Chapter 1
Sound Simulation and Co-simulation for Robotics

Ana Cavalcanti[1], Alvaro Miyazawa[1], Richard Payne[2], and Jim Woodcock[1]

1.1 Introduction

Safety is a major concern in robotics: for example, regulations for industrial robots often require them to be kept in cages, and autonomous vehicles currently cannot be certified for civil aviation. The ability to provide safety evidence can create significant opportunities. Yet, the programming techniques in use involve, on one hand, advanced robotics technology, and, on the other, outdated approaches to validation and verification of software controllers (either in isolation or in the context of a specific robotic hardware platform and environment).

Figure 1.1 indicates, in bold, the artefacts that are currently engineered in typical developments. The other artefacts are either provided on an ad hoc basis by particular tools or missing. In a first development phase, a state machine is often used to define the controller. If relevant, timed and probabilistic behaviours are recorded informally. For simulation, if the tool of choice does not provide adequate hardware and environment models, they need to be adapted or developed. A main difficulty is reasoning about the effect of the environment or recording assumptions about it. Connections between the artefacts are tenuous.

We propose here extensions and restrictions to INTO-SysML [2], a SysML [31] profile for cyber-physical systems. Our goal is to support modelling of robotic applications, including facilities to specify time, probabilities, the robotic platform, and the environment. SysML is a UML-based language that is becoming a de facto standard for systems, rather than just software, modelling. The INTO-SysML profile restricts the use of SysML block and internal block diagrams to characterise a collection of potentially heterogenous (co-)models as typically required for describing cyber-physical systems. These models may be written using a variety of notations adopting a variety of modelling paradigms. The definition of a collection of models

Department of Computer Science, University of York, UK · School of Computing Science, Newcastle University, UK

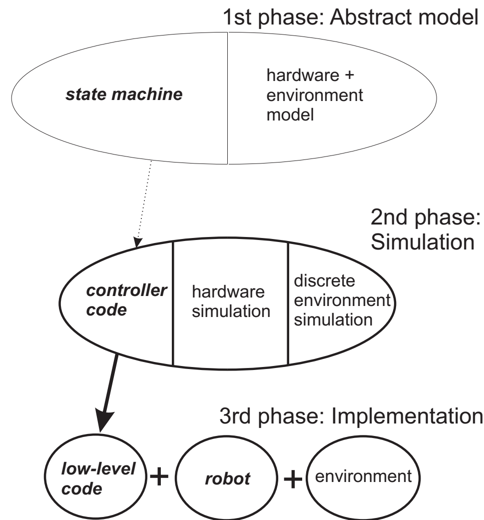

Fig. 1.1: Robot controller development artefacts: current practice

in INTO-SysML identifies how they are described and how they are connected to each other to specify a complete cyber-physical system (CPS).

Our version of the INTO-SysML profile dictates the use of a domain-specific language for discrete modelling of robot controllers: RoboChart [30]. For continuous modelling, we use as an example Simulink [29], which is widely used in industry for simulation of control laws. We can, however, accommodate other notations for continuous modelling. To illustrate our novel SysML profile, we apply it to describe co-models for the chemical detector in [22][1] and its environment.

Tools and facilities for simulation of robotic systems range from APIs [27] to sophisticated engines [35, 24] that embed discrete hardware and environment models. The variety of tools, simulation languages, and facilities for hardware and environment modelling and simulation means that the choice is not obvious, tool-specific knowledge is required, and reuse across tools is difficult.

In the long term, we envisage the automatic generation of tool-independent simulations from models written in INTO-SysML. Typically, for the continuous models, simulation tools are available. For RoboChart, automatic generation of simulations is under development. To support overall simulation of the various heterogeneous models of components, controllers, robotic platforms, and environment, using the most adequate tool for the task, we can adopt co-simulation. This is a technique widely adopted in industry to deal with the increased complexity of cyber-physical systems via the coordinated use of heterogeneous models and tools.

An industry standard, FMI [15] (Functional Mock-up Interface), supports orchestration. It avoids the need for customised coupling of each collection of simulation tools relevant for an application. An FMI co-simulation comprises black-box slave

[1] See `http://tinyurl.com/hdaws7o`.

FMUs (Functional Mockup Units); these wrap simulation models, connected via their inputs and outputs. A master algorithm triggers and orchestrates the simulation of the FMUs. In our envisaged approach, the environment, the controllers, and sometimes the robotic platform are in different FMUs. A co-simulation evolves in steps, which are synchronisation and data exchange points. An FMI API supports the programming of the master algorithm.

An FMI-based co-simulation framework for robotic applications can help developers face the challenge of heterogeneity. Our vision is for a framework that uses automatically generated co-simulations guaranteed to preserve the properties of the co-models described using INTO-SysML.

Besides defining a version of INTO-SysML for robotics, here we also give the INTO-SysML profile a formal behavioural semantics defined using the CSP process algebra [36]. The semantics captures the properties that every realisation of the co-simulation must satisfy. It captures the behaviours of the FMI simulations that orchestrate the executions of the multi-models as specified in SysML.

The use of CSP is a front-end to a semantic model that is described using Unifying Theories of Programming (UTP) [23] and can be extended to deal with continuous time and variables. With this preliminary use of CSP, we enable the use of the model checker FDR [19] for validation. Our semantics complements existing results on UTP semantics of RoboChart [30] and of FMI [9].

We present the background material to our work: SysML and the INTO-SysML profile, FMI, and RoboChart in Sect. 1.2. Sect. 1.3 presents our SysML profile. Sect. 1.4 presents our semantics. We consider related work in Sect. 1.5 and conclude in Sect. 1.6, discussing also our agenda for future work.

1.2 Preliminaries

In what follows, we present the notations used in our work.

1.2.1 SysML and the INTO-SysML profile

SysML [31] is a general-purpose graphical notation for systems engineering applications, defined as an extension of a subset of UML [20]. This extension is achieved by using UML's profile mechanism, which provides a generic technique for customising UML models for particular domains and platforms. A profile is a conservative extension of UML, refining its semantics in a consistent fashion.

There are commercial and open-source SysML tools. These include IBM's Rational Rhapsody Designer,[2] Atego's Modeler,[3] and Modeliosoft's Modelio.[4] They support model-based engineering and have been used in complex systems.

Like a UML model, a SysML model can be described by a set of diagrams. A central notion in SysML is that of a block, which is a structural element that represents a general system component, describing functional, physical, or human behaviour. The SysML Block Definition Diagram (BDD) shows how blocks are assembled into architectures; it is analogous to a UML Class Diagram, but is based on the more general notion of block. A BDD represents how the system is composed from its blocks using associations and other composition relations.

A SysML Internal Block Diagram (IBD) allows a designer to refine a block's structure; it is analogous to UML's Composite Structure Diagram, which shows the internal structure of a class. In an IBD, parts are assembled to define how they collaborate to realise the block's overall behaviour.

SysML includes a number of other diagrams to define state machines, flow charts, requirements, and so on. The BDD and IBD just described are the only ones relevant for our profile for description of co-simulations.

The INTO-SysML profile [2] customises SysML for architectural modelling for FMI co-simulation. It specialises blocks to represent different types of components, that is, co-models, of a CPS, constituting the building blocks of a hierarchical description of a CPS architecture. A component is a logical or conceptual unit of the system, software or a physical entity, modelled as an FMU.

