Function+Data Flow: A Framework to Specify Machine Learning Pipelines for Digital Twinning

Eduardo de Conto, Blaise Genest, Arvind Easwaran

Function+Data Flow: A Framework to Specify Machine Learning Pipelines for Digital Twinning

Eduardo de Conto
Nanyang Technological University
Singapore, Singapore
CNRS@CREATE
Singapore, Singapore
eduardo002@e.ntu.edu.sg

Blaise Genest
IPAL
Singapore, Singapore
CNRS, CNRS@CREATE
Singapore, Singapore
blaise.genest@cnrsatcreate.sg

Arvind Easwaran
Nanyang Technological University
Singapore, Singapore
arvinde@ntu.edu.sg

ABSTRACT

The development of digital twins (DTs) for physical systems increasingly leverages artificial intelligence (AI), particularly for combining data from different sources or for creating computationally efficient, reduced-dimension models. Indeed, even in very different application domains, twinning employs common techniques such as model order reduction and modelization with hybrid data (that is, data sourced from both physics-based models and sensors). Despite this apparent generality, current development practices are ad-hoc, making the design of AI pipelines for digital twinning complex and time-consuming. Here we propose *Function+Data Flow (FDF)*, a domain-specific language (DSL) to describe AI pipelines within DTs. FDF aims to facilitate the design and validation of digital twins. Specifically, FDF treats functions as first-class citizens, enabling effective manipulation of models learned with AI. We illustrate the benefits of FDF on two concrete use cases from different domains: predicting the plastic strain of a structure and modeling the electromagnetic behavior of a bearing.

CCS CONCEPTS

• **Software and its engineering → Orchestration languages**; *Visual languages*; *Data flow languages.*

KEYWORDS

digital twins, machine learning pipeline, dataflow

ACM Reference Format:
Eduardo de Conto, Blaise Genest, and Arvind Easwaran. 2024. Function+Data Flow: A Framework to Specify Machine Learning Pipelines for Digital Twinning. In *Proceedings of the 1st ACM International Conference on AI-Powered Software (AIware '24), July 15–16, 2024, Porto de Galinhas, Brazil.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3664646.3664759

1 INTRODUCTION

Digital twins (DTs) are rapidly emerging as a transformative technology for complex systems across diverse industries [17]. Examples include applications in smart grids and smart cities [10, 19, 34], manufacturing [16, 21, 24] and aviation [32, 33, 36]. The DT market size is projected to grow to US$ 180 - 250 billion by 2032 [13, 15].

The overall ambition of DTs is to represent a physical system over its entire lifespan virtually. To achieve this, twinning leverages simulation and artificial intelligence (AI) for reasoning and decision-making. In addition, a DT can be updated with data to maintain fidelity with the physical counterpart. Despite this ambition, current practices often rely solely on virtual prototypes [12] instead of DTs. These prototypes are typically created using finite element (FE) modeling, computer-aided design (CAD), or computational fluid dynamics (CFD) frameworks, and they allow predicting (accurately, albeit slowly) the nominal behavior of the system before its physical version is even built.

As per Grieves [17], a virtual prototype can be evolved in three phases of increasing complexity to give rise to DTs. Firstly, a digital twin prototype (*DTP*) is obtained from the virtual prototype using reduced order modeling (ROM) or other techniques. This enables several orders of magnitude faster simulations than the CAD/CFD models [18, 27]. Secondly, a digital twin instance (*DTI*) is created by modeling one particular instance of a physical system. This uses historical data from sensors placed on that instance, tuning and adapting (offline) the DTP to account for its deviations from the prototype (e.g., manufacturing errors and impact from operating conditions). Thirdly, the loop between the physical system and the DTI can be closed by updating (online) the latter using real-time sensor data and controlling the actual instance. This paper will focus on the first two phases, i.e., the offline design of DTPs and DTIs using ROM and deviation models, respectively. The online exploitation of DTIs also benefits from the methodologies presented in this work (see Section 6). However, additional advancements specially tailored for real-time update and control are also needed: these advancements will be explored in future research.

Machine learning (ML), a subfield of AI, plays a key role in the design of DTPs and DTIs. To obtain DTPs, while established ROM techniques, such as proper generalized decomposition [9], can reduce the number of physical variables required to model the system by several orders of magnitude, no physics-based model can work directly on the reduced basis. ML addresses this by enabling the creation of real-time models that operate on the reduced basis. These ML models are trained using simulations generated by the original, slower physics-based model. On the other hand, developing a DTI requires integrating data from one particular system instance and comparing it with the nominal DTP model. ML plays a crucial role in this process as well. Two main approaches are possible. Either (1) ML can be employed to combine the DTP and the specific instance

AIware '24, July 15–16, 2024, Porto de Galinhas, Brazil

ACM ISBN 979-8-4007-0685-1/24/07
https://doi.org/10.1145/3664646.3664759

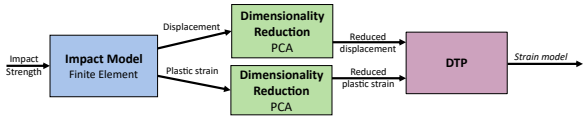

Figure 1: Pipeline for structural health monitoring.

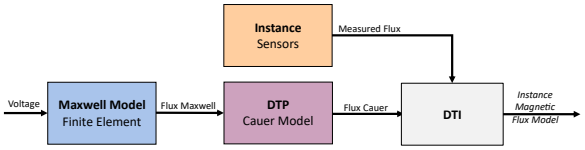

Figure 2: Pipeline to model an active magnetic bearing.

data directly, or (2) ML can be used to create a model of the difference ("ignorance") between the nominal behavior and the actual instance (Hybrid Twins methodology [8]).

We will now describe two *motivating examples* that we will use to illustrate the concepts and steps to obtain a DTP or a DTI in two distinct applications. These are real-world examples based on typical DT applications, as identified in the literature [6, 16].

Structural Integrity Monitoring. The first use-case focuses on structural health monitoring [6], enabling, e.g., predictive maintenance [32]. Here, the goal is to predict the plastic strain of a certain structure given an observed deformation. While currently no (non-destructive) methodology can directly measure the plastic strain, 3D images of the observed deformation can be collected via digital image correlation [2, 31]. To monitor the structural strains, the following pipeline, illustrated in Figure 1, could be used:

(1) Use a (slow) finite element (FE) impact model to simulate different impact strengths and obtain deformation and plastic strain values as output.
(2) Reduce the deformations and plastic strains using principal component analysis (PCA).
(3) Train a DTP (using supervised learning and the above-reduced dataset) to learn a model that predicts the reduced plastic strain from the reduced deformation.

