2023 IEEE/ACM 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS) | 979-8-3503-1192-1/23/$31.00 ©2023 IEEE | DOI: 10.1109/SEAMS59076.2023.00016

Dynamic Runtime Integration of New Models in Digital Twins

Henrik Ejersbo, Kenneth Lausdahl, Mirgita Frasheri, Lukas Esterle
Department of Electrical and Computer Engineering
DIGIT, Aarhus University
Aarhus, Denmark
{hejersbo, mirgita.frasheri, lukas.esterle}@ece.au.dk, kenneth@lausdahl.com

***Abstract*—The development of cyber-physical systems is heavily relying on model-driven approaches. After deployment, these models can be utilised in a Digital Twin setting, acting as virtual replicas of the physical components and reflecting the behaviour of the running system in real-time. Complex systems often consist of numerous models interacting with each other and individual models may need to be updated after deployment. This means that new models need to be integrated and swapped during runtime without interrupting the running system. In this paper, we propose an approach for model-based Digital Twins to replace individual models without stopping or halting the operation of a cyber-physical system. Furthermore, our approach allows to replace not only individual models, but also update the overall structure of the interaction of models in the Digital Twin setting. The use of the proposed mechanism is illustrated through two case-studies with an agricultural robot prototype.**

***Index Terms*—Model Swap, Model-driven engineering, Co-simulation, Digital Twins, Functional Mock-up Interface**

I. Introduction

Cyber-physical systems (CPSs) are computing systems interacting tightly with the physical environments. Controllers implemented in embedded devices have to adjust the actions of the physical counterparts accordingly utilising the limited available resources [1], [2]. The development of such a system is cumbersome as safety guarantees have to be specified and verified. To overcome this, model driven engineering approaches have been developed and utilised, allowing to test and verify the behaviour of systems to be engineered [3], [4]. However, the different parts of a CPS are typically based on different mathematical basis, and hence modelled in different tools and environments, giving rise to challenges when it comes to testing the system as a whole. To tackle such complexity, co-simulation is often used, where models are to be packaged in specific ways, and their execution orchestrated in a coherent joint simulation [5], [6]. The interaction between the different models is captured by a multi-model, which is just a high level representation of the entire system to be simulated. The simulation of these interacting models is coordinated by a dedicated co-orchestration engine (COE).

With the development of Digital Twins (DT), models became useful beyond the design and development phase of CPSs [7]. Indeed, a bi-directional exchange of information can be generated, from the model to the CPS, and *vice versa*, upon deployment of the CPS and the DT. This means, models would not only follow the behaviour of a CPS during its operation in real-time but also perform verification tasks, predict potential maintenance requirements, and support runtime improvement while not necessarily interfering with the actual system [8], [9].

Co-simulation is further utilised to create modular DTs and execute their behaviour at runtime [10]. Different components of a CPS can be modelled in a DT depending on the particular needs of a user. This also allows to replace test hardware together with the models themselves, enabling the gradual introduction of hardware and using hardware-in-the-loop simulations [11] to improve reliability and performance of the underlying CPS. During deployment, it is possible that newer – and better – models become available, due to new information from the operation of the CPS. These new models may be able to follow the behaviour of the CPS more accurately and therefore are required to replace the previous, less accurate, models. However, as both the DT and CPS run in parallel, halting the DT would require to interrupt the operation of the CPS and *vice versa*. Therefore it is desired that these new models replace the old ones in the DT in such a way that the operation of the CPS is uninterrupted.

In this paper, we propose an approach to replace individual models within a system composed of multiple models, so-called multi-models, as well as introducing new information and control flows in the running systems. Specifically, our contributions include:

1) A model swap mechanism that allows to integrate new or upgraded continuous models at runtime in a DT setting. The mechanism proposes a set of general specification elements needed for dynamic models. The swap can be performed without interrupting the operation of the physical system.
2) An approach enabling the change of the structure of the multi-model and co-simulation during runtime without *a priori* knowledge of respective changes before deployment. This can introduce new information and control flows in the multi-model.

Our developed mechanism supports upgrading a model structure where the new replacing structure is not known prior to the construction of the initial model. In this way our mechanism does not require a complete specification of the

979-8-3503-1192-1/23/$31.00 ©2023 IEEE
DOI 10.1109/SEAMS59076.2023.00016

dynamic structure beforehand, but rather allows upgrades to be applied during runtime without interrupting the operation of the upgraded system. In this way it supports that new and better models can be developed over time and deployed at runtime in already operating DTs. It is possible to upgrade/swap individual models as well as the entire structure of the interconnected multi-model.

The effects of a swap can be simulated in a stand-alone setup decoupled from the DT - before being added to the running DT. In this way providing a two-step approach to upgrade models in a running DT. Specifically, we provide an implementation of models following the standard of Functional Mock-up Interface (FMI) by the Modelica Association [12]–[14], widely used by industry. Here, individual models are wrapped in Functional Mock-up Units (FMUs) with clearly defined interfaces to enable co-simulation.

The rest of this paper is organised as follows. Section II describes in detail the design of the proposed mechanism and its implementation, and Section III elaborates the background of necessary concepts relevant for this paper. Section IV discusses the experimental setup and the obtained results. Section V provides a brief overview of the related work in the area. We conclude our paper and outline future work in Section VI.

II. The Model Swap Mechanism

In the following we first present a formal definition of the problem and a conceptual overview of the developed model swap mechanism including a proposed set of general requirements to the information needed when specifying dynamic model structures.

A. Conceptual Overview and Requirements

The overall goal of the model swap mechanism is to replace a continuous model m_k within a DT. The DT can be defined as a tuple (M, C) where M is a set of models $M = \{m_1, ...m_i, ...m_n\}$ and C is a set of connections (o_k, i_j), where o_k describes the outputs of model m_k and i_j the input of m_j.

For swapping a model m_j (original) at runtime, we first have to develop and implement a model m'_j (replacement). Swapping the model itself is as simple as replacing the respective inputs and outputs, i.e., ensuring the inputs to m_j are also received by m'_j and the outputs of m'_j replace the outputs of m_j in all models receiving information originally from m_j. To decide when the new model should receive inputs, we define a dedicated step condition indicating that inputs i'_j of m'_j should be connected when the condition is satisfied. When the `stepCondition` is satisfied, we can define a new set of connections C' as

$$\forall k \; \exists (o_k, i_j) \in C : C' = C \cup \{(o_k, i'_j)\}. \quad (1)$$

Importantly, models can run in parallel and process incoming information. This means that output streams of one model can be processed by multiple receiving models as inputs. Hence we do not need to remove the connection between m_k and m_j when adding a new connection with input to m'_j as outlined in Equation 1. However, while a model may have multiple input interfaces, a dedicated input for a model can only come from a single other model. This is expressed in Equation 2, where the old input stream to a model m_l is removed, when it is replaced by the information from m'_j. This replacement is realised when a dedicated `swapCondition` is satisfied and we can thus define the resulting set of connections C'' as follows

$$\forall l \; \exists (o_j, i_l) \in C' : C'' = (C' \cup \{(o'_j, i_l)\}) \setminus \{(o_j, i_l)\}. \quad (2)$$

A conceptual overview of the mechanism is illustrated in Figure 1. An existing DT, consisting of multiple models, is shown in Figure 1(a) (top). This should be transferred at runtime to an intermediate (but invalid) DT containing a new model m'_2 in Figure 1(b) (middle). This is an invalid configuration as the input i_3 is connected to two different outputs o_2 and o'_2 and it cannot be expected that the model itself can differentiate between information coming from those two outputs. Here $c(\mathrm{o}_2, \mathrm{o}'_2)$ indicates the condition at which the models should be swapped (`swapCondition`). Finally the new instance m'_2 is replacing m_2 in Figure 1(c) (bottom). The replacement is conditionalised by an expression $c(\mathrm{o}_2, \mathrm{o}'_2)$ over a subset of DT variables. The DT in Figure 1(b) may be seen as an extension of the DT in Figure 1(a) where all models of the DT in Figure 1(a) are transferred to the DT in Figure 1(b) with their state preserved. In addition the new m'_2 is added and the swap condition c specifies the trigger condition for the original m_2 to be replaced by m'_2 and its new connections. The DT in Figure 1(c) is the resulting DT after a successful swap of m_2 to m'_2.

In general, the condition for a model swap may be more involved than a simple condition over the output variables of the respective models as depicted in Figure 1. As an example, consider the case where a separate observer component may be needed that enables more elaborate assessment of outputs.