The following types of components are represented in INTO-SysMML using specialised blocks: `System`, `EComponent` (encapsulating component), and `POComponent` (part-of component). A `System` block is decomposed into subsystems: `EComponents`, which are further decomposed into `POComponents`. An `EComponent` corresponds to an FMU. `EComponents` and `POComponents` may be further classified as `Subsystem`, a collection of inner components, `Cyber`, an atomic unit that inhabits the digital or logical world, or `Physical`, an atomic unit in the physical world. Characterising phenomena may be classified as being `discrete` or `continuous`. Decomposition of `EComponents` can be used to provide more information about the structure of the co-models, although that structure is hidden inside an FMU from the point of view of the FMI co-simulation.

To define the system, the components, and their relationships, INTO-SysML comprises two diagram types, Architecture Structure Diagrams (ADs) and Connection Diagrams (CDs), specialising SysML BDDs and IBDs. ADs describe a decomposition in terms of the types of system components and their relations. They emphasise multi-modelling: certain components encapsulate a model built using some modelling tool (such as VDM/RT [26], 20-sim [4] or Open Modelica [18]). CDs include AD block instances to describe the configuration of the system's components, highlighting flow and connectedness of these components.

---

[2] See `sysml.tools/review-rhapsody-developer/`.

[3] See `http://www.atego.com/de/products/atego-modeler/`.

[4] See `www.modelio.org/`.

In our examples, the multi-models are written in Robochart (see Section 1.2.3) and Simulink [29]. The latter, developed by MathWorks, is a graphical programming environment for modelling, simulating, and analysing multi-domain dynamic systems. Its primary interface is a graphical block diagramming tool and a customisable set of block libraries. Simulink is a de facto standard for modelling and simulation of control systems in the automotive and avionics industry.

1.2.2 Functional Mock-up Interface (FMI)

The Functional Mock-up Interface (FMI) [15] is an industry standard for collaborative simulation of separately developed models of CPS components: co-simulation. The key idea is that, if a real product is assembled from components interacting in complex ways, each obeying physical laws (electronic, hydraulic, mechanical), then a virtual product can be created from models of those physical laws and a model of their control systems. Models in these different engineering fields are heterogeneous: they use different notations and simulation tools.

The purpose of FMI is to support this heterogeneous modelling and simulation of a CPS by using the most convenient tools to deal with the different models. FMI is used in a number of different industry sectors, including automotive, energy, aerospace, and real-time systems integration. There is a formal development process for the standard, and many tools now support FMI.

As mentioned above, an FMI co-simulation consists of FMUs, which are models encapsulated in wrappers, interconnected through inputs and outputs. FMUs are slaves: their collective simulations are orchestrated by a master algorithm. Each FMU simulation is divided into steps with barrier synchronisations for data exchange; between these steps, the FMUs are simulated independently.

A master algorithm communicates with FMUs through the FMI API, whose most important functions are those to exchange data, `fmi2Set` and `fmi2Get`, and that to command the execution of a simulation step, `fmi2DoStep`. Other functions of the FMI API support the low-level management of the FMUs: initialisation, termination, recovery of its state, and so on. They play an important role is supporting the implementation of sophisticated master algorithms. From a conceptual point of view, however, the co-simulation is characterised by the sequence of calls to `fmi2DoStep`, which define the simulation steps, and the associated values input and output using `fmi2Set` and `fmi2Get`.

The FMI standard does not specify master algorithms, but restricts the use of the API functions to constrain how a master algorithm can be defined and how an FMU may respond. Formal semantics for FMI can be found in [5, 9].

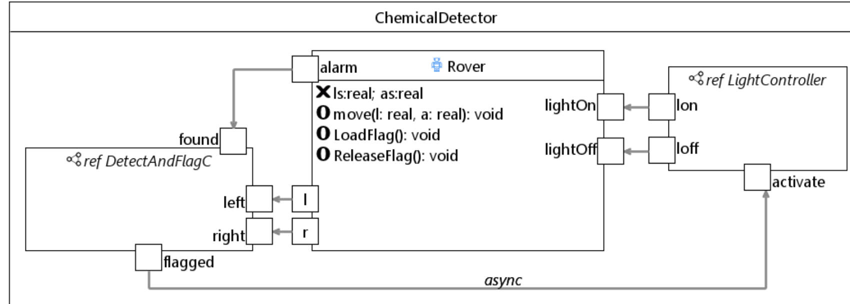

Fig. 1.2: RoboChart module

1.2.3 RoboChart

RoboChart [30] is a diagrammatic notation tailored to the design of robotic systems. RoboChart models use Harel-style statecharts [21], but crucially, also include constructs that embed concepts of robotic applications. They are used to structure models for abstraction and reuse. Moreover, distinctively, the RoboChart state machines use an action language that is both timed and probabilistic.

A RoboChart design centres around a robotic platform and its controllers. Communication between controllers can be either synchronous or asynchronous, but communication between state machines inside a controller is synchronous. The operations in a state machine may be given interface contracts using preconditions and postconditions, may be further defined by other state machines, or may come from a domain-specific API formalised separately. The formal semantics of RoboChart is mechanised in CSP [30] for model checking with FDR [19].

As a simple example, we consider a Rover robot inspired by that in [22]. It is an autonomous vehicle equipped to detect certain chemicals. It randomly traverses a designated area, sniffing its path with its onboard analysis equipment. If it detects a chemical source, it turns on a light and drops a flag as a marker.

A robotic system is specified in RoboChart by a module, where a robotic platform is connected to one or more controllers. A robotic platform is modelled by variables, events, and operations that represent built-in hardware facilities. The ChemicalDetector module for our example is shown in Figure 1.2; it has a robotic platform Rover and controllers DetectAndFlagC and LightC.

The named boxes on the border of Rover declare events. The lightOn and lightOff events can be used to request that the robot built-in light is switched one way or the other. The sensor events l and r record the detection by the robot of a wall on one side or the other. Similarly, the alarm indicates the detection of a chemical source by the built-in sensor in the robot.

The variables ls and as of Rover record its linear and angular speeds. The move(l,a) operation commands the Rover to move with speeds l (linear) and a (an-

gular); it is part of the RoboChart API. The operations LoadFlag() and ReleaseFlag(), on the other hand, are not in the RoboChart API, since they are particular to this example. They are declared, but not further defined.

The two controllers DetectAndFlagC and LightC define the behaviour of Rover. DetectAndFlagC controls the events left, right, found, and flagged (see the bordered boxes), thereby interacting with Rover and LightC. These events are associated with l, r, and alarm of Rover, and activate of LightC, as indicated by the arrows, whose directions define information flow. So, when Rover finds a chemical, it sends an alarm to DetectAndFlagC. LightC uses events lon, loff, and activate to communicate with Rover and DetectAndFlagC.

RoboChart models do not provide specific facilities to specify the robot physical model and the environment. In the next section, we propose a way of considering RoboChart models in the context of a variation of the INTO-SysML profile, especially designed to deal with RoboChart co-models. In Section 1.4, we give a semantics based on the FMI API for the co-simulation specified in SysML.

1.3 Multi-modelling of robots in SysML

We propose the combined use of RoboChart with the notation of a simulation tool for continuous systems: any of Simulink, 20-sim, or OpenModelica, for instance. For illustration, we consider here control-law diagrams used in Simulink.

The goal is to support the addition of detailed models for the robotic platform and for the environment. To identify these multi-models and their relationship, we propose to adapt the INTO-SysML profile.

