Article:
Gil, Santiago, Badyal, Arjun, Miyazawa, Alvaro et al. (2026) A model-based approach for co-simulation-driven digital twins in robotics. Robotics and Autonomous Systems. 105240. ISSN: 0921-8890

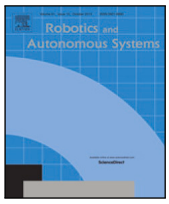
https://doi.org/10.1016/j.robot.2025.105240

Robotics and Autonomous Systems 196 (2026) 105240

Contents lists available at ScienceDirect

Robotics and Autonomous Systems

journal homepage: www.elsevier.com/locate/robot



A model-based approach for co-simulation-driven digital twins in robotics

Santiago Gil [a,*], Arjun Badyal [b], Alvaro Miyazawa [b], Peter Gorm Larsen [a], Ana Cavalcanti [b]

[a] Department of Electrical and Computer Engineering, Aarhus University, Finlandsgade 22, Aarhus, 8200, Denmark
[b] Department of Computer Science, University of York, Deramore Lane, York, YO10 5GH, United Kingdom

ARTICLE INFO

Keywords:
Robotics
Co-simulation
Digital twin
Model-based engineering

ABSTRACT

A digital twin (DT) for a robot can support its development and deployment; it is a valuable resource for simulation and monitoring. Creating a DT for a robot, however, is not an easy task, involving heterogeneous simulation models potentially developed by several stakeholders. This paper proposes a systematic and highly automated approach to develop a DT for a robot based on diagrammatic models and on an industry standard for co-simulation: the Functional Mockup Interface (FMI). Our modelling notation is RoboSim, a tool-independent framework to model, verify, and generate code for control software and for simulations of physical robotic platforms. We take advantage of RoboSim's facilities for structured modelling and code generation to obtain results that help bridge the reality gap and produce DTs with less engineering effort. We present here our technique, using a manufacturing cell as a case study, and its assessment based on existing criteria for DT frameworks. The evaluation establishes that our technique provides significant coverage (specifically, 60%) of the Digital Twinning spectrum.

1. Introduction

A concept first introduced by Grieves [1], a Digital Twin (DT) is a key enabling technology of the digital industry, usually called Industry 4.0 [2]. A DT represents a physical object, the Physical Twin (PT), throughout that object's life-cycle, and provides services, such as, monitoring, optimisation, testing, virtual commissioning, and visualisation, among others [3,4]. The realisation of a DT relies on other enabling technologies [5], including, simulation, Internet of Things (IoT), and information and communication technologies.

DTs have attracted the attention of the robotics community as a key technology to integrate robots in virtual environments [6]. The field has been described as *DT-incorporated Robotics* by Mazumder et al. in a survey [7]. Applications indicated in [7] include virtual and augmented reality, smart manufacturing via simulation and DT-based maintenance, haptic telerobotics, and DT-aided AI. Research surveyed in [7] covers space, medical and rehabilitation, industrial, and soft robotics, as well as the area of human–robot interaction.

Although the use of DTs in robotics looks promising, there is still a gap in terms of frameworks for realising DTs in this domain. In general, DT applications are tailored to case-specific needs, limiting the wider adoption of this technology [8]. Recent surveys on DT platforms [9] and open-source DT frameworks [10] provide a general overview of specialised tooling for DT engineering; however, they do not mention any requirement for applications in robotics. Similarly, existing simulators for robotics do not provide specialised support for integrating their simulations with PTs or with external software that takes advantage of the data provided by a simulation to provide the value-added services of a DT. This lack of support leads to high development and deployment costs of DTs for robotic systems [7,11].

Modelling and simulating robots, as required for a DT, however, is a complex task that requires expertise and multidisciplinary knowledge [12]. This complexity is increased when modelling DT-enabled robotic systems since other assets, including data interfacing, services provided by the DT, and synchronisation need to be considered [7], challenging the integration and accuracy behind the (heterogeneous) components of the DT.

Co-simulation [13] has been favoured as a key technology to reduce the complexity of combining DT assets, namely, simulations of heterogeneous components, and communication and synchronisation infrastructure. Co-simulation enables the composition of simulation code for heterogeneous models and simulators, and has been proved to address the challenge of development of hierarchical DTs [14,15]. Co-simulation also addresses some of the needs for scalability of DTs, since multiple simulations, obtained from different vendors or repositories, can be integrated to obtain simulations of more complex systems [8].

* Corresponding author.
E-mail address: sgil@ece.au.dk (S. Gil).

https://doi.org/10.1016/j.robot.2025.105240
Received 2 December 2024; Received in revised form 7 March 2025; Accepted 21 October 2025
Available online 28 October 2025

There are several standards for co-simulation [13]. In the work presented here, we adopt an industry standard: the Functional Mock-up Interface (FMI) [16], which provides guidance and resources for the development of both the components of a co-simulation and for the co-simulation as a whole.

In FMI, the components of a co-simulation, known as Functional Mock-up Units (FMUs), implement an Application Program Interface (API) that enables their use in a co-simulation. In this setting, FMUs can be replaced by other FMUs [17] to achieve different co-simulation setups, promoting modularity and reuse. Moreover, there is support for integrating FMI co-simulations into a DT platform [18,19]. Creating FMUs, however, requires modelling and coding effort.

This paper proposes and applies an approach to model-based development of DTs for robotics using RoboSim [20], a tool-independent modelling framework for design, verification, and execution of simulations of robotic systems. For the control software, the RoboSim diagrammatic platform-independent behavioural and timed models are based on state machines defined in terms of events and operations representing services of the robotic platform. For the platform itself, the RoboSim models are block diagrams that define links, joints, sensors, and actuators, and (differential) equations defining their behaviour. Finally, a mapping between the platform-independent software model and the robot physical model define how the services used in the state machines are provided by the robotic platform.

RoboSim provides support for automatic verification and code generation. In this paper, we propose the use of RoboSim to create DTs. Our approach, taking advantage of RoboSim's capabilities, can be used to develop verified co-simulations. We support highly automated integration of code automatically generated from RoboSim models to an FMI interface, and thus, enable co-simulation setups based on RoboSim models.

Due to the modularity of RoboSim models (composed of a platform-independent software model, a platform model, and a separate mapping between them) the generated code used for the (co-)simulation of the control software can be directly deployed in the real robot, avoiding coding mistakes. This is in contrast with approaches where the code for simulation of the software and platform are intertwined and written using the API of a simulator. In that case, the control software is not a self-contained component that can be easily reused. With our approach, we also provide ready support for co-simulation with hardware-in-the-loop, since the mapping can be realised to connect to an FMU for the platform simulation, or to a physical platform.

Importantly, in defining how the services used in the software are implemented using the facilities of the platform, the mapping encodes assumptions made about the capabilities of sensors and actuators. So, in face of a reality gap between the co-simulation and a deployment, those assumptions can be revisited, without impact on the simulation (FMUs) of the software or platform. Our experiments described here suggest that, moreover, the reality gap is reduced by our modular approachsince the same components can be used plug-&-play in both simulation and deployment, reducing and localising potential issues. They can also be readily replaced by upgraded, calibrated, or adapted components.

We showcase our work using a robotic arm that is part of the Flex-cell, a platform for manufacturing using cooperative robotics developed by Technicon ApS.[1] It has been used in the assessment of other works for DT engineering [21–23]. Here, we use the simulator CoppeliaSim[2] [24] to run the worked out example.

[1] https://technicon.dk/en/

[2] https://www.coppeliarobotics.com/

Remainder. Section 2 presents the background of our work, followed by related work in Section 3. Section 4 introduces our case study and its RoboSim models. Section 5 elaborates on our approach to combine RoboSim capabilities with co-simulation to achieve DTs. Our approach is evaluated, using independent criteria, and discussed in Section 6. Section 7 discusses the main limitations of this work. Finally, Section 8 provides concluding remarks and directions for future work.

2. Background

This section describes the context of our work: co-simulation (Section 2.1), DTs (Section 2.2), and the key role of a co-simulation in developing a DT (Section 2.3).

2.1. Co-simulation

Co-simulation is an emerging technology that extends the capabilities of simulation to heterogeneous systems, with behaviour coupled by combining simulation composites [13]. Simulation offers the ability to run experiments in a risk-free scenario to answer *what-if* questions without involving the real system [25]. Using co-simulation, it is possible to answer the same questions for large systems, where the simulation is composed of heterogeneous simulators that need to be intrinsically interconnected to properly represent the real system. Here, we propose the use of co-simulation also to preserve the structure of design models, improving traceability and reuse during development of robotic systems. It enables the separate consideration of the heterogeneous components of a robotic system, and is also appropriate to deal with multi-robot systems.

Co-simulation is also relevant to hardware-in-the-loop simulations [26], which interact with external equipment. This kind of co-simulation, known as hybrid co-simulation, addresses the issue of coordinating the passage of real time for external equipment and the discrete-time and event-based simulation components.

As noted, a simulation component in the context of FMI is an FMU, a black-box *.zip* file that includes the source code of the implementation of the FMI API functions, miscellaneous resources, and a *ModelDescription.xml* file, which describes the variables, model structure (input and output ports and parameters), and other metadata of the FMU [16]. Such FMUs can be used for *model exchange* and *co-simulation*, as we do here.

An FMI co-simulation requires a master algorithm, usually in the form of an orchestration engine, which coordinates the stepping of the simulation in each FMU and the connections among FMUs that represent the coupling. In this work, we use the Maestro[3] [27] master algorithm, which has wide coverage of the FMI features, and additional support for fault injection and FMU swapping. Our results, however, can be used with any master algorithm faithful to the FMI standard.

Other standards for co-simulation include the IEEE 1516-2010 Standard for Modeling and Simulation (M&S) High Level Architecture,[4] the Distributed Co-simulation Protocol,[5] and the IEEE 1730-2022 Recommended Practice for Distributed Simulation Engineering and Execution Process.[6] As said, FMI, in contrast to these, defines a co-simulation modality [28] via a detailed API that guides the development of both the FMUs and a master algorithm. This provides a clear target for (automatic) generation of co-simulations.

In this work, we use Maestro, RabbitMQ FMU (RMQ FMU)[7] [29,30], which enables a communication channel between the real world and

[3] https://github.com/INTO-CPS-Association/maestro

[4] https://standards.ieee.org/ieee/1516/3744/

[5] https://dcp-standard.org/

[6] https://standards.ieee.org/ieee/1730/10715/

[7] https://github.com/INTO-CPS-Association/fmu-rabbitmq

co-simulation settings, and UniFMU[8] [31], which facilitates the implementation of FMUs by wrapping existing code in popular programming languages to implement the FMI interface. We present these tools as needed to describe our approach.

2.2. Digital twins

According to the first definition by Grieves [1], DTs are representations in the virtual space of physical assets in the real space, connected via data and information that tie both spaces. Over the years, however, the definitions and interpretations for DT have become varied and ambiguous [32,33]. A highly-adopted definition is part of the classification provided by Kritzinger et al. [34], which differentiates a DT from its downgraded versions, Digital Shadow (DS) and Digital Model (DM), as follows. A DT has automatic bi-directional data flow to and from its PT, whereas a DS only has automatic uni-directional data flow coming from its PT, and a DM has no automatic data flow capabilities at all.

This definition, however, does not cover other relevant aspects of DTs known as the *DT constellation* [35]. These include the models and code adopted, reasoning mechanisms and enablers, and services provided to deliver value in view of a business goal [33]. Here, we adopt the definition in Fitzgerald et al. [36] for a DT-enabled system, as the delimitation of the joint system where a PT and a DT co-exist, and the DT offers a range of services without compromising the PT's operation.

The orchestration of the DT constellation plus the support communication between real and virtual spaces are key goals of any engineering process to implement a DT. Such a process is known as DT engineering or *Digital Twinning* [36,37]. At its heart is the definitions of the simulation (models and code) used by the DT.

In particular, modelling and simulation techniques provide foundational bases for DTs, and need to ensure that they are reliable representations of their physical counterparts. This area, however, involves multidisciplinary knowledge and methods, including geometry, physics, behaviour, and rule modelling [38]. Most DT approaches in the literature are case-specific [8,34], which challenges the standardisation and reuse of existing DTs and their components in different settings. Our work presented here uses a domain-specific approach tailored for robotics, but it is application-independent.

2.3. The role of co-simulation in digital twins

DTs are model-centric abstractions, which require a combination of a range of modelling, simulation, and communication components [38], and a wide range of enabling technologies [5]. The coupling of such a variety of components requires interoperability.

For modelling and simulation components, co-simulation provides key features for combining heterogeneous simulation units, and therefore, enabling the exchange of information between different modelling and simulation components of a DT [39]. Additionally, a co-simulation can also be enabled with communication components, which can be used to synchronise DTs and PTs during run-time [30], and thus, close the loop between DTs and PTs for bi-directional data exchange.

Hence, co-simulation, the focus of our work here, plays an important role as a key enabling technology in DT engineering, an argument that is also supported by Hatledal et al. [40]. In fact, given existing frameworks for DT engineering, including those adopted in our work, the development of the DT services to establish a DT constellation is a largely technological problem. The creative effort is in the development of simulation models and code, for which use of the technique presented here brings guidance and automation.

In the next section, we present works on DT for robotics, and explain how our approach provides a contribution over and above existing technology.

[8] https://github.com/INTO-CPS-Association/unifmu

3. Related work

Although the adoption of DT in robotics is relatively recent, it is trending [7]. DT has featured in several contributions in human–robot collaboration [11,41] and robot–robot cooperation. Recent reviews by Baratta et al. [41] and Ramasubramanian et al. [11] indicate that most solutions for DTs in the context of human–robot collaboration are built on top of robotics, 3D, and simulation software rather than specialised tools for DTs. So, extra effort is likely to be needed to set up DTs.

Some of the approaches in DTs for human–robot collaboration include: Martínez et al. [42] proposes a DT demonstrator to support flexible manufacturing tasks using a human–robot interaction setting enabled by a Universal Robots UR3 robotic arm performing cutting operations. Two similar approaches have been presented by Malik and Brem [43], and by Bilberg and Malik [44], who propose DT approaches for assisting assembly tasks in human–robot collaboration settings to address cost and engineering-effort challenges in flexible automation solutions. Their collaborative cells are enabled by Universal Robots UR5e and UR10 robotic arms. The demonstrators above are impressive, but no general approach for DT development is put forward.

The Robot Operating System (ROS) has been widely used to achieve DT implementations for robotics in a variety of applications [45–49]. It provides a suitable communication interface to bind different robotic platforms to their corresponding DTs or DSs. In these approaches, however, code is developed directly, rather than automatically generated. In addition, ROS does not provide a complete solution for DTs, as it does not provide native facilities for communication with a PT and integration with a DT constellation.

Regarding cooperative robots, Gil et al. [21] have proposed a compositional modelling approach in the development of DTs. It is showcased in a cooperative-robotics case study composed of a Kuka LBR iiwa 7 robotic arm and a Universal Robots UR5. This approach has been extended in [22] to integrate robot skills [50] and evaluated with the same case study. This work has been proved to reduce the engineering effort of DT development by enabling the reuse of skills.

Reuse is also supported by the RoboSim approach to modelling. RoboSim control software models can be combined with a RoboSim physical model for any robot that can be used to realise the services required by the software. Symmetrically, a physical robot model can be used with any control-software whose required services it can provide. With this separation, we are able to restrict the dependence of the co-simulation on external, usually closed-source, software. Specifically, only the simulation of the physical platform depends on the use of a simulator (CoppeliaSim [24] in our example). This leads to higher fidelity of the simulation used in a DT, which is a key challenge for DTs in robotics [44].

To alleviate the complexity of engineering DTs, some tools, the so-called *DT platforms*, have been developed. These platforms use metamodels to provide a well-defined semantic structure that speeds up engineering and the real-to-virtual interfacing. A well-known metamodel is Asset Administration Shell (AAS) [51], which has now become part of the IEC 63278-1 standard.