Electromagnetic Bearing Modeling. The second use-case relates to the modeling of an active magnetic bearing [29], facilitating real-time analysis of the device's behavior. The goal is to predict the induced magnetic flux based on the voltage applied to the device. A recent work [16] proposed to combine a slow FE model with a ROM (Cauer) to achieve a fast and accurate pipeline. The following pipeline, depicted in Figure 2, is used:

(1) Leverage an FE model based on the Maxwell Equations to accurately calculate the magnetic flux (*Flux Maxwell*) within the system based on the applied voltage. This model is computationally expensive.
(2) Obtain a faster DTP (denoted as *Cauer Model*) allowing for much faster simulations. This DTP is a model of the magnetic bearing, described as an electric circuit with several branches of resistor and inductance elements. The output of this step is *Flux Cauer*, the magnetic flux induced within the equivalent circuit, a (linear) approximation of Flux Maxwell.
(3) Integrate historical data from a specific magnetic bearing instance (e.g., sensor measurements) to obtain the DTI.

Challenges in Specifying ML Pipelines for Twinning. In conventional data-driven pipelines used in classical ML applications, the objective is to train *one* model to accomplish *one* task. The model does not need to be manipulated, and many operations can be implicit (e.g. the training and the inference of models/functions do not need to be distinguished). Given that the application of ML to DTP/DTI design is a relatively new research area, the development practices, methodologies, and tools, are not yet fully established. Compared to the standard ML pipelines, the following challenges arise:

Distinction Between Training and Inference. In supervised learning, training the model and running inference from the model are different steps of the process. For instance, while the latter (inference) takes input data X (e.g. voltage) and outputs data Y (e.g. predicted flux), the former (training) receives pairs (X, Y), where Y is the ground truth answer from the query X. Note that Figure 2 lacks a clear distinction between *learning* the Cauer Model and *inferring the flux* predicted by the Cauer. Also, the input of each step is implicit.

Function/Model Manipulation. By contrast with standard ML pipelines, twinning requires several models/functions. These functions include projection to a reduced basis, several DTPs, and the DTI itself, where some functions are instrumental in generating others. Hence, operations on these models must be either explicit or they must adhere to a very rigid implicit pipeline.

Pipeline Diversity. Although twinning involves the same basic operations (reduced order modeling and combining data from different sources), the specific pipelines employed heavily depend on the application domain or even the individual application. Figures 1 and 2 illustrate this variability. In the electromagnetic bearing pipeline (Figure 2), both the DTP (Cauer Model) and the FE (Maxwell Model) have the same input/output. However, the structural health monitoring pipeline (Figure 1) demonstrates a situation in which the *input* of the DTP (displacement) is an *output* of the FE impact model. That is, the pipeline is not directly reducing the Impact Model.

Proposal. To address the challenge of *pipeline diversity*, we adopt the visual dataflow paradigm [20], allowing for an intuitive and adaptable pipeline description.

Concerning *function/model manipulation*, we propose a novel Function+Dataflow (FDF). FDF is a domain-specific language (DSL) for the specification of ML pipelines used for DT design. FDF enables the manipulation of functions learned by the ML pipeline. To achieve this, FDF extends the traditional dataflow by incorporating *functions as first-class citizens*. The function is defined as another type of flow besides data streams, as in [14], a generic data-flow programming language, as well as in dataflow-based DSLs for different contexts (specifically, quantum-classical combination [30] and data engineering [11]). This addition allows us to:

- *Decouple* the learning of a function (e.g. how to project data into a PCA reduced basis) and the usage of the function (actual projection of data into the reduced basis).
- *Manipulate* the models explicitly, allowing their use and reuse as necessary. For example, in the structural health monitoring case, we can reuse the reduced basis projections.

- *Infer and track* the input and output data type for each function within the pipeline automatically. This capability allows for implicit type-checking and can offer significant benefits to the users: we can suggest valid inputs to the users, or warn them of potential incompatibilities, avoiding bugs. Notice that actual data types are never required from the user, avoiding a tedious process.

2 RELATED WORKS

Machine Learning Workflow. Recent advancements and the deployment of AI and ML in critical applications have led to a surge of tools to structure ML pipelines (e.g., Kedro [1], MLflow [7], Apache Airflow[4], Kubeflow [22]). These tools allow the description of a machine learning pipeline and, in addition, allow, among other things, tracking experiment results, performing model versioning, and monitoring the model performance. The ML pipeline in these systems is typically represented by directed acyclic graphs.

Despite their advantages, the current tools still require a significant integration effort: they focus on generating a single ML model, which is not explicitly represented in the pipeline [23]. One exception is the Transformers Library [35], but this library only supports predefined pipelines for common tasks in the natural language processing domain (object detection, summarization, etc.). In all cases, the essential task of the workflow is to produce *one* model which can only be recovered and used externally after the learning is completed.

As described in the "Function/Model Manipulation" challenge, the design of DTs involves the creation and manipulation of multiple interrelated models and functions. For instance, the DTP may be needed to create the DTI, etc. A possible way to handle model manipulation within dataflow is to transmit the model as a data token. While technically feasible, this method discards valuable information (e.g. the input/output type of the models).

To overcome these limitations, FDF includes a *dedicated function flow* (used to transmit learned models and functions). Valuable information about the functions learned in the ML workflow can thus be preserved and transmitted. This includes the data types accepted as input and produced as output.

Twin Builders from CAD/CFD tools. Commercial software vendors specializing in CFD/CAD, such as Ansys and Siemens, have incorporated ROM capabilities into their existing software toolkits (e.g., Ansys Twin Builder [3], Siemens Simcenter Amesim [28]). These functionalities typically follow a similar workflow: execute simulations within a CAD/CFD environment and export them to a dedicated ROM module with limited user-controllable parameters. Notably, none of these tools offer an open, dataflow-based pipeline that can be customized for specific use cases. Therefore, the results are often inconsistent and heavily dependent on whether the predefined and proprietary workflows available are suitable for the domain of interest. The core novelty of our methodology is its ability to make the ROM pipeline *fully customizable* using FDF. This flexibility is crucial for adapting the model to diverse application contexts. Importantly, our approach can still offer predefined, yet fully customizable workflow templates.

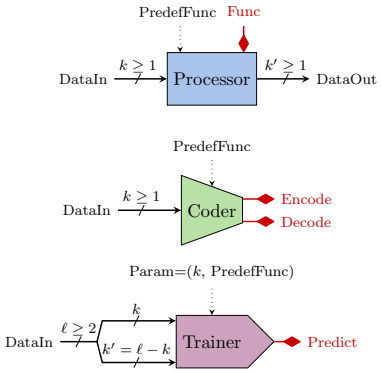

Figure 3: Visual syntax for boxes of Function+Data Flow: Processor on top, Coder in the middle, and Trainer at the bottom. The processor executes either a function *Func* learned by an earlier box in the pipeline or a *predefined* function *PredefFunc*. The value k in the Trainer's Param specifies the number of input ports to consider as X. The remaining ports are the Y in the (X, Y) supervised learning pairs.

3 FUNCTION+DATA FLOW SYNTAX

We now describe the syntax of the Function+Data Flow (FDF) pipeline. We first provide an overview of the rationale and visual syntax of the components of the pipeline, and then formalize it.

3.1 Syntax Overview

An FDF pipeline is composed of boxes (as shown in Figure 3) that represent the different processing steps. Each box has several input and output ports. There are two types of ports: function ports (in red), sending/receiving a single *learned function*, and data ports (in black) sending/receiving a batch of data.