When specifying a scenario for a dynamic model swap, we identify the following required specification parts:

- `swapInstance`: are new instances of a model to be added and eventually replacing an existing model.
- `swapConnection`: are new connections to and from a swap model. The new connections of a swap instance are activated (connected) in a conditional manner for both the inputs and the outputs.
- `stepCondition`: defines the condition at which the swap instance (new/updated model) is introduced to the overall DT. When the step conditions are satisfied, the inputs are connected (Equation 1 and Figure 1(b)). The step condition may, e.g., be needed in situations where the initial output of a new instance needs to be synchronized with the output of some other instance before stepping.
- `swapConditon`: defines the condition for completing the swap connections and connecting the outputs of the new model accordingly to the other models of the system. The new model becomes fully active with both inputs and outputs connected. At the same time, the inputs

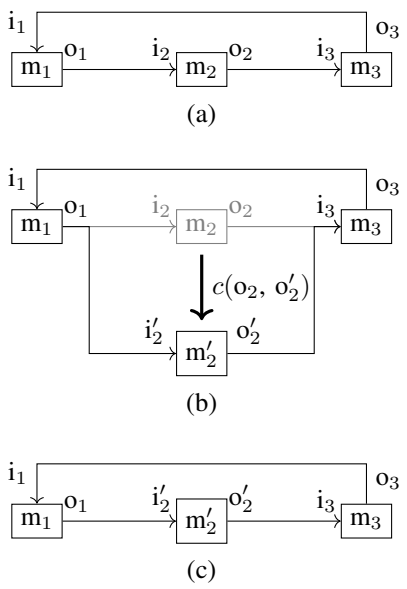

Fig. 1: Conceptual overview of the model swap mechanism: (a) Initial DT with multiple models, input/output variables and connections; (b) Intermediate DT with both m_2 and m_2' and a swap from m_2 to m_2' based on a condition over variables; (c) Final DT after the swap.

and outputs of the old model are disconnected and the old model is not executed anymore (Equation 2 and Figure 1(c)).

- `transferInstance`: are existing instances present in the already executing DT that are to be transferred to the updated DT unchanged and with all state preserved.
- `parameterInitialization`: is required for all newly added instances to ensure a continuous operation when swapping instances, i.e., initial state of a new instance is similar to the final state of the replaced instance. This could, e.g., be the initial heading parameter of a vehicle model, that in case the vehicle model was updated, would need to be initialized to the current heading of the existing model.

In addition to the proposed specification constructs, the internal simulation time of a co-simulation also needs to be transferred from one scenario to the next one (swapped) to properly continue an executing co-simulation. This is always required and thus not explicitly stated by our added specification constructs.

Our proposed set of specification constructs can be seen as a simple and general set of specification requirements for multi-model-based DTs, that can handle a broad set of cases and that avoids the complexity of full state serialization. Our added constructs do not impose new requirements to model instances but can be handled entirely by the entailing orchestration component, coordinating the individual models. Furthermore, the proposed approach does not limit us to generate the exact same connections as the previous model. This means, the new specification may introduce different or new control flows (i.e., different in- and outputs from and to the swapped model) without interrupting the ongoing operation of the CPS.

B. Specification Interpretation

A co-simulation generally consists of three phases: initialization, execution/simulation, and termination [15]. During the initialization phase a COE, able to execute the multi-model and coordinate the individual simulation models, first loads and instantiates all specified models. Then it calculates the initialization order of the interconnected models based on the topological ordering of model connections - including handling of algebraic loops using fixed point iteration [16].

Thereafter the COE constructs the initialisation code from the initialisation graph created in the previous step which properly represents dependencies between inputs and outputs of the utilised models. When the specification contains model transfers (entry `modelTransfer`), the initialisation code in the generated swap specification needs to avoid initialising ports of transferred model instances (as they have been already initialised). In this situation, only the initialisation of swap model instances needs to be considered. This is achieved by pruning the edges representing connections into transfer instances from the initialisation code. As a result, only the new models will be initialised, in the proper order, i.e., respecting the dependencies on other models.

In the execution phase the simulation loop will be executed. This loop iterates the simulation steps of all models and gets and sets input and output variables. When the description of the co-simulation scenario, executed by the COE, contains model swaps, the generated simulation loop code needs to reflect the `stepCondition`, `swapConditon`, and `swapConnection` sub-entries.

The `stepCondition` entry specifies a condition for the swap instance to enter a state where it may be stepped by the orchestration engine, i.e., including the model in the simulation loop. This condition allows a swapped-in model to be initialized (enter and leave state *Initialization Mode*) and then started in a controlled manner by a condition over model variables - including its own newly initialized variables. Additionally, when this condition is satisfied, the inputs to the new model are active, i.e., they get values from a specific output of an existing model (see Equation 1). This condition is useful when a scenario needs synchronization between newly swapped-in models and already executing models. The step condition is evaluated in the simulation loop as a *trigger condition* in the sense that once it evaluates to `true` it keeps this value. The trigger is evaluated at the start of each simulation loop iteration. Valid step conditions are expressions build from standard Boolean, relational, and arithmetic operators applied to model variables and literal constants as operands.

The `swapCondition` entry specifies the condition for the swap instance connections (`swapConnections`) to become effective. Importantly, the `swapCondition` has to occur together with or after the `stepCondition`. The swap connections affect the code generated to set linked model variables (by `setXXX`), in other words it activates the outputs of the

new model (see Equation 2). Note, that, for each connected input port of a model instance the setting of the port may have to be conditionalised by the swap condition. The rules controlling this are based on the source of the connection being from swap instances or not. E.g. for an existing (transferred) model instance with an input connected from a swap instance, the setting of that input needs to be guarded by the swap condition of the connected swap instance. This is due to the outputs of that swap instance only being enabled once the swap condition becomes true.

C. Execution Overview

A schematic overview of the model swap execution is shown in Figure 2. The figure illustrates the interaction between the orchestration engine and two models, where the simulation starts in m1 and swaps in m1′. Each simulation loop iteration starts by executing a *transfer point*. Here it is checked if a new swap specification is available in a configured file location and if so its validity for transfer is checked. If the transfer is valid, a transfer to a new orchestration interpreter context is performed, now interpreting the new specification including the swap in of m1′. This orchestration interpreter context is coordinating the execution of the individual models in the correct order. The new context continues the co-simulation with the original m1 transferred and with m1′ being loaded from scratch.

Since newly loaded models may have separate conditions to execute individual steps in the simulation, the co-orchestration engine must be able to advance individual models in time only if their respective step condition is valid. This means, that in a given global communication step some models may progress (valid step condition) while others don't (invalid step condition). In Figure 2, the condition in the block for setting inputs (setXXX) intends to illustrate that setting any input may have to be guarded by the swap condition or its negation. This depends on the source port of the connection coming from a model being swapped in or out. If the source model is not part of a swap, the input setting may be unguarded. We slightly misuse the standard UML notation 'opt' to define a condition that may be applied to all behaviours in the block. The block for stepping the models (doStep) illustrates that stepping a swapped in model is guarded by the step condition, and stepping the replaced model is guarded by the negation of the swap condition. The final block for getting outputs (getXXX) illustrates that outputs are read from a swapped in model if the swap condition is true, and otherwise from the original.

