A Programming Model for Heterogeneous CPS from the Physical Point of View

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

***Abstract*—The emergence of Cyber-Physical Systems leads to an integration of the digital and physical world through sensors and actuators. Programming such systems is error-prone and complex as a plethora of changing heterogeneous devices is involved. 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. Additionally, the interactions between different types of distributed sensors and actuators have to be coordinated, which 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. 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.**

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

I. Introduction

Because of emerging trends like the Internet of Things [1], Smart Grid [2], automated warehouse logistics [3], and Industry 4.0 [4] an increasing number of devices are interconnected and have access to a multitude of different sensors and actuators in their environment. The emergence of such Cyber-Physical Systems (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 [5]. Apart from being distributed over space, each of the devices may be connected to a multitude of sensors and actuators. They possess varying capabilities, regarding what they measure and how they influence their environment. As the main goal of the developer is to monitor and control a physical system, he or she has to consider these capabilities when designing his or her application. In classic programming models for CPS, the developer 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 increases the difficulty of designing applications. The semantics of controlling actuators and interpreting sensor readings are not always clear with respect to their influence on the physical and digital world, respectively. Our goal is to relieve the programmer from having to convert distributed measurements of physical phenomena to a digital representation and subsequently having to translate digital computations to a variety of actuator influences on the physical environment.

This paper presents a programming model for reducing the complexity of designing applications 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 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). 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) handles this conversion transparently by utilizing sensor and actuator specifications in addition to the programmer's physical system and target state descriptions. The RTE maintains a digital representation of the physical system by interpreting sensor measurements. Additionally, it takes advantage of a constraint solver to compute sufficient actuator inputs to reach a target state. These computations are based on the programmer's specification and the digital representation of the system. The RTE chooses a sufficient set of actuators and sensors at each point in time, based on the required physical inputs and outputs to control and observe the system. 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. 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.

As a running example we use a set of robots interacting with a soccer ball. This example offers all the system traits that are of interest to us. There are different sensors and actuators attached to each robot and the system consists of multiple physical objects of interest (i.e., the robots and the ball).

This paper is structured as follows. Section II reviews the related work. Section III describes the concept of our

 ISBN: 978-1-61208-989-8

approach. It depicts the programmer's view on the system and the functionality of the RTE. Section IV 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 types of sensors or actuators. Under such circumstances, programming errors are easily introduced as current solutions rely on the developer to handle the interpretation of physical phenomena through digital computations.

Approaches like *Aggregate Computing* [6] 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.

Other propositions like *Spatial Views* [7] or *Spatial Programming* [8] allow the programmer to control specifically, which part of the code is executed in which region. Additionally, the developer statically specifies, which sensors and actuators are required for the execution of the corresponding parts of the program. Thus, it is not possible for the programmer to take changing types of sensors and actuators into account.

Physical modeling languages like *Modelica* [9] or *Simulink* [10] 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* [11], *Hovering Data Clouds* [12] or *Egocentric Programming* [13] 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.

The presented solutions 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 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.

III. Concept

Our programming model provides means for the developer to define the properties of a physical system of interest, such that the RTE is able to transparently create a digital representation of it. Apart from that, the programmer is able to specify desired target states for the physical system. This allows the RTE to deduce required actuator actions to cause an appropriate state change. The following sections present a system model to introduce our definition of a physical system. Subsequently, we introduce the developer's view on the system, and an execution model for the RTE.