There are three user-specified boxes in FDF, each associated with a different task in the twinning workflow:

- Processor: for typical data processing (including applying functions learned by other boxes),
- Coder: for learning a reduced basis (or unsupervised clustering) and the associated projection and inverse projection, and
- Trainer: for learning a function with supervised ML.

Each box is represented by different polygon shapes, as shown in Figure 3. There are also two implicit (i.e., non-depicted) boxes, namely FuncOut and DataIO, that represent the pipeline's external input/output in terms of functions and data, respectively.

We now outline the syntax of each pipeline box. A summary is provided in Table 1, their visual representation is given in Figure 3 and a description is given in the following.

The *Processor boxes* are represented by a light blue rectangle, as in the top of Figure 3. It has multiple input data ports to receive the data for processing, and multiple output data ports to return the processed data. The function to execute is either provided through an input function port (function learned by the previous boxes) or is a predefined function from a library, as specified by box parameter.

The *Coder boxes* are represented by a pale green trapezoid, as in the middle of Figure 3. It has multiple input data ports to receive

Table 1: Summary of Box Syntax

	Box Type				
Characteristics	Processor	Coder	Trainer	FuncOut	DataIO
Representation	Light blue rectangle	Pale green trapezoid	Pale violet pentagonal	Invisible	Invisible
Application	Data processing	Unsupervised learning	Supervised learning	Function source/sink	Data source/sink
Inputs Data Ports Inputs Function Ports	One or more Zero or one	One or more Zero	Two or more Zero	Zero Zero or more	Zero or more Zero
Output Data Ports Output Function Ports	One or more Zero	Zero One or Two	Zero One	Zero Zero	Zero or more Zero

the data from which to compute the reduced basis and one or two output function ports to return the encoder and decoder functions (i.e., the projection and inverse projection onto the reduced basis, respectively). Coder boxes have no input function port: the specific encoding/decoding algorithm is a predefined function from a library, as specified by the parameter of the box. For instance, "PCA (99%)" specifies a principal component analysis capturing $\geq$ 99% of the variance of the original data.

The *Trainer boxes* are represented by a pale violet pentagon, as in the bottom of Figure 3. It has $\ell \geq 2$ input data ports to receive the supervised (X, Y) pairs, with $k < \ell$ ports providing X and the remaining $k' = \ell - k$ providing Y, and it has one output function port to return a function that predicts Y given X. The number k is provided in the first component of the Trainer box parameter. The Trainer boxes have no input function port: the specific ML training algorithm is provided in the second component of the Trainer box parameter and is a predefined function from a library (e.g. PyTorch). For instance, "NN (50, 50, SGD)" indicates that stochastic gradient descent (SGD) shall be used to learn a feedforward neural network with 2 hidden layers, each with 50 nodes.

Finally, the *implicit boxes (FuncOut and DataIO)* represent the input/output dependencies of the pipeline. *FuncOut* has no input/output data ports, no output function port, but it can accept any number of input function ports. A function is sent to FuncOut to export it to external pipelines. *DataIO* has no input/output function port, and it has any number of data input/output ports. The output ports are sources for the different data batches used by the FDF pipeline. The input ports are data sinks, i.e., they store data, allowing them to be persisted in disk for further analysis.

3.2 Formal Syntax

An FDF pipeline is defined as $P = (\mathcal{B}', boxclass, \mathcal{P}, portclass, box, src, param)$, where:

- $\mathcal{B} = \{b_1, \ldots, b_n\}$ are the user-defined boxes. We denote $\mathcal{B}' = \mathcal{B} \sqcup \{DataIO, FuncOut\}$.
- $boxclass : \mathcal{B} \to \{\text{Processor, Coder, Trainer}\}$ defines the class of each box.
- $param : \mathcal{B} \to \mathcal{L} \cup \mathbb{N} \times \mathcal{L}$ provides the parameters of a given box, where $\mathcal{L}$ is a library of predefined functions.
- $\mathcal{P} = \mathcal{P}^I \sqcup \mathcal{P}^O = \{1, \cdots, m\}$ is the ordered set of natural numbers up to m, the number of ports. It is partitioned into the sets of input ports $\mathcal{P}^I$ and output ports $\mathcal{P}^O$.
- $portclass : \mathcal{P} \to \{\text{Data, Function}\}$ provides the class (Data or Function) of each port.
- $box : \mathcal{P} \to \mathcal{B}'$ is a function associating each port with the box it belongs to.
- $src : \mathcal{P}^I \to \mathcal{P}^O$ is a function associating every input port with the output port providing its data/function. src is such that the class of an input and the associated output ports are the same: $\forall p \in \mathcal{P}^I, portclass(p) = portclass(src(p))$.

3.3 Example

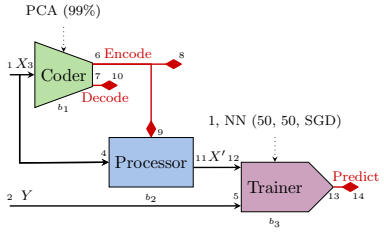

Figure 4: Minimal FDF pipeline with annotations

We now review a minimal FDF pipeline to show the one-to-one correspondence between the visual syntax (Figure 3) and the formal syntax just described. The pipeline is given in Figure 4 and is described in the following:

(1) Use a Coder box with a predefined function "PCA (99%)" to obtain a PCA basis of X with 99% of accuracy,
(2) Use a Processor box executing the learned function "Encode" and retrieve the reduced data X' from (full) X,
(3) Use a Trainer box with a Param "1, NN (50, 50, SGD)" to learn a function *Predict* that predicts Y from X. This function is learned with stochastic gradient descent on a neural network with 2 hidden layers of 50 nodes each.

Here is the formal definition of this FDF pipeline P:

- $\mathcal{B} = \{b_1, b_2, b_3\}$.
- $boxclass(b_1)$ = Coder; $boxclass(b_2)$ = Processor; $boxclass(b_3)$ = Trainer.

- $param(b_1)$ = PCA(99%); $param(b_3)$ = 1, NN (50, 50, SGD)
- $\mathcal{P} = \mathcal{P}^I \sqcup \mathcal{P}^O$, with $\mathcal{P}^I = \{3, 4, 5, 8, 9, 10, 12, 14\}$, $\mathcal{P}^O = \{1, 2, 6, 7, 11, 13\}$.
- $portclass(p) = \begin{cases} (Data), & \text{if } p \in \{1, 2, 3, 4, 5, 11, 12\} \\ (Function), & \text{if } p \in \{6, 7, 8, 9, 10, 13, 14\} \end{cases}$
- $box(p) = \begin{cases} DataIO, & \text{if } p \in \{1, 2\} \\ FuncOut, & \text{if } p \in \{8, 10, 14\} \\ b_1, & \text{if } p \in \{3, 6, 7\} \\ b_2, & \text{if } p \in \{4, 9, 11\} \\ b_3, & \text{if } p \in \{5, 12, 13\} \end{cases}$
- $src(3) = 1; src(4) = 1; src(5) = 2; src(8) = 6; src(9) = 6; src(10) = 7; src(12) = 11; src(14) = 13$.