Lehner et al. [9] have analysed three DT platforms using requirements derived from the ISO 23247 standard and with influence of existing literature and the authors' expertise. Gil et al. [10] have also surveyed and analysed open-source DT frameworks using criteria inspired by existing requirements for DTs in the literature and the ISO 23247 standard. That survey covers frameworks for IoT-based, infrastructure, and geospatial applications, and for co-simulation; a domain-specific framework for airplanes is also analysed. Criteria suggested in [9,10] are used here to evaluate our work.

Sciullo et al. [52] proposed a generic framework for engineering DTs using the concept of Web of Things (evolving from the IoT concept) and extending it to support data-driven behavioural models generated from observations. In this approach, only data-driven models are allowed, and these are not necessarily designed for robots or robot applications.

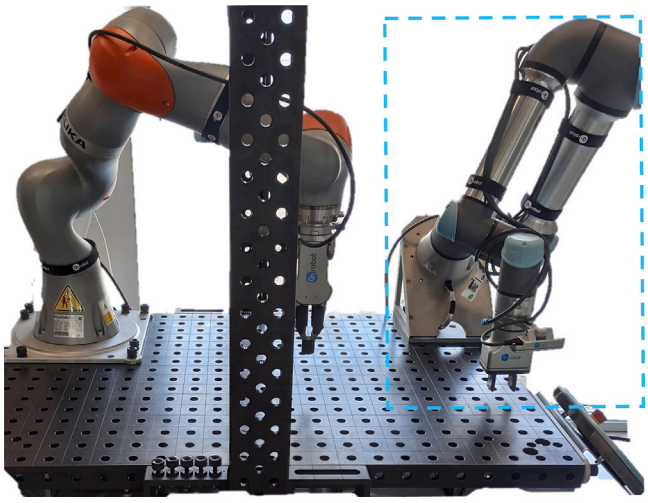


Fig. 1. Flex-cell case study. The dashed box in blue frames the UR5e, which is the scope robot for this research.

Therefore, several tailored models and their integration may be required to represent the different aspects of the robot behaviour needed for a given use case.

Two recent approaches have bound DT platforms with FMI-based co-simulation. The DT Manager in [18] provides an interface to attach existing DT platforms to black-box simulations, including FMUs. The Digital Twin as a Service (DTaaS) platform in [19] provides a hub for engineering DTs focused on reusability and integration with co-simulation tools. Although these two works can be used for the development of DTs for robotics, they do not use robot-specific simulation engines (such as Gazebo, CoppeliaSim, for instance) to simulate robots' functionalities. Therefore, these tools require external dependencies and domain knowledge if they are used for applications in robotics.

A more specialised approach to DT engineering in robotics is presented by Li et al. [53], where a multimodel framework that supports geometric, physical, and rule modelling is proposed. Although this approach is quite promising, the deployment of the resulting DT is done in a stand-alone fashion by combining individual software and hardware tools and the models.

In summary, the works mentioned above have not focused on the needs of DT engineering for robotics, or do not provide sufficient generalisability for realising DTs of robots while keeping the consistency between the engineering of PTs and DTs, leading to high development expenses time- and resource-wise [7].

4. Case study and RoboSim

Our Flex-cell is a manufacturing cell composed of two robotic arms; we use a Kuka LBR iiwa 7 and a UR5e. This setup has been used to demonstrate methods for DTs in cooperative systems in previous works [21,22,54]. We use it to demonstrate our approach, and in particular focus on the UR5e robot. Fig. 1 shows the Flex-cell, with the UR5e on the right.

In Section 4.1, we give an overview of the Flex-cell. Section 4.2 describes its RoboSim model for the control software; Section 4.3, its RoboSim physical model; and Section 4.4, the mapping between the two.

For modelling and code generation, we use RoboTool [55], which supports creation, validation, and verification of RoboSim models. This includes models of control software (called d-models), of robotic platforms (p-models), and of platform mappings that describe how the sensors and actuators of a p-model implement the services used in a d-model. For d-models, RoboTool provides facilities for textual and graphical development, specification of properties in controlled English, and automatic generation of mathematical models for verification with the FDR4 model checker [56]. In addition, a C and Rust code generator is under development and has been used to produce implementations in our work. For p-models, RoboTool provides plugins for the textual and graphical definition of models, and automatic generation of Simulation Description Format (SDF) models [57], used here, and of mathematical descriptions for verification.

4.1. Flex-cell

The physical Flex-cell used in the work presented here is located at the Digital Transformation Lab (DTL) in Ringkøbing-Skjern, Denmark. It is intended for use in cooperative (robot–robot) and collaborative (human–robot) assembly in a shared working space: the Flex-cell's plate. Such a plate has symmetrically distributed holes, which allow modelling and programming using a discretised space based on the holes.

Currently, the Flex-cell is used mainly on the assembly of 2D structures on the Flex-cell's plate using two cooperative robots. The applications considered, primarily pick-and-place, focus on creating different kinds of shapes using pegs or LEGO bricks, through attaching different types of OnRobot grippers, and the use of robot skills [50] and planners to create such shapes. For this reason, we focus here on a pick-and-place application, where bricks are moved from a source to a target position, using 2FG7 and RG6 OnRobot grippers.

Some of the existing libraries to enable the remote and automatic control of the Flex-cell robots and grippers include the URInterface[9] (for communication with the UR5e), the Kukalbrinterface[10] (for communication with the Kuka LBR iiwa 7), PyModbusTCP[11] (for communication with the OnRobot grippers), and the Robotics Toolbox Python [58] (for calculating the inverse kinematics of the robotic arms). In the work presented here, we use a custom API developed by DTL, on top of the URInterface and The Robotics Toolbox Python libraries, and the PyModbusTCP library.

For example, it is possible to send commands to ask one of the robotic arms to move to a specific hole in a 16×24 grid. To achieve this, we have used an API that translates commands in a discretised space to robot-specific angular positions, including the rotational transformations to ensure that the grippers are always pointing downwards. More specifically, the robots accept the command `movediscrete(X,Y,Z)`, where $X \in \mathbb{N} \cap [0, 15]$ and $Y \in \mathbb{N} \cap [0, 23]$ define a hole combination on the plate, and $Z \in \mathbb{N} \cap [0, 9]$ defines the height. The three axes are discretised according to the hole-to-hole distance, which is 50 mm.

RoboSim models for the application and for the robotic arm are presented next.

4.2. RoboSim d-model

A RoboSim d-model is defined by a module block, including a robotic-platform block and one or more controller blocks. The robotic-platform block declares events and operations that describe services that are available for the software. For our example, the module, called DiscreteGrippingArm, is shown in Fig. 2.

The robotic-platform block RoboticArmGripper abstracts the services provided by the UR5e platform. For example, moveDiscreteCommand is an event representing the function of the same name of our custom API. The values communicated by the event, a tuple of type MovementArgs, represent the parameters of the API function. All

[9] https://gitlab.au.dk/clagms/urinterface
[10] https://github.com/sagilar/kukalbrinterface
[11] https://pypi.org/project/pyModbusTCP/

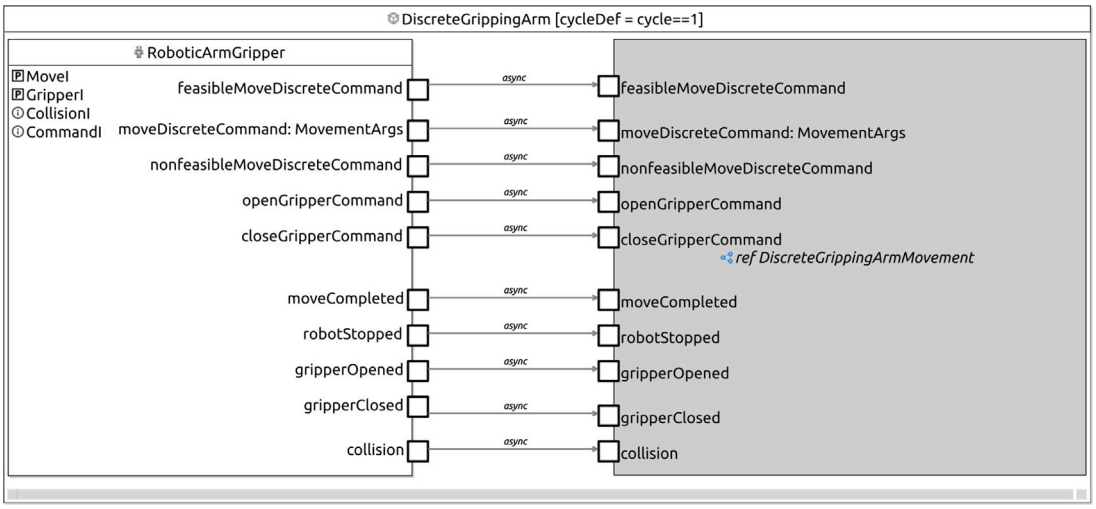


Fig. 2. RoboSim module for the Flex-cell.

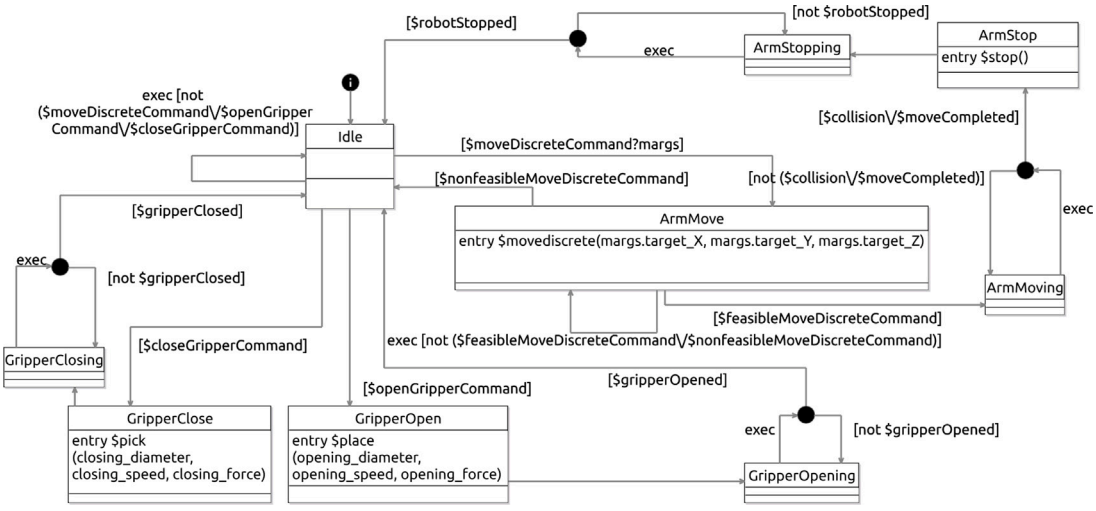


Fig. 3. d-model state machine chart for the UR5e.

events and operations of RoboticArmGripper correspond to a function of the custom API.

The controller block DiscreteGrippingArmMovement is a reference (indicated by the keyword *ref*) to the actual block definition. It is omitted here, but the complete model is available.[12] The behaviour of a controller is given by state machines combined in parallel. Thus, RoboSim d-models can capture parallelism at the controller and machine levels. For our example, the controller has a single state machine SimDiscreteGrippingArmMovement (shown in Fig. 3). So, in this case, the controller just passes input events from the robotic platform to the state machine, and output events and operation calls from the machine to the platform.

RoboSim d-models embed the simulation paradigm, so that evolution is defined by the passage of time, with cycles based on a length of time. For our example, the module (see Fig. 2) defines that the cycle is 1 time unit; specification of the value of a time unit is determined when generating code. In each cycle, all inputs are read and processed, producing the outputs (events and operations call) infinitely fast, before time then passes. Complete information about the inputs is available at the start of each cycle, and the model can observe both the occurrence and non-occurrence of events.

SimDiscreteGrippingArmMovement specifies the behaviour of the system and contains eight states. The state machine starts on the state `Idle` (indicated by the initial junction – black circle with an ***i*** inside), and uses the input events of the platform (for instance, `gripperClosed`) and the current state to decide which state to move to next. State and transitions may execute actions that use the output events and operations of the robotic platform to actuate the robot. For instance, the action in the state `GripperClose` is executed when the state is activated and calls the platform operation `pick`.

SimDiscreteGrippingArmMovement is representative of the pick-and-place applications executed on the Flex-cell. For each pick or place action, the state machine loops from Idle via three available paths, depending on the command received. If no command is received, the guard of the self-transition of Idle is true, and SimDiscreteGrippingArmMovement stays in Idle, but moves on to the next cycle. This is indicated by the keyword exec in the self-transition label.

If the moveDiscreteCommand event is input, then SimDiscreteGrippingArmMovement transitions to the state ArmMove. There, the entry action calls the operation moveDiscrete of the robotic platform. With the assumption that the inputs feasibleMoveDiscreteCommand and nonfeasibleMoveDiscreteCommand are not available in the same cycle, SimDiscreteGrippingArmMovement stays in ArmMove, where a self-transition signals evolution to the next cycle via exec.

[12] https://github.com/INTO-CPS-Association/DigitalTwins_RoboSim

If a nonfeasibleMoveDiscreteCommand is input, SimDiscreteGrippingArmMovement goes back to Idle. If feasibleMoveDiscreteCommand is input instead, SimDiscreteGrippingArmMovement goes to ArmMoving. There, in each cycle (after an exec) a decision is made at a junction (represented by a black circle). If the input does not indicate a collision or moveCompleted, the machine stays in ArmMoving, otherwise it goes to the state ArmStop. There, the behaviour is similar, waiting for the input robotStopped, when SimDiscreteGrippingArmMovement goes back to Idle.

The paths of SimDiscreteGrippingArmMovement to handle the inputs openGripperCommand and closeGripperCommand are specified in a similar way. They represent the behaviours for the picking and placing operations, respectively. A specific application can execute the loops in SimDiscreteGrippingArmMovement as many times as objects to be picked and placed. This can be defined, for example, by a task planner. Our d-model can be deployed in any gripper-enabled robotic arm supporting discrete movement commands.

4.3. p-model

A p-model provides a diagrammatic description of a robot rigid body's components (links, bodies, joints, sensors, and actuators). These components are defined by blocks (of a SysML-like block diagram) that form a graph whose edges represent containment relationships and flexible connections via joints. The behaviour of these components can be specified by (differential) equations that are useful for mathematical proofs. In this paper, we do not make use of this feature, but a full account of the p-model notation is in [59].

The p-model for our example (a robotic arm from Fig. 1) is sketched in Fig. 4. In RoboSim, the p-model block defines the origin and inertial frame from which those of all other blocks are determined. The p-model indicates the components that form (are contained in) the rigid body. Containment relationships (lines with a diamond on the side of the container block) define relative positions to that origin.

In our example, the p-model block contains four links (base_link, shoulder_link, upper_arm_link, forearm_link) corresponding to the first four links (from the base) of the UR5e. A p-model can also include parts, which refer other p-models. In our example, the Wrist is described in a part. It is omitted here, but the full p-model is available[12]. Fixed or flexible connections can be defined between blocks, representing whether or not one block moves with respect to the other. In Fig. 4, the joint shoulder_pan_joint is contained in base_link and is flexibly connected (dashed line with an arrow to the flexibly connected link) to shoulder_link. This means that shoulder_link moves with respect to shoulder_pan_joint, and so, with respect to base_link.

In our example, the joints between each link in Fig. 4 are described as being of type Revolute with the constant AXIS defining their orientation. This is a kind of joint available in the RoboSim library, which can be extended to include customised blocks, including parts. Bodies (blocks labelled body) describe the geometry of the links; in our example, they are Cylinders.

Our p-model includes sensors for the position, velocity (shown in Fig. 4), and torque at each joint. In a p-model, sensors have outputs that are the inputs to a platform mapping (described in the next section), which in turn can be used to define input events in the d-model. In our example, the velocity outputs of the sensors are used to define when the event robotStopped occurs. Conversely, inputs of an actuator block connect to the outputs of the platform mapping, which can be used to define output events and operations in the d-model.

We can generate SDF files automatically from a p-model. Further details about the p-model notation and the SDF generation can be found in [59]. This SDF file can then be parsed by CoppeliaSim to be used for visualisation and by its physics engine. Fig. 5 shows the visualisation of our example.

