A Programming Model for Heterogeneous CPS from the Physical POV with a Focus on Device Virtualization

Martin Richter, Christine Jakobs, Theresa Werner, Matthias Werner
Operating Systems Group
Chemnitz University of Technology
09111 Chemnitz, Germany
email: {martin.richter, christine.jakobs, theresa.werner, matthias.werner}@informatik.tu-chemnitz.de

***Abstract*—The emergence of cyber-physical systems leads to an integration of the digital and physical worlds through sensors and actuators. Programming such systems is error-prone and complex as a plethora of different devices is involved, each of which may be mobile and unreliable. In existing approaches, the developer views the world from the digital point of view. He or she has to implicitly interpret digital values as sensor measurements of the environment or as control values, which influence the environment through actuators. This leads to an increase of complexity as the number of sensors and actuators in cyber-physical systems is ever-increasing and different types of devices may become available during the runtime of the system. These devices may be located in different physical contexts that may bear no relation to each other. Therefore, the developer has to take the properties of the environment into account when designing his or her application as he or she has to acknowledge the impact of varying devices being located in different contexts. Additionally, he or she has to consider the coordination of interactions between different types of distributed sensors and actuators. This increases the likelihood of errors in the programmer's implicit interpretations of the digital values. Current approaches mainly focus on providing abstractions from the distribution and heterogeneity of the system, but fail to explicitly address the impact of digital calculations on the physical world and vice versa. We present a programming model, which reverses the view of the developer on the system. It allows him or her, to take the perspective of the physical system of interest and to explicitly describe its desired behavior. Therefore, a virtualization of devices is possible. This allows to transparently handle failing, moving, as well as emerging sensors and actuators.**

***Keywords*—cyber-physical systems; programming model; context awareness; heterogeneity; virtualization.**

I. Introduction

This paper expands on our work presented in [1]. There, a programming model for Cyber-Physical Systems (CPS) is introduced which allows the developer to take a physical point of view. The paper at hand provides a more detailed description of the model and further elaborates on the possibilities of introducing device virtualization, especially concerning sensors.

Sophisticated programming models as well as device virtualization play a crucial role in enabling the developer to create adaptable and robust applications for CPS. This proves especially valuable in the contexts of the Internet of Things [2], Smart Grid [3], automated warehouse logistics [4], and Industry 4.0 [5] as an increasing number of possibly unreliable devices are interconnected and have access to a multitude of different sensors and actuators in their environment. The emergence of such CPS leads to an integration of digital computations and the physical world. This entanglement raises multiple challenges, which do not exist in classical distributed systems [6]. Apart from being distributed over space, the different devices possess varying capabilities, regarding what they measure and how they influence their environment. Therefore, the developer has to ascertain that the employed sensors and actuators are able to achieve his or her goals with respect to affecting and monitoring the physical world. Additionally, the devices may be unreliable and mobile. They may therefore fail, move between different physical contexts of the system (e.g., between two different rooms within a house), or leave the system entirely. This leads to changes in the semantics of their abilities with respect to which parts of the physical world they observe or influence.

Current approaches leave the task of managing the continuously changing set of heterogeneous devices to the programmer. When compared to classical distributed systems, this leads to an impairment of the portability of applications and an unproportional increase in complexity for the design process. Applications for CPS are currently created via classic programming models, where the application implicitly converts sensor measurements to a digital representation of the physical phenomenon of interest (e.g., reading a value from a register of a sensor). Based on this digital representation, the programmer's application performs calculations of which the results are implicitly converted to impacts on the physical world (e.g., writing a value into a register of an actuator). This procedure further increases the difficulty of designing applications as digital values do not directly translate to observations of and influences on the physical world. Our goal is to relieve the programmer from having to work with this implicit conversion of distributed measurements of physical phenomena to a digital representation and subsequently the translation of digital computations to a variety of actuator influences on the physical environment.

This paper presents a programming model for reducing the complexity of application design for heterogeneous CPS. To achieve this, we provide the developer a new view on the system. We reverse the programmer's perspective, such that he or she no longer directly controls the devices through digital computations. Instead, he or she describes the properties of the physical system of interest and how these properties should

evolve over time to reach a desired target state. The developer is concerned with the CPS' effect on the environment (i.e., the desired state change) rather than the cause (i.e., the controlled actuators). This perspective is comparable to declarative programming where the properties of a solution are described rather than the procedure of achieving the solution.

As the programmer designs the application from the view of the physical system, he or she does not have to implicitly translate physical phenomena to digital representations and vice versa anymore. Rather, the Runtime Environment (RTE) transparently handles this conversion by utilizing sensor and actuator specifications in addition to the programmer's physical system and target state descriptions. As the RTE is able to decide which sets of sensors and actuators are suited for the tasks at hand, this approach enables the virtualization of sensors and actuators. Therefore, the developer does not have to account for failing or moving devices as the RTE is able to transparently evaluate and choose alternatives. To achieve this, the RTE maintains a location-aware digital representation of the physical system by interpreting sensor measurements.

Not all sensor measurements may be related to each other due to their positioning and environmental circumstances. Therefore, the RTE requires a description of environmental contexts to precisely identify locations of interest within the system that can be observed by available sets of sensors. Each environmental context refers to a region of space that restricts the interpretability of sensor measurements. For example, a picture taken by a camera in one room of a house may not be interpretable for other rooms of the house. The definitions of contexts allow the RTE to decide which sensor readings stand in relation to each other and for which locations they provide relevant information. Therefore, it is able to create a more precise and less error-prone digital representation of the physical environment while taking the motion of sensors and the resulting physical context changes into account.

The RTE uses the digital representation of the system to compute sufficient actuator inputs to reach a target system state. It achieves this by utilizing a constraint solver which takes the digital representation of the system and the programmer's target state description as inputs. Its computations provide a sufficient set of actuator inputs to reach the desired target state. Hence, our programming model abstracts from complex conversions between digital computations and physical phenomena. Moreover, it provides transparency to the developer with respect to changing device configurations (i.e., through motion, failure, or emergence). It is intended to be used in applications utilizing a variable set of arbitrary sensors and actuators to measure and influence physical systems with well-understood properties and dynamics.

From an Operating System (OS) perspective, the presented programming model enables the decoupling of the system and application programmers for CPS. Therefore, a system programmer is not bound to certain applications anymore. This allows him or her to provide required system functionalities to the RTE (e.g., device drivers) as well as commonly utilized abstractions to the application programmers (e.g., libraries for the specification of the physical system). Additionally, the application developer is not bound to hardware specifics anymore due to the employment of the RTE and unified interfaces. This makes the development of applications for CPS more robust as the reusability of code is enhanced and the portability of applications is improved. For the remainder of the article, we refer to the application developer as developer or programmer, and the system developer will be specifically labeled so.