A. System Model

The programmer desires to influence a physical system through digital computations, such that a certain goal is reached. A physical system Σ is part of the environment and consists of a set of physical objects O. Each object $o \in O$ features a state $\vec{s}_o$, which comprises of multiple properties s_i:

$$\vec{s}_o(t) = \begin{bmatrix} s_1(t) & s_2(t) & \dots & s_n(t) \end{bmatrix}^T \tag{1}$$

Each property s_i is an interpretation g_i of one or more sensor measurements $\vec{m}$ of the environment at each point in time t:

$$s_i(t) = g_i(\vec{m}, t) \tag{2}$$

A measurement comprises of a value in a unit, which can be represented by a combination of possibly multiple SI base units. The set of all sensors makes up the input interface of the CPS. They allow the CPS to recognize physical objects through measurements of the environment. Subsequently, the RTE interprets these measurements to derive the objects' properties. For example, one property of a physical object may be its shape. One way to measure the shape of an object is to interpret the measurements of a digital camera. It transforms the reflected light of the environment (i.e., its wavelength or frequency) into a pixel array, which then can be interpreted to identify the shape of the object.

Properties of physical objects may change over time, which leads to a change of their state $\vec{s}'$. This change of state can be caused by internal dynamics (e.g., a rolling ball) or external influences $\vec{u}(t)$ (e.g., a ball being kicked). External influences are the impact of actuator actions on the properties of a physical object (e.g., a force acting on the ball during the kick). The change of state at each point in time is a function f of the object's state and the corresponding external influences.

$$\vec{s}'(t) = f(\vec{s}, \vec{u}, t) \tag{3}$$

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:

$$\vec{u}_\Sigma = \begin{bmatrix} \vec{u}_{o_1}(t) & \vec{u}_{o_2}(t) & \dots & \vec{u}_{o_m}(t) \end{bmatrix}^T \quad (4)$$

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

$$\vec{s}_\Sigma(t) = \begin{bmatrix} \vec{s}_{o_1}(t) & \vec{s}_{o_2}(t) & \dots & \vec{s}_{o_m}(t) \end{bmatrix}^T \quad (5)$$

The change of the state of the physical system $\vec{s}'_\Sigma$ 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:

$$\vec{s}'_\Sigma(t) = f_\Sigma(\vec{s}_\Sigma, \vec{u}_\Sigma, t) \quad (6)$$

We limit our approach to viewing actuator actions as external influences on physical objects. This stands in contrast to regarding any interactions between arbitrary physical objects as external influences. 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 counter measures by the CPS.

B. The Programmer's View

In our programming model, the developer views the system from a 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. The RTE requires these descriptions to determine, which sensors are necessary for observing the physical objects and how they 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 programming interface and for which of the RTE actions they are necessary.

1) Physical Object Specification: Since the physical system consists of possibly multiple physical objects of interest, each of which possess a designated state and behavior, 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.

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	Estimate when objects reach the target state through internal dynamics.
Req.3	Physical objects' reactions to external influences	Estimate when objects reach the target state through external influences.
Req.4	Target state description	Calculate actuator actions to reach a target system state.

Through inheritance, a class may extend the state and behavioral descriptions of other classes. This simplifies the description of different types of objects, which partly have a similar state and behavior. 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 1 shows an example of a ball, which extends the class of a moving object.

The state of an object of a given class is the vector of its attribute values. The value of an attribute is a digital representation of a physical object property, i.e., an interpretation

```
Class MovingObject extends PhysicalObject {
    MovingObject() {
        this.p = Position(?[m],?[m],?[m]);
        this.v = Velocity(?[m],?[m],?[m]);
        this.a = Acceleration(?[m],?[m],?[m]);
    }
    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 = Sphere(radius==30[cm]);
        this.m = Mass(mass=0.3[kg]);
    }
    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 1. Example for physical object specifications.

of sensor measurements (see Section III-A). The type of an attribute describes which property it is (e.g., a shape). For a programmer to declare such attributes comfortably, a library provides commonly used classes. Further, the developer specifies the characteristics of the property for the given physical object in the constructor (e.g., the radius of a sphere). Such characteristics allow to distinguish multiple similar physical objects, if necessary. This allows the RTE to identify physical objects reliably and continuously through the available sensor measurements. In Figure 1, a ball is defined as a spherical object with a radius of exactly 30 centimeters. Its position, velocity, and acceleration are unknown and therefore have to be measured by sensors or computed by the RTE, which is syntactically indicated by the corresponding question marks.

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, which represent inputs from actuators. They affect the digital object's state and correspond to external influences on the physical object. For example, in Figure 1, 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 towards 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 of achieving the desired results.

Figure 1 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 actuators 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.

2) Target State Description: We chose the concept of constraint programming [14] 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.

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

Figure 2 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 enemy player positions. All players of the own team should be close to an enemy player (i.e., closer than one meter); there should always be one player between the ball and the own goal; there should always be one player between any enemy player and the ball. This positioning allows to intercept the ball, prevents undisturbed passes and enemy attempts 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.

C. Runtime-Environment

The RTE maintains a set of physical object descriptions, which specify the digital representation of the physical system. It continuously evaluates 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, based on the current state of the physical system. In its evaluations the RTE takes into account the actuator requirements in addition to the available actuators, since they narrow down the available physical inputs.