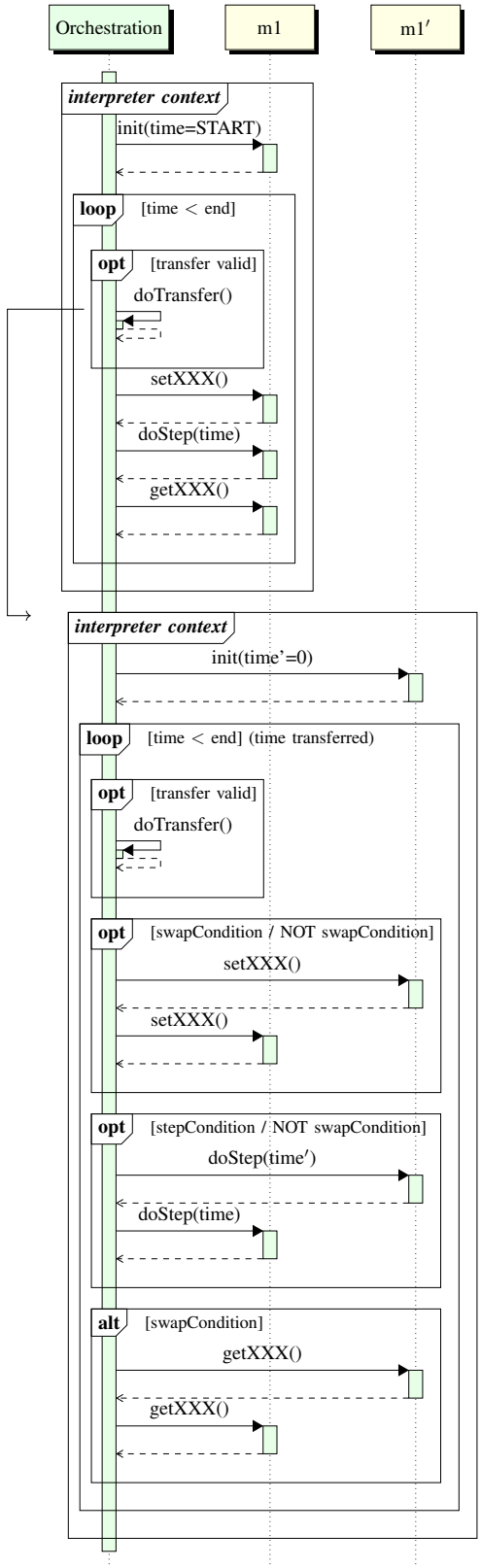

Fig. 2: Swap specification execution overview.

III. Background and Implementation

In this section we cover the concepts and tools relevant for the implementation and experimentation in the context of this paper. Specifically, we discuss the Functional Mock-up Interface (FMI) and FMI-based co-simulation, a standard widely used in industry, as well as Maestro, a co-orchestration engine developed for co-simulation. Furthermore, we highlight a data-broker based on the Advanced Message Queuing Protocol (AMQP) to facilitate exchange of information between co-simulation (i.e., the Digital Twin) and the corresponding hardware (i.e., the physical counterpart of the DT). We then

describe how the conceptual model is realized into a functioning extension of Maestro for a multi-model DT.

A. FMI-based Co-simulation

The design and development of CPSs requires expertise from multiple domains, due to the combination of (networked) software and hardware components. These components may depend on different mathematical basis and could be modelled using different tools. As individual models/solvers are designed and tested in their corresponding environments, the need arises to test and verify them as a whole. A common technique to achieve this goal is co-simulation, which couples individual units and executes them coherently in a joint simulation. This requires a common standard, describing how units are packaged and interfaced. In this paper, we adopted the industry standard Functional Mock-up Interface (FMI) [17], [18]. Within this standard, simulation units are referred to as Functional Mock-up Units (FMUs). FMUs implement a set of c-interfaces (as per the standard), provide a model description file that describes their parameters, input and outputs, and are packaged in a specific way.

A co-simulation can be specified through a multi-model file which describes which FMUs are to be included, the connections between them, and what values to set the parameters to. The execution of a co-simulation is carried out by a co-orchestration engine (COE), which implements a co-orchestration algorithm. The latter describes how the COE can interact with the FMUs, in terms of the order of retrieving outputs and setting inputs, through `getXXX` and `setXXX` functions respectively, and progressing them in time through the `doStep` function. Two typical examples of an orchestration algorithm are the Gauss-Seidel and Jacobi. Both run a loop from a defined start time until a defined end time. The former sets the inputs of the first FMU in a co-simulation, requests a `doStep` such that said FMU progresses from time h to $h + \Delta h$, collects the outputs, and proceeds to the next FMU to be stepped to time $h + \Delta h$. The latter, sets the inputs for all FMUs, requests a `doStep` that progresses all involved FMUs to time $h + \Delta h$, collects all outputs, and proceeds to the next time step. In this paper we use Maestro2 as our COE, henceforth simply called Maestro, which implements the Jacobian [19] approach. The Maestro engine consists of the domain specific language called Maestro Base Language (MaBL), an interpreter of the language and utilities to assist in specifying co-simulations in MaBL. It is cross-platform, based on the JVM, and offers interaction both through a web interface and command line.

The FMI specification defines an FMU interface to serialize all state of instances and having this feature generally implemented would ensure that the entire state of an FMU could be serialized and loaded into the context of a new compatible (swap) instance. However, such serialization may be difficult to implement generally with non-trivial FMU implementations containing, e.g. multi-threading models. Multi-threading would require serialising and saving state of the executing thread scheduling context as well as the FMU states. With the constructs we propose, a user can handle a broad set of cases avoiding the complexity of full state serialization. Furthermore, the extensions to the Maestro engine fully support the specified constructs and further allow for dynamic loading of swap specifications to facilitate workflows where new models are dynamically developed and integrated into running simulations. In the following sections we present details of the realization of our mechanism in the Maestro framework. We first describe the implementation of the specification format and then describe the corresponding MaBL translation and interpretation.

B. RMQFMU: A bridge to the real world

There are cases in which it is desirable to feed data – live or historical – to a co-simulation. As an example, consider the case in which a user would like to build a DT using the aforementioned co-simulation tools. In this scenario, the co-simulation/DT should be able to connect to the external world, e.g., to a piece of hardware or a fully-developed robot, and exchange data back and forth in real-time. To enable such bi-directional communication, we have previously proposed the RMQFMU [20], [21], an AMQP based FMU that serves as a data-broker between a co-simulation and any external system able to send/receive data through a RabbitMQ server.

RMQFMU can be configured through a set of parameters, covering the connection to the server, such as username and password, as well as other parameters that shape its internal behaviour, such as the *maxage* among others, which refers to the age of the data to be considered valid in the DT. Additionally, it is possible to specify the inputs and outputs of the FMU through its model description file. The RMQFMU steps in time when it has valid data from an external source, and sets its outputs to the received values. Conversely the FMU will send data to an external listener every time its inputs are set to a value different to those in the previous time-step.

C. Specification Format

The configuration file for the (multi-model) DT is in JSON format, with an example given in Figure 3. Such file has standard data elements to specify model instances and locations (lines 2–7 in Figure 3), connections (lines 8–17 in Figure 3), and parameters (lines 29–32 in in Figure 3) for a co-simulation, which is used to execute the DT. The model swap mechanism extends the standard format with data elements to specify the new parts proposed at the end of Section II-A, and is given in bold in Figure 3, specifically lines 13–28.

The new configuration elements are specified by entries `modelSwaps` and `modelTransfers`. An element in the `modelSwaps` entry has a model instance to be replaced as a key and an object as a value. This object specifies the `swapInstance` (new model replacing another model), a conditional expression (`stepCondition`) defining when the model replacement can be started (entering the execution phase) and have its inputs enabled, and a conditional expression (`swapCondition`) defining when the model replacement output connections will be enabled and the replaced

```
{
  "fmus": {
    "{x1}": "./fmu1.fmu",
    "{x2}": "./fmu2.fmu",
    "{x3}": "./fmu3.fmu"
    "{x4}": "./fmu2p.fmu"
  },
  "connections": {
    "{x1}.fmu1.o1": [ "{x2}.fmu2.i2" ],
    "{x2}.fmu2.o2": [ "{x3}.fmu3.i3" ],
    "{x3}.fmu3.o3": [ "{x1}.fmu1.i1" ]
  },
  "modelSwaps": {
    "fmu2": {
      "swapInstance": "fmu2p",
      "stepCondition": "(true)",
      "swapCondition": "(fmu2p.o2 - fmu2.o2 <1)",
      "swapConnections": {
        "{x1}.fmu1.o1": [ "{x4}.fmu2p.i2" ],
        "{x4}.fmu2p.o2": [ "{x3}.fmu3.i3" ]
      }
    }
  },
  "modelTransfers": {
    "fmu1": "fmu1",
    "fmu2": "fmu2",
    "fmu3": "fmu3"
  }
  "parameters" : {
    {x2}.fmu2.p1" : 2,
    {x4}.fmu2p.p1" : 2
  }
}
```

Fig. 3: Configuration file for co-simulation scenario with model swap

model may be unloaded and its connections removed. The step condition and the swap condition apply to the input/output connections of the exchanged models (`swapConnections`). In the example configuration in Figure 3, the new instance is `{x4}.fmu2p`. The input connection from `{x1}.fmu1.o1` to `{x4}.fmu2p.i2` will be enabled by the step condition (in this case simply `true`), whereas the output connection from `{x4}.fmu2p.o2` to `{x3}.fmu3.i3` will be enabled by the swap condition. In our example, this is triggered when the difference between the outputs of the new model `fmu2p.o2` and the old model `fmu2.o2` is below the specified threshold. Transferring the state of one model into a new model is done by utilizing the keyword `parameters`. Important, however, only states that are exposed by the old model as, e.g., outputs or parameters can be transferred as parameters to the new model.