This paper is structured as follows. Section II reviews the related work. Section III presents two running examples for illustrative purposes. Section IV depicts our system model. Section V describes the application programmer's view on the system. Section VI presents the RTE as a link between the programmer's specifications and the physical world (i.e., sensors, actuators, and physical objects). Section VII supplies a conclusion and an outlook for future work.

II. Related Work

A CPS incorporates the digital and the physical world. The configuration of such heterogeneous distributed systems may change at any point in time due to device failures and the emergence of new sensors or actuators. Under such circumstances, programming errors are easily introduced as current solutions rely on the developer to work out the physical semantics of the digital inputs and outputs of varying sets of devices.

Approaches like *Aggregate Computing* [7] focus on convergence. They enable the developer to write an application for a set of computational nodes situated in a given region. The computations of each node take place on the basis of its local state and its neighbors states. Therefore, the behaviors of the nodes in a region converge over time. Such approaches abstract from the distribution of the system. Nevertheless, they are only suited for homogeneous CPS since a converging node behavior implies that the devices possess similar capabilities.

Physical modeling languages like *Modelica* [8] or *Simulink* [9] enable the developer to describe the properties and the behavior of a physical system. These approaches are designed for the simulation of physical systems and for code generation purposes for non-distributed systems. Here, the developer explicitly handles the heterogeneity of the system. The main goal of physical modeling languages is to draw conclusions on the design of a system rather than controlling and observing it directly in a distributed fashion.

Approaches like *Regiment* [10], *Hovering Data Clouds* [11] or *Egocentric Programming* [12] provide mechanisms for the rule-based aggregation and dissemination of environmental data in a distributed CPS. The goal of these propositions is to monitor the environment, rather than to influence it. The programmer therefore has to utilize additional frameworks to describe the desired changes of the physical system state.

Other propositions like *Spatial Views* [13] or *Spatial Programming* [14] allow the programmer to control specifically, which part of the code is executed in which region within the system. These regions can be interpreted as environmental

contexts in which certain devices are situated. They do not take the impact of the physical regions of space on the capabilities of the devices into consideration. For example, multiple robots being located in different closed rooms may not be able to cooperate but if they are located on two hills located next to each other their camera measurements may be related to each other. The discussed programming models do not allow this differentiation. Regarding the control of devices, the developer statically specifies which types of sensors and actuators are required for the execution of the program for different spatial regions. Thus, the programmer cannot take changing types of devices into account.

The presented programming models tackle challenges like providing distribution transparency or managing heterogeneity. The programmer's main concern still is the management of digital data, which obstructs him or her from focusing on the main goal: influencing the physical environment. Our programming model reverses the developer's view on the system. He or she describes the properties and the desired behavior of the physical system from which the RTE deduces the required digital computations while managing a possibly changing set of heterogeneous devices. Additionally, we take the impact of sensors being located in different contexts into account. Our approach provides the programmer with the possibility of defining the relevant physical contexts for determining which sensor measurements are related.

III. Running Examples

This section presents two running examples that we will use for illustrative purposes: robot soccer and a street surveillance system. The former focuses on the control of actuators and their impact on the physical world while the latter is mainly concerned with observing the environment. This allows us to demonstrate the expressive power of our programming model in the domains of observing as well as influencing the physical environment of the CPS. The rest of the paper will show that both use cases can be adequately treated by the developer by utilizing the presented programming model.

A. Robot Soccer

The first running example consists of a set of robots that interact with a soccer ball. There are different sensors and actuators attached to each robot and the system consists of multiple physical objects of interest (i.e., the robots, the ball, and the goal). The robots possess different properties such as varying masses and different maximum velocities. Both characteristics influence the robots' capabilities to move and kick the ball. Naturally, the developer has to take the mobility of the objects into account as well. Therefore, this example offers all the system traits that are of interest to us when focusing on influencing the physical environment.

Figure 1 shows an exemplary team of three robots R_1, R_2, and R_3 as well as a ball B. Two cameras are installed on the robots R_1 and R_2 which are therefore mobile. Their fields of view are depicted as dashed lines. One static camera is located above the playing field. Its field of view is represented by the dotted line.

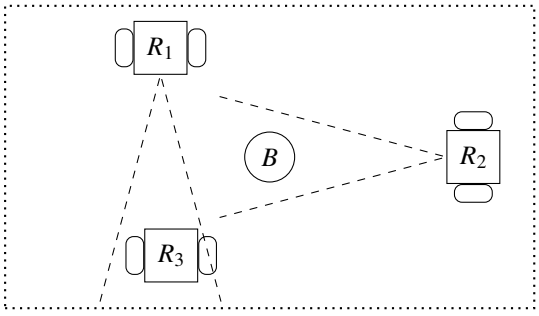


Figure 1. Robot soccer with three robots R_1, R_2, and R_3, as well as a ball B. R_1 and R_2 possess a camera and a stationary camera is installed above the playing field. The fields of view of the cameras are indicated by dashed and dotted lines.

B. Street Surveillance System

As a second running example, we utilize a street surveillance system. Its purpose is the identification of environmental hazards close to the road such as wildfires or burning cars. The system consists of multiple static cameras located at the roadside and dashboard cameras situated within the cars driving by. Additionally, cars possess temperature sensors measuring their interior that are located under the driver seat as well as temperature sensors measuring the exterior of the car situated behind the bumper bar. Each car has tinted windows in the back which do not allow outside cameras to observe the inside of a car. In contrast, cameras within the car are able to observe locations outside through these windows. This example offers all relevant features for identifying physical objects of interest via sensors that are constrained by varying environmental contexts (e.g., tinted windows).

Figure 2 shows an example of such a street surveillance system with two static cameras SC_1 and SC_2 on the roadside and one camera mounted on the dashboard of a car C. Similar to Figure 1 static cameras' fields of view are depicted as dotted lines and mobile fields of view are depicted as dashed lines.

IV. System Model

This section describes our system model. It is divided into modeling physical objects, sensors, and actuators.

A. Physical Objects

The programmer desires to influence a physical system through digital computations such that a certain goal is reached. A physical system Σ consists of a set of locations X_Σ in which a set of physical objects of interest O is situated. Each object $o \in O$ takes up a region of space $X_o \subseteq X_\Sigma$ and features a state $\vec{z}_o$ which comprises multiple properties $z_o^{(i)}$ (e.g., color and shape):

$$\vec{z}_o(t) = \left[z_o^{(1)}(t) \quad \dots \quad z_o^{(w)}(t) \right]^T \tag{1}$$

Each property is characterized by a type and a value, e.g., a ball's shape may be round and its color red.