```
Defense {
  double k, l;

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

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

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

Figure 2. Examples of defensive positioning in robot soccer.

Figure 3 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 them to the other modules. The following paragraphs describe the functionalities of the RTE.

1) Interpreter: The interpreter offers an interface to the programmer for registering physical object specifications. It extracts three basic types of information from them:

(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.
(iii) The actuator requirements for providing input signals, i.e., what conditions have to be met for an actuator to be able to supply an input to the physical system.

The interpreter creates a vector of state variables $\vec{x}_c$ from the state description of a given class c. The constraint solver is later able to assign values to these variables. Each state variable stands for a physical object's property, i.e., a set of interpreted sensor measurements. The state variables are utilized in the state equation of the physical object. This equation is created from the set of methods M of the given class. Each method $m \in M$ describes a change of state $\vec{x}'_{c,m}$ for an object of the given class. A method may take parameters (external influences caused by actuators). The interpreter converts them to input variables $\vec{u}_m$. 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_m describes this change of state:

$$\vec{x}'_{c,m}(t) = f_m(\vec{x}_c, \vec{u}_m, t) \tag{7}$$

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

$$\vec{x}'_{c,m}(t) = A_m \vec{x}_c(t) + B_m \vec{u}_m(t) \tag{8}$$

If the class' overall behavior 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:

$$\vec{x}'_c(t) = \sum_{m \in M} \vec{x}'_{c,m}(t) = \sum_{m \in M} (A_m \vec{x}_c(t) + B_m \vec{u}_m(t)) \tag{9}$$

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

$$\vec{x}_c(k+1) = \sum_{m \in M} (A_m \vec{x}_c(k) + B_m \vec{u}_m(k)) \tag{10}$$

For each new class the interpreter appends the new classes state to the existing system state $\vec{\underline{x}}_\Sigma$. Therefore, a new system state $\vec{x}_\Sigma$ is created (see Section III-A):

$$\vec{x}_\Sigma(k) = \left[\vec{\underline{x}}_\Sigma(k) \quad \vec{x}_c(k)\right]^T \tag{11}$$

Hence, a similar approach is used for the state change equations. The new system state change equation is a concatenation of the old system state change equation and the new classes state change equation:

$$\vec{x}_\Sigma(k+1) = \left[\vec{\underline{x}}_\Sigma(k+1) \quad \vec{x}_c(k+1)\right]^T \tag{12}$$

Moreover, the interpreter creates constraints from the actuator requirements for each method of a class. These constraints allow allocating the available actuator inputs to the corresponding input variables. The target state description is added to the constraint system and restricts the possible system states and system state changes. From the given system of equations, requirements, and constraints, the constraint solver is able to compute a set of actuator inputs, which lead to a target state.

2) 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, similarly to the data aggregation and dissemination process described in [12]. This allows to gather and distribute data of the system, based on given rules (i.e., according to the object specification).