IV. Experiments

In this section, we present two experiments of applying our developed model swap mechanism, providing insights into the way the mechanism can be used and the importance of the separate swap specification parts. In our experiments, we utilise a DT setting of a field robot prototype. Specifically, we utilise the desktop version (Desktop Robotti) (Figure 4) of a large agricultural field robot (Robotti) [22], developed by AgroIntelli. The Desktop Robotti enables us to experiment ideas in the lab, at a low cost. The DT consists of a multi-model executed in a co-simulation environment. With this setup we update models at runtime and can thereafter observe changed behaviour of the robot in real-time. First, we retrofit the robot with a mechanism for time discrepancy detection between the DT and its physical part. Second, we update the speed controller of the robot to change the maximum allowed velocity without interfering with the robots ongoing operation. This experiment will in addition illustrate the importance of having the separate step and swap conditions.

A. Desktop Robotti with Time Discrepancy Detection

In our experiment, we use the proposed model swap mechanism in the development of a DT for a prototype of an agricultural field robot to replace a message broker model with a new version with extended functionality during run-time. The extended functionality encompasses an orchestration synchronisation mechanism (OSA) which has been developed for the Maestro framework [23] that can detect and try to mitigate observed time discrepancies between a PT and a DT. The OSA mechanism, has a designated FMU that will detect discrepancies between simulation and wall-clock time based on PT and DT having synchronized wall clocks. The introduced detection FMU provides a new out-of-sync output that can be used by other FMUs to get informed of any time discrepancy occurrences and react appropriately.

Fig. 4: The Desktop Robotti.

In the experiment the OSA mechanism enables the DT to temporarily disable sending any data back to the robot in case of an out-of-sync situation, caused by, e.g., network degradation. This to prevent that the robot operates with out-of-sync data from the DT.

The initial DT of the robot consists of three FMUs. A data broker, an actuation model, and a vehicle model. The broker FMU enables the transmission of data into and out of the co-simulation in real-time. In the experiment, the broker FMU forwards steering controls (steering angle and speed) into the co-simulation matching the controls sent to the robot. In addition, it carries safety control commands (like a safety stop command) from the co-simulation back to the robot.

In the context of model swapping the most interesting part concerns the run-time update of the broker FMU `RmqFmu`, which is replaced by a new model named `RmqFmu2`. The swap requires a state synchronization with the replaced instance.

Therefore, in the following we will focus our discussion on the broker FMU replacement.

Figure 5a shows the co-simulation output from the broker FMU restricted to the steering angle data originating from the RabbitMQ server. In this experiment, the steering angle is a sine waveform generated by the robot control software. The swap specification is provided at simulation time 5sec (vertical magenta line) and since both step and swap conditions are true, the swap is instant.

As is clear from the figure, the transition from before the swap (blue) to after (green) breaks continuity in the steering angle output. This discontinuity is a result of the instant swap, and indicates that some form of synchronization is required. If this swap had been immediately carried out in a real DT setting it might have caused unforeseen problems since the DT would be missing a portion of real operational data. A stand alone co-simulation, including the upgraded broker FMU from the start, would not have been showing this issue. However, being able to simulate the real swap in a co-simulation setting, provides means to investigate and mitigate such issues up front.

To remedy the issue, the swap condition is changed such that the output steering angle of the existing broker FMU must match the output of the replacement FMU. The resulting co-simulation output is given in Figure 5b, and the corresponding (updated) swap specification part is shown in Figure 6.

In the output it can be seen that after the swap (timestep 5s), the `RmqFmu` and `RmqFmu2` outputs are both following the sine wave, but the outputs are offset by a constant 5s time interval. Thus, the original `RmqFmu` keeps being connected and is not replaced by `RmqFmu2`.

The real issue is caused by the `RmqFmu` having an internal queue state. When enabling the new `RmqFmu2`, its queue will be initially synchronized to contain the next unacknowledged message from the server. The original `RmqFmu` may have messages already in its internal queue that are read and acked with the server and thus the new `RmqFmu2` instance may read and output an initial message from the server being later than all (not yet processed) messages in the `RmqFmu` incoming queue.

To remedy this issue, we can make use of the step condition of our specification language, by stating that the `RmqFmu2` steering angle must be synchronized with the `RmqFmu` output before stepping. I.e., `RmqFmu2` will initialize and synchronize its first message with the RabbitMQ server. Thereafter, it will start stepping only when the original `RmqFmu` has processed any messages in its queue. Finally, we allow `RmqFmu2` to be swapped in at this instant, so the `stepCondition` and `swapCondition` are triggered at the same time. The resulting output and corresponding specification can be seen in Figure 5c and Figure 7, respectively.

In the simulation output it can be seen, that at simulation time 5s when the swap specification is enabled, the output used until time 10s is from the original `RmqFmu` (orange). The step condition of `RmqFmu2` is enabled at time 10s and the swap condition also becomes enabled so the output used is from `RmqFmu2` (green). Notice, that using the steering angle alone as synchronization state between the FMUs, may in general be insufficient to provide a continuous switch. Consider, e.g., Figure 5c and suppose that `RmqFmu2` has been initialized to the state of `RmqFmu` at time approximately 13s instead of time 10s. At time 13s the steering angle appears identical to the one at 10s, so the step condition may be detected as satisfied at 10s and `RmqFmu2` will start stepping at this instant causing a similar problem as in Figure 5a. In this example, we can conjoin the synchronization condition with, e.g., the message timestamp or sequence number to become unique to remedy this potential issue.

B. Limiting the speed of the Desktop Robotti

In our second experiment the maximum speed of the Desktop Robotti should be limited to a certain value. To enable this, we introduce a new speed controller to regulate the velocity of the robot. This new controller overwrites manual commands from the operator. This can become relevant when previous speed have caused problems in the path planning or travelling of the robot (e.g., muddy slopes require slower speeds to achieve required traction). The input for the FMU controlling the speed limit (`SpeedControlFmu`), is the current speed of the robot `act_speed`. This speed limiter is replaced by a new `SpeedControlFmu2` as soon as the speed of the robot is below a predefined threshold. This could be achieved by expecting the robot to enter this speed (e.g., when it has to slow down in a corner), or by synthesising a new controller to ensure this target requirement is achieved as proposed by Nahabedian et al. [24]). In this example, the `stepCondition` is true when the new specification is defined, allowing the new FMU to be simulated and executed along the FMU scheduled to be replaced. As soon as the `swapCondition` has been satisfied, the `swapInstance` becomes operational.

Figure 9 highlights the parameters of this experiment. The blue solid line indicates the desired speed defined by the user. The speed assigning to the electrical motors of the CPS by the FMU is illustrated with a solid line in magenta. The initial specification runs until time step 5.9 (blue background) when the new specification is defined. The new specification is operational immediately but does not swap the model yet as the swap condition is not reached until time step 10.5 (orange background). At this point, the actual speed (`act_speed`) of the CPS, illustrated as solid orange line, drops below the `swapCondition` (illustrated as red dashed line). By satisfying the `swapCondition`, the model is replaced and the actuated control is limited by the new `SpeedControlFmu2`, overwriting the desired user input. This swap is illustrated by the green background. From the time of the model swap, the goal is to change the speed limit to a maximum of 30cm/s. This is indicated by a green dashed line.

From our measurements, one can follow that the model swap is performed after the speed of the system is below the desired threshold and the new model is utilised as expected afterwards, i.e., under the speed cap enforced by the new controller `SpeedControlFmu2`.