B. Sensors

The RTE creates instances of such physical object properties by utilizing interpretation methods. They are provided by the system developer and interpret the available sensor measurements as described in [15]. The RTE instantiates a property by performing one or more interpretation methods on a set of suitable sensor outputs (e.g., performing image recognition on camera outputs to identify objects of a given shape). Each instance of a property is valid for a region in space. The extent of this region depends on multiple factors:

1) the method chosen for interpreting the sensor measurements (e.g., spatial interpolation for a set of temperature sensors or triangulation for multiple cameras),
2) the locations a sensor may be able to measure (e.g., a camera takes measurements of a cone in front of it and a temperature sensor measures a single location), and
3) the environmental contexts of the sensors as they restrict their measured locations (e.g., a camera within a closed room can not take measurements of locations outside the room).

The RTE requires a sensor capability model to decide which sensors can be utilized for the available interpretation methods to observe the desired physical object properties. This model is implemented by the system developer through a corresponding driver that provides the required information to the RTE. In our capability model, a sensor s is specified by the following five tuple.

$$s = (q, \mu, \vec{x}(t), \vec{p}(t), X(\vec{x}, \vec{p}, t)) \tag{2}$$

A sensor provides measurements of a physical quantity q for a region in space X. This region is dependent on the sensor's position $\vec{x}(t)$ and other sensor-specific parameters $\vec{p}(t)$ (e.g., the orientation and angle of view for a camera). Both parameters $\vec{x}$ and $\vec{p}$ may change over time due to changes in the configuration of the sensor and its mobility. A sensor provides its measurements in the form of a digital output signal v which it creates by performing a measurement process μ on its physical input.

$$v(t) = \mu(q, t) \tag{3}$$

An example of this is a camera measuring electromagnetic radiation in a cone in front of it which it transforms into a digital array of pixels.

A sensor belongs to a certain sensor class based on the physical quantity it observes and its measurement process. Based on these sensor classes, the RTE determines which sensor outputs are suitable for each interpretation method. The RTE may utilize multiple different interpretation methods $\vec{m}_z$ for instantiating a property z. For example, for determining the shape of an object it may utilize image recognition on cameras as well as methods for interpreting the output of laser scanners. This allows the conversion of various types of sensor outputs to a more holistic and precise digital representation of the physical object. Additionally, different sensors of the same class may replace each other when providing inputs for methods based on their failure or motion. The RTE chooses the set of methods for determining the properties of a physical object based on the available interpretation methods, the corresponding property of interest, and the currently accessible sensors.

Figure 3 gives an overview of the creation of the state vector of a physical object o, multiple sensors s_j, and multiple different interpretation methods $\vec{m}_{z_o^{(k)}}$ for each of the properties $z_o^{(k)}$ of the object o. Dotted lines refer to a mapping that may or may not be used. These mappings depend on the object's properties, the sensors' measurands, and the required method inputs.

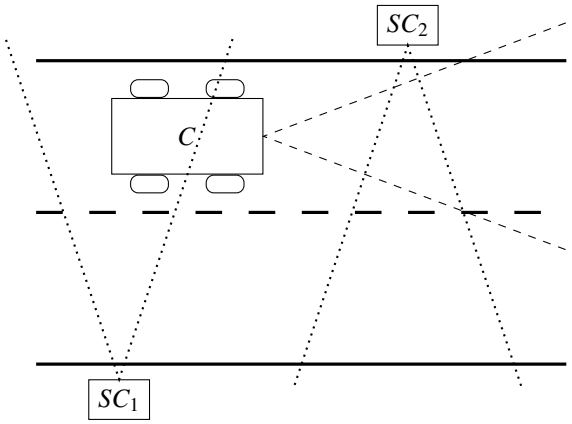


Figure 2. Street surveillance system with one car C that incorporates one interior temperature sensor, one exterior temperature sensor, and a dashboard camera. On the roadside two stationary cameras SC_1 and SC_2 are installed. The cameras' fields of view are indicated by dashed and dotted lines.

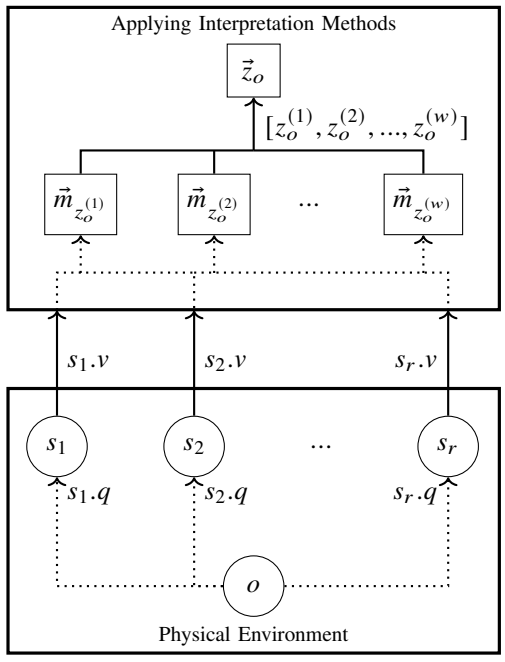


Figure 3. Creation of the state vector $\vec{z}_o$ of a physical object o.

Additionally, the RTE has to take the environmental contexts C of sensors into account as these restrict the locations $s.X$ for which a sensor s provides measurements. The contexts also allow the RTE to reason on whether sensor measurements stand in relation to each other. An example of this are camera measurements from different closed rooms that can not be utilized for a method like triangulation. In contrast, if the cameras are situated in the same room and their measurement regions overlap they may be used for such a method. Recognizing these relations allows the RTE to decide which sensor measurements may be used as inputs for an interpretation method. Therefore, the provision of environmental contexts is essential for determining the properties of physical objects of interest. An environmental context is a possibly mobile region in space that influences a physical quantity. The measurement regions of sensors that observe this physical quantity may be constrained by the corresponding contexts. This depends on whether the context is inward-blocking or outward-blocking:

- inward-blocking implies that sensors from the outside are not able to observe locations within the context, and
- outward-blocking refers to sensors located within the context not being able to observe locations outside of the context.

An example of this is a car with tinted windows. Cameras from the outside are not able to observe the inside of the car but the opposite is possible.