4.4. Platform mapping

A RoboSim platform mapping connects the services (events and operations) from the robotic-platform block of the d-model to the p-model. It is represented by a block, including an internal block for each service of the d-model. For our example, we have 13 service-blocks: 10 for events, and three for operations.

Each of the three operations, namely, movediscrete, pick, and place, are defined using the external interfaces of the robot: pick and place are defined using the PyModbusTCP interface (for the gripper), and movediscrete using the custom API defined in terms of the URInterface for the actual robotic arm.

For some events, there is a one-to-one relationship with the value of a variable of the platform: they occur if a boolean variable in the platform or in the custom API is true or false. These variables can be declared in the platform mapping. Other events, such as robotStopped, moveCompleted, and collision, can be directly defined in terms of the outputs of sensors defined in the p-model. For robotStopped, the block is presented in Fig. 6. The equations define the conditions that need to hold for the event to occur: the velocity (output_vel) recorded by each of the sensors VSJ1 to VSJ6 for the joints velocities need to be 0.0 m/s.

RoboSim has various facilities for validation and verification of properties of a model. They can be used before code generation to ensure that the obtained simulation or deployment satisfies relevant properties. Next, we describe our technique and resources to generate a DT from a d-model, a p-model, and a mapping.

5. Our digital twinning approach for robotics

In this section, the steps to develop co-simulation-driven DTs from RoboSim models are presented. First, the desired development scenario, leveraging RoboSim models and co-simulation to develop DTs, is explained: we describe the various artefacts in Section 5.1 and the proposed technique in Section 5.2. From there, the development of implementation artefacts is described and illustrated using our case study (from Section 4). Their wrapping into FMUs and connections to define an FMI co-simulation are described in Sections 5.3 and 5.4. The co-simulation resources available to support our digital twinning approach are explained in Section 5.5. Finally, Section 5.6 presents a co-simulation experiment using our case study.

5.1. Artefacts

Fig. 7 provides an overview of the main artefacts involved in our approach to developing DTs based on RoboSim models. RoboSim is used to define the *model artefacts* (d-model, p-model, and platform mapping).

As noted, a d-model can be used to automatically generate a (C) implementation of the control software. A p-model can be used to automatically generate SDF files, useful to create robot simulations using tools like CoppeliaSim. To bring these together, we need a platform-mapping interface: an implementation of a RoboSim platform mapping that enables the connection between the C code for the d-model and the robot (either simulated or real). These are the *implementation artefacts* in Fig. 7 described in Section 5.3.

The implementation artefacts enable the development of a co-simulation using an FMI implementation, with the C code for the d-model and platform-mapping interface components wrapped as FMUs. These are depicted as the *co-simulation artefacts* in Fig. 7.

To drive a co-simulation run, providing the required inputs to the robot, we use an additional FMU, identified in Fig. 7 as the controller FMU. Using this extra FMU, we can construct a co-simulation without modifying the code that is automatically generated from the d-model and the platform-mapping interface.

The SDF file is not an active component, and so not wrapped as an FMU. Instead, we use CoppeliaSim to control the simulated robot

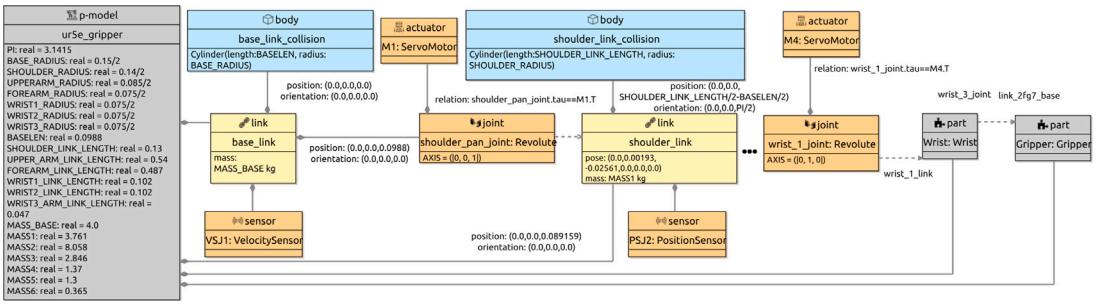


Fig. 4. This is a simplified version of the p-model; full version at github.com/INTO-CPS-Association/DigitalTwins_RoboSim.

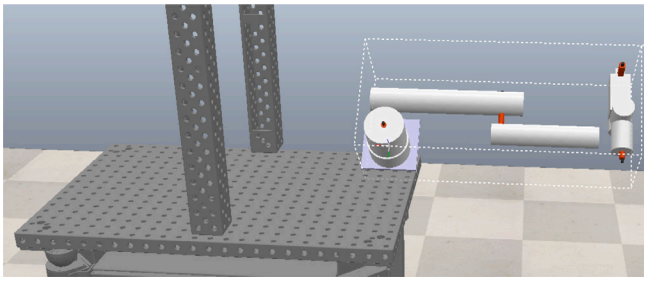


Fig. 5. Rendering of the p-model SDF for the UR5e in CoppeliaSim.

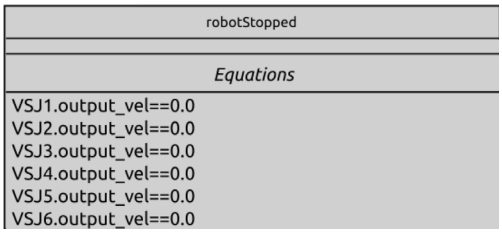


Fig. 6. Block for robotStopped in the platform mapping.

defined in SDF; this includes providing communication endpoints for the simulated robot. We can use other simulators, such as Gazebo, as well, and in future work we will wrap the simulator and the SDF files as a single FMU. CoppeliaSim can import the SDF files generated from RoboSim, run simulations based on position control, and be remotely integrated using its Remote API.[13] The co-simulation artefacts are further described in Section 5.4. Section 5.5 elaborates on how to use those artefacts to enable DTs.

5.2. Methodology

To provide guidance throughout the process of developing co-simulation-driven DTs using RoboSim, a 13-step methodology is proposed here. It starts with the definition of the RoboSim models and goes all the way to orchestrating the co-simulation to enable the DT of a given robot. Fig. 8 summarises the 13 steps, giving a short description for each of them.

Steps 1 and 2 involve writing the RoboSim models for the given robot and generating their C and SDF implementations. This generation is fully automated.

Steps 3 and 4 refer to the implementation artefacts in Fig. 7. Our framework uses a collection of pre-defined templates for Python classes and functions that implement the concepts of a RoboSim platform mapping: events, operations, actions, and equations. In **Step 3**, we use these templates to define classes and functions corresponding to the definitions of the RoboSim model and the interface of the actual robot or its simulation. More details are in Section 5.3.1.

Step 4 sets up the interface of the robotic platform, either in the real robot, for a co-simulation with the hardware in the loop, or in the simulator, CoppeliaSim in our example, using the SDF file. The details of this step are further described in Section 5.3.2.

Steps 5 to 13 are concerned with the FMI integration of the co-simulation artefacts described in Fig. 7. **Step 5** defines the overall design of the input and output ports and variable management to enable the FMUs and the interaction among them (see Section 5.4.1). This is based on the architecture of the co-simulation shown in Fig. 7, reflecting the events and operations of the d-model, and, therefore, of the platform mapping.

Steps 6 and 7 are about the integration of the platform mapping interface into an FMU, that is, the Platform Mapping FMU described in Section 5.4.2. **Step 6** defines the structure of the FMU based on the input and output structure of the RoboSim platform mapping, and **Step 7** defines a step-based evolution for the mapping, in line with the behaviour of a d-model and of its implementation, and with the FMI paradigm.

Similarly, **Steps 8–11** integrate the d-model C code into the d-model FMU; see Section 5.4.3. Finally, **Steps 12 and 13** design the behaviour of simulation scenarios using the controller FMU (Section 5.4.4).

The result of each step for our example is online[12].

5.3. Developing the implementation artefacts

As said, the d-model and the p-model can be used to automatically generate their corresponding implementations as C code and SDF files (which are the results of Steps 1 and 2 in Figs. 8 and 9). For the platform mapping, we create an interface specific to the robot endpoint to be used for communication: in the simulator or in the real robot. Section 5.3.1 elaborates on the engineering of the platform-mapping interface, and Section 5.3.2 on the setup of the robot endpoints.

5.3.1. Platform-mapping interface

Step 3 creates the platform-mapping interface using templates for classes and functions that we have developed and described here. The task in this step is the definition of (a) the endpoint: simulator or robot; (b) state variables, corresponding to outputs of sensors, inputs of actuators, properties of links and joints, and any data to be recorded in simulation; and (c) interface-specific implementations of the events, operations, equations, and actions of the RoboSim platform mapping.

We have developed class templates for `Events` and `Operations`, anonymous functions templates for `Actions` and `Equations`. These functions are in the next step linked to the robot endpoints, so they are automatically triggered when an event occurs or an operation is called, forwarding the functionality to and from the robotic platform. In total, we have templates for three Python classes, namely `InputEvent`, `Operation`, and `Mapping`, and for functions offering generic functionality, including starting and stopping the interface; grouping up

[13] https://manual.coppeliarobotics.com/en/remoteApiOverview.htm

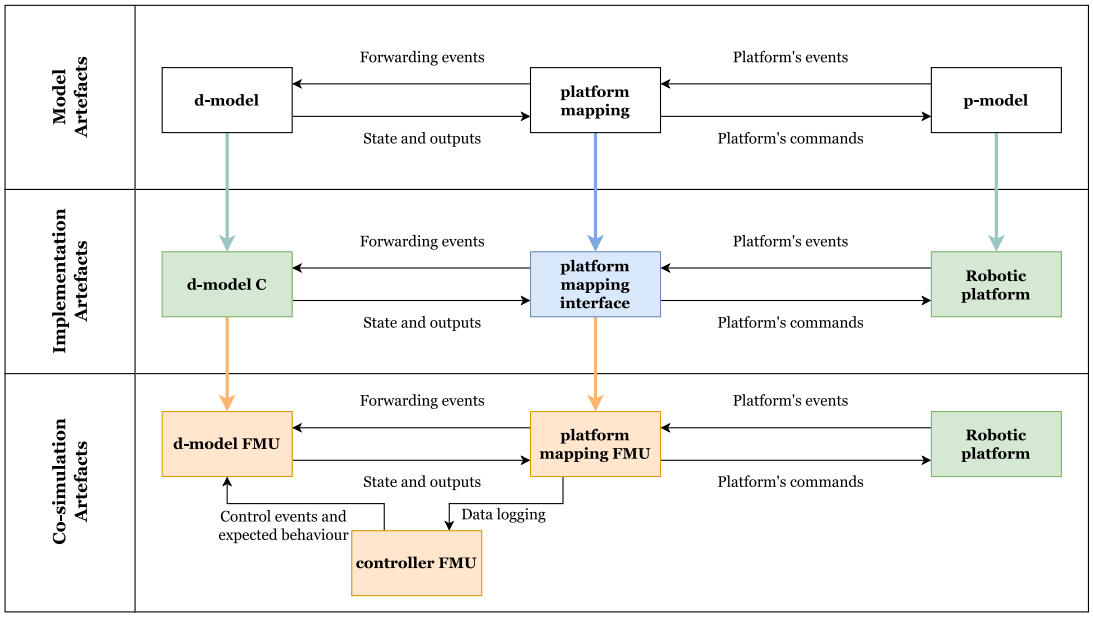


Fig. 7. Three-level architecture for achieving co-simulation-driven DTs extending RoboSim's capabilities. In green: current capabilities to generate implementable code. In blue: model-based implementation of the platform mapping interface (see Section 5.3.1). In orange: extended capabilities to integrate RoboSim implementations with co-simulation (see Section 5.4).

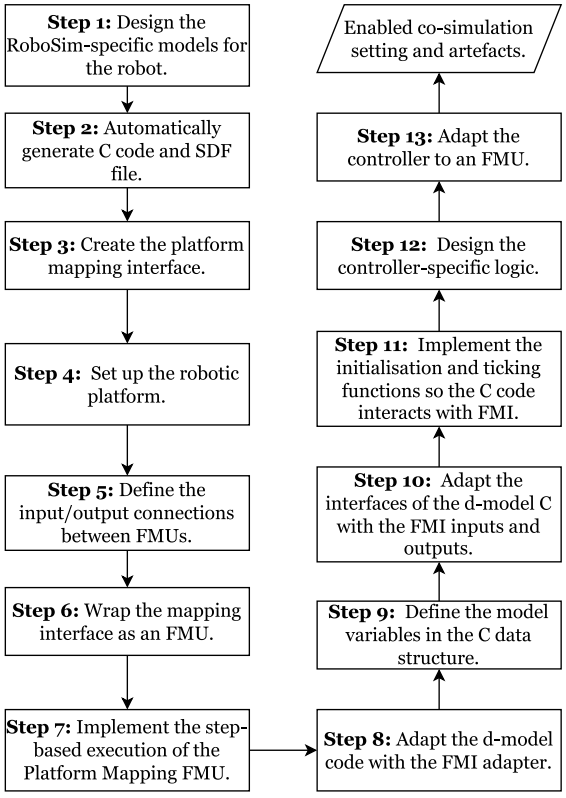


Fig. 8. DT engineering with RoboSim: 13-step technique.

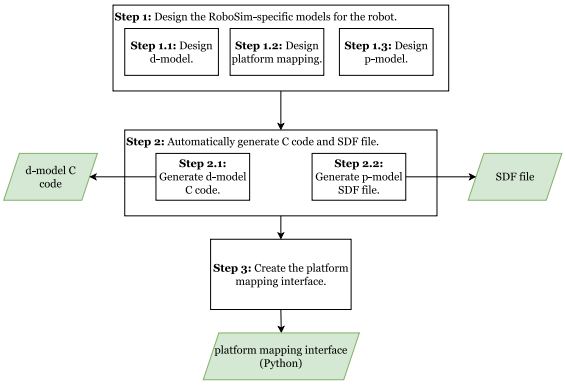


Fig. 9. DT engineering with RoboSim: Steps 1 to 3.

sets of actions and equations into events and operations; executing operations; and getting events.

To create the platform-mapping interface, we need to adapt the template for the class `Mapping`, which is related to communication with an endpoint: the robot or its simulation. In our work presented here, two interfaces are considered, namely, the CoppeliaSim Remote API (to enable the simulated-robot endpoint) and the URInterface (to enable the real-robot endpoint). Other protocols or interfaces can be used to enable different robot endpoints using and adapting our templates.

An instance of `Mapping` is the active component of the platform-mapping interface (see Fig. 7). To use our template, we need first to define the attributes of the class `Mapping`; they record the outputs of the sensors, the inputs of the actuators, and other properties of links and joints in the p-model used in the platform mapping or that need to be recorded during the simulation.

A `Mapping` instance needs to be started using the method `start_mapping()`. The implementation of the d-model and of the robotic platform (simulation or real robot) use the instance of `Mapping` to get input events and execute operations using its methods `get_event(event_name,args)` and `execute_operation(operation_name,args)`. Finally, the `Mapping` instance can be shutdown using the method `stop_mapping()`. In our work, we have developed templates for all these methods.

To implement the methods `get_event(...)` and `execute_operation(...)`, we can use the `InputEvent` and

Operation classes. For an input event in the d-model, we need to create an instance of the InputEvent class with the event's name and the set of equations defined in the RoboSim platform mapping. As illustrated in Listing 1 for the robotStopped event, for an event called ev, we use the constructor of InputEvent to create an object ev, recording the name of the event, and its equations. The equations equations_ev are given in the form of lists of anonymous (lambda) functions. In the example, the equations require that the variables recording the outputs of the velocity sensors for the joints are all 0.0.