4 FUNCTION+DATA FLOW SEMANTICS

To define the semantics of an FDF pipeline, we first introduce the directed FDF graph associated with an FDF pipeline, which will define an explicit order in which the FDF pipeline is executed.

4.1 FDF Graph

The FDF directed graph $G(P) = (\mathcal{P}, \mathcal{E})$ associated with an FDF pipeline $P = (\mathcal{B}', boxclass, \mathcal{P}, portclass, box, src, param)$ is defined as follows. For all ports $p, q \in \mathcal{P}$, we have $(p, q) \in \mathcal{E}$ iff either:

- $p \in \mathcal{P}^O, q \in \mathcal{P}^I$ and $p = src(q)$. These are the edges between the boxes, or
- $p \in \mathcal{P}^I, q \in \mathcal{P}^O$ and $box(p) = box(q) \in \mathcal{B}$, which excludes the two implicit boxes $DataIO, FuncOut$. This models the *(complete)* internal dependencies in each box.

Note that output ports associated with the implicit box $DataIO$ will have no predecessor in $G(P)$, and input ports associated with the implicit boxes $DataIO, FuncOut$ will have no successors.

We call the directed graph $G(P)$ and the FDF pipeline P *well-formed* if $G(P)$ is a directed acyclic graph (DAG). To be executable, P needs to be well-formed. We will thus assume in the following that P is well-formed. Note that this can be tested in linear time.

Example. In Figure 4, we have the following edges:

$$\mathcal{E} = \{(1,3), (1,4), (2,5), (6,8), (6,9), (7,10), (11,12), (13,14), (3,6), (3,7), (4,11), (9,11), (5,13), (12,13), (13,14)\}$$

4.2 FDF Execution

Well-formedness allows us to decide the order in which to execute boxes, which boxes may be executed in parallel, and which must be executed before/after another. We say that $b \in \mathcal{B}$ (so not an implicit box) is a direct predecessor of b', denoted $b \lessdot b'$ whenever $\exists (p, q)$ with $p = src(q)$, $box(p) = b$ and $box(q) = b'$. The execution of the box b' is blocked until all the explicit boxes $b \lessdot b'$ have been executed. In addition, a box executes only if the batches for all its input data ports have the same number of samples. The execution semantics depend on the class $boxclass(b)$ of the FDF box b. We describe in the following how each box is executed in FDF.

Processor. Running a Processor box (i.e., $boxclass(b)$ = Processor) with k input data ports and k' output data ports consists of the following steps:

(1) Load the function f to execute and the input data DataIn. The function f is provided either as a predefined function PredefFunc, defined by the parameter, or as an input function port (in which case f is a learned function, the output of a previous Coder or Trainer box of the pipeline). DataIn are batches in the input data ports of b.
(2) Execute f for each sample of the batch DataIn (granted that f accepts a vector of size k as input and produces a vector of size k' as output).
(3) Return the processed DataOut in the k' output data ports of the Processor.

Coder. Running a Coder box (i.e., $boxclass(b)$ = Coder) with two output function ports consists of the following steps:

(1) Load the DataIn from the input data port.
(2) Load the predefined function PredefFunc, provided by the parameter of b, specifying how to obtain the reduced basis from a batch of data.
(3) Run PredefFunc on DataIn to obtain the 2 functions Encode and Decode[1] based on the reduced basis. Return these functions in their corresponding output function port.

Trainer. Running a Trainer box (i.e., $boxclass(b)$ = Trainer) with ℓ input data ports consists of the following steps:

(1) Load $Param = (k, PredefFunc)$ defining a number $k < \ell$ and a predefined function PredefFunc. The number k is used in the next step to distinguish X values from Y values in the (X, Y) supervised learning pairs. PredefFunc specifies how to learn the model from the supervised pairs (X, Y).
(2) Load the required batch of (X, Y) pairs, with X obtained from the first k input data ports, and Y from the last $k' = \ell - k$ input data ports.
(3) Run PredefFunc on the batch of (X, Y) pairs to obtain the function Predict for the generalization of $X \mapsto Y$.

5 IMPLICIT TYPING

In FDF, we can leverage the fact that functions are first-class citizens to infer extra information about them. Here we explain how to automatically infer (implicit) types for the data and functions generated within the pipeline, based on the FDF syntax and semantics.

Implicit typing endows the FDF pipeline with advantages commonly associated with statically typed languages, e.g. being less prone to errors and easier to maintain [5, 26]. The typing is *implicit*, that is, it does not require the user to manually define explicit types, which can be laborious [25], and even infeasible in the case of DT design due to the dynamic nature of the functions learned within the pipeline. For instance, when applying PCA to obtain a reduced basis preserving 99% of the variance of the original dataset, the explicit output type (i.e., number of dimensions) depends on the training data: explicit typing would not be feasible in this case.

5.1 Implicit Types

First, for all (data or function) port $i \in \mathcal{P} = \{1, \ldots, m\}$, we define default$(i) = i$. Now, we associate an *implicit* data type type(p) to each data port and to each function port.

[1] Some algorithms specified in PredefFunc may only provide one of these two functions. Thus, Coder may have only one function output in some cases.

For *data ports*, we define the set $\mathcal{D} = \{1, \ldots, m\}$ of Data Types, which is the same as the set of ports $\mathcal{P}$. By default, port p has implicit type $\text{type}(p) \leftarrow \text{default}(p)$, but it may be given type $\text{type}(p) \leftarrow \text{default}(q) < \text{default}(p)$ if it is known that the types of ports p, q are the same. In general, some numbers in $[1, m]$ will not be used, as several ports will have the same implicit type.

For *function ports*, the set of function types is defined as $\mathcal{F} = \cup_{i\in\mathbb{N}}\mathcal{D}^i \times \cup_{j\in\mathbb{N}}\mathcal{D}^j$. That is, a function f with k inputs and k' outputs will have the implicit type $((\text{type}_1, \ldots, \text{type}_k), (\text{type}_{k+1}, \ldots, \text{type}_{k+k'}))$, where $\text{type}_{i\leq k}$ is the implicit type of the i-th input, and type_{k+i} is the implicit type of the i-th output of f for $i \leq k'$.

By default, a *Coder* box b, with $\text{type}_1, \ldots, \text{type}_k$ denoting the types of input data ports of b, generates:

- a function Encode of type $((\text{type}_1, \ldots, \text{type}_k), (\text{type}_{Out}))$. type_{Out} is a fresh type never seen before and represents the data on the reduced basis, and
- a function Decode of type $((\text{type}_{Out}), (\text{type}_1, \ldots, \text{type}_k))$.

This default can be changed by providing extra information in the library containing the predefined function specified by the Coder's parameter. For example, if the parameter of b calls a normalization procedure, we could have $\text{type}_{Out} = \text{type}_1$.