In conclusion, a context c is characterized by the following five tuple.

$$c = (\vec{x}(t), \vec{r}(t), X(\vec{x}, \vec{r}, t), q, \neg in, \neg out) \tag{4}$$

The vector $\vec{x}$ depicts the anchor location of the context which may change over time. The vector $\vec{r}$ relates to the orientation of the context which may also change over time. The spatial extent of the context is described by the set X. It depends on the context's anchor location and its orientation to be fit correctly into space. Additionally, the context influences a physical quantity q depending on the boolean attributes $\neg in$ (inward-blocking) and $\neg out$ (outward-blocking). A sensor is situated within a context if its location $s.\vec{x}$ is in the spatial region $c.X$ of the context c. As contexts may overlap, a sensor may be located in multiple contexts at once. A context c influences the spatial interpretability of the output of a sensor s if the context influences the same physical quantity the sensor measures (i.e., $s.q = c.q$), and either

1) the sensor is located within the context and the context is outward-blocking (i.e., $out_c^s = s.\vec{x} \in c.X \wedge \neg c.out$), or
2) the sensor is located outside the context and the context is inward-blocking (i.e., $in_c^s = s.\vec{x} \notin c.X \wedge \neg c.in$).

The set of all contexts $C^s \subseteq C$ that influence a sensor s can therefore be described by the following equation.

$$C^s = \{c \in C : (s.q = c.q) \wedge (out_c^s \vee in_c^s)\} \tag{5}$$

Based on these contexts C^s, a sensor's interpretable locations $s.X$ are narrowed down to the actually observed locations $s.X'$.

$$s.X' = s.X \cap \Big(\bigcap_{\substack{c_{out} \in C_s \\ out_c^s = true}} c_{out}.X \Big) \setminus \Big(\bigcup_{\substack{c_{in} \in C_s \\ in_c^s = true}} c_{in}.X \Big) \tag{6}$$

Therefore, the sensors' outputs are exclusively valid for these locations. The results of interpretation methods are also only valid for certain locations. This depends on the used sensors (i.e., their observed locations $s.X'$) and the methods themselves. For example, interpolation of temperature sensor readings increases the size of the set of valid locations and triangulation via cameras decreases it. For the valid locations of method results the RTE evaluates whether given physical object properties are present and if so instantiates corresponding state vectors for the physical objects of interest.

In conclusion, based on the presented information regarding sensor capabilities, their environmental contexts, and the utilized interpretation methods, the RTE is able to determine regions of space in which physical objects of interest are present (i.e., X_o for each object o). Subsequently, it is able to instantiate their state representations $\vec{z}_o$.

C. Actuators and Internal Object Dynamics

The properties of a physical object may change over time, which leads to a change of its state $\vec{z}'_o$. This change of state can be caused by internal dynamics (e.g., a rolling ball) or external influences $\vec{u}_o$ due to actuator actions (e.g., a ball being kicked). The change of state at each point in time is a function f of the object's state and the corresponding external influences.

$$\vec{z}'_o(t) = f(\vec{z}_o, \vec{u}_o, t) \tag{7}$$

An actuator takes a digital signal as input and transforms it into one or more actions that affect their environment. These actions have measurable impacts on the properties of physical objects. For example, a gripper arm performs the action of grabbing an object. This action can be measured as a force (in Newton) acting on the object from two directions. The set of all actuators makes up the output interface of the CPS. The external influences $\vec{u}_\Sigma$ on the physical system of interest are the concatenation of the external influences on the different physical objects.

$$\vec{u}_\Sigma(t) = \begin{bmatrix} \vec{u}_{o_1}(t) & \dots & \vec{u}_{o_m}(t) \end{bmatrix}^T \tag{8}$$

The state $\vec{z}_\Sigma$ of the system is a concatenation of the different physical object states $\vec{z}_{o_i}$.

$$\vec{z}_\Sigma(t) = \begin{bmatrix} \vec{z}_{o_1}(t) & \dots & \vec{z}_{o_m}(t) \end{bmatrix}^T \tag{9}$$

The change of the state of the physical system $\vec{z}'_\Sigma$ depends on the internal dynamics of the physical objects that populate the system and their external influences. The function f_Σ describes the system's state change.

$$\vec{z}'_\Sigma(t) = f_\Sigma(\vec{z}_\Sigma, \vec{u}_\Sigma, t) \tag{10}$$

We regard actuator actions as external influences on physical objects (e.g., a robot kicking a ball). This stands in contrast

to viewing any interactions between arbitrary physical objects as external influences (e.g., a ball rolling against another ball). Considering all possible interactions between any physical objects would lead to an explosion in complexity, as there may be an arbitrary number of specified and unspecified physical objects. Instead, we treat interactions between objects as disturbances, which may or may not require countermeasures by the CPS.

V. The Application Programmer's View

In our programming model, the developer views the system from the standpoint of physics. He or she provides specifications for the objects that populate the physical system. These specifications encompass information on the properties of the physical objects (i.e., their state) and a definition of their behavior, based on internal dynamics and external influences. Additionally, the developer provides environmental context descriptions that restrict the capability of sensors to observe their environment. These descriptions allow the RTE to decide which sensor measurements stand in relation to each other and for which regions the measurements are valid. This enables the RTE to decide in which regions the given physical object properties are present, based on the available sensors, the utilized interpretation methods, and the relevant context regions. The RTE requires all of these specifications to determine which sensors are necessary for observing the physical objects and how the objects react to given actuator inputs.

For the RTE to decide which actions have to be taken by the actuators to reach a target state, a target state description is necessary. This description refers to the whole physical system rather than a single physical object, as relative relationships between physical objects may be of interest to the programmer. The target state description spans a state space, because different states may satisfy the goal of the developer. Table I summarizes the described requirements for the functionality of the RTE and for which of its actions they are necessary.

TABLE I. REQUIREMENTS FOR RTE ACTIONS.

ID	Specification Requirement	RTE Actions
Req.1	Physical objects' properties	Recognizing objects and comparing the current system state with the target state space.
Req.2	Physical objects' internal dynamics	Estimating when objects reach the target state through internal dynamics.
Req.3	Physical objects' reactions to external influences	Estimating when objects reach the target state through external influences.
Req.4	Target state description	Calculating actuator actions to reach a target system state.
Req.5	Environmental context specifications	Determining which sensor measurements are related and localizing objects.

A. Physical Object Specification

The physical system consists of possibly multiple physical objects of interest, each of which possesses a designated state and behavior. Hence, the object-oriented programming paradigm fits the described requirements and system model. A class enables the developer to specify attributes (state) and methods (change of state) of a physical object. From such a class, the RTE creates a digital representation of a physical object whenever it recognizes the corresponding properties of the described object in the environment. If the RTE recognizes multiple objects of the same class, multiple instances are created. As a physical system may encompass a variety of physical objects, the programmer may have to provide multiple different class specifications.