Similarly, for an operation, we need to create an instance of the Operation class with the operation name and a set of actions, which define the operation. The actions are given in the form of lists of anonymous functions. For instance, an assignment to a variable var of a value var_target is written lambda var_target: set_variable_target("var",var_target).

```
equations_robotStopped = [
    lambda : get_joint_velocity("j0") == 0.0,
    lambda : get_joint_velocity("j1") == 0.0,
    lambda : get_joint_velocity("j2") == 0.0,
    lambda : get_joint_velocity("j3") == 0.0,
    lambda : get_joint_velocity("j4") == 0.0,
    lambda : get_joint_velocity("j5") == 0.0
]
robotStopped = InputEvent(name="robotStopped",
     equations=equations_robotStopped)
```

Listing 1: Definition in Python of the robotStopped event including its equations in the Platform Mapping Interface.

In summary, the platform-mapping interface allows the definition of the components of a platform mapping using several robotics libraries. So, it is easy to map the d-model events and actions into platform-specific commands via APIs and library-specific interfaces.

Although the use of our templates as described in this section is not automated, they provide clear guidance for the step. Moreover, the definition of the templates is the first step to increase the automation of our technique. The inputs needed are the definition of the p-model variables that need to be recorded under simulation, and the development of the robot endpoints. For particular endpoints, a high level of automation can be achieved, and we will tackle this in future work.

5.3.2. Enabling the robotic platform

Once the platform-mapping interface is created, Step 4, detailed in Fig. 10, setups the robotic platform. The steps to be carried depend on whether we use the real robot (Step 4a), or a simulation for it (Step 4b). Setting up the physical robot requires enabling the robot's interface to be remotely controlled. Setting up the simulated robot requires setting up the interfaces and potential scripts in the simulation engine. For our example, where CoppeliaSim is used, the SDF file needs to be imported and adapted to the static objects (the Flex-cell plate) (Step 4b.1). In addition, CoppeliaSim needs to have installed the plugin CoppeliaSim Remote API so the robot can be controlled and monitored from the platform-mapping interface (Step 4b.2).

5.4. Integration with FMI

As noted, the integration with FMI involves several steps. In Section 5.4.1, we describe how to define the connections between the FMUs as suggested in Fig. 7. In Section 5.4.2, we describe how to create the platform-mapping FMU. Section 5.4.3 describes the development of the d-model FMU, and Section 5.4.4, the development of the controller FMU. They are used as described in Section 5.5 to define an FMI co-simulation.

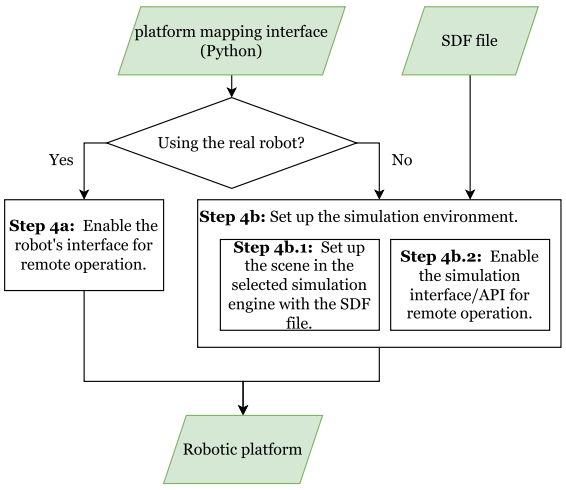


Fig. 10. DT engineering with RoboSim: Step 4.

5.4.1. Connections between FMUs

Since FMUs work as black boxes that are only accessible via their input and output ports at every co-simulation step, the events and operations of the d-model and platform mapping become inputs and outputs of their FMUs. Some of these also become inputs and outputs of the controller FMU (see Fig. 7).

Inputs and outputs are part of what is known as the data structure of an FMU, which is recorded in a ModelDescription.xml file mentioned in Section 2.1 as well as in the implementation of the FMU. This is the object of Step 5 (see Fig. 11 and next section). Through the ModelDescription.xml files, the data for the co-simulation artefacts are shared among the different FMUs via the master algorithm. An extra specification file specifies the connections between the FMUs.

For a d-model FMU, the ModelDescription.xml file must contain an entry for each event or operation of the robotic-platform block of the d-model. If the event communicates values, or the operation has arguments, then these need to be declared. For example, for the input event robotStopped, the entry is shown in Listing 2, which defines the event name, channel/port ("3"), causality ("input"), and data type ("Boolean"). This means that the d-model FMU receives an input event robotStopped of type Boolean via the port number 3, every time there is a co-simulation step.

```
<ScalarVariable name="robotStopped" valueReference="3"
    causality="input" variability="discrete">
      <Boolean start="false"/>
</ScalarVariable>
```

Listing 2: Declaration of the input event robotStopped in the file ModelDescription.xml of the d-model FMU.

Similarly, the file ModelDescription.xml for the platform-mapping FMU has an entry for each event or operation of the robotic-platform block of the d-model. The causality in the ModelDescription.xml files of the d-model FMUs and platform mapping FMU are swapped, since inputs of the d-model FMUs are outputs of the platform-mapping FMU, and vice-versa. For instance, the causality for the event robotStopped, in Listing 2 for the d-model, is "input"; in the platform-mapping FMU, it is "output".

Finally, for the controller FMU, there are entries in the file ModelDescription.xml corresponding to the events and operation calls, and their communicated values or arguments, if any, that are to be controlled or observed via the simulation. For a fully autonomous robot, there may not be a need for a controller FMU at all, but in general, the controller FMU allows the simulation to provide inputs from an external source or record data about the simulation. In our example, we

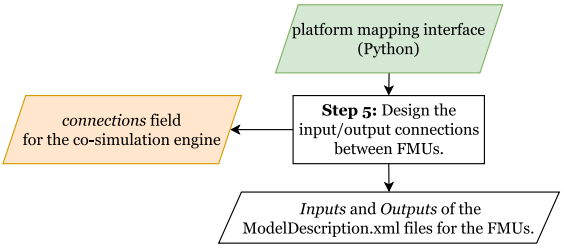


Fig. 11. DT engineering with RoboSim: Step 5.

have entries for the `moveDiscreteCommand`, its arguments `MovementArgs`, `closeGripperCommand`, and `openGripperCommand` declared in Fig. 2. This means that the co-simulation supports the testing of scenarios characterised by the use of these commands in particular orders and with particular argument values.

Based on all the above `ModelDescription.xml` files, we can create the specification of the connections for the co-simulation as a whole. We do that by attaching output ports to any number of input ports in the form of lists. In our example, and in particular for Maestro, Listing 3 shows the connections that enable the co-simulation artefacts of Fig. 7. This listing reflects the behaviour of the coupling between the FMUs, that is, input events coming from the robotic platform (through the platform-mapping FMU) or the testing scenario (the controller FMU) are connected to the d-model FMU. Similarly, the operations from the d-model FMU are connected to the platform-mapping FMU.

```
"controllerFMU.closeGripperCommand" :["dmodelFMU.closeGripperCommand"],
"controllerFMU.openGripperCommand" :["dmodelFMU.openGripperCommand"],
"controllerFMU.moveDiscreteCommand" :["dmodelFMU.moveDiscreteCommand"],
"controllerFMU.MovementArgs_target_X" :["dmodelFMU.MovementArgs_target_X"],
"controllerFMU.MovementArgs_target_Y" :["dmodelFMU.MovementArgs_target_Y"],
"controllerFMU.MovementArgs_target_Z" :["dmodelFMU.MovementArgs_target_Z"],
"mappingFMU.feasibleMoveDiscreteCommand" :["dmodelFMU.
        feasibleMoveDiscreteCommand"],
"mappingFMU.robotStopped" :["dmodelFMU.robotStopped"],
"mappingFMU.gripperOpened" :["dmodelFMU.gripperOpened"],
"mappingFMU.collision" :["dmodelFMU.collision"],
"mappingFMU.gripperClosed" :["dmodelFMU.gripperClosed"],
"mappingFMU.nonfeasibleMoveDiscreteCommand" :["dmodelFMU.
        nonfeasibleMoveDiscreteCommand"],
"mappingFMU.moveCompleted" :["dmodelFMU.moveCompleted"],
"dmodelFMU.movediscrete" :["mappingFMU.movediscrete"],
"dmodelFMU.pick" :["mappingFMU.pick"],
"dmodelFMU.place" :["mappingFMU.place"],
"dmodelFMU.target_X" :["mappingFMU.target_X"],
"dmodelFMU.target_Y" :["mappingFMU.target_Y"],
"dmodelFMU.target_Z" :["mappingFMU.target_Z"],
"dmodelFMU.closing_diameter" :["mappingFMU.closing_diameter"],
"dmodelFMU.closing_speed" :["mappingFMU.closing_speed"],
"dmodelFMU.closing_force" :["mappingFMU.closing_force"],
"dmodelFMU.opening_diameter" :["mappingFMU.opening_diameter"],
"dmodelFMU.opening_speed" :["mappingFMU.opening_speed"],
"dmodelFMU.opening_force" :["mappingFMU.opening_force"],
"dmodelFMU.stop" :["mappingFMU.stop"]
```

Listing 3: Specification of input/output connections for the co-simulation.

5.4.2. Platform-mapping FMU

The left column of Fig. 12 describes the steps to create the platform mapping FMU from the platform-mapping interface (explained in Step 3), using the templates we have developed. To develop it, we use UniFMU as the wrapping mechanism.

For Step 6, UniFMU provides a Python template that contains the structure of the implementation of an FMU, following the FMI standard, with minimal contents. This template needs to be adapted to bridge the implementation artefact with the FMI interface.

An FMU implementation has a resources folder, to which we need to add the platform-mapping interface, so that it becomes an internal

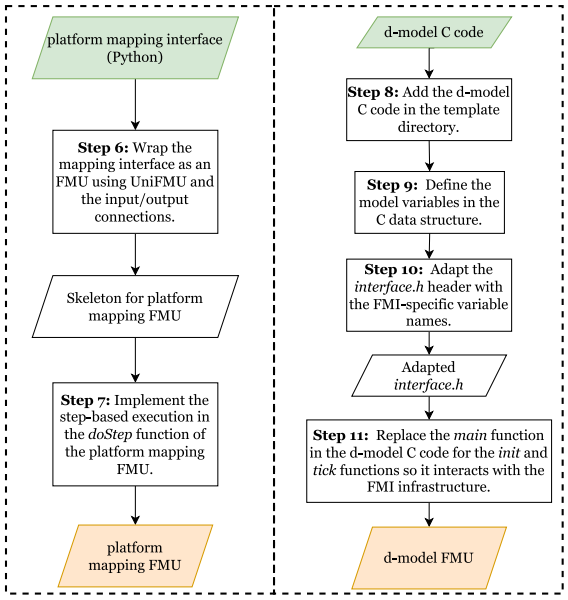


Fig. 12. DT engineering with RoboSim: Steps 6 and 7 related to the platform mapping FMU, and Steps 8-11 for the d-model FMU.

library. In the FMU's resources folder, the file `Model.py` contains the code defining the behaviour of the FMU. For the platform-mapping FMU, we have a template for the stepped logic for a co-simulation that uses the `Mapping` instance, adapting its interface to that mandated by FMI. All we need to do is to create an instance of the class `Mapping` in the way explained in Section 5.3.1. In addition, we need to define the input and output data structure of the FMU (as recorded in the file `ModelDescription.xml`) using instance fields of `Model.py`. The content in `Model.py` provides the implementation of the FMI interface. The content in `ModelDescription.xml` provides the FMU structure and metadata. The templates provided assist in the setup of these files consistently.

Following the FMI standard, `Model.py` defines the active behaviour of the FMU using the method `fmiDoStep`, which describes each step of the co-simulation. For Step 7, we have templates that can be used, based on the information in the `ModelDescription.xml` file, to implement `fmiDoStep`. This ensures that the platform-mapping FMU forwards the right data to the right ports (as illustrated in Listing 3, for instance).

The code for `fmiDoStep` (1) resets the outputs; (2) reads the inputs from the d-model FMU (through FMI) and from the robotic platform (through the platform-mapping interface); and (3) calls the methods of the platform-mapping interface to determine whether events have occurred or to call operations. We have templates for the code of each of these tasks.

Since the UniFMU template already contains the binary files required by FMI, our worked out implementation only needs to be zipped with a `.fmu` extension to get the functional platform-mapping FMU.

5.4.3. d-model FMU

Fig. 12 shows, on the right, the steps to implement the d-model FMU using templates we have created. They come with a `makefile`, which compiles the C application into a binary file, and wraps it as an FMU, which can be used by the co-simulation.

In contrast with the platform-mapping interface, written in Python, the d-model code is in C. So, we use the FMI C library[14] to create the

[14] https://github.com/modelica/fmi-standard

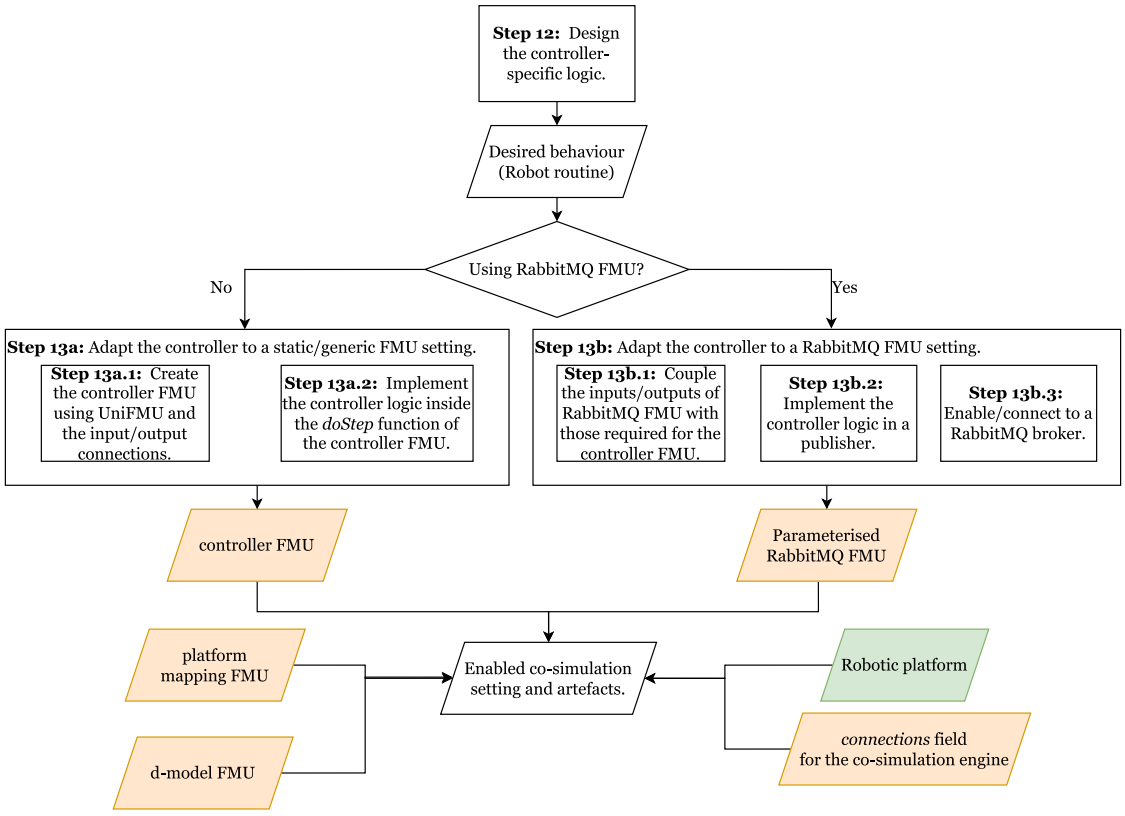


Fig. 13. Orchestration of the co-simulation artefacts: Steps 12 to 13.

d-model FMU. We have a template to use the generated d-model code. In Step 8, we add that code in the template directory. The next steps make additional changes to the result.

We need to define a C structure (`struct`) named `ModelData` (in the `defs_fmi.h` header) to create a shared data space for the code of the d-model and of the FMI infrastructure (Step 9). The definition of `ModelData` covers all the inputs and outputs of the d-model C code from the RoboSim model (and so matches the `ModelDescription.xml` file).