A *Trainer* box b with ℓ input data ports, generates, by default, a function Predict of type $((\text{type}_1, \ldots, \text{type}_k), (\text{type}_{k+1}, \ldots, \text{type}_\ell))$, where $\text{type}_1, \ldots, \text{type}_\ell$ are the types of the input data ports of b, and k is the number provided in the first component of the parameter of the Trainer box b. This default can also be changed in the library.

5.2 Type propagation and checking

We now explain how to propagate the types automatically between ports. The types are propagated via the FDF Graph topological order. We assume, without loss of generality, that the port numbering follows the topological order.

Note that the type checking may return warnings to the user if it does not have enough information to ensure that the two types are equal. In this case, the user would either confirm that the two ports have the same type or rectify the pipeline if the type mismatch is genuine. The user can also add explicit *type annotations* before running the type checking, to provide this information. Specifically, if different ports $p_1, p_2, \ldots, p_s$ have the same annotation (except exponents), then they have the same implicit type. Thus, we can set $\text{type}(p_1) = \ldots = \text{type}(p_s) \leftarrow min_{j\leq s}(\text{type}(p_j))$.

The type propagation proceeds in three main steps. The *first step* is to propagate types for the DataIO output data port. Let $\{1 \ldots, r\}$ be the output data ports of DataIO (that is, the first r ports of the FDF pipeline). By default, each port $i \leq r$ will have a different data type: $\text{type}(i) \leftarrow \text{default}(i) = i$. The user may add annotations to specify otherwise.

The *second step* is to propagate the types through the ports which are associated with a box b. First, each input port $p \in \mathcal{P}^I$ with $box(p) = b$ copies the implicit data type from the corresponding output port $src(p)$, that is $\text{type}(p) \leftarrow \text{type}(src(p))$.

Finally, the *third step* is to compute the implicit type for each output port of b. Note that, in general, an output port p can either have its default type, $\text{type}(p) = \text{default}(p)$, or it can have a type $\text{type}(p) = \text{type}(q) < \text{default}(p)$, propagated from a previous port q (through possibly several boxes). The implicit type depends on $boxclass(b)$ as follows.

Coder Type Propagation. Let p_{Out} be the output function port of b (if there are two output function ports, take the minimal $\text{default}(p)$: there is at least one output function port). Then, a box b with $boxclass(b) = \text{Coder}$ has $\text{default}(p_{Out})$ as the output type of Encode (and the input type of Decode). This guarantees by construction that this type has not been used before. Let $\text{type}_1, \ldots, \text{type}_r$ be the implicit types of the input data ports of b, and let p_{Encode}, p_{Decode} be the two output function ports. We define:

$$\text{type}(p_{\text{Encode}}) \leftarrow ((\text{type}_1, \ldots, \text{type}_r), (\text{default}(p_{Out})))$$
$$\text{type}(p_{\text{Decode}}) \leftarrow ((\text{default}(p_{Out})), (\text{type}_1, \ldots, \text{type}_r))$$

Trainer Type Propagation. A box b with $boxclass(b) = \text{Trainer}$ has a single output function port. We define its implicit type as follows. Let k be the number provided by Param; let $\text{type}_1, \ldots, \text{type}_\ell$ be the implicit types of the input data ports of b; let p be the output function port of b. Then:

$$\text{type}(p) \leftarrow ((\text{type}_1, \ldots, \text{type}_k), (\text{type}_{k+1}, \ldots, \text{type}_\ell))$$

Processor Type Propagation. For $boxclass(b) = \text{Processor}$, we have two cases. The *first case* is when b has a input function port p_F. We denote $\text{type}(p_F) = ((\text{type}_1, \ldots, \text{type}_k), (\text{type}'_1, \ldots, \text{type}'_{k'}))$. Before propagating the type, we must ensure the following conditions are absent. If a condition is detected, we raise an error or warning:

(1) *Mismatch in the number of input/output.* A mismatch error occurs if the number of input data ports of b is not k or if the number of output data ports of b is not k'. The user should fix the pipeline.
(2) *Inconsistent input type.* An inconsistent input type warning occurs when $\exists j \in [1, k] : \text{type}(p_j) \neq \text{type}_j$, i.e., the type expected as input by p_F does not match the type of port p_j. To fix this inconsistency, the user can either tell that the two types are identical (for instance, by annotation) or fix the pipeline.

If no such problem is encountered, we can set $\text{type}(p_{k+j}) \leftarrow \text{type}'_j$ for all $j \leq k'$, and propagate to the next boxes.

The *second case* is when b has no input function port, but instead, its parameter specifies a predefined function PredefFunc from a library. The library is unaware of the implicit types propagated in one particular FDF pipeline. Further, a function from a library can be polymorphic, accepting several types as input, another reason to not impose strong typing. The library can provide however weak type information: first, its number k of input ports and k' of output ports (If the input vectors can take any size, then the input is multiplexed into a single port and $k = 1$; similarly for the output). The library can also specify a partition $P_1, \ldots, P_r$ with $P_1 \sqcup P_2 \sqcup \cdots \sqcup P_r$ of the set of all (input and output) ports of the function. Each partition represents the fact that types should be equal within the partition.

Similar to the first case, we must first ensure the following two conditions are absent:

(1) Mismatch in the number of inputs or outputs.
(2) Inconsistent input type. This condition occurs if two ports from the same partitions have different types, i.e., if $\exists i \leq r :$ $(p, q) \in P_i$ and $\text{type}(p) \neq \text{type}(q)$.

If a warning is raised, the user can fix it as in the first case, e.g. by providing the information that the two types are the same. Once there are no more warnings or errors, the type propagation proceeds as follows. For all $i \leq r$, either:

(1) there is an input data port $p \in P_i$: we set $\text{type}(q) \leftarrow \text{type}(p)$ for all the output ports q in P_i, as the library specified that these output data ports have the same implicit type as the input data port p, or
(2) there is no such input data port: we set a fresh $\text{type}(q) \leftarrow \min_{q \in P_i} \text{default}(q)$ for all the output ports q in P_i.

6 APPLICATION TO MOTIVATING EXAMPLES

Now we illustrate how the FDF formalism can be applied to the motivating examples described in Section 1.

6.1 DTP for Material Strain Prediction

Recall that the first motivating example (see Figure 1) aims to predict the plastic strain of a structure from an observed deformation [6]. Figure 5 describes an FDF pipeline generating a Strain Model DTP.

From a design of experiments exploring impacts with different strengths and repetitions, we obtain a correlation between the deformation ΔU and the plastic strain ϵ_p using a (slow) finite element model (first Processor box).

Both these datasets (set of deformations and set of strains) have high dimensionality (>1000 dimensions). First, we reduce the dimensions of each dataset, using PCA to learn a reduced basis (tens of dimensions) allowing us to recover 99.9% of the precision. The learning of the PCA basis (in the Coder box, that is generating the encoding $\mathcal{E}$ and decoding $\mathcal{D}$ functions) is decoupled from applying it in the second Processor box. This produces a reduced dataset $r\Delta U$. The same is true for the reduced plastic strain $r\epsilon_p$. Notice that the i-th reduced plastic strain corresponds to the i-th reduced deformation. Last, we learn a neural network (with 2 layers of 50 nodes each) generalizing the function from reduced displacement $r\Delta U$ to reduced strain $r\epsilon_p$.