A RoboChart module includes exactly one robotic platform, for which it gives a very abstract account. As already said, a RoboChart robotic platform defines just the variables, events, and operations available. In many cases, the operations are left unspecified, or are described just in terms of their effect on variables. In our ChemicalDetector module, for example, the operation move is specified just in terms of its effect on the variables ls and as of the robotic platform. There is no account of the actual laws of physics that control movement.

In a Simulink model, on the other hand, we can define the expected effect of the operations on the actual behaviour of the robot by capturing the laws of physics. In addition, we can also capture physical features of the environment that have a potential effect on the robot. To integrate the models, however, they need to share and expose events and variables. It is the purpose of the SysML model to depict the multi-models and their connections via events and variables.

Next, we present the extensions (Section 1.3.1) and restrictions (Section 1.3.2) to the INTO-SysML profile that we require, mainly for the specification of RoboChart and Simulink model composition. They are mostly confined to the Architecture Structure Diagrams. Some are of general interest, and some support the use of RoboChart as a co-model, in particular. In Section 1.3.3 we explain how we expect the resulting INTO-SysML profile to be used.

#	Description	INTO-SysML/RoboCalc
E1	The components of the `System` block can include `LComponent` blocks.	Both.
E2	We can have specialisations of `LComponent` blocks with stereotypes `Environment` and `RoboticPlatform` to group models for the environment and for the robotic platform. We can have any number of `Environment` blocks, but at most one block `RoboticPlatform`. These blocks can themselves be composed of any number of `LComponent` or `EComponent` blocks.	INTO-SysML may be extended to include an `Environment` block; however, `RoboticPlatform` should be RoboChart-specific.
E3	The type of a flow port optional.	RoboChart-specific.
E4	The `Platform` of an `EComponent` is a `String` and can include any simulation tools. Alternatively, RoboChart and Simulink need to be admitted.	RoboChart-specific.

Table 1.1: Overview of proposed extensions to the INTO-SysML profile

1.3.1 Extensions to INTO-SysML

The extensions are outlined in Table 1.1, and the restrictions later in Table 1.2. Both tables identify if the changes are specific to the needs of multi-models involving RoboChart diagrams, or if they are more generally useful for cyber-physical systems and, therefore, could be included in the original INTO-SysML profile.

Figure 1.3 presents the definition of the Architecture Structure Diagram for the multi-models for the chemical detection system. The block ChemicalDetector represents the RoboChart module in Figure 1.2. The interface of a RoboChart module is defined by the variables and events in its robotic platform, which become ports of the SysML block that represents the module. The blocks Arena, WallSensor, MobilityHw, and ChemDHw represent Simulink models.

The first extension (E1) is about new `LComponent` blocks, which can be used to group models of the robotic platform or of the environment. They are logical blocks: they do not correspond to an actual component of the co-simulation. In our example, the `LComponent` block Rover represents the models for the robotic platform. It is composed of three `EComponent` blocks, representing Simulink models for the hardware for wall sensing, mobility, and chemical detection.

In fact, it is possible to define Rover as a `RoboticPlatform` block using the extension E2. On the other hand, we have just one `EComponent` that models one aspect of the environment, namely, the Arena where the robot moves.

Event-based communications are required in RoboChart, so we propose in E3 that ports may not carry any values. For instance, in our model, left and right are events of the robotic platform, as defined in RoboChart. As already said, they represent an indication of the presence of a wall from the sensors in the robotic platform. They carry no values; they are events declared without a type.

The INTO-SysML profile does not include operations on blocks. This is due to the fact that blocks are intended to inform FMI model descriptions. The FMI stan-

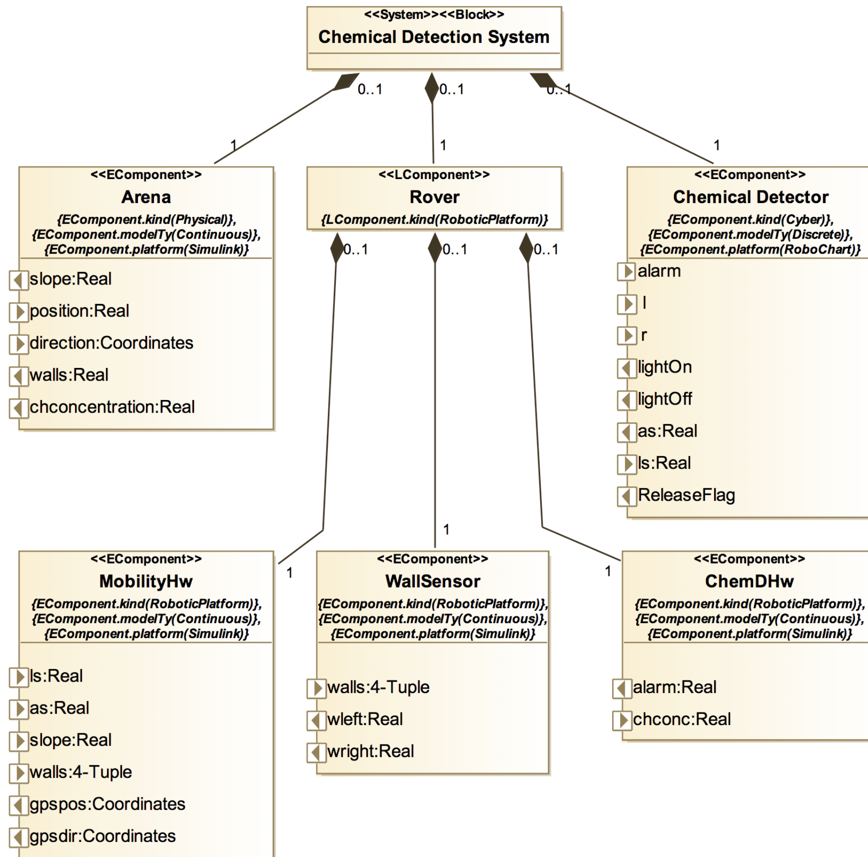

Fig. 1.3: Chemical Detector Architecture Structure Diagram

dard considers interactions to be in the form of typed data passed between FMUs—this data is shared at each time step of a simulation. As such, the profile does not natively support the concepts of event-based or operation-based interactions. The new typeless ports representing RoboChart events are, therefore, handled by encoding event occurrence using real numbers 0.0 and 1.0.

Finally, the extension E4 includes extra values, namely, RoboChart and Simulink, for defining the platform of an `EComponent` block.

For the Connection Diagram, no changes to the INTO-SysML profile are required, except that we can have instances of `LComponent` blocks as well. For our example, the diagram is shown in Figure 1.4.