The RoboSim code generator for d-models supports the use of the generated code in a variety of contexts without the need to change the implementation of the state machines. The d-model C code is split into several modules. We need to adapt just the module providing the interface of inputs and outputs, in the `interface.h` header file (Step 10), and implement an `init` and a `tick` function to use the code for the d-model in the context of the FMI infrastructure (Step 11).

The `interface.h` header relies on two functions, `read_inputs` and `write_outputs`, which translate between the internal and external representations of inputs and outputs. The automatically generated code provides a command-line implementation of these functions, where `read_inputs` reads a sequence of strings from the command line and converts them into events of the RoboSim model, and `write_outputs` receives events of the RoboSim model and prints them to the command line. For integration with the FMI interface, `read_inputs` and `write_outputs` must be adapted to deal with the data in `ModelData`, instead of the command-line entries. Our templates provide code for dealing with both events and operation calls.

Similarly, the `main` function automatically generated for the d-model is for a terminal interface. To implement the FMI `init` function, we extract the initialisation code from the generated `main` function.

By limiting the modifications to `interface.h`, which does not encode the behaviour of the RoboSim model, we guarantee that adapting the code to FMI does not introduce errors in the d-model implementation. So, any verification done at the model level remains valid.

5.4.4. Controller FMU

Steps 12 and 13 are detailed in Fig. 13 and they deal with realising the controller FMU. In Step 12, we write the code that defines the co-simulation scenarios to be carried out. It can be done by hand, but can be generated automatically from a test suite. Ongoing work provides support for such code generation for other platforms [60]. For our example, the code for the controller FMU gives commands, using the services implemented in the platform-mapping FMU, to simulate specific tasks, determining where the robot should move, when to pick or place, and how many times, to build shapes on the 2D working space of the Flex-cell's plate.

To implement the FMU, we have support for the use of UniFMU (Step 13a) or RMQ FMU (Step 13b) using templates we have created. As for the platform-mapping FMU, the controller FMU based on UniFMU is created from a minimal UniFMU template. Steps 13a.1 and 13a.2 are similar to Steps 6 and 7.

The use of UniFMU, however, limits the co-simulations to pre-programmed tasks, that is, they do not change over time. As an alternative, and workaround for this limitation, the RMQ FMU can be used. RMQ FMU enables interaction with a co-simulation via a RabbitMQ channel to feed and retrieve data from a co-simulation in real-time. Using RMQ FMU, the controller is not implemented as an FMU. Instead, it publishes commands via a RabbitMQ broker over specific topics, and RMQ FMU forwards them to the co-simulation in real-time.

In Step 13b.1, we create an instance of RMQ FMU and update its `ModelDescription.xml` so it matches the connections in Section 5.4.1. For example, to achieve the bridging between the real world and the co-simulation using RMQ FMU, Listing 3 is modified from line

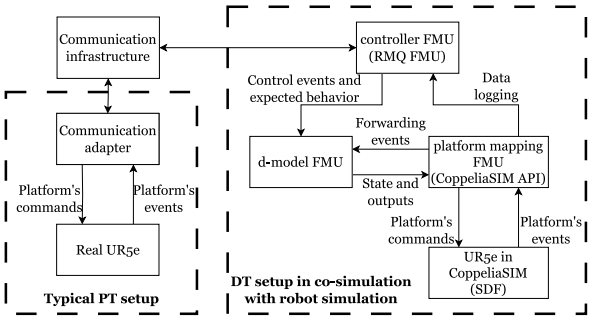


Fig. 14. Co-simulation block diagram using a typical PT setup.

one to line six as shown in Listing 4. The differences are just that references to the controllerFMU are replaced with references to RMQ FMU. This shows the generality and reusability of our approach.

```
"RMQFMU.closeGripperCommand" : ["dmodelFMU.closeGripperCommand"],
"RMQFMU.openGripperCommand" : ["dmodelFMU.openGripperCommand"],
"RMQFMU.moveDiscreteCommand" : ["dmodelFMU.moveDiscreteCommand"],
"RMQFMU.MovementArgs_target_X" : ["dmodelFMU.
    MovementArgs_target_X"],
"RMQFMU.MovementArgs_target_Y" : ["dmodelFMU.
    MovementArgs_target_Y"],
"RMQFMU.MovementArgs_target_Z" : ["dmodelFMU.
    MovementArgs_target_Z"]
```

Listing 4: Specification of input/output connections with RMQ FMU.

Since RMQ FMU depends on a publisher that sends messages periodically to the RabbitMQ broker, in Step 13b.2, we use the code from Step 12 to send commands to the broker at particular times, following the co-simulation steps. To bind that code and RMQ FMU, in Step 13b.3, a RabbitMQ broker is enabled, and both RMQ FMU and the code are connected to that broker.

After Step 13, the co-simulation artefacts (see Fig. 7) are ready to be orchestrated by the master algorithm as discussed in the section below.

5.5. A co-simulation-driven digital twin with RoboSim

Fig. 14 presents a typical setup for a DT and a PT, which relies on the assumption that the PT's existing communication capabilities are sufficient to bind it to its DT via some interface or a combination of adapters. In Fig. 14, the PT consists of a real UR5e robot and a cyber-element, the communication adapter, that can be controlled from the outside by a communications infrastructure that is customised to the adapter.

In this work, we illustrate an approach in which both the PT and the DT run as co-simulations, using the artefacts obtained as described in Section 5.4. Fig. 15 shows the setup; its right-hand outer block, describing the DT, is the same as that in Fig. 14, but here the PT, represented by the left-hand outer block, is also a co-simulation. As previously said, we can create two setups for a co-simulation: (i) one where the robot is simulated, for example, in CoppeliaSim, to enable comparisons with the PT; and (ii) another that integrates the hardware in the loop, with the real robot software realised by co-simulation steps. These setups use the same FMUs, except only that two different instantiations of the platform-mapping FMU bind the two different interfaces and the two different exchanges in RabbitMQ. One performs the physical setup, and another, the digital setup. In the first setup, we obtain a PT; in the second, a DT. In comparison with a typical setup, we can more easily synchronise the PT and the DT since both execute their processes discretely by time steps.

For both the DT and the PT, we adopt the TwinManager architecture proposed by Gil et al. [18,23]. It enables the integration of FMI co-simulations through a unified interface with DT services and with hardware components to provide also a PT. For the DT, the artefacts generated by our technique fit perfectly in the TwinManager architecture. The DT in Fig. 15 (and also Fig. 14) is composed by the co-simulation artefacts from Fig. 7. For the PT, we set up a similar structure. As said, structurally, as highlighted in Fig. 15, the only difference between the DT and the PT is in the platform mapping FMU and, of course, in the PT we have the real robot, and in the DT, a simulator providing the functionality of that robot. The other FMUs are exactly the same as those already presented.

We note that a full setup of the DT constellation also requires the integration of the services (also known as benefits). These services are case-specific and also depend on particular use cases, such as, collision detection, visualisation, self-calibration, and what-if analyses, which are common in robotic applications. Through RMQ FMU and the TwinManager components implementing the communication infrastructure, our approach provides interfaces that ensure these services can be easily added to both setups, enhancing reusability and interoperability, and lowering costs.

Once all the components are initialised and under the control of the TwinManager, bi-directional connectivity is enabled. By default, the TwinManager works in a synchronous manner, but with RMQ FMU takes advantage of RabbitMQ to enable asynchronous messaging. The services, which depend on the use case, are coupled with the TwinManager, so they use its interfaces to execute specific tasks, such as, deviation checking, visualisation, calibration, and so on. We do not focus on a particular service, since for these, other tools and enablers are available. For example, AURT [61] for calibration, and the DTaaS platform [19] for cloud deployment and integration with other infrastructure services, Unity, CoppeliaSim, Gazebo, and so on, and for visualisation. In [23,54], for instance, we use DTaaS.

With our modular approach, based on the structure of RoboSim models, we have a high level of reusability. The only two components that need major modifications to change from a DT to a PT, or vice-versa, are the platform-mapping interface and the robotic platform associated to modifications in Steps 3 and 4. (These are highlighted in Fig. 15 by the inner dashed red (left) and blue (right) boxes.) Another minor change is that the instances of the two RMQ FMUs need to bind two different exchanges for the data stream of both setups. The other components can be reused as-is.

Some components shown in Fig. 15 can also be reused in other case studies. For example, for any other robotic arm controlled by discrete movement commands, the same d-model FMU can be reused since the RoboSim state machine is platform independent. This means that we can easily extend our example to the complete Flex-cell case study: both the UR5e and the Kuka lbr iiwa 7 can use the same d-model FMU without any modifications, since both are controlled in the same way, as explained in Gil et al. [54]. Similarly, the controller FMU (or its alternative RMQ FMU) can also be reused since the logic and the interfaces among the controller FMU, the platform mapping FMU, and the d-model FMU would not change. As discussed in our evaluation in the next section, this has a positive impact on engineering effort, and the reality gap.

The full set of artefacts resulting from applying each of the steps of our technique to our case study is available in the GitHub repository provided[12]. Next, we discuss the use of the generated co-simulations.

5.6. Demonstration

To illustrate the operation of the co-simulations of our example, we describe an experiment where the DT-enabled system is provided with services for what-if analysis, visualisation, and synchronised execution of discrete movement commands for deviation checking given a plan. These services have been designed for the Flex-cell (see [54] for more detail). To showcase our work, we run a co-simulation that can be executed on the DT, decoupled from its PT, for what-if analysis, or on both the DT and the PT, for deviation checking.

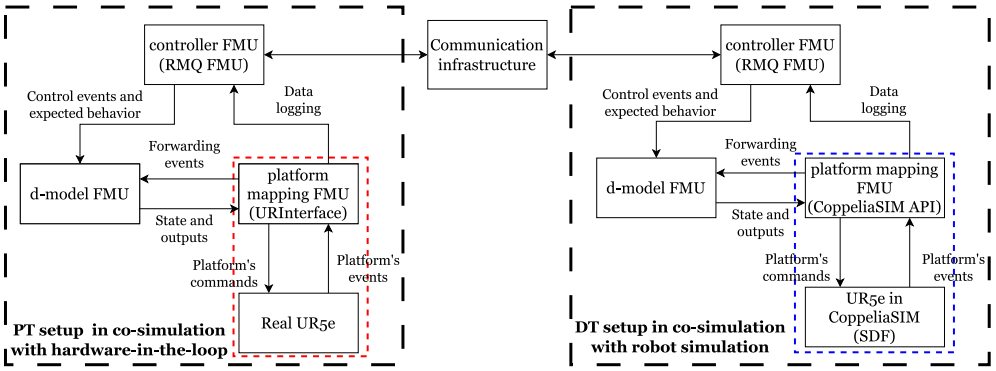


Fig. 15. Co-simulation block diagram following our approach.

```
"mappingFMU.q0" : ["RMQFMU.q0"],
"mappingFMU.q1" : ["RMQFMU.q1"],
"mappingFMU.q2" : ["RMQFMU.q2"],
"mappingFMU.q3" : ["RMQFMU.q3"],
"mappingFMU.q4" : ["RMQFMU.q4"],
"mappingFMU.q5" : ["RMQFMU.q5"]
```

Listing 5: Specification of input/output connections for monitoring joint positions with RMQ FMU.

Experiment settings. The co-simulation total time is 20.0 s with fixed step sizes $\Delta_t = 0.5$ s, 0.25 s, and 0.1 s. The co-simulation contains 28 output-input pairs, namely, 22 connections coming from Listing 3 with the modifications of Listing 4 to enable RMQ FMU, and six connections from Listing 5 to enable the runtime monitoring of the joint angle positions used for the services of what-if analysis and deviation checking. Our experiment executes four `moveDiscreteCommand` with target positions $(0, 23, 1)$, $(3, 20, 2)$, $(8, 10, 0)$, and $(1, 13, 0)$ for the X, Y, and Z axes, respectively. These commands are published from the script attached to the RabbitMQ broker or directly from the TwinManager application at steps occurring at times $t_1 = 1.0$ s, $t_2 = 5.5$ s, $t_3 = 8.0$ s, and $t_4 = 15.0$ s. The behaviour of the co-simulation(s) during execution is published to the same RabbitMQ broker and monitored periodically. Such a monitoring can be implemented using either the TwinManager or an external RabbitMQ subscriber. For post-analysis, the data after execution are also stored as CSV files.

For this demonstration, we use a fixed set of target positions. However, the commands for a particular task execution can be changed programmatically, either via the external RabbitMQ publisher or directly via the TwinManager through a planning service. Moreover, plans generated by automated planners can also be used through one of these two alternatives.

Results. Fig. 16 gives the results of the co-simulation experiments for the two setups given in Fig. 15 (one for the PT and one for the DT). As said, the co-simulations can be run independently, on the PT or the DT, or synchronously on both at the same time depending on the expected use case. Similarly, the input–output connections for controlling or monitoring the behaviour of the PT and the DT can also be updated depending on the use case. Connections are defined in the same way for both the PT and the DT, since the control and monitoring interfaces for them are the same architecturally, and the differences rely only on the underlying interfaces of the platform-mapping interface.

In Fig. 16, we can observe the behaviour of RMQ FMU when receiving and forwarding the move commands at the given times. This generates the behaviour propagated through the co-simulation components and the robotic platform. Subplots (1) and (2) in Fig. 16 show the commands and their arguments from the RMQ FMU at different time steps. Subplot (3) in Fig. 16 shows the arguments of the `movediscrete` operation of the d-model FMU after having received the commands from the RMQ FMU. These operation calls, executed by the d-model FMU, convey the behaviour to the robotic platform via the platform-mapping FMU; this is illustrated by the angular position of *joint 1* in subplot (4).

Since the co-simulation is performed in steps, when it is parametrised with a step size of $\Delta_t = 0.5$ s, for example, the command published at $t_1 = 1.0$ s is received by the RMQ FMU at $t = 1.5$ s (see subplots (1) and (2) of Fig. 16), that is, $t = t_1 + \Delta_t$, and forwarded to the d-model FMU at the same time step. On the next time step, that is, $t = t_1 + 2 \times \Delta_t$ ($t = 2.0$ s), the d-model FMU processes the inputs from the RMQ FMU and then outputs the requested command(s) to the platform-mapping FMU based on its state machine (see subplot (3) of Fig. 16). On the next time step, that is, $t = t_1 + 3 \times \Delta_t$ ($t = 2.5$ s), the platform mapping FMU gets the command from the d-model FMU and proceeds with the platform-specific functions at that point. A similar sequence of steps apply for the other commands too. It means that there is at least a delay of three time steps between when the command being sent by the publisher and when the command is executed by the robotic platform. This is visually detectable in the simulation with step size $\Delta_t = 0.5$ s in subplots (3) and (4) of Fig. 16; however, a smaller step size, such as $\Delta_t = 0.25$ s and $\Delta_t = 0.1$ s, makes this delay negligible, as observed in subplots (3) and (4) of Fig. 16.

In subplot (4) of Fig. 16, we can also notice some minor differences between the simulated and the real robot. We can pinpoint these differences as follows: (1) different initial value between simulation and real robot since the simulation is starting from zeros and the real robot from its previous state; (2) dynamics of the model, which is evident in the transient states; and (3) execution delays when executing commands on the real robot due to network and computation overhead. In particular for (3), these execution delays are due to synchronisation issues between the RMQ FMU and the co-simulation, which is further increased when using smaller step sizes since the co-simulation execution with RMQ FMU cannot cope with the hard real-time requirements of the running system.

However, with our approach, we can at least clearly identify the cause of the reality gap. There cannot be a problem with the software simulation, since its code is reused without changes in the PT and in the DT. The gap is clearly related to invalid assumptions made by the physics engine in the simulation of the joints as mentioned in point (2) above. Point (1) can be easily solved by setting the initial value of the simulation to be that of the real robot. Finally, point (3) is unavoidable.

5.7. Generalisability of the method

In order to assess the generalisability of this approach, we follow a multi-case design approach [62] using a second example in the form of a mobile robot. For this second example, we only focus on the co-simulation artefacts of the simulated robot.