Exploitation. Once the different functions have been learned, the DTP can be exploited to obtain the plastic strain from actual 3D images of a deformation. An exploitation pipeline can be described in FDF as well: the pipeline is composed of a series of Processor boxes using the learned functions.

The pipeline, illustrated in Figure 6, starts by fitting the 3D image on the finite element mesh of the structure, to obtain a ΔU map,

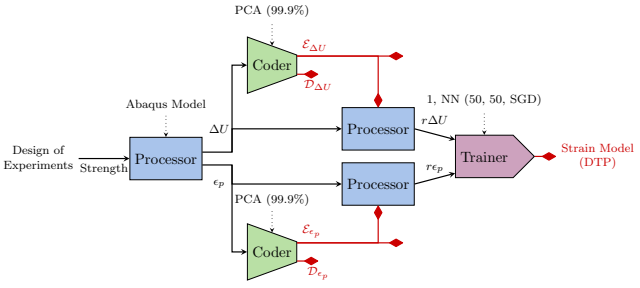

Figure 5: An FDF pipeline to learn a Strain Model DTP.

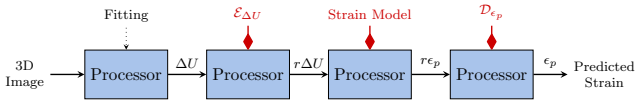

Figure 6: Exploitation pipeline for Material Strain Prediction.

using a predefined *Fitting* function. Then, the encoder $\mathcal{E}_{\Delta U}$ that has been learned is used to obtain the reduced $r\Delta U$, which can be input to the learned Strain Model. A reduced $r\epsilon_p$ is obtained, which is decoded using $\mathcal{D}_{\epsilon_p}$ into a full dimensionality strain map on the original mesh of the structure, that can be interpreted by experts.

FDF facilitates the design of this pipeline in three ways. First, it allows describing visually the dataflow necessary to learn a function, which is non-standard here as it involves routing an output ΔU of the first Abaqus Model as the input of the DTP. Second, it enables the easy export of functions from the FDF learning pipeline and their easy reuse at the correct place in the exploitation pipeline. Lastly, implicit typing helps to prevent user mistakes. For example, it can prevent users from mistakenly feeding ΔU to the Strain Model instead of the expected reduced $r\Delta U$ in case they forget to use $\mathcal{E}_{\Delta U}$.

6.2 DTI of a Magnetic Bearing Instance

Recall that the second motivating example (see Figure 2) aims to predict the magnetic flux given an applied voltage profile, in a particular instance of a bearing [16]. Figure 7 describes an FDF pipeline to generate the nominal DTP and then the DTI by tuning to the particular instance of a bearing.

From a design of experiments on various voltage time series (V_n^E) (different amplitude and shape), we obtain, using the (slow) Maxwell's equation (first Processor box), the output sequence (ϕ_n^M) of the magnetic flux induced in the system. A (fast) Cauer model is trained (first Trainer box) from these input-output time series to generalize them accurately. This gives the nominal Cauer model.

To account for the characteristics of a specific magnetic bearing instance (deviations from the nominal model during manufacturing or operation), we introduce an Ignorance Model [8]. This model captures the difference between the nominal Cauer model and the instance's actual behavior observed through historical data collected from it. Specifically, we obtain the predicted flux time series (ϕ_n^C) from the nominal Cauer model based on some historical voltage (V_n^H). We then compare this predicted flux with the associated historical flux (ϕ_n^H) measured on the specific magnetic bearing instance. The last Processor box computes the (time series) difference between the predicted and actual historical behavior. Lastly, we train a Long Short-Term Memory (LSTM) network on these supervised pairs $((V_n^H), (\Delta\phi_n))$, resulting in the Ignorance model.

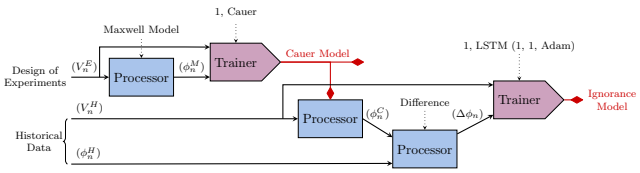

Figure 7: An FDF pipeline to learn a Magnetic Bearing DTI.

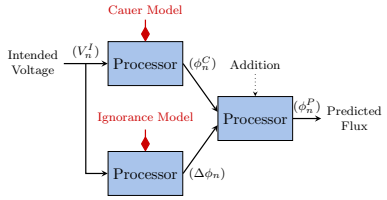

Figure 8: The exploitation pipeline for a Magnetic Bearing.

Note that FDF makes the design easy, in particular for manipulating and reusing the learned functions: the Cauer model is reused to infer flux from data (V_n^H) different from the one it has been learned on (V_n^E). This is unlike the encoder $\mathcal{E}_{\Delta U}$ in Figure 5 that is used on the same dataset that it has been trained with. Hence, not decoupling learning and inference would hinder such generality. Type-checking would also help the designer here. Note that superscripts ^{E} and ^{H} in the annotation are disregarded for the type equality check, and the type propagation will know that both have the same type from annotations.

Exploitation. Figure 8 details the exploitation pipeline. This pipeline aims to predict the magnetic flux ϕ_n^P that would be triggered on the particular instance of the magnetic bearing based on an intended voltage input profile. This prediction allows us to assess whether the bearing will operate as intended with the given voltage profile or if the voltage profile needs to be adjusted.

The pipeline proceeds as follows. The intended voltage time series (V_n^I) is fed into both the learned Cauer and Ignorance models. The Cauer model predicts the nominal flux response (ϕ_n^C), while the Ignorance model predicts the discrepancy $(\Delta\phi_n)$ between this particular instance and the nominal Cauer model. These two components are then integrated (by a simple addition $(\phi_n^C + \Delta\phi_n)$) to obtain the predicted DTI flux.

Alternate DTI pipeline. The pipeline in Figure 7 is very accurate when the discrepancy between the instance and the nominal Cauer model directly depends upon the applied voltage. For example, the higher the voltage, the more the instance flux deviates from the nominal Cauer model. However, in other types of discrepancies, there may be a more complex interplay, e.g., thresholding of instance's flux wrt to the Cauer model. In such cases, employing different operators, such as composition instead of difference, could lead to a more accurate model: the Cauer flux (ϕ_n^C) is directly linked to the discrepancy of the flux of the instance in this case.

We finally demonstrate once more the generality and ease of using FDF, by proposing an alternate DTI pipeline to specify a pipeline using composition instead of difference. We illustrate such a variant of the FDF pipeline in Figure 9.