Through inheritance, a class may extend the state and behavioral descriptions of other classes. This simplifies the specification of different types of objects that partially share their state and behavior descriptions. For example, a car and a ball both possess the properties of moving objects (i.e., position, velocity, and acceleration) and they also have similar internal dynamics in the sense that their position changes with their velocity and their velocity changes with their acceleration. The specific differences in the behavior and properties of balls and cars are then described in their specific classes respectively, e.g., how external influences affect their positions, velocities, and accelerations. Figure 4 shows an example of a ball that extends the class of a moving object.

```
Class MovingObject extends PhysicalObject {
    MovingObject() {
        this.p = Position(m) : true;
        this.v = Velocity(m/s) : true;
        this.a = Acceleration(m/s^2) : true;
    }
    motion(ElapsedTime delta) {
        this.v = this.a + delta * this.a;
        this.p = this.p + delta * this.v;
    }
}

Class Ball extends MovingObject {
    Constructor Ball() {
        this.s = Shape : sphere(radius==30cm);
        this.m = Mass : mass==0.3kg;
    }
    Requirement(Act(v) == Act(m) AND
                Act(v).position == this.p)
    kick(Velocity v, Mass m) {
        this.v = 1/(this.m + m) * (this.m * this.v +
             m * v + m * 0.8 (v - this.v));
    }
}
```

Figure 4. Example for physical object specifications.

The programmer provides the state description of a physical object of interest by providing a set of tuples (τ_i, r_i) where τ_i is the type of a property (e.g., a shape) and r_i is a rule for further specifying the characteristics of the object (e.g., its shape has to be a sphere with a radius of 30 centimeters). Based on the type τ_i, the RTE determines which interpretation methods to

utilize for evaluating the available sensor measurements (see Section IV). The RTE continuously assesses the results of these methods with respect to whether the corresponding rule r_i is satisfied. If it is satisfied, the object attribute is instantiated with the methods' result. The logical representation of the object's state vector is the instantiation of all its attributes.

For a programmer to declare such attributes comfortably, the system developers provide libraries that supply definitions of commonly used attribute types including their domains (e.g., the type `Shape` which encompasses values like `sphere` and `cube`). The values of a type may be further constrained depending on their characteristics (e.g., the shape `sphere` can be additionally characterized by its radius). In Figure 4, a ball is defined as a spherical object with a radius of exactly 30 centimeters. Its position, velocity, and acceleration are not specific to the object and therefore the corresponding rules always evaluate to `true`.

The methods of a class describe the state change induced by the physical object's internal dynamics and its reactions to external influences. Each method has access to the object's state and describes a change of its state. A method's calculations may depend on parameters that represent inputs from actuators. They affect the digital object's state and correspond to external influences on the physical object. For example, in Figure 4, the method `kick` takes the actuator's velocity and its mass as parameters, which influence the velocity of the ball after the impact.

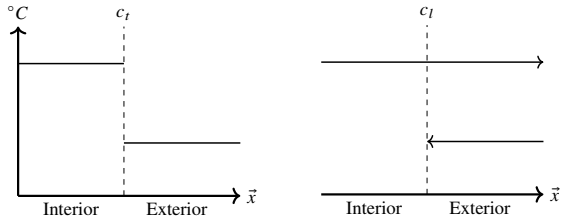
Multiple actuators may provide the inputs to an object's method. This enables the RTE to coordinate a variety of actuator actions for better efficiency or to supply inputs, which a single actuator may not be able to provide. For example, if a building component has to be clamped, two forces on opposite sides of the component have to be at work toward its center. From a result perspective, it does not matter whether this is accomplished by one single actuator or two independent actuators.

Depending on the properties of the physical object and the method parameters, the programmer may have to specify requirements for the inputs from actuators. For example, an actuator has to be close to a component to exert a force on it. The actuator requirements may incorporate the following information:

- the origin of the inputs to the method (e.g., they have to be provided by the same actuator),
- the actuators' states (e.g., their positions), and
- the object's state (e.g., its position).

This allows the RTE to choose actuators capable of influencing the given object and achieving the desired results.

Figure 4 depicts two methods for the classes `MovingObject` and `Ball`. Method `motion` describes the change of position and velocity, based on the object's velocity and acceleration, respectively. The `delta` parameter stands for the elapsed time between two evaluations of the method. Method `kick` takes two parameters `v` and `m`, which correspond to an actuator's velocity and mass, respectively. For this method, requirements for the actuator inputs are given. They specify that both mass and velocity have to be provided by the same actuator and that the actuator has to be situated at the position of the ball. The method calculates the approximate velocity of a ball after being kicked by an actuator with a coefficient of restitution of 0.8.



(a) Discontinuity in the interpretation of the output of a temperature sensor within a car by a context c_t.

(b) Discontinuity in the interpretation of the output of a camera sensor caused by a context c_l.

Figure 5. Discontinuities in interpretations of sensor measurements caused by contexts.

B. Environmental Context Specification

The RTE localizes physical objects within the system by employing interpretation methods on sensor measurements and inspecting whether the given object properties are present and the corresponding rules are fulfilled. To achieve this, the RTE requires a description of environmental contexts within the system (see Section IV). An environmental context restricts the regions for which a sensor provides measurements. As already mentioned, a context consists of an anchor location, a spatial extent, an influenced physical quantity, and an annotation of whether it is inward- and/or outward-blocking. Each context may either have a static anchor location or be bound to a physical object of interest. The latter option may lead to the context being mobile.

The borders of a context exist wherever discontinuities arise in the interpretation of a sensor measurement between the sensor's location and the location for which the measurement is interpreted. Figure 5a exemplifies this for a temperature sensor located within a car. From its temperature readings no inferences can be made on the ambient temperature outside the car and the readings of interior as well as exterior sensors do not stand in any relation to each other. Something similar can be observed when considering light passing through a tinted window. In this case, the discontinuity is only present if the light passes through the window in one direction (see Figure 5b). The programmer has to consider these discontinuities to determine for which regions in space he or she has to define contexts. The RTE offers a module to support the developer in this regard. It infers the possibly utilized interpretation methods from the object property descriptions of the programmer. The methods allow the RTE to reason which sensor classes may be required to observe these properties. Based on this, it is able to provide the programmer the sets of relevant physical quantities. Subsequently, the developer is able to examine the system space for corresponding contexts that influence these physical quantities.

When specifying a context, the programmer utilizes a similar paradigm when compared to specifying a physical object. He or she creates a class that describes the required parameters for a context, i.e., its anchor location, orientation, spatial extent, influenced physical quantity, and whether it is inward- and/or outward-blocking. Figure 6 shows an example of the specification of a context for the temperature within a car. The context is inward- and outward-blocking as there is no relation between temperature readings from the interior and the exterior of the car. The position and orientation of the car are determined by the RTE through available positioning systems (e.g., GPS and/or camera triangulation). The developer provides the definition of the spatial extent of the context in the form of a CAD model to ensure sufficient precision. In most cases, such models are in use already such that the programmer is able to utilize them. Otherwise, he or she has to utilize modeling tools to create the definition of the spatial extent.