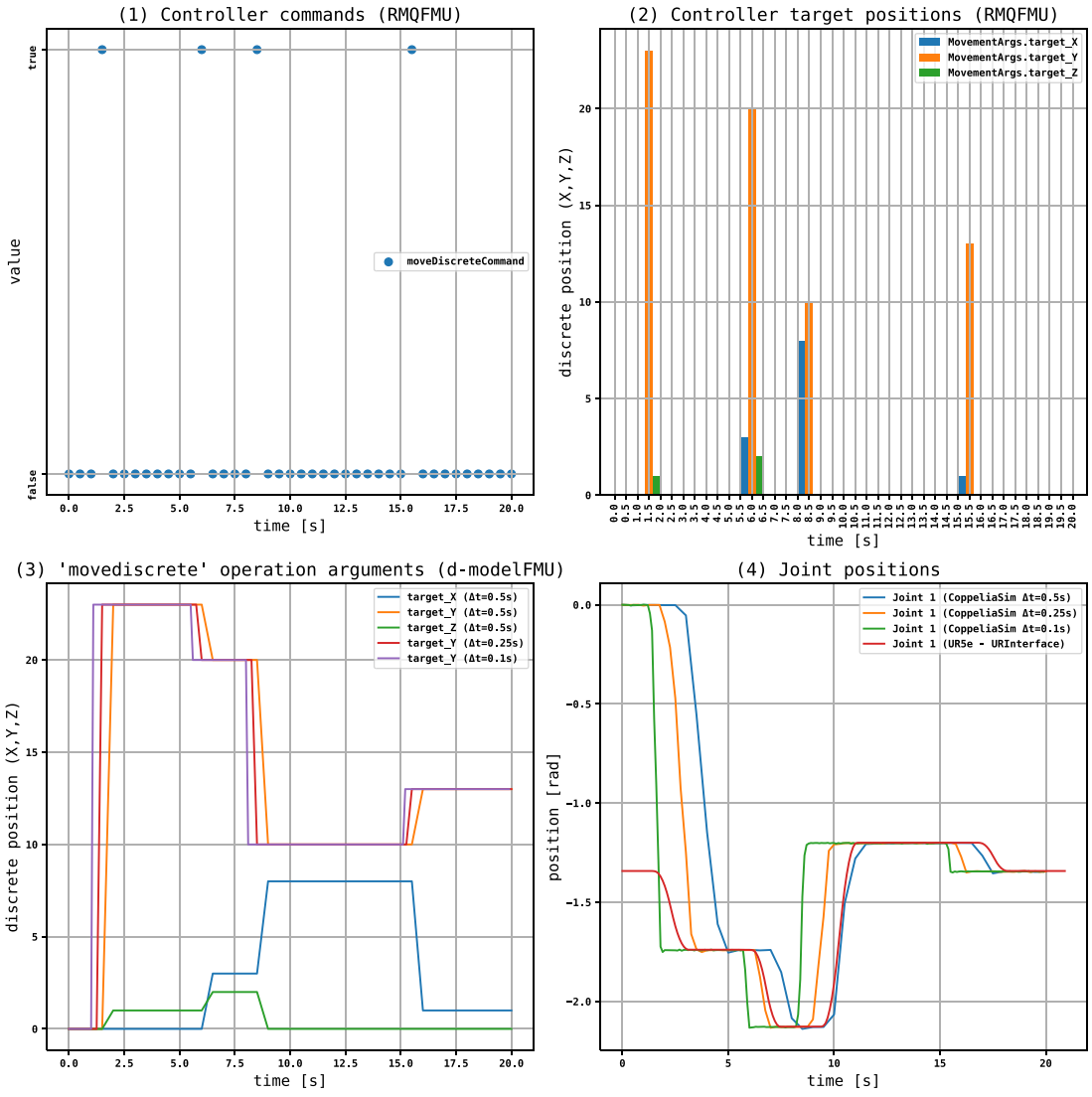


Fig. 16. Result of the co-simulation experiments for the physical and digital setups in Fig. 15.

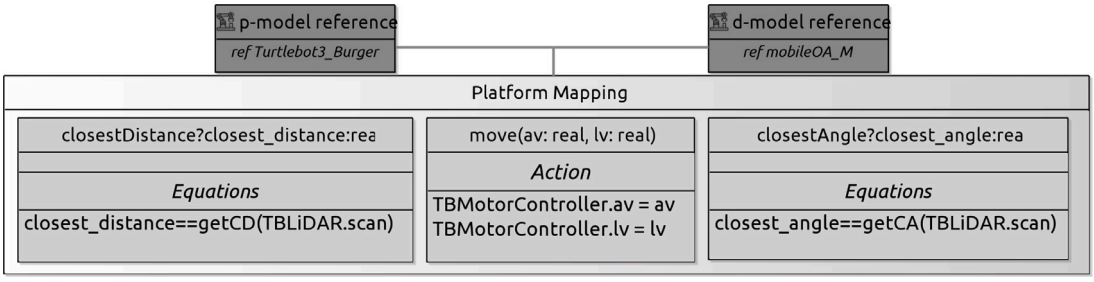


Fig. 17. Platform mapping between mobile robot and d-model.

This second case study consists of a software controller for a mobile robot that performs a random walk whilst avoiding obstacles.[15] The d-model for this example has input events `closestDistance` and `closestAngle`, communicating values coming from a LiDAR scan when there is data available (in a buffer). The d-model also uses one operation of the robot, namely, `move`, which has two parameters, `av` and `lv`, which convey the angular and linear velocities that are to be sent to the robot's velocity controller.

Fig. 17 shows the platform mapping for the example. It includes the mappings for `move`, `closestDistance`, and `closestAngle`. For `move`, the mapping specifies that the arguments `av` and `lv` are assigned to inputs of a motor of the robot represented by `TBMotorController`. The scan data from the sensors, represented by `TBlidar.scan`, must be preprocessed to obtain the input values `closest_distance` and `closest_angle` for the events `closestDistance` and `closestAngle`. These case-specific functions, called `getCD` and `getCA`, are provided in the platform-mapping interface, which integrates the platform via a ROS2 interface. In this way, the robotic platform is bound through the platform-mapping FMU to send velocity commands

[15] A video for a simulation is available at[12].

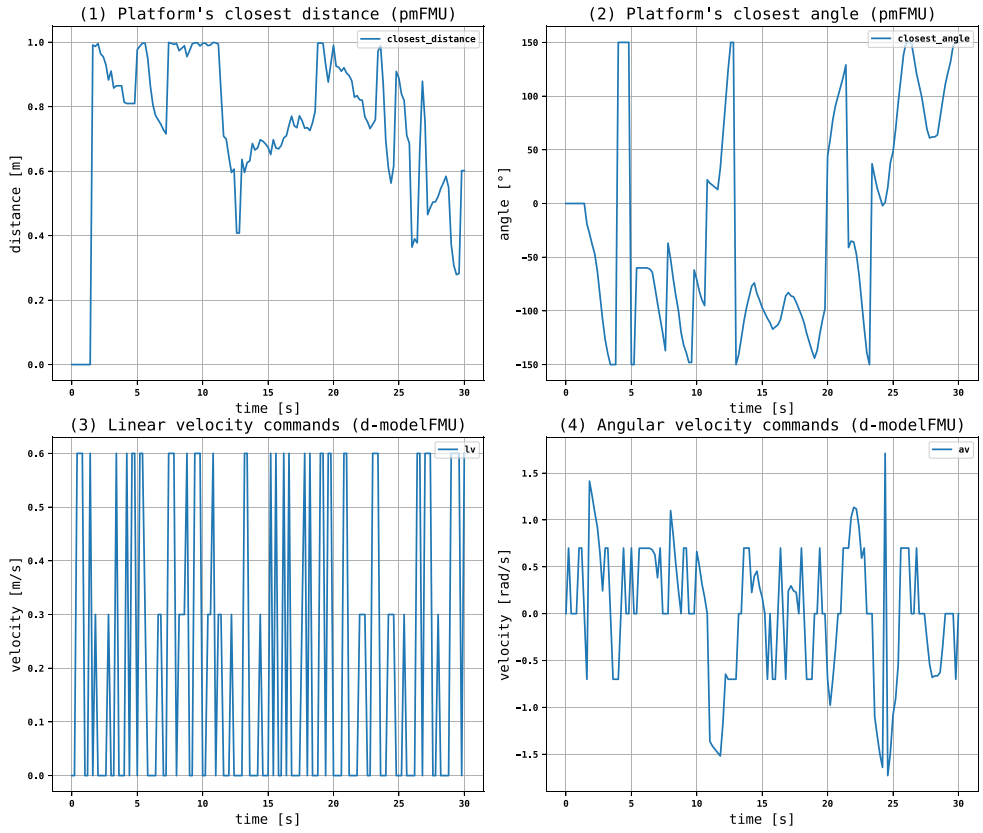


Fig. 18. Results of running the co-simulation artefacts of the mobile robot.

and receive scan data from Gazebo via its underlying platform-mapping interface. In this instance, we do not need to produce a p-model and generate an SDF description for it, because we can use an SDF description made available by the manufacturer of a suitable robot that can be simulated in Gazebo.

The d-model FMU and the platform-mapping FMU encapsulate the autonomous behaviour of the robot's controller. We do not need to use the controller FMU or RabbitMQ FMU, because this is a fully autonomous robot not requiring inputs. Our framework, however, allows the user to easily add these additional FMUs if required, for example, for monitoring, testing, or interacting with the robot when behaving autonomously. The integration of the co-simulation artefacts following the steps up to Section 5.4 for this example are represented in the connections between the d-model FMU and the platform-mapping FMU in Listing 6.

The results of running these co-simulation artefacts with a step size $\Delta_t = 0.2$ s for a total time of 30.0 s are provided in Fig. 18. It is possible to see how the behaviour of the mobile robot changes via the commands for updating the linear and angular velocities (in subplots (3) and (4)), according to its proximity to objects using the scan data for closest distance and closest angle (in subplots (1) and (2)). The full implementation of this example including a video of the co-simulation is provided on the GitHub repository[12].

```
"mappingFMU.closestDistance" :["dmodelFMU.closestDistance"],
"mappingFMU.closest_distance" :["dmodelFMU.closest_distance"],
"mappingFMU.closestAngle" :["dmodelFMU.closestAngle"],
"mappingFMU.closest_angle" :["dmodelFMU.closest_angle"],
"dmodelFMU.move" :["mappingFMU.move"],
"dmodelFMU.lv" :["mappingFMU.lv"],
"dmodelFMU.av" :["mappingFMU.av"]
```

Listing 6: Specification of input/output connections for the mobile robot co-simulation.

In the next section, we assess our work using benchmarking criteria for DT frameworks.

6. Evaluation

In this section, we conduct an assessment of our approach based on existing independent criteria that have been used to assess DT frameworks in a case-agnostic manner in Lehner et al. [9] and Gil et al. [10], with the metrics explained in Section 6.1. The results are presented in Section 6.2 and discussed in Section 6.3. In Section 7, we discuss the limitations of our approach.

6.1. Evaluation metrics

To assess our work, we consider the independent *requirements* and *criteria* from Lehner et al. [9] and Gil et al. [10], omitting those that overlap. (More precisely, *Communication* in [10] is merged with *Bidirectional Synchronisation* in [9], *Support for physical interventions* in [10] is merged with *Modifiability* in [9], and *Interoperability* in [10] is merged with *Platform interoperability* and *System interoperability* in [9].)

To provide a quantitative evaluation, we follow the metrics defined in the reference works: a three-range scale for the requirements by Lehner et al. [9], and the combination of quantitative and qualitative metrics for the criteria by Gil et al. [10]. These refer to the support or capabilities of our approach related to each criterion. More precisely, each criterion is assessed with a reference to its maximum possible value (if applicable), that is, $\frac{assessment}{Range_{MAX}}$, with 0 as the minimum value. Finally, we sum the scores for each criterion with reference to the maximum possible value to understand the coverage of our approach in broader terms of Digital Twinning. For those criteria that cannot be quantitatively scored, we use the *Not applicable* label instead of a score.

6.2. Evaluation results

Table 1 summarises the evaluation of our approach against the criteria and metrics as defined in Section 6.1. The details are described in what follows.

Table 1
Summary of assessment of our approach.

Criterion	Assessment	Score
From Lehner et al. [9]		
Bidirectional Synchronisation	◐	1/2
Convergence	◐	1/2
Verification & Validation	●	2/2
Real-time behaviour	◐	1/2
Automation protocols	○	0/2
Platform interoperability	●	2/2
System interoperability	●	2/2
Domain expert involvement	●	2/2
Connection and data security	○	0/2
Modifiability	●	2/2
Reusability	●	2/2
Continuous Integration and Deployment	●	2/2
Provisioning	○	0/2
From Gil et al. [10]		
Storage	Files	Not applicable
Support for analytics	Level 1	1/4
Compositionality	Aggregable FMUs	1/2
Scalability	Moderate	1/2
Standardisation	Behaviour	Not applicable
Reproducibility	Yes	1/1
Community support	Research group(s)/GitHub repository	Not applicable
Total score		21/35

Bidirectional Synchronisation (1/2): our approach requires a combination of manual and automatic connection realisation to interface the DT and the PT.

Convergence (1/2): our approach provides independent access to the state and parameters of both the PT and the DT. This supports convergence by ensuring that deviations between these parameters can be detected, adjusted, and rectified when operating in synchronised mode. These capabilities, however, need to be implemented through other DT services.

Verification & Validation (2/2): with RoboSim's tools and techniques, which provide a framework for formal verification and validation, our approach can ensure the system satisfies key expected properties.

Real-time behaviour (1/2): our approach provides the interfaces to deal with multiple robotic platforms either real or simulated, with a quasi-real time support, but it has only been tested at the edge and not on the cloud.

Automation protocols (0/2): our approach does not natively support the automation of communication protocols for the domain of industrial automation. These could be implemented as part of the platform-mapping interface, but we have no demonstrators for that.

Platform interoperability (2/2): our approach provides standardised interfaces, as defined in the FMI standard, to both read and write data. DT data can be retrieved and updated using the `fmiGet` and `fmiSet` interfaces, which changes depending on the FMI version and the data type. The structural information is also accessible via the `ModelDescription.xml` file.

System interoperability (2/2): our approach provides support for connections and for automatically implementing interactions between different devices using FMI channels and the worked out platform-mapping interface for specific robotic platforms. As detailed in the previous section, this support is provided by the connection between FMUs defined as in Listing 3, which is interpreted by the co-simulation master algorithm. The interoperability with external systems depends on the design of the platform-mapping interface, described in Section 5.3.1, and other integrations with, for example, RMQ FMU via RabbitMQ (see Section 5.4.4).

Domain-expert involvement (2/2): relying on RoboSim's capabilities, which provides a suite for graphical modelling using Eclipse, the user can model physical devices (using the p-model notation), types and data (using the RoboSim libraries), and even the control software (using the d-model notation), in a structured way and with a graphical interface.

Connection and data security (0/2): our approach does not natively support mechanisms to ensure security for the connections and for the data.

Modifiability (2/2) the modularity of our approach, supported by RoboSim and FMI, enables the modification or replacement of modules, or more precisely, FMUs, during runtime [17], without impact on other components, as long as interfaces are preserved.

Reusability (2/2): our approach provides several components that can be reused between different projects, such as platform-independent control software (d-models), tests for scenarios (controller FMU and RMQ FMU), FMI connections, model and data type definitions in the physical model, and implemented software. Reuse across different platforms and co-simulation setups is also immediate, as illustrated here.

Continuous integration and deployment (2/2): Similar to *Modifiability*, our modular approach enables the modification, replacement, and addition of modules, which can be either FMUs or services working with those, which enable the continuous integration and continuous deployment of the DT-enabled system.

Provisioning (0/2): our approach does not natively support cloud computing as required in this criterion. However, this feature could be enabled using compatible tools, such as the above-mentioned DTaaS platform.

Storage (Not applicable): any approach, including ours, can store data in files, but several levels of sophistication are possible (such as SQL/NoSQL databases, time-series databases, or cloud storage, for instance). It is very difficult to provide a score to compare these possibilities, so we refrain from that.

