic Research (Grant No. FFL15- 0032), by Digital Futures, and by the Swedish Research Council (Grant No. 2018-04329). I would like to thank the anonymous reviewers, Elias Castegren, and Oscar Eriksson for detailed feedback and comments on the paper. I would also like to thank Anders Nilsson, Calin Capitanu, Moa Andersson for their initial effort of practically exploring the approach proposed in this paper.

REFERENCES

[1] Tom Addis and Jan Addis. 2009. *Drawing programs: the theory and practice of schematic functional programming.* Springer Science & Business Media.

[2] Rajeev Alur, Costas Courcoubetis, Thomas A Henzinger, and Pei-Hsin Ho. 1993. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid systems*. Springer, 209–229.

[3] Albert Benveniste, Benoît Caillaud, and Mathias Malandain. 2020. *The Mathematical Foundations of Physical Systems Modeling Languages.* Research Report RR-9334. Inria. 112 pages. https://hal.inria.fr/hal-02521747

[4] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. 2019. Pyro: Deep Universal Probabilistic Programming. *Journal of Machine Learning Research* 20, 28 (2019), 1–6.

[5] David C Black, Jack Donovan, Bill Bunton, and Anna Keist. 2009. *SystemC: From the ground up.* Vol. 71. Springer Science & Business Media.

[6] Timothy Bourke and Marc Pouzet. 2013. Zélus: a synchronous language with ODEs. In *Proceedings of the 16th International Conference on Hybrid Systems: Computation and Control (HSCC).* ACM, 113–118.

[7] David Broman. 2010. *Meta-Languages and Semantics for Equation-Based Modeling and Simulation.* Ph.D. Dissertation. Department of Computer and Information Science, Linköping University, Sweden.

[8] David Broman. 2019. A Vision of Miking: Interactive Programmatic Modeling, Sound Language Composition, and Self-Learning Compilation. In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering (SLE '19).* ACM, 55–60.

[9] David Broman, Christopher Brooks, Lev Greenberg, Edward A. Lee, Michael Masin, Stavros Tripakis, and Michael Wetter. 2013. Determinate Composition of FMUs for Co-Simulation. In *Proceedings of the International Conference on Embedded Software (EMSOFT).* IEEE.

[10] David Broman and Peter Fritzson. 2009. Higher-Order Acausal Models. *Simulation News Europe* 19, 1 (2009), 5–16.

[11] David Broman and Henrik Nilsson. 2012. Node-Based Connection Semantics for Equation-Based Object-Oriented Modeling Languages. In *Proc. of the Fourteenth International Symposium on Practical Aspects of Declarative Languages (PADL 2012) (LNCS)*, Vol. 7149. Springer-Verlag, 258–272.

[12] David Broman and Jeremy G. Siek. 2012. *Modelyze: a Gradually Typed Host Language for Embedding Equation-Based Modeling Languages.* Technical Report UCB/EECS-2012-173. EECS Department, University of California, Berkeley.

[13] David Broman and Jeremy G Siek. 2018. Gradually Typed Symbolic Expressions. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM).* ACM, 15–29.

[14] Benoît Caillaud, Mathias Malandain, and Joan Thibault. 2020. Implicit structural analysis of multimode DAE systems. In *Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control.* 1–11.

[15] Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A probabilistic programming language. *Journal of Statistical Software* 76, 1 (2017).

[16] François E. Cellier. 1991. *Continuous System Modeling.* Springer-Verlag, New York, USA.

[17] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. 2017. Scade 6: A formal language for embedded critical software development. In *International Symposium on Theoretical Aspects of Software Engineering (TASE).* IEEE, 1–11.

[18] Fabio Cremona, Marten Lohstroh, David Broman, Edward A Lee, Michael Masin, and Stavros Tripakis. 2017. Hybrid co-simulation: it's about time. *Software & Systems Modeling* (2017), 1–25.

[19] Dassault Systems. [n.d.]. DYMOLA Systems Engineering: Multi-Engineering Modeling and Simulation based on Modelica and FMI. http://www.dymola.com [Last accessed: September 9, 2020].

[20] Johan Eker, Jorn Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Sonia Sachs, and Yuhong Xiong. 2003. Taming heterogeneity - the Ptolemy approach. *Proc. IEEE* 91, 1 (January 2003), 127–144.

[21] OpenBugs Online Examples. [n.d.]. Pumps: conjugate gamma-Poisson hierarchical model. http://www.openbugs.net/Examples/Pumps.html [Last accessed: February 2, 2020].