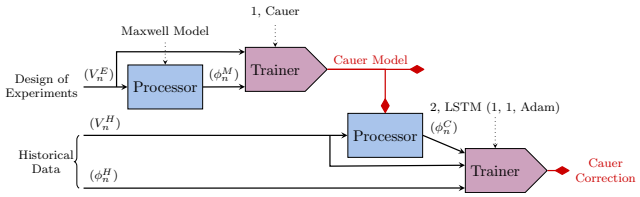

Figure 9: A variant of the Magnetic Bearing DTI pipeline.

The Cauer model is obtained using the same process as before. However, the second Trainer box now has as additional input (ϕ_n^C) on top of (V_n^H). This is reflected in the "2" in the first component of its parameter. Similarly, we adapt the exploitation pipeline (see Figure 10): now the Cauer correction is applied *after* the Cauer model's prediction, and it takes the flux (ϕ_n^C) as input to make its final prediction, thus implementing composition.

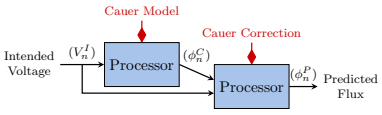

Figure 10: Variant of the exploitation pipeline for a Magnetic Bearing with composition.

7 CONCLUSION

This paper introduces the Function+Data Flow (FDF) language, a novel domain-specific language designed to streamline the creation of machine learning pipelines for digital twins. FDF addresses shortcomings in existing formalisms: *generality* to twin in many different domains and applications is hindered when the pipeline is rigid and proprietary (twin builders from software vendors specialized in CAD/CFD) and *manipulation of functions* is hard when they are not represented explicitly (traditional machine learning workflows).

FDF bridges the first gap by leveraging open machine learning workflows and concepts of dataflow. This flexibility allows FDF to encompass the diverse requirements of various digital twinning domains, as showcased in two motivating examples in electromagnetic and structural engineering. To bridge the second gap, FDF extends the classical dataflow by treating functions as first-class citizens. This enables users to manipulate and combine these functions freely, a crucial aspect of digital twinning. Furthermore, FDF enjoys the maintainability and debuggability of typed languages, through the introduction of user-friendly implicit typing, which removes the burden of explicitly specifying every data type in the pipeline. In essence, FDF integrates the well-established software engineering principles of abstraction into the design of machine learning pipelines for digital twinning.

Future work shall include developing the FDF framework, e.g. through complete libraries, and by extending FDF to support control applications and online learning.

ACKNOWLEDGMENTS

We thank our reviewers for their constructive feedback. We thank Amine Ammar and Joel Moutarde for their suggestions and inputs on the motivating examples. This research is part of the program DesCartes and is supported by the National Research Foundation, Prime Minister's Office, Singapore under its Campus for Research Excellence and Technological Enterprise (CREATE) program.

REFERENCES

[1] Sajid Alam, Nok Lam Chan, Laura Couto, Yetunde Dada, Ivan Danov, Deepyaman Datta, Tynan DeBold, Jitendra Gundaniya, Yolan Honoré-Rougé, Stephanie Kaiser, Rashida Kanchwala, Ankita Katiyar, Ravi Kumar Pilla, Huong Nguyen, Nero Okwa, Juan Luis Cano Rodríguez, Joel Schwarzmann, Dmitry Sorokin, Merel Theisen, Marcin Zabłocki, and Simon Brugman. 2024. *Kedro*. Kedro. https://github.com/kedro-org/kedro

[2] American Society of Mechanical Engineers 2023. *Complexities of Capturing Large Plastic Deformations Using Digital Image Correlation: A Test Case on Full-Scale Pipe Specimens*. International Conference on Offshore Mechanics and Arctic Engineering, Vol. Volume 3: Materials Technology; Pipelines, Risers, and Subsea Systems. American Society of Mechanical Engineers. https://doi.org/10.1115/OMAE2023-102308 arXiv:https://asmedigitalcollection.asme.org/OMAE/proceedings-pdf/OMAE2023/86854/V003T04A033/7040915/v003t04a033-omae2023-102308.pdf

[3] Ansys 2024. *Ansys Twin Builder | Create and Deploy Digital Twin Models*. Ansys. https://www.ansys.com/products/digital-twin/ansys-twin-builder

[4] Apache Airflow 2024. *Apache Airflow*. Apache Airflow. https://airflow.apache.org/

[5] Justus Bogner and Manuel Merkel. 2022. To Type or Not to Type? A Systematic Comparison of the Software Quality of JavaScript and Typescript Applications on GitHub. In *Proceedings of the 19th International Conference on Mining Software Repositories* (2022-10-17) *(MSR '22)*. Association for Computing Machinery, New York, NY, USA, 658–669. https://doi.org/10.1145/3524842.3528454

[6] Amaury Chabod. 2022. Digital Twin for Fatigue Analysis. *Procedia Structural Integrity* 38 (Jan. 2022), 382–392. https://doi.org/10.1016/j.prostr.2022.03.039

[7] Andrew Chen, Andy Chow, Aaron Davidson, Arjun DCunha, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Clemens Mewald, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, Avesh Singh, Fen Xie, Matei Zaharia, Richard Zang, Juntai Zheng, and Corey Zumar. 2020. Developments in MLflow: A System to Accelerate the Machine Learning Lifecycle. In *Proceedings of the Fourth International Workshop on Data Management for End-to-End Machine Learning* (Portland, OR, USA) *(DEEM'20)*. Association for Computing Machinery, New York, NY, USA, Article 5, 4 pages. https://doi.org/10.1145/3399579.3399867

[8] Francisco Chinesta, Elias Cueto, Emmanuelle Abisset-Chavanne, Jean Louis Duval, and Fouad El Khaldi. 2020. Virtual, Digital and Hybrid Twins: A New Paradigm in Data-Based Engineering and Engineered Data. *Arch Computat Methods Eng* 27, 1 (Jan. 2020), 105–134. https://doi.org/10.1007/s11831-018-9301-4

[9] Francisco Chinesta, Pierre Ladeveze, and Elías Cueto. 2011. A Short Review on Model Order Reduction Based on Proper Generalized Decomposition. *Arch Computat Methods Eng* 18, 4 (Nov. 2011), 395–404. https://doi.org/10.1007/s11831-011-9064-7

[10] William Danilczyk, Yan Sun, and Haibo He. 2019. ANGEL: An Intelligent Digital Twin Framework for Microgrid Security. In *2019 North American Power Symposium (NAPS)*. IEEE, Wichita, KS, USA, 1–6. https://doi.org/10.1109/NAPS46351.2019.9000371

[11] Alexis De Meo and Michael Homer. 2022. Domain-Specific Visual Language for Data Engineering Quality. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments* (Auckland, New Zealand) *(PAINT 2022)*. Association for Computing Machinery, New York, NY, USA, 48–56. https://doi.org/10.1145/3563836.3568727

[12] Francesco Ferrise, Monica Bordegoni, and Umberto Cugini. 2013. Interactive Virtual Prototypes for Testing the Interaction with New Products. *Computer-Aided Design and Applications* 10, 3 (Jan. 2013), 515–525. https://doi.org/10.3722/cadaps.2013.515-525