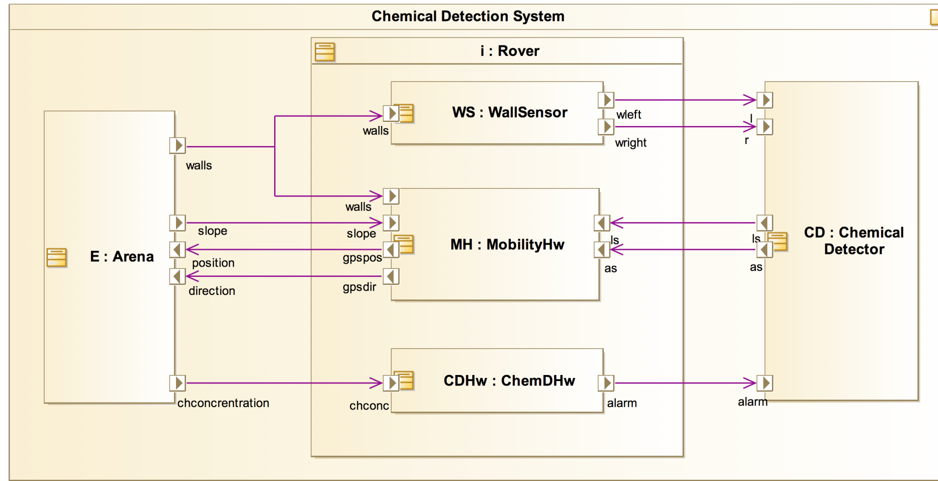

Fig. 1.4: Chemical Detector Connection Diagram

#	Description	INTO-SysML / RoboCalc
R1	There is exactly one `System` block and it cannot have flow ports.	Both.
R2	An `LComponent` block has no ports.	Both.
R3	In a diagram, we can have only one `EComponent` of `ComponentKind cyber` that must have `Platform` as RoboChart.	RoboChart-specific.
R4	When the kind of a variable is `parameter` and when the `Direction` of a `FlowPort` with a type is `in`, its initial value must be defined.	Both.
R5	Ports of a RoboChart block should be connected to the robotic platform, and not to the environment.	RoboChart-specific.
R6	No use of `POComponent` is needed, if Simulink is used. If textual continuous models are adopted, these blocks can be used.	RoboChart-specific.
R7	Typeless ports of a RoboChart block can be connected only to ports of type real of a Simulink block.	RoboChart-specific.

Table 1.2: Overview of proposed restrictions to INTO-SysML profile

1.3.2 Restrictions on INTO-SysML

As for the restrictions in Table 1.2, we have in R1 and R2 constraints on flow ports of `System` and `LComponent` blocks. Basically, a `System` block, Chemical Detection System in our example, is unique and can have no ports.

An `LComponent` block, like Rover in Figure 1.3, cannot have ports either. Since it is the `EComponent` blocks that represent multi-models, only them can contribute with inputs and outputs. For this reason, it does not make sense to include extra ports in an `LComponent` block, which just groups multi-models.

We recall that a `cyber` component models software aspects of the system. They should be specified using RoboChart. So, R3 requires that there is just one `cyber` component: the RoboChart module that is complemented by the Simulink diagrams. In our example, this is the ChemicalDetector block.

For simulation, parameters and inputs need an initial value as enforced by the restriction R4. For a parameter, this is the default value used in a simulation, unless an alternative value is provided. For the inputs, these are initial conditions that define the first set of outputs in the first step of the simulation. In our example, the initial values for position and direction, for instance, are (0.0,0.0) and (1.0,1.0). So, initially, the robot is stationary at a corner of the arena.

A controller can only ever sense or influence the environment using the sensors and actuators of the robotic platform. With R5, we, therefore, require that there is no direct connection between the controller and the environment. For example, Arena, representing a Simulink model of the environment, has an output port walls, that identifies as a 4-tuple the distances from the current position to the walls in the directions left, right, front, and back. This port is connected to the input port of the same name in WallSensor. It is this component that provides ports wleft and wright connected to the input ports l and r of ChemicalDetector.

We note that FMI does not have vector (array) types. So, strictly speaking, instead of ports with vector types, we should have separate ports for each component of the vectors. The inclusion of vectors in the FMI standard is, however, expected. We, therefore, make use of them in our example.

We require with R6 that no `POComponent` is included. For further detail in the models, we use RoboChart and Simulink, which are both diagrammatical. Of course, if only a textual continuous model is available, `POComponent` blocks can improve readability of the overall architecture of the system.

The only restriction relevant to a Connection Diagram is R7. Only ports of compatible types can be connected; compatibility between RoboChart and Simulink types is as expected. On the other hand, as already said, events in the RoboChart model that do not communicate values are represented by typeless ports of its SysML block. These ports can be connected to ports of a Simulink block representing a signal of type real. The values 0.0 and 1.0 can be used to represent the absence or occurrence of the event, for example.

The Simulink model for MobilityHw is shown in Figure 1.5. The input ports of MobilityHw, namely, ls, as, slope, and walls (see Figure 1.3) correspond to the input ports of the same name in the Simulink diagram. Moreover, the output ports gpspos and gpsdir correspond to the output ports current_position and current_direction of the Simulink diagram.

The Simulink diagram consists of two main sybsystems: rotate takes an angle and a vector as input and provides as output the result of rotating the vector by the given angle; and accelerate takes a linear speed (ls) and a slope and calculates the necessary acceleration profile to reach that velocity taking the slope into account. Essentially, given an initial_position and initial_direction, angular and linear speeds as and ls, a slope, and the distances walls to obstacles, the model calculates the movement of the robot. Restrictions over the speed are established by

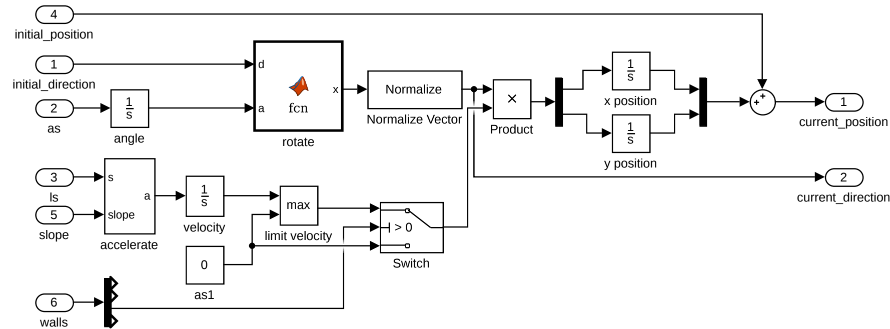

Fig. 1.5: MobilityHw Simulink Diagram

the block limit velocity. Restrictions over the position are described by the block Switch, which sets the speed to zero if there is a wall in front of the robot.

1.3.3 Use of INTO-SysML with RoboChart

In addition to the proposed required changes to the INTO-SysML profile, there are some specific methodological issues to consider when defining a multi-model for RoboChart using the INTO-SysML profile. We outline these below.

Given a RoboChart module, the corresponding `EComponent` block has a particular form. First of all, it must have: kind as `cyber`; the `Platform` as RoboChart; and the `ModelType` as `discrete`. This is illustrated in Figure 1.3.

The variables of the RoboChart robotic platform become output ports. In our example, the variables ls and as of the Rover in Figure 1.2 become output ports of ChemicalDetector in Figure 1.3. The variables record the speeds required by the controller. This is used to define the behaviour of the mobility hardware.

The events of the RoboChart robotic platform are part of the visible behaviour of the RoboChart module. For this reason, they become flow ports in the `cyber` `EComponent`. In our example, we have events l, r, alarm, lightOn and lightOff in the block ChemicalDetector of the Architecture Structure Diagram in Figure 1.3, just like in the Rover of the module in Figure 1.2.

In the RoboChart module, the definition of the direction of the events is from the point of view of the software controllers. In the Architecture Structure Diagram, the point of view is that of the hardware and the environment. So, their directions are reversed. For instance, the Rover in Figure 1.2 can send the events l and r to the controller DetectAndFlagC indicating the presence of a wall on the left or on the right. In ChemicalDetector, however, these are input events. The relevant part of the hardware itself is modelled by the `EComponent` WallSensor, where the matching events wleft and wright are indeed outputs.