[22] Martin Fowler. 2009. Illustrative Programming. https://martinfowler.com/bliki/IllustrativeProgramming.html. [Last accessed: June 28, 2019].

[23] Peter Fritzson, Peter Aronsson, Håkan Lundvall, Kaj Nyström, Adrian Pop, Levon Saldamli, and David Broman. 2005. The OpenModelica Modeling, Simulation, and Software Development Environment. *Simulation News Europe* 15, 44/45 (2005), 8–16.

[24] Hauke Fuhrmann and Reinhard von Hanxleden. 2010. On the Pragmatics of Model-Based Design. In *Proceedings of the 15th Monterey Workshop 2008 on the Foundations of Computer Software. Future Trends and Techniques for Development (LNCS)*, Vol. 6028. Springer, 116–140.

[25] Sébastien Furic. 2009. Enforcing model composability in Modelica. In *Proceedings of the 7th International Modelica Conference.* Linköping University Electronic Press, 868–879.

[26] Edward I George, UE Makov, and AFM Smith. 1993. Conjugate likelihood distributions. *Scandinavian Journal of Statistics* (1993), 147–156.

[27] George Giorgidze and Henrik Nilsson. 2008. Embedding a Functional Hybrid Modelling Language in Haskell. In *Proceedings of the 20th International Symposium on the Implementation and Application of Functional Languages.*

[28] George Giorgidze and Henrik Nilsson. 2009. Higher-Order Non-Causal Modelling and Simulation of Structurally Dynamic Systems. In *Proceedings of the 7th International Modelica Conference (Linköping Electronic Conference Proceedings).* Linköping University Electronic Press, Como, Italy, 208–218.

[29] George Giorgidze and Henrik Nilsson. 2011. Mixed-level Embedding and JIT Compilation for an Iteratively Staged DSL. In *Proceedings of the 19th Workshop on Functional and (Constraint) Logic Programming (WFLP 2010) (LNCS)*, Julio Mariño (Ed.), Vol. 6559. Springer-Verlag, 48–65.

[30] Cláudio Gomes, Casper Thule, David Broman, Peter Gorm Larsen, and Hans Vangheluwe. 2018. Co-simulation: a survey. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1–33.

[31] Noah D Goodman and Andreas Stuhlmüller. 2014. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org. Accessed: 2018-10-02.

[32] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. 1991. The Synchronous Data Flow Programming Language LUSTRE. *Proc. IEEE* 79, 9 (1991), 1305–1320.

[33] David Harel. 1987. Statecharts: A visual formalism for complex systems. *Science of Computer Programming* 8, 3 (1987), 231–274.

[34] Alan C. Hindmarsh, Peter N. Brown, Keith E. Grant, Steven L. Lee, Radu Serban, Dan E. Shumaker, and Carol S. Woodward. 2005. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Programming Languages and Systems* 31, 3 (2005), 363–396.

[35] Christoph Höger. 2019. *Compiling Modelica : about the separate translation of models from Modelica to OCaml and its impact on variable-structure modeling.* Doctoral Thesis. Technische Universität Berlin, Berlin.

[36] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. 1997. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA).* ACM, New York, NY, USA, 318–326.

[37] National Instruments. [n.d.]. What is LabView? - National Instruments. https://www.ni.com/labview/ [Last accessed: January 8, 2020].

[38] Juraj Kubelka, Romain Robbes, and Alexandre Bergel. 2018. The road to live programming: insights from the practice. In *Proceedings in the 40th International Conference on Software Engineering (ICSE).* IEEE, 1090–1101.

[39] Peter Kunkel and Volker Mehrmann. 2006. *Differential-Algebraic Equations Analysis and Numerical Solution.* European Mathematical Society.

[40] Kim G Larsen, Paul Pettersson, and Wang Yi. 1997. UPPAAL in a nutshell. *International journal on software tools for technology transfer* 1, 1-2 (1997), 134–152.

[41] Edward A Lee. 2016. Fundamental Limits of Cyber-Physical Systems Modeling. *ACM Transactions on Cyber-Physical Systems* 1, 1 (2016), 3.

[42] Edward A Lee and Haiyang Zheng. 2005. Operational semantics of hybrid systems. In *International Workshop on Hybrid Systems: Computation and Control.* Springer, 25–53.

[43] MathWorks. [n.d.]. Simscape - Model and simulate multidomain physical systems. https://www.mathworks.com/products/simscape [Last accessed: February 1, 2020].