The programmer provides classes, which are specifications of the physical objects encompassing the physical system. The observer recognizes an object of a class, if the sensor measurements relate to the class attributes and the corresponding attribute description in the class' constructor.

The observer interprets the sensor measurements (see Section III-A) based on given rules. These rules describe the relation between physical object properties and the corresponding

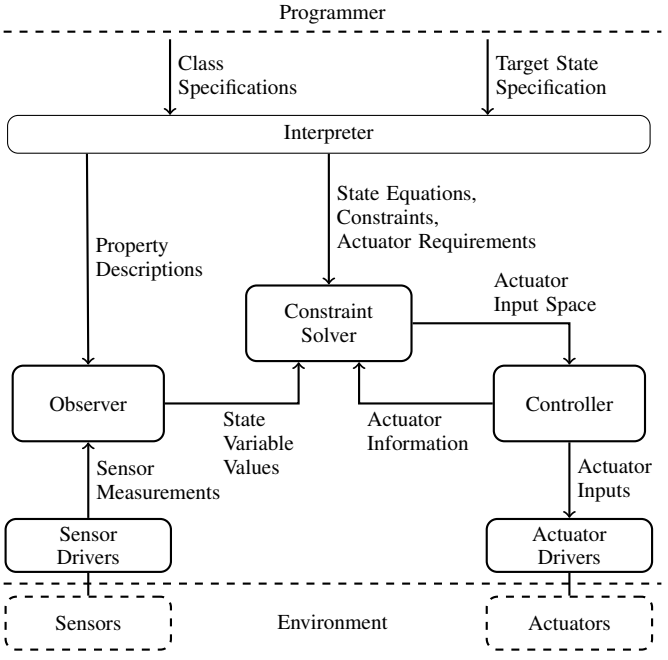

Figure 3. Architecture of the Runtime-Environment.

measurements. The observer maintains all the created object instances by updating their states. These updates are applied whenever new sensor values are available, which relate to the objects' properties. Moreover, the module populates the state variables of the constraint system with the corresponding attribute values (i.e., object states).

3) 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's state equation, the measured current state, the actuator requirements, and information about the currently available actuators. The solver evaluates the constraints periodically to check whether a target state is reached and to update the set of actuator inputs.

Mathematically, the constraint solver's task is to find a point in time t_1, which lies before a given deadline d, and 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 the chosen point in time t_1, such that all constraints hold for the state at time t_1:

$$\vec{x}_\Sigma(t_1) = \vec{x}_\Sigma(t_0) + \int_{t_0}^{t_1} \vec{x}'_\Sigma(t)dt,\ t_1 \leq d \tag{13}$$

4) Controller: The controller module manages the set of available actuators. For each actuator, the controller module maintains information about the actuator's state 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 constraints for determining, which actuators are able to provide the required inputs.

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

IV. 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 given inputs and internal dynamics. This abstracts from the need to manage a changing set of actuators and sensors directly, as the required information by the programmer is reduced to defining the influences of actuators on the system and specifying properties of physical objects.

We present a RTE, which encompasses an interpreter, an observer module, a controller module, and a constraint solver. The observer module maintains a digital representation of the physical system's state, based on the physical object descriptions. 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.

The presented programming model abstracts from implicit conversions between digital computations and physical phenomena. Therefore, the physical semantics of a program are made explicit. They are easier to understand and errors in the translation between digital and physical quantities are prevented. Additionally, the RTE transparently handles a set of changing 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 sensor and actuator specifications, which allows deducing their properties with regard to how they observe and 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, 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 an optimal utilization of the devices have to be ensured.

References

[1] 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.

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

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.* Springer, 2004, pp. 258–272.

[8] 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.

[9] P. Fritzson, *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3*, 2nd ed. John Wiley & Sons, 2014.

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

[11] 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.

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

[13] 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.* ACM, 2002, pp. 21–30.

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

 ISBN: 978-1-61208-989-8