Operations of the RoboChart robotic platform, on the other hand, may or may not be part of the visible behaviour of the module. So, they are not necessarily included. For instance, move andLoadFlag are not in the `EComponent` Rover. On the other hand, ReleaseFlag is included in the ChemicalDetector to indicate that a call to this operation is visible: we can see the flag dropped.

The flow ports of the `cyber EComponent` can become flow ports also of one of the `EComponent` blocks that represent the robotic platform. It is possible, however, that there are extra flow ports for communication between the models of the robotic platform and of the environment. In our example, for instance, an extra flow port slope is used for the environment model Arena to inform the hardware model MobilityHw of the inclination of the floor, which has an effect on the hardware control of movement to achieve the targetted speed.

In the next section, we define a semantics for our extended profile. Given an INTO-SysML model, like that in Figures 1.3 and 1.4, our semantics defines the FMI simulations that define inputs and outputs corresponding to the ports of the blocks in the AD, and that, at each step of the simulation, connect these ports as described in the CD. This establishes a correctness criterion for (a model of) an FMI simulation, which includes extra components: a master algorithm, and wrappers that allow communication between the FMUs and the master algorithm.

1.4 Semantics

A CSP semantics for INTO-SysML is already defined in [2]. Our semantics here is different in two ways: it considers extensions and restrictions described above, and it is based on events that represent calls to one of three functions of the FMI API: `fmi2Set`, `fmi2Get`, and `fmi2DoStep`. In contrast, the semantics in [2] identifies events with flows. It gives a simulation view of the model, where behaviour proceeds in steps, but a data flow is one interaction. In FMI, a flow is established by a pair of calls to `fmi2Set` and `fmi2Get` functions.

As a consequence of our approach here, our semantics is useful to define specifications for FMI simulations. In [9], we present a CSP semantics for such simulations that can be automatically generated from a description of the FMUs and their connections, and a choice of master algorithm. The semantics presented here can be used, for example, to verify the correctness of those models.

A CSP specification is defined by a number of processes that communicate via channels. The system and each of its components are defined by a process. Communication is synchronous, atomic, and instantaneous.

The CSP process that defines the semantics of an INTO-SysML model uses communications on the following channels.

channel $fmi2Get : FMI2COMP \times PORT \times VAL \times FMI2STATUSF$
channel $fmi2Set : FMI2COMP \times PORT \times VAL \times FMI2STATUS$
channel $fmi2DoStep : FMI2COMP \times TIME \times NZTIME \times FMI2STATUSF$

The types of these channels match the signature of the corresponding FMI API functions. $FMI2COMP$ contains indices for each of the used instances of `EComponent` blocks, which represent FMUs in the INTO-SysML profile.

$PORT$ contains indices, unique to each `EComponent` instance, to identify ports, which represent input and output variables of the FMU. VAL is the type of valid values; we do not model the SysML or the FMI type system. For ports corresponding to a RoboChart event, special values (perhaps just 0 and 1) represent absence or presence of an event occurrence. VAL must include these values.

In FMI, there is one `fmi2Get` and one `fmi2Set` function for each data type. For simplicity, however, we consider just one generic channel for each of them, since the overall behaviour of these functions is the same.

$FMI2STATUS$ and $FMI2STATUSF$ contain flags returned by a call to the API functions. In our model, all calls return the flag $fmi2OK$, indicating success. So, the scenarios that it defines do not cater for the possibility of errors.

Finally, the types $TIME$ and $NZTIME$ define a model of time, using natural numbers, for instance. In the case of $NZTIME$, it does not include 0, since `fmi2DoStep` does not accept a value 0 for a simulation step size.

For example, if we consider that the WallSensor FMU has index $WallSensor$, its port wleft has index $WSwleft$, and VAL includes the real numbers, then the communication $fmi2Get.WallSensor.WSwleft.1.fmi2OK$ models the successful recording of the value 1 for the variable corresponding to the port wleft in the FMU for WallSensor. Similarly, $fmi2DoStep.WallSensor.1.2.fmi2OK$ records a successful request for the same FMU to take a step at time 1, with a step size 2.

In what follows, we use the above channels to define CSP processes that correspond to `EComponent` instances (Section 1.4.1), and to co-simulations defined by a Connection Diagram for a `System` block (Section 1.4.2).

1.4.1 `EComponent` instances

The process $EComponent(ec)$ that defines the semantics of an `EComponent` block instance of index ec is specified in Figure 1.6. We define a number of local processes $Init(setup)$, $TakeOuptuts(outs)$, $DistributeInputs(inps)$, $Step$, and $Cycle$ used to define the behaviour of $EComponent(ec)$ as the initialisation defined by the process $Init$, followed by the process $Cycle$.

We use functions $Parameters$, $Inputs$, $Outputs$, and $Initials$, which, given an index ec, identify the parameters, input and output ports, and input ports with initial values, of the block instance ec. Instances of the same `EComponent` block have the same parameters, inputs, outputs, and input ports with initial values.

In $Init(setup)$, the parameters and the input ports identified in $setup$ are initialised using values defined by a fifth function $InitialValues$, which, given a block ec and a parameter or input port var, gives the value of the parameter or the initial value of the port. Initialisation is via the channel $fmi2Set$. Each variable var in $setup$ is initialised independently, so we have an interleaving ($|||$) of initialisations.

$$
\begin{array}{l}
\mathit{EComponent}(ec) = \mathrm{let} \\
\quad \mathit{Init}(\mathit{setup}) = \\
\qquad (|||\, \mathit{var} : \mathit{setup} \bullet \mathit{fmi2Set}.ec.\mathit{var}.\mathit{InitialValues}(ec, \mathit{var}).\mathit{fmi2OK} \longrightarrow \mathit{SKIP}) \\
\quad \mathit{TakeOutputs}(\mathit{outs}) = \\
\qquad \mathit{sync} \longrightarrow (|||\, \mathit{var} : \mathit{outs} \bullet \mathit{fmi2Get}.ec.\mathit{var}?x.\mathit{fmi2OK} \longrightarrow \mathit{SKIP}) \\
\quad \mathit{DistributeInputs}(\mathit{inps}) = \\
\qquad \mathit{sync} \longrightarrow (|||\, \mathit{var} : \mathit{inps} \bullet \mathit{fmi2Set}.ec.\mathit{var}?x.\mathit{fmi2OK} \longrightarrow \mathit{SKIP}) \\
\quad \mathit{Step} = \mathit{sync} \longrightarrow \mathit{fmi2DoStep}.ec?t?ss.\mathit{fmi2OK} \longrightarrow \mathit{SKIP} \\
\quad \mathit{Cycle} = \mathit{TakeOutputs}(\mathit{Outputs}(ec));\ \mathit{DistributeInputs}(\mathit{Inputs}(ec));\ \mathit{Step};\ \mathit{Cycle} \\
\mathrm{within} \\
\quad \mathit{Init}(\mathit{Parameters}(ec) \cup \mathit{Initials}(ec));\ \mathit{Cycle}
\end{array}
$$

Fig. 1.6: CSP model of an `EComponent` block

Each initialisation is defined by a prefixing ($\rightarrow$) of a communication on $\mathit{fmi2Set}$ to the process SKIP, which terminates immediately.