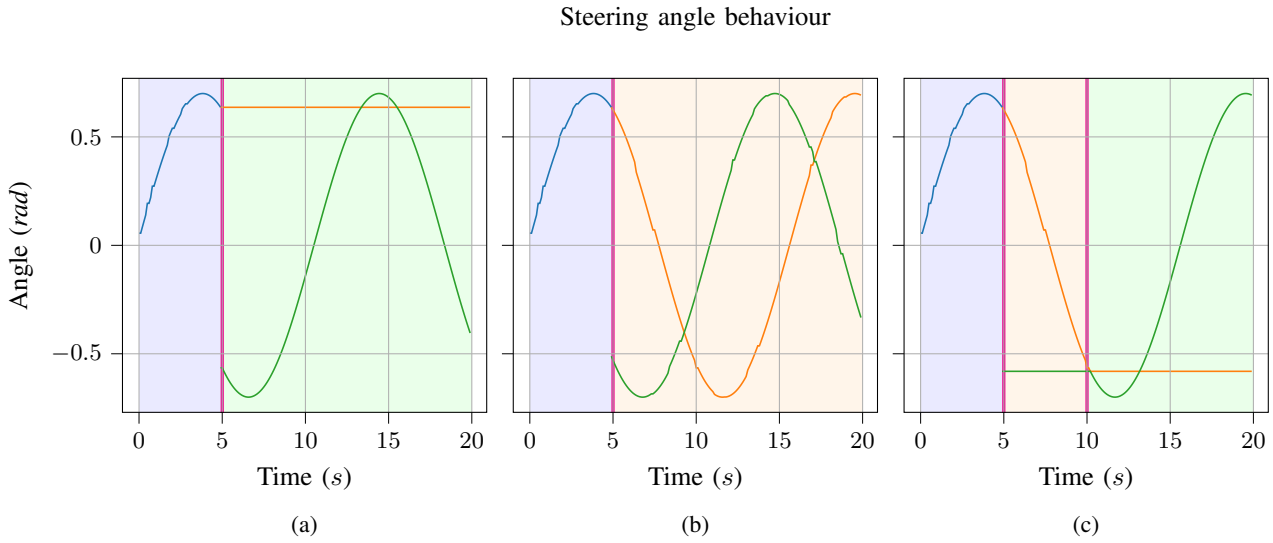

Fig. 5: (a) Instant swap of `RmqFmu` for `RmqFmu2`. Start and swap conditions true. `RmqFmu` angle output before swap (blue), `RmqFmu2` immediately connected output after swap (green). (b) Conditional swap of `RmqFmu` for `RmqFmu2`. Start condition true and swap conditional. `RmqFmu` angle output before swap (blue), `RmqFmu` stays connected output after swap (orange). (c) Conditional swap of `RmqFmu` for `RmqFmu2`. Both step and swap conditional. `RmqFmu` angle output before swap (blue), `RmqFmu` stays connected output after swap (orange) until swap condition where `RmqFmu2` gets connected (green).

```
"stepCondition": "(true)",
"swapCondition": "(RmqFmu.steering_angle == RmqFmu2.
    steering_angle)",
```

Fig. 6: Second configuration of the `RmqFmu` model swap.

```
"stepCondition": "(RmqFmu.steering_angle == RmqFmu2.
    steering_angle)",
"swapCondition": "(RmqFmu.steering_angle == RmqFmu2.
    steering_angle)",
```

Fig. 7: Final configuration of the `RmqFmu` model swap.

```
"stepCondition": "(true)",
"swapCondition": "(act_speed < 15)",
```

Fig. 8: Configuration of the `SpeedControlFmu` model swap.

V. Related Work

Developing Cyber-physical systems is a cumbersome task involving modelling physical interactions of software systems as well as the interaction among different networked, embedded systems. One of the key approaches to reduce the development complexity for CPSs is Model-driven Engineering (MDE) [25]. With MDE, systems are developed using compositions of models. This allows for rapid adaptations of developed systems by replacing individual models, leading to a blurry boundary between development and deployment [26] leading to DevOps (development and operations). Here, systems are designed to be improved and updated during runtime. Ahmed et al. [27] survey the state-of-the-art for dynamically updating traditional software systems after deployment. A specific problem arises when models need to be updated in CPSs, where the computational system has to directly interact, verify, and control physical properties whether they are of discrete or continuous nature [2]. Over the past years, dedicated DevOps approaches have been developed to consider the co-evolution of models and deployed systems [28], [29]. Dobaj et al. [30] propose a DevOps approach to continuously improve DTs for CPS.

Jørgensen [31] proposed structural dynamic models, the creation of larger models by combining simple, small-scale models. In addition, allowing adjustments of their parameters through control mechanisms. Several extentions have been proposed over the years: Ören [32] considered the discontinuities in the differential equations of models and how to model them, while Barros [33] proposed taxonomies to model multi-models. Yilmaz and Ören [34] introduce multi-models specifically for multi-agent systems. Uhrmacher [35] introduces an implementation-independent formalism for models whose description entails the possibility of changing their own state and behaviour utilising transition functions as part of the models. However, the approach is not directly applicable to CPSs and the continuous domain.

Nilsson and Giorgidze [36] proposed functional hybrid modelling as an approach to non-causal modelling to switch between pre-defined models. At around the same time, Zimmer [37] developed equation-based language to explicitly define adaptations in variable-structure systems, i.e., models where equations change during the time of execution.

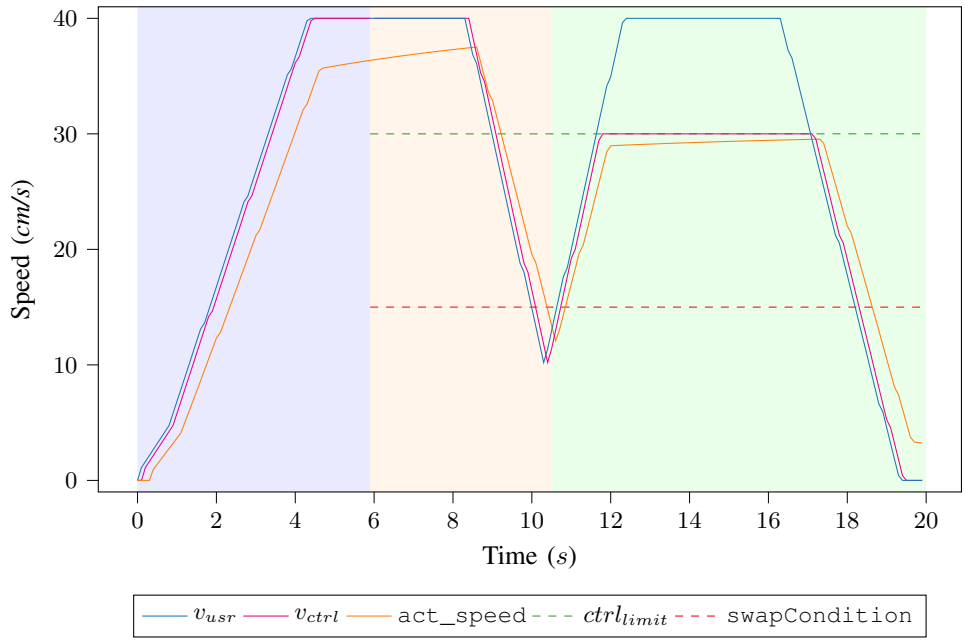

Fig. 9: Speed adaptation through replacing the underlying model at runtime. Blue solid line defines the velocity desired by the user (v_{usr}), the red solid line indicates the velocity defined by the controller (v_{ctrl}), and the solid orange line illustrates the actual speed of the CPS (`act_speed`). Blue background from time 0 to 5.9 utilises the original model. The new specification takes place at time 5.9 (orange background), defining the model swap. The model is swapped at time 10.4 when the speed is below the desired threshold. The horizontal dashed lines indicate the `swapContition` in red (i.e., lower threshold to perform the model swap), and the new maximum speed allowed for the new `SpeedControlFmu2` in green.

Mehlhase [38] proposes to utilise models with multiple, different modes. She further introduced a Python-based framework allowing to switch between different modes of the model. The framework utilises end values from the previous mode to initialise the new mode when switching between them. In contrast, we allow to dynamically change the model structure and introduce new models and underlying model structures at runtime without suspending the simulation. Karner et al. [39]–[41] explores swapping models in the co-simulation in order to exploit the trade-off between simulation time and accuracy. This trade-off arises as more complex models require more time to be simulated and executed while faster models lack accuracy. In a similar way, Lavin et al. [42] swap more complex models for simpler models, using statistical approximation, at runtime. These swaps are performed without halting or interrupting the running simulation. However, they do not elaborate on the applicability of their approach in digital twin settings. Furthermore, the discussed approaches focus on model replacement alone without potential changes to the underlying control flow in the simulation of the multi-model.