Support for analytics (1/4): our approach provides the API interfaces in accordance with the FMI standard and via the platform-mapping interface, but does not support more advanced analytics methods natively.

Compositionality (1/2): our approach supports compositionality in terms of aggregation of FMUs to compose larger systems, but the FMI standard does not support fully hierarchical composition: an FMU cannot compose other FMUs, for example.

Scalability (1/2): this criterion focuses on the scalability of the running platform to support addition of an increasing number of DT

instances. First of all, our approach supports coupling heterogeneous subcomponents into larger systems due to the use of co-simulation; this is key for scalability of DTs [8]. On the other hand, the computational and networking demands of an FMU is higher than those of a typical information system. Based on our knowledge and experience, for a robotic system realised by a number of FMUs in the order of thousands and above, we expect that we would have a problem. Therefore, we believe that our approach may be limited to tens or hundreds of FMUs, and therefore, it has a moderate scalability.

Standardisation (Not applicable): our approach provides standardised interfaces at the behavioural level following the industrial FMI standard 2.0.

Reproducibility (1/1): we provide reproducibility and documentation of our approach through the publicly available GitHub repository and refer to external resources for the dependencies regarding RoboSim tools and co-simulation tools. We provide templates and three examples; one following the UR5e case study, one following the mobile robot used in Section 5.7, and one for a UR3e.

Community support (Not applicable): *Creator:* our approach is created by the collaboration of two research groups based in different countries. *# of releases:* 0 releases, 61 commits at the time of writing. *Documentation:* GitHub repository and external resources.

6.3. Discussion of evaluation results

Of the 35 maximum points, our approach for DTs of robotic platforms scores 21 points, that is, 60% of support and capabilities of the Digital Twinning spectrum according to the criteria in [9,10]. We can highlight that the scores for the criteria *Platform and System interoperability*, *Modifiability*, *Reusability*, and *Continuous integration and deployment* of our approach are high: **(2/2)**, due to various reasons arising from our adoption of the FMI standard and the definition of FMUs based on the structured RoboSim models. These include support for platform-independent and interoperable components, models, and interfaces, among others, and reuse in heterogeneous software systems, enabled by a modular approach that uses FMI and RoboSim. Similarly, the score for the criteria *Verification & Validation* and *Domain-expert involvement* is also high, **(2/2)**, due to our reliance on RoboSim technology, which provides a modelling suite with graphical user interface and native support for formal verification.

In terms of *Reproducibility*, our approach is also highly scored: **(1/1)**. This is based on results described in this paper. As said, we provide the templates and documentation of the templates and case studies and usage of the technique in general on the GitHub repository provided, which can also be used for questions, issues, new case studies, and potential updates coming up.

Lower scores, that is, (0/2), are due to the criteria *Automation protocols*, *Connection and data security*, and *Provisioning*. *Automation protocols* refers to a very particular niche in the industrial-automation domain; although we are not supporting any automation protocol currently, this aspect could be easily adapted by integrating the OPC UA communication protocol (or any other of those referred in *Automation protocols* in [9]) to the platform-mapping interface as the robot-specific interface in Steps 3 (Section 5.3.1) and 4 (Section 5.3.2).

The criterion *Connection and data security* focuses on security mechanisms for the connection and data. Currently, the FMI standard does not focus on these aspects beyond providing intellectual-property protection with the encapsulation of FMUs. The connection between the platform-mapping interface and the robotic platform, however, can be designed to incorporate both authentication and encryption mechanisms for secure connection. This would improve the score in this criterion for at least some parts of our approach.

Similarly, *Provisioning* refers to a particular niche application area of DTs when deployed on the cloud. Currently, we do not support this capability; however, relying on complementary tools, such as the DTaaS platform [19], this aspect, including the connection and data security aspects, can be improved. Additionally, this would also enable the analysis of the real-time capabilities of our approach, considered in the criterion *Real-time behaviour*, when using connections on the cloud.

Compositionality and *Scalability* have score **(1/2)**. So, our approach performs moderately well, that is, it can be used to incrementally build larger DT systems by smaller composites, following a hierarchical aggregation method defined by FMU connections rather than pure compositions, and can support a moderate number of DT instances, in the range of tens or hundreds depending on the machine running the implementations. This is likely to be enough for robotic applications, with the likely challenges coming from robot teams, which we are addressing as part of our agenda for future work.

Finally, future work on improving the automation of our artefacts (and the services bound to those artefacts) will also help in terms of the criteria *Bidirectional synchronisation*, *Convergence*, and *Support for analytics*. As explained, our approach provides the interfaces and the (co-)simulation capabilities to bind other services of the DT constellation to perform some tasks. If we considered an improved worked-out version of the DT constellation, our score in these criteria would improve. There is nothing intrinsic to our approach that makes such additional work more or less difficult.

Our approach, however, has some limitations, which we identify and describe in the next section.

7. Discussion of limitations

Application areas Our framework cannot, for example, deal with soft robots, because the RoboSim p-model notation is tailored for the description of rigid-body robots. We also do not consider technology for autonomous vehicles in general. Although RoboSim has been used in separate work for drones, the combination of tools provided here is not tailored for customised simulators normally used in the transportation domain. Nevertheless, within their scopes, RoboSim and FMI are application-agnostic, so our results are relevant for many application areas and domains.

Reusability In our approach, reusability of artefacts is as follows: (1) within the same case study for the deployment of the DT-enabled system, i.e., for the deployment of the PT using any platform providing the services required by the RoboSim d-model, and for as many DTs as there are d-models featuring different behaviours of the PT; and (2) the partial reuse in other case studies with architectural similarity [63]. Artefacts for different case studies may need to be re-engineered.

Complexity behind integrating with FMI. Our approach for integrating implementations of RoboSim models with FMI is promising, but requires knowledge of the FMI standard to understand how to properly set the artefacts. Our technique also requires some manual work, especially on the data structures and connections, but further automation is possible for future work.

Non-self-contained p-model implementation. Currently, the implementation for the p-model is an automatically generated SDF file, which we need to import into a robotics simulation engine. This creates external dependencies outside our control. Ongoing work is taking advantage of the behavioural specifications (via equations) in the p-model to generate faithful simulations, which can then be integrated with FMI.

Development of co-simulation tests. Our technique requires the development of code to execute the simulation scenarios of interest: the controller FMU. Future work will adapt test-generation techniques currently available for RoboSim to produce test drivers as FMUs.

Time delays due to stepped simulation. As discussed in Section 5.6, since we are following the FMI standard, three time steps are needed to propagate the commands from the publisher to the robot (or its simulation). A solution is use of FMI version 3.0, which has support for FMUs with different clocks. With that, we can set the time of the FMUs to accommodate the extra cycles for communication while keeping the overall clock of the master algorithm defined by that of the robot (simulation or real robot). This would also allow observation of internal computations in the d-model code.

Subjectivity in the evaluation process. The assessment presented in Section 6 has been conducted using the guidelines provided by two independent set of criteria for DT frameworks. On the other hand, it is based on the authors' knowledge and experience. This can lead to bias and subjectivity in the assessment and its reporting. Mitigation has been done via robust discussion and revision within the whole group.

8. Concluding remarks

This paper proposes a systematic, highly automated, technique to create FMI co-simulations that can be used to realise DTs in robotics. They are based on RoboSim, a modelling framework to design robotic simulations. The approach is in 13 steps, which we have demonstrated using an UR5e robotic arm of a manufacturing cell, and evaluated using criteria adopted in key works on DT, many of which have been independently defined.

Our findings show the feasibility to couple RoboSim with FMI to achieve implementations for DTs in the robotics domain while reducing the implementation effort. This is thanks to the automatic code generation capabilities of RoboSim and the templates that we have presented here. Since RoboSim supports advanced forms of (formal) verification, with our technique we can potentially take advantage of that to produce simulation for which mathematical analysis is available.

Additionally, the findings also show that RoboSim's structured approach to modelling has enabled the development of a flexible technique that can develop DT-enabled systems using different robots, either real or simulated ones. This is achieved by slightly adapting the templates to the co-simulation artefacts.

Future work

Future work will improve the automation of our technique. This will involve automatic generation of code for the platform mapping for specific robot interfaces. Out of the 13 steps of our technique, steps 2, 5, 8, 9, and 11 are already fully (or nearly fully) automated, but the tasks in steps 3, 6, 7, 10, 12, and 13 can also be fully automated using tools that implement the specification of these steps presented in this paper.

Another line of future work is a simple extension of our technique to reflect the structure of parallel RoboSim d-models, composed of several state machines, in the co-simulation. In this way, we can have an FMU for each state machine and improve further the reusability of implementation and co-simulation artefacts. Reusability can also be improved by support to integrate the implementation of the p-model directly into FMUs, so the co-simulation setting becomes self-contained, with no dependencies on external software.

RoboSim is being extended with support to define d-models that use controllers specified by the parameters of a trained neural network. This is similar to the approach already described in [64] for RoboChart [65]. Future work will enable code generation for these components. With that, the same approach used here can be applied to develop data-driven DTs.

Finally, our case study is a single-robot system. Future work will extend our technique to multi-robot systems, possibly with humans-in-the-loop.

Code availability

The prototypical implementation and source code are publicly available on GitHub[12].

CRediT authorship contribution statement

Santiago Gil: Software, Visualization, Methodology, Validation, Investigation, Formal analysis, Writing – original draft, Writing – review & editing, Conceptualization. **Arjun Badyal:** Software, Investigation, Visualization, Writing – original draft, Validation. **Alvaro Miyazawa:** Writing – original draft, Software, Resources. **Peter Gorm Larsen:** Supervision, Writing – review & editing, Funding acquisition, Writing – original draft, Conceptualization, Project administration. **Ana Cavalcanti:** Project administration, Writing – review & editing, Validation, Investigation, Writing – original draft, Supervision, Conceptualization, Methodology.

Funding

This work has been partially funded by the Ringkøbing-Skjern Municipality, Denmark, under the Framework Collaboration Agreement for Aarhus University Digital Transformation Lab-Skjern. Funding has also been provided by the Royal Academy of Engineering, United Kingdom under Grant CiET1718/45, the UKRI (UK Research and Innovation Council) Grants No EP/R025479/1 and EP/V026801/1, and by EU Horizon RoboSapiens under agreement number 101133807.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

The authors are grateful to the Poul Due Jensen Foundation, which has supported the establishment of the Centre for Digital Twin Technology at Aarhus University. The authors also want to thank Maurizio Palmieri, who provided the C structure and guidelines to integrate the d-model C code with the FMI Interface, and Daniella Tola, who assisted in that integration. Finally, we thank anonymous referees for their insightful comments that helped us improve our paper.

Data availability

No data was used for the research described in the article.

References

[1] M. Grieves, Digital Twin: Manufacturing Excellence through Virtual Factory Replication, (March) 2014, pp. 1–7.

[2] P. Leitao, F. Pires, S. Karnouskos, A.W. Colombo, Quo Vadis Industry 4.0? Position, Trends, and Challenges, IEEE Open J. Ind. Electron. Soc. 1 (2020) 298–310, http://dx.doi.org/10.1109/ojies.2020.3031660.

[3] G. Barbieri, A. Bertuzzi, A. Capriotti, L. Ragazzini, D. Gutierrez, E. Negri, L. Fumagalli, A virtual commissioning based methodology to integrate digital twins into manufacturing systems, Prod. Eng. 15 (3–4) (2021) 397–412, http://dx.doi.org/10.1007/s11740-021-01037-3.

[4] D. Jones, C. Snider, A. Nassehi, J. Yon, B. Hicks, Characterising the Digital Twin: A systematic literature review, CIRP J. Manuf. Sci. Technol. 29 (2020) 36–52, http://dx.doi.org/10.1016/j.cirpj.2020.02.002.

[5] A. Fuller, Z. Fan, C. Day, C. Barlow, Digital Twin: Enabling Technologies, Challenges and Open Research, IEEE Access 8 (2020) 108952–108971, http://dx.doi.org/10.1109/ACCESS.2020.2998358, arXiv:1911.01276.

[6] W. Xu, J. Cui, L. Li, B. Yao, S. Tian, Z. Zhou, Digital twin-based industrial cloud robotics: Framework, control approach and implementation, J. Manuf. Syst. 58 (PB) (2021) 196–209, http://dx.doi.org/10.1016/j.jmsy.2020.07.013.

[7] A. Mazumder, M. Sahed, Z. Tasneem, P. Das, F. Badal, M. Ali, M. Ahamed, S. Abhi, S. Sarker, S. Das, M. Hasan, M. Islam, M. Islam, Towards next generation digital twin in robotics: Trends, scopes, challenges, and future, Heliyon 9 (2) (2023) e13359, http://dx.doi.org/10.1016/j.heliyon.2023.e13359.

[8] S.A. Niederer, M.S. Sacks, M. Girolami, K. Willcox, Scaling digital twins from the artisanal to the industrial, Nat. Comput. Sci. 1 (5) (2021) 313–320, http://dx.doi.org/10.1038/s43588-021-00072-5.

[9] D. Lehner, J. Pfeiffer, E.-F. Tinsel, M.M. Strljic, S. Sint, M. Vierhauser, A. Wortmann, M. Wimmer, Digital Twin Platforms: Requirements, Capabilities, and Future Prospects, IEEE Softw. 39 (2) (2022) 53–61.

[10] S. Gil, P.H. Mikkelsen, C. Gomes, P.G. Larsen, Survey on open-source digital twin frameworks–A case study approach, Softw.: Pr. Exp. 54 (6) (2024) 929–960, http://dx.doi.org/10.1002/spe.3305.

[11] A.K. Ramasubramanian, R. Mathew, M. Kelly, V. Hargaden, N. Papakostas, Digital twin for human–robot collaboration in manufacturing: Review and outlook, Appl. Sci. 12 (10) (2022) http://dx.doi.org/10.3390/app12104811, URL https://www.mdpi.com/2076-3417/12/10/4811.

[12] B. Siciliano, O. Khatib, Springer Handbook of Robotics, 2016, pp. 1–2227, http://dx.doi.org/10.1007/978-3-319-32552-1.

[13] C. Gomes, C. Thule, D. Broman, P.G. Larsen, H. Vangheluwe, Co-simulation: A survey, ACM Comput. Surv. 51 (3) (2018) http://dx.doi.org/10.1145/3179993.

[14] C. Gomes, B. Meyers, J. Denil, C. Thule, K. Lausdahl, H. Vangheluwe, P. De Meulenaere, Semantic adaptation for FMI co-simulation with hierarchical simulators, Simulation 95 (3) (2019) 241–269, http://dx.doi.org/10.1177/0037549718759775.

[15] J. Michael, J. Pfeiffer, B. Rumpe, A. Wortmann, Integration Challenges for Digital Twin Systems-of-Systems, in: SESoS, in: SESoS, IEEE/ACM, 2022, pp. 9–12, http://dx.doi.org/10.1145/3528229.3529384.

[16] S.T. Hansen, C.A.G. Gomes, M. Najafi, T. Sommer, M. Blesken, I. Zacharias, O. Kotte, P.R. Mai, K. Schuch, K. Wernersson, C. Bertsch, T. Blochwitz, A. Junghanns, The FMI 3.0 standard interface for clocked and scheduled simulations, Electronics 11 (21) (2022) http://dx.doi.org/10.3390/electronics11213635, URL https://www.mdpi.com/2079-9292/11/21/3635.

[17] H. Ejersbo, K. Lausdahl, M. Frasheri, L. Esterle, Dynamic runtime integration of new models in digital twins, in: 2023 IEEE/ACM 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS, 2023, pp. 44–55, http://dx.doi.org/10.1109/SEAMS59076.2023.00016.

[18] D. Lehner, S. Gil, P.H. Mikkelsen, P.G. Larsen, M. Wimmer, An architectural extension for digital twin platforms to leverage behavioral models, in: 2023 IEEE 19th International Conference on Automation Science and Engineering, CASE, 2023, pp. 1–8, http://dx.doi.org/10.1109/CASE56687.2023.10260417.