Cycle defines a cyclic behaviour in three phases for an `EComponent`. It continuously provides values for its outputs, as defined by $\mathit{TakeOutputs}(\mathit{Outputs}(ec))$, takes values for its inputs, as defined by $\mathit{DistributeInputs}(\mathit{Inputs}(ec))$, and then carries out a step of simulation, as defined by Step. As already said, $\mathit{fmi2Get}$ is a channel used to produce the values of the outputs, $\mathit{fmi2Set}$ is used to take input values, and $\mathit{fmi2DoStep}$ is used to mark a simulation step.

The process $\mathit{TakeOutputs}(\mathit{Outputs}(ec))$ offers all the outputs var in the set $\mathit{Outputs}(ec)$ via the channel $\mathit{fmi2Get}$ in interleaving. The particular value x output is not defined (as indicated by the ? preceding x in the communication via $\mathit{fmi2Get}$). This value can be determined only by a particular model (in RoboChart or Simulink, for instance) for the `EComponent`. A synchronisation on a channel sync, that is, a communication without data passing, is used to mark the start of the outputting phase before the interleaving of communications on $\mathit{fmi2Get}$.

The process $\mathit{DistributeInputs}(\mathit{Inputs}(ec))$ is similar, taking the inputs in the process $\mathit{Inputs}(ec)$ via $\mathit{fmi2Set}$. Finally, the process Step, after accepting a sync, takes an input via $\mathit{fmi2DoStep}$ of a time t and a step size ss, and terminates.

In the next section, the semantics of a co-simulation uses the parallel composition ($[\![...]\!]$) below of instances of $\mathit{EComponent}(ec)$ for each `EComponent` block instance. The processes $\mathit{EComponent}(ec)$ synchronise on sync to ensure that they proceed from phase to phase of their cycles in lock step.

$$
\mathit{BlockInstances} = ([\![\{\mathit{sync}\}]\!]\ ec : \mathit{FMI2COMP} \bullet \mathit{EComponent}(ec)) \setminus \{\mathit{sync}\}
$$

The communications on sync, however, are hidden ($\setminus$). Therefore, as already indicated, the collective behaviour of the block instances is specified solely in terms of communications on the FMI API channels: $\mathit{fmi2Get}$, $\mathit{fmi2Set}$, and $\mathit{fmi2DoStep}$.

$$
\begin{array}{l}
\mathit{Connections} = \mathrm{let} \\
\quad \mathit{Init} = |||\, ec : \mathit{FMI2COMP} \bullet \\
\qquad (|||\, \mathit{var} : \mathit{Initials}(ec) \bullet \mathit{fmi2Set.ec.var?x.fmi2OK} \longrightarrow \mathit{SKIP}) \\
\quad \mathit{Step} = \|\; c : \mathit{ConnectionIndex} \bullet [AC(c)]\, \mathit{Connection}(c) \\
\quad \mathit{Cycle} = \mathit{Step};\ \mathit{Cycle} \\
\mathrm{within} \\
\quad \mathit{Init};\ \mathit{Cycle}
\end{array}
$$

Fig. 1.7: CSP model of a Connection Diagram

1.4.2 Co-simulation

A Connection Diagram for a `System` block instance characterises a co-simulation by instantiating blocks of the Architecture Diagram and defining how their ports are connected. This is captured by the CSP process $\mathit{CoSimulation}$ defined in the sequel. We note that the instances of `LComponent` blocks plays no role in the co-simulation semantics, since these blocks do not represent any actual component of a co-simulation, but just a logical grouping of co-models.

Besides $\mathit{BlockInstances}$ above, $\mathit{CoSimulation}$ uses the process $\mathit{Connections}$ shown in Figure 1.7. This is defined in terms of a parallel composition ($\|$) Step of processes $\mathit{Connection}(c)$ that define each of the connections c in a Connection Diagram, identified by indices in a set $\mathit{ConnectionIndex}$.

The behaviour of $\mathit{Connections}$ is defined by the sequential composition of the local processes Init followed by Cycle. Init initialises in interleaving the variables corresponding to input ports of each component ec using the channel $\mathit{fmi2set}$ as before. Cycle continuously behaves like Step described above.

A $\mathit{Connection}(c)$ process takes an output from the source port of the connection and gives it to the target port. In our example, we can, for instance, give the connection between the ports wright of WallSensor and r of ChemicalDetector the index 3. In this case, $\mathit{Connection}(3)$ is as follows, where $\mathit{WallSensor}$ and $\mathit{ChemicalDetector}$ are the indices in $\mathit{FMI2COMP}$ for these `EComponent` block instances, and $\mathit{WSwright}$ and CDr are variables corresponding to wright and r.

$$
\begin{array}{l}
\mathit{Connection}(3) = \mathit{fmi2Get.WallSensor.WSwright?x.fmi2OK} \longrightarrow \\
\qquad\qquad\qquad\quad \mathit{fmi2Set.ChemicalDetector.CDr.x.fmi2OK} \longrightarrow \mathit{SKIP}
\end{array}
$$

$\mathit{Connection}(3)$ ensures that the value x output via wright is input to the r port.

In the parallel composition in the process Step, each process $\mathit{Connection}(c)$ is associated with an alphabet $AC(c)$, which includes the communications over $\mathit{fmi2Get}$ and $\mathit{fmi2Set}$ that represent the connection it models. In our example, $AC(3)$ contains all communications over $\mathit{fmi2Get}$ with parameters $\mathit{WallSensor}$ and $\mathit{WSwright}$, and over $\mathit{fmi2Set}$ with parameters $\mathit{ChemicalDetector}$ and CDr.

The use of these alphabets in the parallelism that defines $Step$ ensures that, if there are several connections with the same source port, they share the output in the port by synchronising on that communication. In our example, the output $Awalls$ of $Arena$ is shared between the processes for the connections between Arena and WallSensor and between Arena and MobilityHw.

Finally, the semantics for the co-simulation defined by a Connection Diagram for a System instance is the parallel composition below.

$$CoSimulation = BlockInstances \llbracket FMIGetSet \rrbracket Connections$$

The processes synchronise on communications on the set $FMIGetSet$ containing the union of the alphabets of the $Connection(c)$ processes.

If there are ports that are not associated with a connection, their corresponding communications via $fmi2Set$ or $fmi2Get$ are not included in $FMIGetSet$. These communications are restricted only by the process $BlockInstances$. In our example, the ports lightOn and lightOff, for instance, are not connected to any other ports. Their outputs are visible to the environment of the chemical detection system, but not connected to any other modelled components.

The behaviour defined by $CoSimulation$ specifies a cyclic simulation whose steps contains three phases: all outputs are taken in any order, used to provide all inputs, also in any order, and then the time advances via a simulation step of each multi-model. As already said, a CSP semantics for an FMI co-simulation is available [9]. With that, $CoSimulation$ can be used as a specification to validate an FMI co-simulation where the FMUs correspond to the EComponent block instances and must be orchestrated as indicated in the connection diagram.

1.5 Related work

Another graphical domain-specific language for robotics is presented in [12]. It also supports design modelling and automatic generation of platform-independent code. It was defined as a UML profile, with all the advantages and problems entailed by this; reasoning about non-functional properties is envisaged.