Replacing or swapping models in MDE, after systems have been deployed, is not entirely new. The work on models@run.time [43] has introduced the idea of replacing individual models while the actual system is executed. This allows for runtime adaptation of the behaviour of the respective system. This enables to provide assurance for the behaviour of self-adaptive systems [44]. The need for explicit mechanisms and approaches to implement this runtime change is highlighted by Bennaceur et al. [45] or Götz et al. [46], among others. The recent and very extensive survey on models@run.time [47] highlights the challenge of replacing models during runtime. Fouquet et al. [48] propose an approach to replace models at runtime in CPS but focus on memory utilisation and reboot delay when adding new firmware. They do not explicitly consider the dynamics and continuous change to which a CPS is exposed.

Heinzemann et al. [49] present a modeling language which allows to specify platform-independent models of hierarchical re-configurations. Their reconfiguration protocol guarantees atomicity, consistency, and isolation properties and real-time constraints by design. Furthermore, it can be utilised to verify properties in discrete and continuous physical environments.

When developing self-adaptive systems which will undergo changes after their deployment, the underlying adaptation needs to be defined and implemented as well. Goldsby et al. [50] propose different levels of requirements engineering and respective processes including a dedicated level for adaptation scenarios and adaptation infrastructure engineering. Braberman et al. [51] propose a reference architecture for configuration and behaviour of self-adaptive systems. More recently, Weyns and Usman [52] bring forward a formally founded approach to develop and implement self-adaptive systems.

Trusting autonomous adaptive systems able to change their

own behaviour is often challenged due to the limited predictability of the systems [53]. Zhang and Cheng [54] introduces formal models for adaptive systems enabling verification of the systems state before and after the adaptation. They focus on verification of software behaviour resulting from collaborative of multiple software components. In a similar way, Han et al. [55] proposed a refinement-based modelling and verification approach for self-adaptive systems. Formal methods for a self-adaptive system and specifically runtime verification of its properties has been well researched [56]–[60]. Recently Finkbeiner and Passing [61] proposed formal methods to synthesise compositional systems. The compositional system generates contracts and certificates, allowing to formulate process requirements for the full system. Bernardeschi et al. [62] use formal verification to find acceptable intervals of values for design parameters in the development of mechatronics systems. Temperekidis et al. [63] propose an approach beyond monitoring individual models. They present a runtime verification module for the master algorithm and co-simulation tool, allowing them to monitor the predicates and events of all involved models of a co-simulated system. Hansen et al. [64] verify that an orchestration algorithm respects all contracts related to the simulation tool's implementation and how to synthesize such tailored orchestration algorithms.

A lot of work has gone into ensuring safety requirements for CPS when changing the behaviour of the system using model-swapping. LaManna et al. [65] propose an approach to define criteria for when a system can safely change its current behaviour. In such a situation, the system can disregard its current obligations and change its behavior to satisfy the new specification. Ghezzi et al. [66] further propose to check the correctness of updates. However, they assume that the system being executed eventually reaches a state in which they can safely update the current behavioural models. To overcome this shortcoming, Nahabedian et al. [24], [67] not only proposed to check the correctness of the future behaviour but also to automatically generate a controller to guide the system into a safe transition state. More details on control-theoretical software adaptation can be found in an extensive survey by Shevtsov et al. [68].

Bellman et al. [69] highlight the importance of self-integrating systems and their ability to master continuous change. They argue that models need to be able to replace models during execution. With changing models and underlying software, we also have to reconsider our testing approaches in a dynamically changing world. Bertolino and Inverardi [70] highlight the challenges and outline approaches to tackle them. We argue that not only models need to be replaced but also model structures need to be adaptable at runtime and tested accordingly.

Recently, Gomes et al. [18] present the FMI 3.0 standard and specifically discuss the different types of clock-based simulations. They discuss synchronous clocked simulations and scheduled execution, where the former utilises a common clock for all events while the latter relies on real-time simulation of black-box models.

Finally, Brockhoff et al. [71] discuss the potential of process mining for process discovery and even process prediction. While this is not directly related to adapting simulation models and replacing models at runtime, the outlined approaches could be utilised to automatically identify changes in processes and trigger a model exchange dynamically.

VI. Conclusion and Future Work

Digital Twins, as virtual replicas of cyber-physical systems, can follow the behaviour of their physical counterparts in real-time, and thus provide support for a range of operations such as monitoring, prediction, maintenance, learning, reconfiguration, and self-adaptation, among others. However, the realization of such systems is not trivial, partly due to the fact that DTs may run alongside their CPS for a long time, resulting in a divergence between the real (actual) and modeled behaviour of the system. Such divergence can be due to inaccurate models, where the error may accumulate over time, but also due to the wear and tear of the physical system or the changes in the environment that render what initially were adequate models useless. In either case, there is a clear need of mechanisms that allows users to update models, preferably during runtime, to avoid costs related to the entire shutdown and restart of the whole system. In this paper we tackle precisely this problem, and propose such mechanisms that enable us to update the DT during runtime, in a controlled manner. We investigate the utility of these mechanisms in two case-studies concerning update of models in a DT for a prototype of an agricultural robot. Our results show that the swap of models can be done in a controlled manner, leaving the system in a stable state after the swap has taken place.

Nevertheless, there are several issues that need to be tackled in future research. Among others, we have to incorporate safety and security aspects in the DevOps cycle, to ensure the updates operate correctly and within defined safety requirements [72], [73]. Furthermore, we have observed in the results a deviation between the Digital Twin and the CPS, and we speculate that this is introduced due to communication delays and inaccurate models. In future research it is crucial to close this *reality gap* between models and physical world, and provide bounds for the tolerated error, such that the whole system reacts more accurately and in a timely manner. To achieve this, we require an interplay between runtime model calibration [74] and synchronisation [75] of the DT and the CPS.

Acknowledgements

We would like to thank Jakob Levisen Kvistgaard for his help with the Desktop Robotti case-study. We would also like to thank Innovation Foundation Denmark and ITEA for funding the UPSIM project. In addition we acknowledge the Poul Due Jensen Foundation that funded our basic research for engineering of digital twins.

REFERENCES

[1] J. Shi, J. Wan, H. Yan, and H. Suo, "A survey of cyber-physical systems," in *Proceedings of the International Conference on Wireless Communications and Signal Processing (WCSP)*, 2011, pp. 1–6.

[2] L. Esterle and R. Grosu, "Cyber-physical systems: challenge of the 21st century," *e & i Elektrotechnik und Informationstechnik*, vol. 133, no. 7, pp. 299–303, 2016.

[3] J. C. Jensen, D. H. Chang, and E. A. Lee, "A model-based design methodology for cyber-physical systems," in *Proceedings of the International Wireless Communications and Mobile Computing Conference*, 2011, pp. 1666–1671.

[4] I. Graja, S. Kallel, N. Guermouche, S. Cheikhrouhou, and A. Hadj Kacem, "A comprehensive survey on modeling of cyber-physical systems," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 15, p. e4850, 2020, e4850 cpe.4850.

[5] P. G. Larsen, C. Thule, K. Lausdahl, V. Bandur, C. Gamble, E. Brosse, A. Sadovykh, A. Bagnato, and L. D. Couto, "Integrated Tool Chain for Model-Based Design of Cyber-Physical Systems," in *The 14th Overture Workshop: Towards Analytical Tool Chains*, P. G. Larsen, N. Plat, and N. Battle, Eds. Cyprus: Aarhus University, Department of Engineering, November 2016, pp. 63–78, ECE-TR-28.

[6] C. Gomes, C. Thule, D. Broman, P. G. Larsen, and H. Vangheluwe, "Co-simulation: a Survey," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 49:1–49:33, May 2018.

[7] M. W. Grieves, "Product lifecycle management: the new paradigm for enterprises," *International Journal of Product Development*, vol. 2, no. 1-2, pp. 71–84, 2005.

[8] J. S. Fitzgerald, P. G. Larsen, and K. G. Pierce, "Multi-modelling and co-simulation in the engineering of cyber-physical systems: Towards the digital twin," in *From Software Engineering to Formal Methods and Tools, and Back - Essays Dedicated to Stefania Gnesi on the Occasion of Her 65th Birthday*, ser. Lecture Notes in Computer Science, M. H. ter Beek, A. Fantechi, and L. Semini, Eds., vol. 11865. Springer, 2019, pp. 40–55.

[9] D. Jones, C. Snider, A. Nassehi, J. Yon, and B. Hicks, "Characterising the digital twin: A systematic literature review," *CIRP Journal of Manufacturing Science and Technology*, vol. 29, pp. 36–52, 2020.