[44] MathWorks. [n.d.]. The Mathworks - Simulink - Simulation and Model-Based Design. http://www.mathworks.com/products/simulink/ [Last accessed: January 8, 2020].

[45] Sven Erik Mattsson and Gustaf Söderlind. 1993. Index reduction in differential-algebraic equations using dummy derivatives. *SIAM Journal on Scientific Computing* 14, 3 (1993), 677–692.

[46] Modelica Association 2017. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling - Language Specification Version 3.4.* Modelica Association. Available from: http://www.modelica.org.

[47] MODELISAR Consortium and Modelica Association. October 2, 2019. Functional Mock-up Interface for Model Exchange and Co-Simulation – Version 2.0.1. Retrieved from https://www.fmi-standard.org.

[48] Lawrence M Murray and Thomas B Schön. 2018. Automated learning with a probabilistic programming language: Birch. *Annual Reviews in Control* 46 (2018), 29–43.

[49] Christoph Nytsch-Geusen, Thilo Ernst, André Nordwig, Peter Schneider, Peter Schwarz, Matthias Vetter, Christof Wittwer, Andreas Holm, Thierry Nouidui, Jürgen Leopold, et al. 2005. MOSILAB: Development of a Modelica based generic simulation tool supporting model structural dynamics. In *Proceedings of the 4th International Modelica Conference TU Hamburg-Harburg*, Vol. 2.

[50] Constantinos C. Pantelides. 1988. The Consistent Initialization of Differential-Algebraic Systems. *SIAM J. Sci. Statist. Comput.* 9, 2 (1988), 213–231.

[51] John D Pryce. 2001. A simple structural analysis method for DAEs. *BIT Numerical Mathematics* 41, 2 (2001), 364–394.

[52] Claudius Ptolemaeus (Ed.). 2014. *System Design, Modeling, and Simulation using Ptolemy II.* Ptolemy.org. http://ptolemy.org/books/Systems

[53] Trygve Reenskaug. 1979. MODELS - VIEWS - CONTROLLERS. Xerox PARC technical note December 1979, https://folk.uio.no/trygver/themes/mvc/mvc-index.html [Last accessed: February 2, 2020].

[54] Fredrik Ronquist, Jan Kudlicka, Viktor Senderov, Johannes Borgström, Nicolas Lartillot, Daniel Lundén, Lawrence Murray, Thomas B. Schön, and David Broman. 2020. Universal probabilistic programming: a powerful new approach to statistical phylogenetics. *bioRxiv* (2020). https://doi.org/10.1101/2020.06.16.154443

[55] rr development team. 2020. rr: lightweight recording & deterministic debugging. https://rr-project.org. [Last accessed: September 11, 2020].

[56] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. 1981. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics* 11, 2 (1981), 109–125.

[57] Steven L Tanimoto. 1990. VIVA: A visual language for image processing. *Journal of Visual Languages & Computing* 1, 2 (1990), 127–139.

[58] Steven L Tanimoto. 2013. A perspective on the evolution of live programming. In *Proceedings of the 1st International Workshop on Live Programming*. IEEE Press, 31–34.

[59] Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. 2018. An introduction to probabilistic programming. *arXiv preprint arXiv:1809.10756* (2018).

[60] Yentl Van Tendeloo, Simon Van Mierlo, and Hans Vangheluwe. 2019. A Multi-Paradigm Modelling approach to live modelling. *Software & Systems Modeling* 18, 5 (2019), 2821–2842.

[61] Reinhard Von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien. 2014. SCCharts: sequentially constructive statecharts for safety-critical applications: HW/SW-synthesis for a conservative extension of synchronous statecharts. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 372–383.

[62] Reinhard Von Hanxleden, Edward A Lee, Christian Motika, and Hauke Fuhrmann. 2012. Multi-view modeling and pragmatics in 2020. In *Monterey Workshop (LNCS)*, Vol. 7539. Springer, 209–223.

[63] Frank Wood, Jan Willem Meent, and Vikash Mansinghka. 2014. A New Approach to Probabilistic Programming Inference. In *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research)*, Samuel Kaski and Jukka Corander (Eds.), Vol. 33. PMLR, Reykjavik, Iceland, 1024–1032.

[64] Dirk Zimmer. 2008. Introducing Sol: A General Methodology for Equation-Based Modeling of Variable-Structure Systems. In *Proceedings of the 6th International Modelica Conference*. Bielefeld, Germany, 47–56.