Model-based engineering of robotic systems is also advocated in [39], where a component-based framework that uses UML to develop robotics software is presented. The communication between components are realized through a set of communication patterns such as *request/response* and *publish/subscribe*, which define the visibility of components. The goal of that work, however, does not include definition of a formal semantics for controllers, like we do for RoboChart.

Some domain-specific languages focus on a particular application (like programming self-reconfigurable robots [41] and service robots [7]). GenoM3 [28] supports the description of robotic applications in terms of its execution tasks and services. It has recently been given a timed behavioural semantics [17].

There are (famously) many different semantics for UML state machines [13]. Kuske *et al.* [25] give semantics for UML class, object, and state-machine diagrams using graph transformation. Rasch and Wehrheim [34] use CSP to give semantics for extended class diagrams and state machines. Davies and Crichton [10] also use CSP to give semantics for UML class, object, statechart, sequence, and collaboration diagrams. Broy *et al.* [6] present a foundational semantics for a subset of UML2 using state machines to describe the behaviour of objects and their data structures. RoboChart state machines have a precise semantics in CSP in the spirit of [34] and [10]; however, for the sake of compositionality, RoboChart state machines do not include history junctions and inter-level transitions.

UML 2.0 includes a timing diagram, a specific kind of interaction diagram with added timing constraints. The UML-MARTE profile [43] provides richer models of time based on clocks, including notions of logical, discrete, and continuous time. The Clock Constraint Specification Language (CCSL) provides for the specification of complex timing constraints, including time budgets and deadlines. This is accomplished with sequence and time diagrams; it is not possible to define timed constraints in terms of transitions or states like in RoboChart.

UML-RT [42] encapsulates state machines in capsules; inter-capsule communication is through ports and is controlled by a timing protocol with timeouts. More complex constraints, including deadlines, are specified only informally.

The work in [33] defines a semantics for a UML-RT subset in untimed *Circus* [45]. An extension to UML-RT is considered in [1] with semantics given in terms of CSP+T [46], an extension of CSP that supports annotations for the timing of events within sequential processes. The RoboChart timed primitives are richer and are inspired by timed automata and Timed CSP [40].

Practical work on master algorithms for use in FMI co-simulations includes generation of FMUs, their simulations, and hybrid models [3, 32, 14, 11]. FMUs can encapsulate heterogeneous models; Tripakis [44] shows how components based on state machines and synchronous data flow can be encoded as FMUs. In our approach, we have a hybrid co-simulation, but each `EComponent` is either `discrete` or `continuous`. Extensions to FMI are required to deal with that [5].

Savicks [37] shows how to co-simulate Event-B and continuous models using a fixed-step master algorithm. Savicks does not give semantics for the FMI API, but supplements reasoning in Event-B with simulation of FMUs within Rodin, the Event-B platform, applying the technique to an industrial case study [38]. The work does not wrap Event-B models as FMUs, and so it does not constitute a general FMI-compliant co-simulation. Here, we do not consider the models of FMUs, but plan to wrap CSP-based models of Simulink [8] and RoboChart [30] to obtain CSP-based FMUs models that satisfy the specification in [9].

1.6 Conclusions and future work

In this paper, we have extended and restricted the INTO-SysML profile to deal with robotic systems. For modelling the controller(s), we use RoboChart. For modelling the robotic platform and the environment, we use Simulink. The approach, however, applies to other languages of same nature: event-based reactive languages to define software and control-law diagrams. We have also given a behavioural semantics for models written in the profile using CSP. The semantics is agnostic to RoboChart and Simulink, and captures a co-simulation view of the multi-models based on FMI.

Our semantics can be used in two ways. First, by integration with a semantics of each of the multi-models that defines their specific responses to the simulation steps, we can obtain a semantics of the system as a whole. Such semantics can be used to establish properties of the system, as opposed to properties of the individual models. In this way, we can confirm the results of (co-)simulations via model checking or, most likely, theorem proving, due to scalability issues.

As already mentioned, the CSP model is a front-end for a UTP predicative semantics. It is amenable to theorem proving using Isabelle [16].

There are CSP-based formal semantics for RoboChart [30] and Simulink [8] underpinned by the UTP. Our next step is their lifting to provide an FMI-based view of the behaviour of models written in these notations. With that, we can use RoboChart and Simulink models as FMUs in a formal model of a co-simulation as suggested here, and use CSP and the UTP to reason about the co-simulation. For RoboChart, for example, the lifting needs to transform inputs of values 0.0 and 1.0 on ports for typeless events to synchronisations. For Simulink, the notion of get and setting values, and simulation steps is more directly recorded.

It is also relatively direct to wrap existing CSP semantics for UML state machines [34, 10] to allow the use of such models as FMUs in a co-simulation. In this case, traditional UML modelling can be adopted.

Secondly, we can use our semantics as a specification for a co-simulation. The work in [9] provides a CSP semantics for an FMI co-simulation; it covers not only models of the FMUs, but also a model of a master algorithm of choice. The scenario defined by an INTO-SysML model identifies inputs and outputs, and their connections. The traces of the FMI co-simulation model should be allowed by the CSP semantics of the INTO-SysML model. This can be verified via model checking.

As indicated in Figure 1.1, currently there is no support to establish formal connections between a simulation and the state machine and physical models (of the robotic platform and the environment). The SysML profile proposed here supports the development of design models via the provision of domain-specific languages based on diagrammatic notations and facilities familiar to the engineering and robotics community for clear connection of models. Complementarily, as explained above, the profile semantics supports verification of FMI-based co-simulations.

There are plans for automatic generation of simulations of RoboChart models [30]. The semantics we propose can be used to justify the combination of these simulations with Simulink simulations as suggested above.

Acknowledgements

This work is funded by the INTO-CPS EU grant and EPSRC grant EP/M025756/1. The authors are grateful to Wei Li, Pedro Ribeiro, Augusto Sampaio, and Jon Timmis, for many discussions on RoboChart and simulation of robotic applications. No new primary data was created during this study.