C. Target State Description

The preceding sections depicted how the developer specifies physical objects and environmental contexts such that the RTE is able to localize objects of interest. To infer required actuator actions the RTE requires a target state description. We chose a variant of declarative programming, i.e., constraint logic programming [16], as a programming interface for describing the target state. It allows the developer to specify the properties of a solution to a problem rather than how to reach the solution. This fits our requirements, as the developer describes a desired physical system state and the RTE deduces the sufficient actuator inputs to the physical system. This approach abstracts from the individual actuators. Therefore, the developer is able to focus on the impact of the actuators' actions on the individual physical objects rather than controlling individual devices.

Since relations between the different states of physical objects may be of interest, the programmer defines target states based on the overall system state. To analyze whether a target state is reached and whether the system state develops correctly, the RTE evaluates the set of constraints periodically.

Figure 7 shows an example of a defending constraint for a game of robot soccer. The target state refers to the positions of the players of the own team with respect to the ball, goal, and opposing player positions:

- all players of the own team should be close to an opposing player (i.e., closer than one meter),
- there should always be one player between the ball and the own goal, and
- there should always be one player between any opposing player and the ball.

This positioning allows to intercept the ball and prevents undisturbed passes as well as attempts of the opponent to score. To reach this objective, the RTE has to coordinate the available actuators, such that the physical properties of the robots (i.e., their positions and velocities) are changed accordingly.

```
Context CarInteriorTemp {
    this.physicalQuantity = Temperature;
    this.anchorLocation = Car.position;
    this.orientation = Car.orientation;
    this.spatialExtent = fitCAD(
        this.anchorLocation,
        this.orientation,
        carInteriorCAD
    );
    this.inwardBlocking = true;
    this.outwardBlocking = true;
}
```

Figure 6. Example for a context specification for the interior of a car.

```
Defense@OpponentOffense {
  double k, l;

  ∀player, opp ∈ MyPlayer × OppPlayer:
    distance(player.p, opp.p) <= 1.0[m];

  ∃player ∈ MyPlayer, ∀goal ∈ MyGoal, ∀ball ∈ Ball:
    goal.p + k * (goal.p - ball.p) == player.p;

  ∀ball ∈ Ball, ∀opp ∈ OppPlayer, ∃player ∈ MyPlayer:
    ball.p + l * (ball.p - opp.p) == player.p;
}
```

Figure 7. Examples of defensive positioning in robot soccer.

D. Application Context Specification

Depending on the present physical objects and their configuration (e.g., their positioning), the developer may desire to provide different target state descriptions. For example, in robot soccer, the programmer's team has to defend if the opponents possess the ball. Vice versa, if the developer's team possesses the ball it should attack to score a goal. Application contexts allow the programmer to describe such relations between physical objects of interest. They form constraints that allow the RTE to decide which target state description to follow at each point in time. Therefore, each target state description has to be bound to an application context (see Figure 7). Similarly to the target state description, the developer utilizes constraints to describe application contexts. Figure 8 shows an example of an application context that describes when defensive player behavior should be adopted in robot soccer. This occurs whenever an opponent possesses the ball, the ball moves toward the programmer team's goal, and an opponent moves toward the programmer team's goal.

VI. Runtime Environment

The RTE maintains a set of physical object descriptions provided by the programmer. It continuously utilizes interpretation methods to evaluate sensor measurements of the CPS environment to determine the state of the physical objects populating the physical system. Moreover, the RTE continuously evaluates the constraint system for the target state and

application context specifications. In its evaluations, the RTE takes into account the actuator requirements in addition to the available actuators since they narrow down the possibilities for the available physical inputs.

Figure 9 depicts the architecture of the RTE. It consists of four modules, which are executed in a distributed fashion: the interpreter, the observer, the controller, and the constraint solver. Furthermore, it facilitates drivers for sensors and actuators. They provide an interface for utilizing the devices and supply information on their state (e.g., their positions and orientation) to the other modules. The following paragraphs describe the functionalities of the RTE.

A. Interpreter

The interpreter offers an interface to the programmer for registering class specifications for physical objects, target state constraints, as well as environmental and application context specifications. It extracts three basic types of information from the class descriptions for physical objects:

(i) the state description of the physical object (i.e., what the properties of the object are and how it differs from other objects),
(ii) the behavioral description (i.e., how the object's state changes, based on internal dynamics and external influences), and
(iii) the actuator requirements for providing input signals (i.e., what conditions have to be met for an actuator to be able to supply a desired input to the physical system).

The interpreter creates a vector of state variables $\vec{z}_c$ from the state description of a given class c. Each state variable stands for a physical object property (i.e., a set of results from interpretation methods performed on sensor measurements). The state variables are utilized in the state equation of the physical object. This equation is created from the set of methods G_c belonging to the given class c. Each method $g \in G_c$ describes a change of state $\vec{z}'_{c,g}$ for an object of the given class. Such a method's parameters correspond to external influences caused by actuators. The interpreter converts them to a vector of input variables $\vec{u}_g$ for the method g. Each method describes a change of state, which depends on the state of the object, the specified internal dynamics, and reactions to external influences. The function f_g describes this change of state.

$$\vec{z}'_{c,g}(t) = f_g(\vec{z}_c, \vec{u}_g, t) \tag{11}$$

```
OpponentOffense {
  ∃opp ∈ OppPlayer, ∀ball ∈ Ball, ∀goal ∈ MyGoal :
    distance(ball.p, opp.p) <= 0.5[m];
    angle(ball.v, goal.p − ball.p) < 90[deg];
    angle(ball.v, goal.p − ball.p) > 180[deg];
    angle(opp.v, goal.p − opp.p) < 90[deg];
    angle(opp.v, goal.p − opp.p) > 180[deg];
}
```

Figure 8. Examples of an application context for defensive player behavior in robot soccer.