[13] Fortune Business Insights 2024. *Digital Twin Market Size, Share | Growth Analysis Report [2032]*. Fortune Business Insights. https://www.fortunebusinessinsights.com/digital-twin-market-106246

[14] Alex Fukunaga, Wolfgang Pree, and Takayuki Dan Kimura. 1993. Functions as Objects in a Data Flow Based Visual Language. In *Proceedings of the 1993 ACM Conference on Computer Science - CSC '93*. ACM Press, Indianapolis, Indiana, USA, 215–220. https://doi.org/10.1145/170791.170832

[15] Gartner 2022. *Emerging Technologies: Revenue Opportunity Projection of Digital Twins*. Gartner. https://www.gartner.com/en/documents/4011590

[16] Chady Ghnatios, Sebastian Rodriguez, Jerome Tomezyk, Yves Dupuis, Joel Mouterde, Joaquim Da Silva, and Francisco Chinesta. 2024. A Hybrid Twin Based on Machine Learning Enhanced Reduced Order Model for Real-Time Simulation of Magnetic Bearings. *Adv. Model. and Simul. in Eng. Sci.* 11, 1 (2024), 3. https://doi.org/10.1186/s40323-024-00258-2

[17] Michael Grieves and John Vickers. 2017. *Digital Twin: Mitigating Unpredictable, Undesirable Emergent Behavior in Complex Systems*. Springer International Publishing, Cham, 85–113. https://doi.org/10.1007/978-3-319-38756-7_4

[18] Dirk Hartmann, Matthias Herz, and Utz Wever. 2018. Model Order Reduction a Key Technology for Digital Twins. In *Reduced-Order Modeling (ROM) for Simulation and Optimization: Powerful Algorithms as Key Enablers for Scientific Computing*, Winfried Keiper, Anja Milde, and Stefan Volkwein (Eds.). Springer International Publishing, Cham, 167–179. https://doi.org/10.1007/978-3-319-75319-5_8

[19] Mina Jafari, Abdollah Kavousi-Fard, Tao Chen, and Mazaher Karimi. 2023. A Review on Digital Twin Technology in Smart Grid, Transportation System and Smart City: Challenges and Future. *IEEE Access* 11 (2023), 17471–17484. https://doi.org/10.1109/ACCESS.2023.3241588

[20] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. 2004. Advances in Dataflow Programming Languages. *ACM Comput. Surv.* 36, 1 (March 2004), 1–34. https://doi.org/10.1145/1013208.1013209

[21] Werner Kritzinger, Matthias Karner, Georg Traar, Jan Henjes, and Wilfried Sihn. 2018. Digital Twin in Manufacturing: A Categorical Literature Review and Classification. *IFAC-PapersOnLine* 51, 11 (2018), 1016–1022. https://doi.org/10.1016/j.ifacol.2018.08.474

[22] Kubeflow 2024. *Kubeflow*. Kubeflow. https://www.kubeflow.org/

[23] Lucy Ellen Lwakatare, Ivica Crnkovic, Ellinor Rånge, and Jan Bosch. 2020. From a Data Science Driven Process to a Continuous Delivery Process for Machine Learning Systems. In *Product-Focused Software Process Improvement*, Maurizio Morisio, Marco Torchiano, and Andreas Jedlitschka (Eds.). Vol. 12562. Springer International Publishing, Cham, 185–201. https://doi.org/10.1007/978-3-030-64148-1_12

[24] Beatriz Moya, Alberto Badías, Icíar Alfaro, Francisco Chinesta, and Elías Cueto. 2022-07-15. Digital Twins That Learn and Correct Themselves. *Numerical Meth Engineering* 123, 13 (2022-07-15), 3034–3044. https://doi.org/10.1002/nme.6535

[25] John-Paul Ore, Sebastian Elbaum, Carrick Detweiler, and Lambros Karkazis. 2018. Assessing the type annotation burden. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 190–201. https://doi.org/10.1145/3238147.3238173

[26] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 155–165. https://doi.org/10.1145/2635868.2635922

[27] Abel Sancarlos, Morgan Cameron, Jean-Marc Le Peuvedic, Juliette Groulier, Jean-Louis Duval, Elias Cueto, and Francisco Chinesta. 2021. Learning Stable Reduced-Order Models for Hybrid Twins. *Data-Centric Eng.* 2 (2021), e10. https://doi.org/10.1017/dce.2021.16

[28] Siemens Digital Industries Software 2024. *Simcenter Systems Simulation*. Siemens Digital Industries Software. https://plm.sw.siemens.com/en-US/simcenter/systems-simulation/

[29] R. Siva Srinivas, R. Tiwari, and Ch. Kannababu. 2018. Application of Active Magnetic Bearings in Flexible Rotordynamic Systems – A State-of-the-Art Review. *Mechanical Systems and Signal Processing* 106 (June 2018), 537–572. https://doi.org/10.1016/j.ymssp.2018.01.010

[30] Seyon Sivarajah, Lukas Heidemann, Alan Lawrence, and Ross Duncan. 2022. Tierkreis: A Dataflow Framework for Hybrid Quantum-Classical Computing. In *2022 IEEE/ACM Third International Workshop on Quantum Computing Software (QCS)*. IEEE, Dallas, TX, USA, 12–21. https://doi.org/10.1109/QCS56647.2022.00007

[31] Amin Darabnoush Tehrani, Zahra Kohankar Kouchesfehani, and Mohammad Najafi. 2020. Pipe profiling using digital image correlation. In *Pipelines 2020*. American Society of Civil Engineers Reston, VA, San Antonio, Texas, USA, 36–45.

[32] Eric J. Tuegel, Anthony R. Ingraffea, Thomas G. Eason, and S. Michael Spottswood. 2011. Reengineering Aircraft Structural Life Prediction Using a Digital Twin. *International Journal of Aerospace Engineering* 2011 (Oct. 2011), e154798. https://doi.org/10.1155/2011/154798

[33] Sebastian Utzig, Robert Kaps, Syed Muhammad Azeem, and Andreas Gerndt. 2019. Augmented Reality for Remote Collaboration in Aircraft Maintenance Tasks. In *2019 IEEE Aerospace Conference*. IEEE, Big Sky, MT, USA, 1–10. https://doi.org/10.1109/AERO.2019.8742228

[34] Zhongju Wang, Long Wang, M Revanesh, Chao Huang, and Xiong Luo. 2023. Short-Term Wind Speed and Power Forecasting for Smart City Power Grid With a Hybrid Machine Learning Framework. *IEEE Internet Things J.* 10, 21 (Nov. 2023), 18754–18765. https://doi.org/10.1109/JIOT.2023.3286568

[35] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. HuggingFace's Transformers: State-of-the-art Natural Language Processing. arXiv:1910.03771 [cs.CL]

[36] Minglan Xiong and Huawei Wang. 2022. Digital Twin Applications in Aviation Industry: A Review. *Int J Adv Manuf Technol* 121, 9 (2022), 5677–5692. Issue 9. https://doi.org/10.1007/s00170-022-09717-9

Received 2024-04-05; accepted 2024-05-04