References

1. K. B. Akhlaki, M. I. C. Tunon, J. A. H. Terriza, and L. E. M. Morales. A methodological approach to the formal specification of real-time systems by transformation of UML-RT design models. *Science of Computer Programming*, 65(1):41–56, 2007.
2. N. Amálio. Foundations of the SysML profile for CPS modelling. Technical Report D2.1a, INTO-CPS project, 2015.
3. J. Bastian, C. Clauß, S. Wolf, and P. Schneider. Master for co-simulation using FMI. In *Modelica Conference*, 2011.
4. J. F. Broenink. Modelling, simulation and analysis with 20-sim. *Computer Aided Control Systems Design*, 38(3):22–25, 1997.
5. D. Broman, C. Brooks, L. Greenberg, E. A. Lee, M. Masin, S. Tripakis, and M. Wetter. Determinate composition of FMUs for co-simulation. In *ACM SIGBED International Conference on Embedded Software*. IEEE, 2013.
6. M. Broy, M. V. Cengarle, and B. Rumpe. Semantics of UML – Towards a System Model for UML: The State Machine Model. Technical Report TUM-I0711, Institut für Informatik, Technische Universität München, February 2007.
7. A. Bubeck, F. Weisshardt, T. Sing, U. Reiser, M. Hagele, and A. Verl. Implementing best practices for systems integration and distributed software development in service robotics-the care-o-bot® robot family. In *2012 IEEE/SICE International Symposium on System Integration*, pages 609–614. IEEE, 2012.
8. A. L. C. Cavalcanti, P. Clayton, and C. O'Halloran. From Control Law Diagrams to Ada via *Circus*. *Formal Aspects of Computing*, 23(4):465–512, 2011.
9. A. L. C. Cavalcanti, J. C. P. Woodcock, and N. Amálio. Behavioural Models for FMI Co-simulations. Technical report, University of York, Department of Computer Science, York, UK, 2016. Available at `www-users.cs.york.ac.uk/~alcc/CWA16.pdf`.
10. J. Davies and C. Crichton. Concurrency and Refinement in the Unified Modeling Language. *Formal Aspects of Computing*, 15(2-3):118–145, 2003.
11. J. Denil, B. Meyers, P. De Meulenaere, and H. Vangheluwe. Explicit semantic adaptation of hybrid formalisms for FMI co-simulation. In *Spring Simulation Multi-Conference*, 2015.
12. S. Dhouib, S. Kchir, S. Stinckwich, T. Ziadi, and M. Ziane. *Simulation, Modeling, and Programming for Autonomous Robots*, chapter RobotML, a Domain-Specific Language to Design, Simulate and Deploy Robotic Applications, pages 149–160. Springer, 2012.
13. R. Eshuis. Reconciling statechart semantics. *Science of Computer Programming*, 74(3):65–99, 2009.
14. Y. A. Feldman, L. Greenberg, and E. Palachi. Simulating rhapsody SysML blocks in hybrid models with FMI. In *Modelica Conference*, 2014.
15. FMI development group. Functional mock-up interface for model exchange and co-simulation, 2.0. `https://www.fmi-standard.org`, 2014.
16. S. Foster, F. Zeyda, and J. C. P. Woodcock. Isabelle/UTP: A Mechanised Theory Engineering Framework. In D. Naumann, editor, *Unifying Theories of Programming*, volume 8963 of *Lecture Notes in Computer Science*, pages 21–41. Springer, 2015.

17. M. Foughali, B. Berthomieu, S. Dal Zilio, F. Ingrand, and A. Mallet. Model checking real-time properties on the functional layer of autonomous robots. In K. Ogata, M. Lawford, and S. Liu, editors, *Formal Methods and Software Engineering*, pages 383–399. Springer, 2016.
18. P. Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Press, 2004.
19. T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. W. Roscoe. FDR3 A Modern Refinement Checker for CSP. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 187–201, 2014.
20. Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1, August 2011.
21. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
22. J. A. Hilder, N. D. L. Owens, M. J. Neal, P. J. Hickey, S. N. Cairns, D. P. A. Kilgour, J. Timmis, and A. M. Tyrrell. Chemical detection using the receptor density algorithm. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(6):1730–1741, 2012.
23. C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.
24. J. Klein and L. Spector. 3D Multi-Agent Simulations in the breve Simulation Environment. In *Artificial Life Models in Software*, pages 79–106. Springer, 2009.
25. S. Kuske, M. Gogolla, R. Kollmann, and H.-J. Kreowski. An Integrated Semantics for UML Class, Object and State Diagrams Based on Graph Transformation. In M. Butler, L. Petre, and K. SereKaisa, editors, *Integrated Formal Methods*, volume 2335 of *Lecture Notes in Computer Science*, pages 11–28. Springer, 2002.
26. P. G. Larsen, N. Battle, M. Ferreira, J. Fitzgerald, K. Lausdahl, and M. Verhoef. The Overture initiative – integrating tools for VDM. *SIGSOFT Software Engineering Notes*, 35(1):1–6, 2010.
27. S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan. Mason: A multiagent simulation environment. *Simulation*, 81(7):517–527, 2005.
28. A. Mallet, C. Pasteur, M. Herrb, S. Lemaignan, and F. Ingrand. Genom3: Building middleware-independent robotic components. In *2010 IEEE International Conference on Robotics and Automation*, pages 4627–4632, 2010.
29. The MathWorks,Inc. *Simulink*. www.mathworks.com/products/simulink.
30. A. Miyazawa, P. Ribeiro, W. Li, A. L. C. Cavalcanti, J. Timmis, and J. C. P. Woodcock. RoboChart: a State-Machine Notation for Modelling and Verification of Mobile and Autonomous Robots. Technical report, University of York, Department of Computer Science, York, UK, 2016. Available at www.cs.york.ac.uk/circus/publications/techreports/reports/MRLCTW16.pdf.
31. OMG. OMG Systems Modeling Language (OMG SysML), Version 1.3, 2012.
32. U. Pohlmann, W. Schäfer, H. Reddehase, J. Röckemann, and R. Wagner. Generating functional mockup units from software specifications. In *Modelica Conference*, 2012.
33. R. Ramos, A. C. A. Sampaio, and A. C. Mota. A Semantics for UML-RT Active Classes via Mapping into *Circus*. In *Formal Methods for Open Object-based Distributed Systems*, volume 3535 of *Lecture Notes in Computer Science*, pages 99–114, 2005.
34. H. Rasch and H. Wehrheim. Checking consistency in UML diagrams: Classes and state machines. In E. Najm, U. Nestmann, and P. Stevens, editors, *Formal Methods for Open Object-Based Distributed Systems*, volume 2884 of *Lecture Notes in Computer Science*, pages 229–243. Springer, 2003.
35. E. Rohmer, S. P. N. Singh, and M. Freese. V-rep: A versatile and scalable robot simulation framework. In *IEEE International Conference on Intelligent Robots and Systems*, volume 1, pages 1321–1326. IEEE, 2013.
36. A. W. Roscoe. *Understanding Concurrent Systems*. Texts in Computer Science. Springer, 2011.
37. V. Savicks, M. Butler, and J. Colley. Co-simulating Event-B and Continuous Models via FMI. In *Summer Simulation Multiconference*, pages 37:1–37:8. Society for Computer Simulation International, 2014.

38. V. Savicks, M. Butler, and J. Colley. Co-simulation Environment for Rodin: Landing Gear Case Study. In F. Boniol, V. Wiels, Y. A. Ameur, and K.-D. Schewe, editors, *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 148–153. Springer, 2014.
39. C. Schlegel, T. Hassler, A. Lotz, and A. Steck. Robotic software systems: From code-driven to model-driven designs. In *14th International Conference on Advanced Robotics*, pages 1–8. IEEE, 2009.
40. S. Schneider. *Concurrent and Real-time Systems: The CSP Approach*. Wiley, 2000.
41. U. P. Schultz, C. D. Johan, and K. Stoy. A domain-specific language for programming self-reconfigurable robots. In *Workshop on automatic program generation for embedded systems (APGES)*, pages 28–36, 2007.
42. B. Selic. Using UML for modeling complex real-time systems. In F. Mueller and A. Bestavros, editors, *Languages, Compilers, and Tools for Embedded Systems*, volume 1474 of *Lecture Notes in Computer Science*, pages 250–260. Springer, 1998.
43. B. Selic and S. Grard. *Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems*. Morgan Kaufmann Publishers Inc., 2013.
44. S. Tripakis. Bridging the semantic gap between heterogeneous modeling formalisms and fmi. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 60–69. IEEE, 2015.
45. J. C. P. Woodcock and A. L. C. Cavalcanti. *Circus*: a concurrent refinement language. Technical report, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD UK, 2001.
46. J. J. Zic. Time-constrained Buffer Specifications in CSP + T and Timed CSP. *ACM Transactions on Programming Languages and Systems*, 16(6):1661–1674, 1994.