[10] D. Tola, T. Böttjer, P. G. Larsen, and L. Esterle, "Towards modular digital twins of robot systems," in *Proceedings of the International Conference on Autonomic Computing and Self-Organizing Systems Companion*, 2022, pp. 95–100.

[11] J. Millitzer, D. Mayer, C. Henke, T. Jersch, C. Tamm, J. Michael, and C. Ranisch, "Recent developments in hardware-in-the-loop testing," *Model Validation and Uncertainty Quantification, Volume 3*, pp. 65–73, 2019.

[12] T. Blochwitz, M. Otter, M. Arnold, C. Bausch, C. Clauß, H. Elmqvist, A. Junghanns, J. Mauss, M. Monteiro, T. Neidhold *et al.*, "The functional mockup interface for tool independent exchange of simulation models," in *Proceedings of the International Modelica Conference*. Linköping University Press, 2011, pp. 105–114.

[13] T. Blochwitz, M. Otter, J. Åkesson, M. Arnold, C. Clauss, H. Elmqvist, M. Friedrich, A. Junghanns, J. Mauss, D. Neumerkel, H. Olsson, and A. Viel, "Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models," in *Proceedings of the International Modelica Conference*. The Modelica Association, 2012, pp. 173–184.

[14] C. Gomes, T. Blochwitz, C. Bertsch, K. Wernersson, K. Schuch, R. Pierre, O. Kotte, I. Zacharias, M. Blesken, T. Sommer *et al.*, "The fmi 3.0 standard interface for clocked and scheduled simulations," in *Proceedings of the International Modelica Conferences*, 2021, pp. 27–36.

[15] C. Thule, M. Palmieri, C. Gomes, K. Lausdahl, H. D. Macedo, N. Battle, and P. G. Larsen, "Towards reuse of synchronization algorithms in co-simulation frameworks," in *Software Engineering and Formal Methods*. Springer International Publishing, 2020, pp. 50–66.

[16] S. T. Hansen, C. Thule, and C. Gomes, "An fmi-based initialization plugin for into-cps maestro 2," in *Proceedings of the International Conference on Software Engineering and Formal Methods*. Cham: Springer International Publishing, 2021, pp. 295–310.

[17] Modelica Association, "Functional Mock-up Interface for Model Exchange and Co-Simulation," https://fmi-standard.org/downloads/, 2020.

[18] C. Gomes, M. Najafi, T. Sommer, M. Blesken, I. Zacharias, O. Kotte, P. R. Mai, K. Schuch, K. Wernersson, C. Bertsch, T. Blochwitz, and A. Junghanns, "The FMI 3.0 Standard Interface for Clocked and Scheduled Simulations," in *Proceedings of the International Modelica Conference*, no. 181, 2021, pp. 27–36.

[19] J. Bastian, C. Clauß, S. Wolf, and P. Schneider, "Master for co-simulation using FMI," in *Linköping Electronic Conference Proceedings*. Linköping University Electronic Press, 2011.

[20] C. Thule, C. Gomes, and K. G. Lausdahl, "Formally verified fmi enabled external data broker: Rabbitmq fmu," in *Proceedings of the Summer Simulation Conference*. San Diego, CA, USA: Society for Computer Simulation International, 2020.

[21] M. Frasheri, H. Ejersbo, C. Thule, and L. Esterle, "Rmqfmu: Bridging the real world with co-simulation for practitioners," in *Proceedings of the International Overture Workshop*, 2021, pp. 66–80.

[22] F. Foldager, O. Balling, C. Gamble, P. G. Larsen, M. Boel, and O. Green, "Design Space Exploration in the Development of Agricultural Robots," in *AgEng conference*, Wageningen, The Netherlands, July 2018.

[23] M. Frasheri, H. Ejersbo, C. Thule, C. Gomes, J. L. Kvistgaard, P. G. Larsen, and L. Esterle, "Addressing time discrepancy between digital and physical twins," *Robotics and Autonomous Systems*, vol. 161, p. 104347, 2023.

[24] L. Nahabedian, V. Braberman, N. D'Ippolito, S. Honiden, J. Kramer, K. Tei, and S. Uchitel, "Assured and correct dynamic update of controllers," in *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. New York, NY, USA: Association for Computing Machinery, 2016, p. 96–107.

[25] M. A. Mohamed, G. Kardas, and M. Challenger, "Model-driven engineering tools and languages for cyber-physical systems–a systematic literature review," *IEEE Access*, vol. 9, pp. 48 605–48 630, 2021.

[26] L. Baresi and C. Ghezzi, "The disappearing boundary between development-time and run-time," in *Proceedings of the Workshop on Future of Software Engineering Research*, 2010, pp. 17–22.

[27] B. H. Ahmed, S. P. Lee, M. T. Su, and A. Zakari, "Dynamic software updating: a systematic mapping study," *IET Software*, vol. 14, no. 5, pp. 468–481, 2020.

[28] B. Combemale and M. Wimmer, "Towards a model-based devops for cyber-physical systems," in *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment: Second International Workshop, DEVOPS 2019, Château de Villebrumier, France, May 6–8, 2019, Revised Selected Papers 2*. Springer, 2020, pp. 84–94.

[29] J. Hugues, A. Hristosov, J. J. Hudak, and J. Yankel, "Twinops - devops meets model-based engineering and digital twins for the engineering of cps," in *Proceedings of the International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, 2020.

[30] J. Dobaj, A. Riel, T. Krug, M. Seidl, G. Macher, and M. Egretzberger, "Towards digital twin-enabled devops for cps providing architecture-based service adaptation & verification at runtime," in *Proceedings of the Symposium on Software Engineering for Adaptive and Self-Managing Systems*. New York, NY, USA: Association for Computing Machinery, 2022, p. 132–143.

[31] S. E. Jørgensen, "Structural dynamic model," *Ecological Modelling*, vol. 31, no. 1, pp. 1–9, 1986.

[32] T. Ören, "Model update: A model specification formalism with a generalized view of discontinuity." in *Proceedings of the Summer Computer Simulation Conference*, 1987, pp. 689–694.

[33] F. J. Barros, "Modeling formalisms for dynamic structure systems," *ACM Transactions on Modeling and Computer Simulations*, vol. 7, no. 4, p. 501–515, 1997.

[34] L. Yilmaz and T. Oren, "Dynamic model updating in simulation with multimodels: A taxonomy and a generic agent-based architecture," *Simulation Series*, vol. 36, no. 4, p. 3, 2004.

[35] A. M. Uhrmacher, "Dynamic structures in modeling and simulation: A reflective approach," *ACM Transactions on Modeling and Computer Simulations*, vol. 11, no. 2, p. 206–232, 2001.

[36] H. Nilsson and G. Giorgidze, "Exploiting structural dynamism in functional hybrid modelling for simulation of ideal diodes," in *Proceedings of the Congress on Modelling and Simulation*. Citeseer, 2010.

[37] D. Zimmer, *Equation-based modeling of variable-structure systems*. ETH Zurich, 2010.

[38] A. Mehlhase, "A python framework to create and simulate models with variable structure in common simulation environments," *Mathematical and Computer Modelling of Dynamical Systems*, vol. 20, no. 6, pp. 566–583, 2014.

[39] M. Karner, E. Armengaud, C. Steger, and R. Weiss, "Holistic simulation of flexray networks by using run-time model switching," in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition*, 2010, pp. 544–549.

[40] M. Karner, C. Steger, R. Weiss, and E. Armengaud, "Optimizing HW/SW co-simulation based on run-time model switching," in *Proceedings of the Forum on specification and Design Languages*. IEEE, 2009, pp. 1–6.

[41] M. Karner, E. Armengaud, C. Steger, and R. Weiss, "Efficient run-time co-simulation model switching for holistic analysis of embedded systems," *International Journal on Embedded Systems*, vol. 5, no. 4, pp. 208–224, 2013.

[42] P. Lavin, J. Young, R. W. Vuduc, and J. Beard, "Online model swapping for architectural simulation," in *Proceedings of the Computing Frontiers Conference*. ACM, 2021, pp. 102–112.

[43] G. Blair, N. Bencomo, and R. B. France, "Models@ run. time," *Computer*, vol. 42, no. 10, pp. 22–27, 2009.

[44] B. H. C. Cheng, K. I. Eder, M. Gogolla, L. Grunske, M. Litoiu, H. A. Müller, P. Pelliccione, A. Perini, N. A. Qureshi, B. Rumpe, D. Schneider, F. Trollmann, and N. M. Villegas, *Using Models at Runtime to Address Assurance for Self-Adaptive Systems*. Cham: Springer International Publishing, 2014, pp. 101–136.