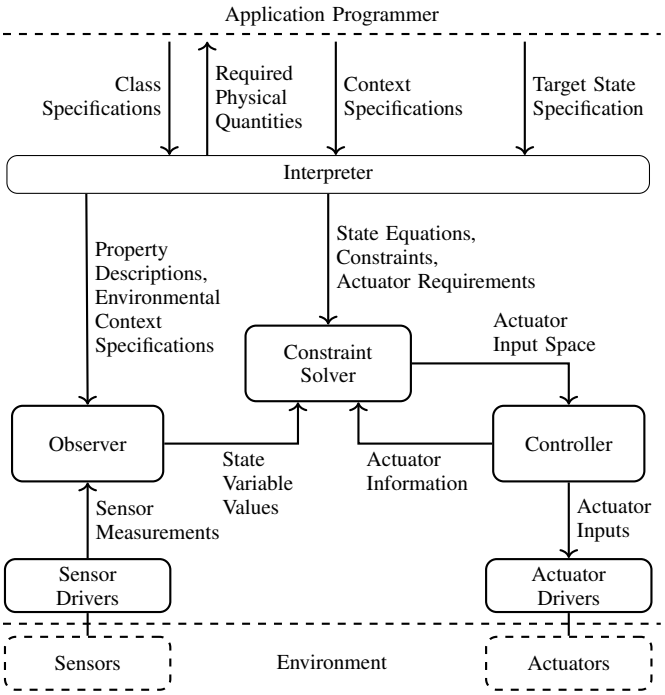


Figure 9. Architecture of the Runtime Environment.

If the function f_g is linear or linearized, the equation can be rewritten as a system of first-order differential equations.

$$\vec{z}'_{c,g}(t) = A_g \vec{z}_c(t) + B_g \vec{u}_g(t) \tag{12}$$

If the overall behavior of the class is linear or linearized, its state change can be described by the sum of all the methods' state changes, as the principle of superposition holds.

$$\vec{z}'_c(t) = \sum_{g \in G_c} \vec{z}'_{c,g}(t) = \sum_{g \in G_c} (A_g \vec{z}_c(t) + B_g \vec{u}_g(t)) \tag{13}$$

Since the constraint solver evaluates the constraints periodically in discrete steps, the interpreter converts the described equation into a time-discrete variant.

$$\vec{z}_c(k+1) = \sum_{g \in G_c} (A_g \vec{z}_c(k) + B_g \vec{u}_g(k)) \tag{14}$$

As the developer may provide multiple classes $c^{(i)}$ for different physical objects of interest, the interpreter creates such a state equation for each of the classes.

Figure 9 includes a depiction of the outputs of the interpreter module. For each method of a class, the interpreter creates constraints from the actuator requirements that are forwarded to the constraint solver. These constraints allow allocating the available actuator inputs to the corresponding input variables. For example, a mobile robot with a specified mass is able to provide the corresponding inputs to the `kick`-method in Figure 4. Additionally, the target state descriptions are added to the constraint system for determining the required actuator actions to reach the programmer's goals (e.g., moving soccer robots such that a defensive positioning is achieved, as described in Figure 7). The interpreter also forwards the

application context specifications to the constraint solver. This allows the solver to decide which target state descriptions are in effect based on the current system state (e.g., choosing the defensive positioning constraints when the opponent is attacking, as described in Section V-D). From the given system of equations, requirements, and constraints, the constraint solver is able to compute a sequence of actuator inputs, which lead to a desired target state.

Furthermore, the interpreter module provides an interface to the programmer that supplies information on which physical quantities are possibly measured to observe the desired physical object properties. To achieve this, the interpreter utilizes information on the available interpretation methods in the system and the state descriptions of the physical objects. From that, the interpreter deduces which physical quantities have to be measured by the required sensor classes to perform each interpretation method for determining the different properties. The interpreter's information on which physical quantities may have to be measured by the system sensors allows the developer to examine the system space for contexts of interest that influence these physical quantities (see Section IV-B). Therefore, the programmer is able to precisely identify such contexts and create the corresponding specifications. He or she subsequently provides these specifications to the interpreter which forwards them to the observer module.

B. Observer

The observer module creates and maintains a digital representation of the state of the physical system. It gathers measurements from the available sensors of the system, similar to the data aggregation and dissemination process described in [11]. This allows for gathering and distributing data of the system based on given rules (i.e., according to the object specification).

These rules can be interpreted as the programmer's specifications for the properties of physical objects of interest. Sensor measurements may not always directly relate to these property descriptions, e.g., the shape of an object is not directly measured by a camera but can be deducted from its output pixel array. The observer achieves such deductions by executing appropriate interpretation methods (see Section IV). Therefore, it utilizes a dictionary of the available interpretation methods within the system. This dictionary maps object properties to the available interpretation methods that are able to measure the corresponding properties. Each interpretation method encompasses information on its required input types. These types refer to the sensor classes that provide corresponding outputs (e.g., the method of image recognition requires the outputs of cameras). The output of the methods themselves corresponds to the requested physical object property and may be valid for a region of space (instead of a single location). This depends on the chosen sensors (i.e., which regions of space they are measuring) and the method itself as it may enlarge or shrink the regions for which the sensor measurements are valid. An example of this is interpolation on temperature sensors. A temperature sensor measures the temperature for a single location in space. When a method like interpolation is performed on a set of temperature sensors, the result of the method is valid for a larger region in space. In contrast, when executing triangulation on a set of cameras the resulting region is smaller than the cameras' measured regions.

When considering the regions for which sensor measurements are valid the observer has to consider the environmental context specifications provided by the programmer. They restrict the interpretability of sensor measurements and allow to reason on which sensor measurements are related (i.e., if their context regions overlap, see [15]). For example, an interpolation between temperature sensors measuring the interior and exterior of a car is not feasible. Therefore, these context descriptions are necessary for the observer module to determine which groups of sensors are suitable as inputs for interpretation methods.

The observer has to consider the regions for which the results of interpretation methods are valid for identifying physical objects of interest. The state of an object can be seen as a set of properties (i.e., attributes) and rules that have to be fulfilled for the object to be present at a given location. the observer assesses whether a rule is fulfilled or not by analyzing the outputs of the corresponding interpretation methods. If the rules for all object properties are fulfilled for a region, the object is present there and the observer instantiates the corresponding state vector. The module may create multiple object instances if there are distinct regions in space in which objects are located. It creates a state vector for each of the regions as it interprets every region as a separate object for each of which the constraint solver maintains a state equation. The observer maintains all the created object instances by updating their states. These updates are applied whenever new results of the corresponding interpretation methods are available. After an update, the module forwards the new state vectors to the constraint system.

C. Constraint Solver

The constraint solver computes a set of sufficient actuator inputs to reach a state of the target state space. As inputs, it takes the system state equation, the measured current state, the actuator requirements, and information about the currently available actuators. Through the actuator requirements, the constraint solver is able to decide which actuators are able to influence the perceived physical objects. By this, location-aware control of the actuators is enabled without the programmer having to explicitly examine individual devices. Through the actuators' influence on the physical world and the internal dynamics of the objects, object states evolve over time. The solver evaluates the constraints periodically to check whether a target state is reached and to update the set of actuator inputs to react to environmental influences (i.e., disturbances) accordingly.