[19] P. Talasila, C. Gomes, P.H. Mikkelsen, S. Gil Arboleda, E. Kamburjan, P.G. Larsen, Digital twin as a service (dtaas): A platform for digital twin developers and users, in: IEEE Smart World Congress, SWC, IEEE, Portsmouth, UK, 2023, pp. 1–8, http://dx.doi.org/10.1109/SWC57546.2023.10448890.

[20] A. Cavalcanti, A. Sampaio, A. Miyazawa, P. Ribeiro, M. Conserva Filho, A. Didier, W. Li, J. Timmis, Verified simulation for robotics, Sci. Comput. Program. 174 (2019) 1–37, http://dx.doi.org/10.1016/j.scico.2019.01.004, URL https://www.sciencedirect.com/science/article/pii/S0167642318301655.

[21] S. Gil, P.H. Mikkelsen, D. Tola, C. Schou, P.G. Larsen, A Modeling Approach for Composed Digital Twins in Cooperative Systems, in: 2023 IEEE 28th International Conference on Emerging Technologies and Factory Automation, ETFA, IEEE, 2023, pp. 1–8, http://dx.doi.org/10.1109/ETFA54631.2023.10275601.

[22] S. Gil, C. Schou, P.H. Mikkelsen, P.G. Larsen, Integrating Skills into Digital Twins in Cooperative Systems, in: 2024 IEEE/SICE International Symposium on System Integration, SII, IEEE, 2024, pp. 1124–1131, http://dx.doi.org/10.1109/SII58957.2024.10417610.

[23] S. Gil, E. Kamburjan, P. Talasila, P.G. Larsen, An architecture for coupled digital twins with semantic lifting, Softw. Syst. Model. 24 (5) (2025) 1379–1404, http://dx.doi.org/10.1007/s10270-024-01221-w.

[24] E. Rohmer, S.P.N. Singh, M. Freese, CoppeliaSim (formerly V-REP): a versatile and scalable robot simulation framework, in: Proc. of the International Conference on Intelligent Robots and Systems, IROS, 2013, pp. 1321–1326, http://dx.doi.org/10.1109/IROS.2013.6696520, www.coppeliarobotics.com.

[25] A. Maria, Introduction to modeling and simulation, in: Winter Simulation Conference Proceedings, 1997, pp. 7–13, http://dx.doi.org/10.1145/268437.268440.

[26] D. Bullock, B. Johnson, R.B. Wells, M. Kyte, Z. Li, Hardware-in-the-loop simulation, Transp. Res. Part C: Emerg. Technol. 12 (1) (2004) 73–89, http://dx.doi.org/10.1016/j.trc.2002.10.002.

[27] C. Thule, K. Lausdahl, C. Gomes, G. Meisl, P.G. Larsen, Maestro: The INTO-CPS co-simulation framework, Simul. Model. Pr. Theory 92 (August 2018) (2019) 45–61, http://dx.doi.org/10.1016/j.simpat.2018.12.005.

[28] A. Falcone, A. Garro, Distributed Co-Simulation of Complex Engineered Systems by Combining the High Level Architecture and Functional Mock-up Interface, Simul. Model. Pr. Theory 97 (August) (2019) 101967, http://dx.doi.org/10.1016/j.simpat.2019.101967.

[29] M. Frasheri, H. Ejersbo, C. Thule, L. Esterle, RMQFMU: Bridging the real world with co-simulation for practitioners, in: H.D. Macedo, C. Thule, K. Pierce (Eds.), Proceedings of the 19th International Overture Workshop, Overture, 2021.

[30] M. Frasheri, H. Ejersbo, C. Thule, C. Gomes, J.L. Kvistgaard, P.G. Larsen, L. Esterle, Addressing time discrepancy between digital and physical twins, Robot. Auton. Syst. 161 (2023) 104347, http://dx.doi.org/10.1016/j.robot.2022.104347.

[31] C.M. Legaard, D. Tola, T. Schranz, H.D. Macedo, P.G. Larsen, A universal mechanism for implementing functional mock-up units, in: 11th International Conference on Simulation and Modeling Methodologies, Technologies and Applications, in: SIMULTECH, Virtual Event, 2021, pp. 121–129.

[32] M. Dalibor, N. Jansen, B. Rumpe, D. Schmalzing, L. Wachtmeister, M. Wimmer, A. Wortmann, A Cross-Domain Systematic Mapping Study on Software Engineering for Digital Twins, J. Syst. Softw. 193 (2022) 111361, http://dx.doi.org/10.1016/j.jss.2022.111361.

[33] E. VanDerHorn, S. Mahadevan, Digital Twin: Generalization, characterization and implementation, Decis. Support Syst. 145 (February) (2021) 113524, http://dx.doi.org/10.1016/j.dss.2021.113524.

[34] W. Kritzinger, M. Karner, G. Traar, J. Henjes, W. Sihn, Digital Twin in manufacturing: A categorical literature review and classification, in: IFAC, in: IFAC, vol. 51, Elsevier, 2018, pp. 1016–1022, http://dx.doi.org/10.1016/j.ifacol.2018.08.474.

[35] B.J. Oakes, A. Parsai, S. Van Mierlo, S. Demeyer, J. Denil, P. De Meulenaere, H. Vangheluwe, Improving digital twin experience reports, in: MODELSWARD 2021 - Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development, 2021, pp. 179–190, http://dx.doi.org/10.5220/0010236101790190.

[36] J. Fitzgerald, C. Gomes, P.G. Larsen (Eds.), The Engineering of Digital Twins, Springer, 2024, http://dx.doi.org/10.1007/978-3-031-66719-0.

[37] M.M. Rathore, S.A. Shah, D. Shukla, E. Bentafat, S. Bakiras, The Role of AI, Machine Learning, and Big Data in Digital Twinning: A Systematic Literature Review, Challenges, and Opportunities, IEEE Access 9 (2021) 32030–32052, http://dx.doi.org/10.1109/ACCESS.2021.3060863.

[38] F. Tao, B. Xiao, Q. Qi, J. Cheng, P. Ji, Digital twin modeling, J. Manuf. Syst. 64 (2022) 372–389, http://dx.doi.org/10.1016/j.jmsy.2022.06.015.

[39] G. Abbiati, C. Gomes, M. Sandberg, Z. Kazemi, S.T. Hansen, P.G. Larsen, Modelling for digital twins, in: J. Fitzgerald, C. Gomes, P.G. Larsen (Eds.), The Engineering of Digital Twins, Springer International Publishing, Cham, 2024, pp. 89–127, http://dx.doi.org/10.1007/978-3-031-66719-0_5.

[40] L.I. Hatledal, R. Skulstad, G. Li, A. Styve, H. Zhang, Co-simulation as a Fundamental Technology for Twin Ships, Model. Identif. Control 41 (4) (2020) 297–311, http://dx.doi.org/10.4173/mic.2020.4.2.

[41] A. Baratta, A. Cimino, F. Longo, L. Nicoletti, Digital twin for human-robot collaboration enhancement in manufacturing systems: Literature review and direction for future developments, Comput. Ind. Eng. 187 (2024) 109764, http://dx.doi.org/10.1016/j.cie.2023.109764, URL https://www.sciencedirect.com/science/article/pii/S036083522300788X.

[42] S. Martinez, A. Mariño, S. Sanchez, A.M. Montes, J.M. Triana, G. Barbieri, S. Abolghasem, J. Vera, M. Guevara, A Digital Twin Demonstrator to enable flexible manufacturing with robotics: a process supervision case study, Prod. Manuf. Res. 9 (1) (2021) 140–156, http://dx.doi.org/10.1080/21693277.2021.1964405.

[43] A.A. Malik, A. Brem, Digital twins for collaborative robots: A case study in human-robot interaction, Robot. Comput.-Integr. Manuf. 68 (2021) 102092, http://dx.doi.org/10.1016/j.rcim.2020.102092.

[44] A. Bilberg, A.A. Malik, Digital twin driven human–robot collaborative assembly, CIRP Ann 68 (1) (2019) 499–502, http://dx.doi.org/10.1016/j.cirp.2019.04.011.

[45] C. Saavedra Sueldo, I. Perez Colo, M. De Paula, S.A. Villar, G.G. Acosta, ROS-based architecture for fast digital twin development of smart manufacturing robotized systems, Ann. Oper. Res. 322 (1) (2023) 75–99, http://dx.doi.org/10.1007/s10479-022-04759-4.

[46] J. Acker, I. Rogers, D. Guerra-Zubiaga, M.H. Tanveer, A.A.A. Moghadam, Low-Cost Digital Twin Approach and Tools to Support Industry and Academia: A Case Study Connecting High-Schools with High Degree Education, Machines 11 (9) (2023) http://dx.doi.org/10.3390/machines11090860.

[47] A. Alves, C. Lagartinho-Oliveira, F. Moutinho, L. Gomes, ROS-Based Digital Twin for Power Wheelchair, in: 1st IEEE Industrial Electronics Society Annual on-Line Conference, ONCON 2022, IEEE, 2022, pp. 1–6, http://dx.doi.org/10.1109/ONCON56984.2022.10127002.

[48] J. Mattila, R. Ala-Laurinaho, J. Autiosalo, P. Salminen, K. Tammi, Using Digital Twin Documents to Control a Smart Factory: Simulation Approach with ROS, Gazebo, and Twinbase, Machines 10 (4) (2022) http://dx.doi.org/10.3390/machines10040225.

[49] A. Sterk-Hansen, B.H. Saghaug, D. Hagen, M.F. Aftab, A ROS 2 and TwinCAT Based Digital Twin Framework for Mechatronics Systems, in: 2023 11th International Conference on Control, Mechatronics and Automation, ICCMA 2023, 2023, pp. 485–490, http://dx.doi.org/10.1109/ICCMA59762.2023.10374978.

[50] M.R. Pedersen, L. Nalpantidis, R.S. Andersen, C. Schou, S. Bøgh, V. Krüger, O. Madsen, Robot skills for manufacturing: From concept to industrial deployment, Robot. Comput.-Integr. Manuf. 37 (2016) 282–291, http://dx.doi.org/10.1016/j.rcim.2015.04.002.

[51] S. Bader, E. Barnstedt, H. Bedenbender, M. Billman, B. Boss, A. Braunmandl, Details of the Asset Administration Shell Part 1 - The exchange of information between partners in the value chain of Industrie 4.0, 2020, p. 473, URL https://www.plattform-i40.de/PI40/Redaktion/DE/Downloads/Publikation/Details-of-the-Asset-Administration-Shell-Part1.html.

[52] L. Sciullo, A. De Marchi, A. Trotta, F. Montori, L. Bononi, M. Di Felice, Relativistic digital twin: Bringing the IoT to the future, Future Gener. Comput. Syst. 153 (2024) 521–536, http://dx.doi.org/10.1016/j.future.2023.12.016, URL https://www.sciencedirect.com/science/article/pii/S0167739X23004788.

[53] X. Li, B. He, Y. Zhou, G. Li, Multisource model-driven digital twin system of robotic assembly, IEEE Syst. J. 15 (1) (2021) 114–123, http://dx.doi.org/10.1109/JSYST.2019.2958874.

[54] S. Gil, B.J. Oakes, C. Gómes, M. Frasheri, P.G. Larsen, Towards a systematic reporting framework for digital twins: A cooperative robotics case study, Simul.: Trans. Soc. Model. Simul. 101 (3) (2025) 313–339, http://dx.doi.org/10.1177/00375497241261406.

[55] A. Cavalcanti, Z. Attala, J. Baxter, A. Miyazawa, P. Ribeiro, Model-Based Engineering for Robotics with RoboChart and RoboTool, in: International Colloquium on Theoretical Aspects of Computing, vol. 13490 LNCS, Springer, 2023, pp. 106–151, http://dx.doi.org/10.1007/978-3-031-43678-9_4.

[56] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, A.W. Roscoe, FDR3 - A Modern Refinement Checker for CSP, in: Tools and Algorithms for the Construction and Analysis of Systems, 2014, pp. 187–201.

[57] SDFormat, SDFormat Home, 2024, http://sdformat.org/.

[58] P. Corke, J. Haviland, Not your grandmother's toolbox–the robotics toolbox reinvented for python, in: ICRA, IEEE, 2021, pp. 11357–11363.

[59] A. Miyazawa, S. Ahmadi, A. Cavalcanti, M. Post, J. Timmis, P. Ribeiro, T. Wright, Diagrammatic physical models of robotic systems, Softw. Syst. Model. 24 (5) (2025) 1549–1593, http://dx.doi.org/10.1007/s10270-025-01270-9.

[60] A.L.C. Cavalcanti, A. Miyazawa, U. Schulze, J. Timmis, Bringing RoboStar and RT-Tester together, in: Jan Peleska's Festschrift, Springer, 2023, pp. 16–33, http://dx.doi.org/10.1007/978-3-031-40132-9_2.

[61] E. Madsen, D. Tola, C. Hansen, C. Gomes, P.G. Larsen, AURT: A Tool for Dynamics Calibration of Robot Manipulators*, in: Proc. of SII, IEEE/SISE, 2022, pp. 190–195, http://dx.doi.org/10.1109/SII52469.2022.9708769.

[62] E.W. Tsang, Generalizing from research findings: The merits of case studies, Int. J. Manag. Rev. 16 (4) (2014) 369–383, http://dx.doi.org/10.1111/ijmr.12024.

[63] R. Wieringa, M. Daneva, Six strategies for generalizing software engineering theories, Sci. Comput. Program. 101 (2015) 136–152, http://dx.doi.org/10.1016/j.scico.2014.11.013.

[64] Z. Attala, A.L. Cavalcanti, J.C.P. Woodcock, Modelling and verifying robotic software that uses neural networks, in: E. Ábrahám, C. Dubslaff, S.L.T. Tarifa (Eds.), Theoretical Aspects of Computing, Springer, 2023, pp. 15–35, http://dx.doi.org/10.1007/978-3-031-47963-2_3.

[65] A. Miyazawa, P. Ribeiro, W. Li, A.L.C. Cavalcanti, J. Timmis, J.C.P. Woodcock, RoboChart: modelling and verification of the functional behaviour of robotic applications, Softw. Syst. Model. 18 (5) (2019) 3097–3149, http://dx.doi.org/10.1007/s10270-018-00710-z, rdcu.be/bh7dI.

Santiago Gil is a postdoctoral researcher at the Department of Electrical and Computer Engineering, Aarhus University, Denmark, where he completed his PhD degree in 2024. He completed his bachelor's and master's degrees in Colombia and moved to Denmark to pursue his PhD studies. His research interests include digital twins, cyber–physical systems, co-simulation, Internet of Things, and digital transformation.

Arjun Badyal is a Ph.D. student in Computer Science at the University of York, where he also received his MMath. His research currently focuses on the verification of robotic systems, through simulation and proof. Arjun is interested in ensuring that verification techniques accommodate the variety of mathematical representations and formulations encountered in physical models of robots.

Alvaro Miyazawa is a lecturer at the Department of Computer Science of the University of York and a member of the RoboStar Centre for Software Engineering for Robotics. Having completed his doctoral research at the University of York, his main research interests are formal semantics and refinement for domain-specific languages and graphical notations and the development of verification strategies to support high levels of automation in verification. He has applied and developed formal techniques in various fields, including systems engineering, safety-critical real-time systems, and robotics. Currently, his research focuses on modelling, testing, simulation and verification for robotics.

Peter Gorm Larsen is a professor and deputy-head of section at the Department for Electrical and Computer Engineering at Aarhus University. He currently leads the AU DIGIT Centre, the AU Centre for Digital Twins, as well as the research group for CyberPhysical Systems. His research areas range from formal methods over cyber–physical systems to digital twins.

Ana Cavalcanti is a Professor at the University of York, UK, and holds a Royal Academy of Engineering Chair in Emerging Technologies. She directs the RoboStar centre on Software Engineering for Robotics. She previously held a Royal Society Industry Fellowship, which provided the ideal opportunity to understand and contribute to the practice of software verification working with QinetiQ. Her scientific achievements have been on design and justification of software development using industry-strength technology. She has chaired the Programme Committee of leading conferences. Currently, she is the Chair of the Formal Methods Europe Board.