[45] A. Bennaceur, R. France, G. Tamburrelli, T. Vogel, P. J. Mosterman, W. Cazzola, F. M. Costa, A. Pierantonio, M. Tichy, M. Akşit *et al.*, "Mechanisms for leveraging models at runtime in self-adaptive software," in *Models@ run. time*. Springer, 2014, pp. 19–46.

[46] S. Götz, I. Gerostathopoulos, F. Krikava, A. Shahzada, and R. Spalazzese, "Adaptive exchange of distributed partial models@ run. time for highly dynamic systems," in *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, 2015, pp. 64–70.

[47] N. Bencomo, S. Götz, and H. Song, "Models@ run. time: a guided tour of the state of the art and research challenges," *Software & Systems Modeling*, vol. 18, no. 5, pp. 3049–3082, 2019.

[48] F. Fouquet, B. Morin, F. Fleurey, O. Barais, N. Plouzeau, and J.-M. Jezequel, "A dynamic component model for cyber physical systems," in *Proceedings of the Symposium on Component Based Software Engineering*, 2012, pp. 135–144.

[49] C. Heinzemann, S. Becker, and A. Volk, "Transactional execution of hierarchical reconfigurations in cyber-physical systems," *Software & Systems Modeling*, vol. 18, no. 1, pp. 157–189, 2019.

[50] H. J. Goldsby, P. Sawyer, N. Bencomo, B. H. Cheng, and D. Hughes, "Goal-based modeling of dynamically adaptive system requirements," in *Proceedings of the International Conference and Workshop on the Engineering of Computer based Systems*. IEEE, 2008, pp. 36–45.

[51] V. Braberman, N. D'Ippolito, J. Kramer, D. Sykes, and S. Uchitel, "Morph: A reference architecture for configuration and behaviour self-adaptation," in *Proceedings of the International Workshop on Control Theory for Software Engineering*, 2015, p. 9–16.

[52] D. Weyns and U. M. Iftikhar, "Activforms: A formally-founded model-based approach to engineer self-adaptive systems," *ACM Transactions on Software Engineering Methodology*, apr 2022.

[53] P. Andras, L. Esterle, M. Guckert, T. A. Han, P. R. Lewis, K. Milanovic, T. Payne, C. Perret, J. Pitt, S. T. Powers *et al.*, "Trusting intelligent machines: Deepening trust within socio-technical systems," *IEEE Technology and Society Magazine*, vol. 37, no. 4, pp. 76–83, 2018.

[54] J. Zhang and B. H. Cheng, "Model-based development of dynamically adaptive software," in *Proceedings of the International Conference on Software Engineering*, 2006, pp. 371–380.

[55] D.-S. Han, Q.-L. Yang, J.-C. Xing, and G.-L. Ma, "Easymodel: A refinement-based modeling and verification approach for self-adaptive software," *Journal of Computer Science and Technology*, vol. 35, no. 5, pp. 1016–1046, 2020.

[56] D. Weyns, M. U. Iftikhar, D. G. De La Iglesia, and T. Ahmad, "A survey of formal methods in self-adaptive systems," in *Proceedings of the International Conference on Computer Science and Software Engineering*, 2012, pp. 67–79.

[57] G. Tamura, N. M. Villegas, H. A. Müller, J. P. Sousa, B. Becker, G. Karsai, S. Mankovskii, M. Pezzè, W. Schäfer, L. Tahvildari, and K. Wong, "Towards practical runtime verification and validation of self-adaptive software systems," in *Software Engineering for Self-Adaptive Systems II*, 2013, pp. 108–132.

[58] A. Filieri and G. Tamburrelli, "Probabilistic verification at runtime for self-adaptive systems," in *Assurances for Self-Adaptive Systems*. Springer, 2013, pp. 30–59.

[59] W. Yang, C. Xu, Y. Liu, C. Cao, X. Ma, and J. Lu, "Verifying self-adaptive applications suffering uncertainty," in *Proceedings of the International Conference on Automated Software Engineering*, 2014, pp. 199–210.

[60] R. Calinescu, S. Gerasimou, K. Johnson, and C. Paterson, "Using runtime quantitative verification to provide assurance evidence for self-adaptive software," in *Software Engineering for Self-Adaptive Systems III. Assurances*. Cham: Springer International Publishing, 2017, pp. 223–248.

[61] B. Finkbeiner and N. Passing, "Compositional synthesis of modular systems," *Innovations in Systems and Software Engineering*, vol. 18, no. 3, pp. 455–469, 2022.

[62] C. Bernardeschi, P. Dini, A. Domenici, M. Palmieri, and S. Saponara, "Formal verification and co-simulation in the design of a synchronous motor control algorithm," *Energies*, vol. 13, no. 16, 2020.

[63] A. Temperekidis, N. Kekatos, and P. Katsaros, "Runtime verification for fmi-based co-simulation," in *Proceedings of the International Conference on Runtime Verification*, vol. 13498. Springer, 2022, pp. 304–313.

[64] S. T. Hansen, C. Thule, C. Gomes, J. van de Pol, M. Palmieri, E. O. Inci, F. Madsen, J. Alfonso, J. Á. Castellanos, and J. M. Rodriguez-Fortun, "Verification and synthesis of co-simulation algorithms subject to algebraic loops and adaptive steps," *International Journal on Software Tools for Technology Transfer*, vol. 24, no. 6, pp. 999–1024, 2022.

[65] V. P. La Manna, J. Greenyer, C. Ghezzi, and C. Brenner, "Formalizing correctness criteria of dynamic updates derived from specification changes," in *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2013, pp. 63–72.

[66] C. Ghezzi, J. Greenyer, and V. P. L. Manna, "Synthesizing dynamically updating controllers from changes in scenario-based specifications," in *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2012, pp. 145–154.

[67] L. Nahabedian, V. Braberman, N. D'Ippolito, S. Honiden, J. Kramer, K. Tei, and S. Uchitel, "Dynamic update of discrete event controllers," *IEEE Transactions on Software Engineering*, vol. 46, no. 11, pp. 1220–1240, 2020.

[68] S. Shevtsov, M. Berekmeri, D. Weyns, and M. Maggio, "Control-theoretical software adaptation: A systematic literature review," *IEEE Transactions on Software Engineering*, vol. 44, no. 8, pp. 784–810, 2018.

[69] K. Bellman, J. Botev, A. Diaconescu, L. Esterle, C. Gruhl, C. Landauer, P. R. Lewis, P. R. Nelson, E. Pournaras, A. Stein *et al.*, "Self-improving system integration: Mastering continuous change," *Future Generation Computer Systems*, vol. 117, pp. 29–46, 2021.

[70] A. Bertolino and P. Inverardi, "Changing software in a changing world: How to test in presence of variability, adaptation and evolution?" in *From Software Engineering to Formal Methods and Tools, and Back*. Springer, 2019, pp. 56–66.

[71] T. Brockhoff, M. Heithoff, I. Koren, J. Michael, J. Pfeiffer, B. Rumpe, M. S. Uysal, W. M. Van Der Aalst, and A. Wortmann, "Process prediction with digital twins," in *Proceedings of the International Conference on Model Driven Engineering Languages and Systems Companion*. IEEE, 2021, pp. 182–187.

[72] H. Yasar and S. E. Teplov, "Devsecops in embedded systems: An empirical study of past literature," in *Proceedings of the International Conference on Availability, Reliability and Security*, 2022.

[73] M. Ugarte Querejeta, L. Etxeberria, and G. Sagardui, "Towards a devops approach in cyber physical production systems using digital twins," in *Proceedings of the International Conferene on Computer Safety, Reliability, and Security Workshops*, 2020, pp. 205–216.

[74] J. Woodcock, C. Gomes, H. D. Macedo, and P. G. Larsen, "Uncertainty Quantificatios and Runtime Monitoring Using Environment-Aware Digital Twins," in *Leveraging Applications of Formal Methods, Verification and Validation: Tools and Trends*, vol. 12479. Springer International Publishing, 2021, pp. 72–87.

[75] M. Frasheri, H. Ejersbo, C. Thule, and L. Esterle, "Rmqfmu: Bridging the real world with co-simulation for practitioners," in *Proceedings of the International Overture Workshop*, 2021, pp. 66–80.