Mathematically, the constraint solver's task is to find a state trajectory for the system state. The trajectory depends on a set of actuator inputs between the current point in time t_0 and

a chosen point in time t_1 before a deadline d, such that all constraints hold for the state at time t_1.

$$\vec{z}_\Sigma(t_1) = \vec{z}_\Sigma(t_0) + \int_{t_0}^{t_1} \vec{z}'_\Sigma(t)dt,\ t_1 \leq d \tag{15}$$

This allows to create a sequence of actions such that a target state is reached according to the programmer's intentions.

D. Controller

The controller module manages the set of available actuators. For each actuator, the controller module maintains information about the actuator's state (e.g., its position, orientation, reach, etc.) and which inputs it is able to provide. It offers this information to the constraint solver whenever an evaluation round starts. This enables the constraint solver to evaluate the actuator requirements for determining which actuators are able to provide the desired inputs.

The controller module uses the constraint solver's results (i.e., a sequence of sufficient inputs to reach a target state) and distributes it to the corresponding actuators. As the module is executed in a distributed fashion, a consistent view of the available actuators and their information has to be maintained and a consensus for distributing the required inputs has to be found.

VII. Conclusion and Future Work

The presented programming model allows the developer to focus on the description of a physical system and its target state. It allows him or her to specify explicitly what a desired state for a physical system is and how this state changes, based on possible actuator inputs and internal dynamics. This abstracts from the need to manage a changing set of actuators and sensors directly, as the information required of the programmer is reduced to defining the influences of actuators on the system and specifying properties of physical objects. Furthermore, the developer has to specify the physical contexts in which the system sensors may reside. This is necessary, as not all sensor measurements may relate to each other due to their capabilities, positioning, and spatial limitations. This necessitates a precise knowledge of the physical system and its characteristics by the programmer. In return, our programming model achieves virtualization of devices such that it provides distribution, motion, and location transparency to the developer.

To facilitate this, we present a RTE that links the programmer's view to the physical devices. It encompasses an interpreter, an observer module, a controller module, and a constraint solver. The observer module maintains a digital representation of the physical system state, based on the physical object descriptions. Additionally, it manages the available sensors, including information on their capabilities such that it is able to derive which sensors are able to observe physical objects of interest. The interpreter translates the programmer's system specification to a set of constraints and equations such that the constraint solver is able to utilize them. The constraint solver derives target states and required actuator inputs for the physical system from the programmer's specification and the current state of the system. The constraint solver's results are passed to the controller module. It utilizes this data to control the corresponding actuators in order to reach a target state. In conjunction, this allows the RTE the location-aware management of devices and physical objects of interest without the developer being directly involved.

Therefore, the presented programming model and RTE abstract from implicit conversions between digital computations and physical phenomena, which leads to the physical semantics of the program being made explicit. They are less complicated to understand and errors in the translation between digital and physical quantities are prevented. Additionally, the RTE transparently handles changing sets of devices as the programmer is concerned with the influences on the physical system of interest rather than their cause.

For future work, we intend to provide a formal description of actuator specifications, which allows deducing their properties with regard to how they influence their environment. Further research will be focused on describing the interactions between arbitrary physical objects, which are currently viewed as disturbances. To test the described approach, we will create a prototypical implementation of the RTE. In this regard, efficient data structures are necessary for the management of physical context information, sensor and actuator capabilities, as well as digital representations of physical objects. Furthermore, we intend to provide verifications of the real-time capabilities of the RTE. Additional research will concentrate on implementing consensus and consistency algorithms for the RTE, as a consistent view of the environment and optimal utilization of the devices have to be ensured.

References

[1] M. Richter, T. Werner, and M. Werner, "A Programming Model for Heterogeneous CPS from the Physical Point of View," in *The Sixteenth International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*, 2022, pp. 1–6.

[2] L. D. Xu, W. He, and S. Li, "Internet of things in industries: A survey," *IEEE Trans. on Industrial Informatics*, vol. 10, no. 4, pp. 2233–2243, 2014.

[3] X. Yu and Y. Xue, "Smart grids: A cyber–physical systems perspective," in *Proc. of the IEEE*, vol. 104, 2016, pp. 1058–1070.

[4] F. Basile, P. Chiacchio, and J. Coppola, "A cyber-physical view of automated warehouse systems," in *2016 IEEE Int. Conf. on Automat. Science and Eng. (CASE)*, 2016, pp. 407–412.

[5] N. Jazdi, "Cyber Physical Systems in the Context of Industry 4.0," in *IEEE Int. Conf. on Automat., Quality and Testing, Robotics*, 2014, pp. 103–105.

[6] E. A. Lee, "Cyber physical systems: Design challenges," in *11th IEEE symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, 2008, pp. 363–369.

[7] M. Viroli *et al.*, "From distributed coordination to field calculus and aggregate computing," *Journal of Logical and Algebraic Methods in Programming*, vol. 109, no. 100486, pp. 1–29, 2019.

[8] P. Fritzson, *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3*, 2nd ed., 2014.

[9] E. Hossain, *MATLAB and Simulink Crash Course for Engineers*, 2022, ch. Introduction to Simulink, pp. 317–359.

[10] R. Newton, G. Morrisett, and M. Welsh, "The regiment macroprogramming system," in *2007 6th Int. Symp. on Inf. Process. in Sensor Networks*, 2007, pp. 489–498.

[11] S. Ebers *et al.*, "Hovering data clouds for organic computing," in *Organic Comput. — A Paradigm Shift for Complex Syst.*, 2011, pp. 221–234.

[12] C. Julien and G.-C. Roman, "Egocentric context-aware programming in ad hoc mobile environments," in *Proc. of the 10th ACM SIGSOFT Symp. on Found.s of Softw. Eng.*, 2002, pp. 21–30.

[13] Y. Ni, U. Kremer, and L. Iftode, "Spatial views: Space-aware programming for networks of embedded systems," in *Lang.s and Compilers for Parallel Comput.*, 2004, pp. 258–272.

[14] C. Borcea, C. Intanagonwiwat, P. Kang, U. Kremer, and L. Iftode, "Spatial programming using smart messages: design and implementation," in *24th Int. Conf. on Distrib. Comput. Syst.*, 2004, pp. 690–699.

[15] M. Richter, C. Jakobs, T. Werner, and M. Werner, "Using Environmental Contexts to Model Restrictions on Sensor Capabilities," in *The Seventeenth International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*, 2023, to be published.

[16] J. Jaffar and M. J. Maher, "Constraint logic programming: A survey," in *The Journal of Logic Programming*, 1994, pp. 503–581.