A Model-Driven Engineering Process for Autonomic Sensor-Actuator Networks

Carlos Vidal, Carlos Fernández-Sánchez, Jessica Díaz, and Jennifer Pérez

Escuela Técnica Superior de Ingeniería de Sistemas Informáticos, CITSEM, Universidad Politécnica de Madrid (UPM), Carretera de Valencia, Km. 7, 28031 Madrid, Spain

Correspondence should be addressed to Jessica Díaz; yesica.diaz@upm.es

Received 11 August 2014; Revised 22 January 2015; Accepted 2 February 2015

Academic Editor: Chih-Yung Chang

Cyber-Physical Systems (CPS) are the next generation of embedded ICT systems designed to be aware of the physical environment by using *sensor-actuator networks* to provide users with a wide range of *smart* applications and services. Many of these smart applications are possible due to the incorporation of *autonomic control loops* that implement advanced processing and analysis of historical and real-time data measured by sensors; plan actions according to a set of goals or policies; and execute plans through actuators. The complexity of this kind of systems requires mechanisms that can assist the system's design and development. This paper presents a solution for assisting the design and development of CPS based on Model-Driven Development: *MindCPS* (doMaIN moDel for CPS) solution. MindCPS solution is based on a model that provides modelling primitives for explicitly specifying the autonomic behaviour of CPS and model transformations for automatically generating part of the CPS code. In addition to the automatic code generation, the MindCPS solution offers the possibility of rapidly configuring and developing the core behaviour of a CPS, even for nonsoftware engineers. The MindCPS solution has been put into practice to deploy a smart metering system in a demonstrator located at the Technical University of Madrid.

1. Introduction

Cyber-Physical Systems (CPS) refer to ICT systems (sensing, actuating, computing, and communication) embedded or software integrated in physical objects, interconnected, and providing citizens and businesses with a wide range of *smart* applications and services [1–3]. Examples of these CPS include smart buildings, cities, energy grids, and water networks. These CPS are designed to monitor and respond to the physical environment, enabling fast, effective *autonomic control loops* between sensing and actuation, possibly with cognitive and learning capabilities [1]. To sense and act upon the physical environment, CPS are often built on wireless sensor/actor networks (WSAN) [4]. Autonomic control loops implement (i) information monitoring, (ii) advanced analysis and processing of historical and real-time data measured by sensors or other external sources, (iii) planning of actions according to a set of goals or policies, and (iv) execution of those plans through actuators. This implementation is supported by real-time or historical knowledge. CPS systems are complex due to factors such as the heterogeneity of sensors and actuators, the definition of complex conditions and patterns for problem detection over a large amount of data and events, the needs for real-time processing, and the implementation of plans for problem solving. This complexity makes engineers require mechanisms and tools that could assist them during the system's design and development.

This paper presents a solution for designing and developing CPS, specifically CPS that are conceptualized as a set of *smart nodes* distributed throughout a WSAN that implement autonomic control loops for smart sensing and actuation. This solution is a Model-Driven Development process (MDD [5]), called *MindCPS* (doMaIN moDel for CPS). MDD is a software development approach in which the focus and primary artefacts of development are models—as opposed to programs—and model transformations [5]. The automated and semiautomated code generation through model transformations provide benefits in terms of increase of productivity, facilitated maintenance and portability thanks to the quality of the produced code [6, 7]. MindCPS provides modelling

primitives for explicitly specifying the autonomic behaviour of CPS. The MindCPS solution also includes the definition of a set of *model-to-code transformations* for automatically generating the following: (i) Java code that implements the "core behaviour" of the smart nodes of a CPS (i.e., the autonomic control loop for smart sensing and actuation and communication through an event publish-subscribe middleware); (ii) the EPL (Event Processing Language) queries of an Esper CEP engine (Esper is a component for complex event processing (CEP) and event series analysis available for Java.) that implement the advanced analysis and processing of real-time events coming from monitored and filtered measurements of sensors; and (iii) the SQL queries of a database manager that implement the advanced analysis and processing of non-real-time data coming from monitored and filtered measurements of external sources, services, or even sensors without real-time restrictions. The automatic code generation increases the productivity of constructing CPS thanks to the development time reduction with regard to the traditional programming (the hand-made development). The advantage offered by the MindCPS solution, in addition to the automatic code generation, is the possibility of rapidly configuring and developing the core behaviour of a CPS, even for nonsoftware engineers, thanks to its graphical and intuitive domain language. Finally, it is important to emphasize that this solution has been iteratively designed by extracting the code generation patterns from our industrial experiences. These code generation patterns have been iteratively refined and enriched by each case study. This makes that most of the generated code had been previously tested in terms of functionality and performance. These industrial experiences together with the MindCPS solution are the result of research initiated in two larger ITEA2 projects, IMPONET (intelligent monitoring of power networks http://innovationenergy.org/imponet/) and NEMO&CODED (networked monitoring & control diagnostic for electrical distribution http://innovationenergy.org/nemocoded/). These projects focused on supporting complex and advanced requirements of *smart grids* [8], specifically supporting enhanced efficiency through sensing and metering technologies, as well as automated control and management techniques based on energy availability and the optimization of power demand. To illustrate the application of the MindCPS solution, we present the implementation of a demonstrator for smart metering following the MindCPS process.

This paper is structured as follows. Section 2 briefly introduces the concepts of CPS and WSAN, autonomic computing, and MDD. Section 3 presents the MindCPS solution for designing and developing autonomic smart nodes of CPS over a WSAN. Section 4 describes the use of this MDD process in a demonstrator for smart metering. Section 5 describes related work. Finally, conclusions and further work are described in Section 6.

2. Background

2.1. Cyber-Physical Systems and WSAN. CPS are composed of devices with embedded sensors that continuously collect and process information from the physical environment, often in real time. These CPS are then able to make decisions and act upon the physical world through actuators. In order to realize these sensing and acting capabilities, CPS are often built on WSAN [4]. However, the deployment of complex software to perform these tasks in WSAN is difficult due to power, computation, and memory limitations of their nodes (i.e., sensors and actuators). To deal with this issue, WSAN usually incorporate other kinds of nodes with higher processing capabilities (e.g., *base stations*) in order to perform more complex operations. These unconstrained nodes, also known as *smart nodes*, are essential for satisfying CPS' requirements for more intelligent and autonomic behaviours as they grow in complexity. Upon these smart nodes, most of the advances made on the area of autonomic computing should be implemented.

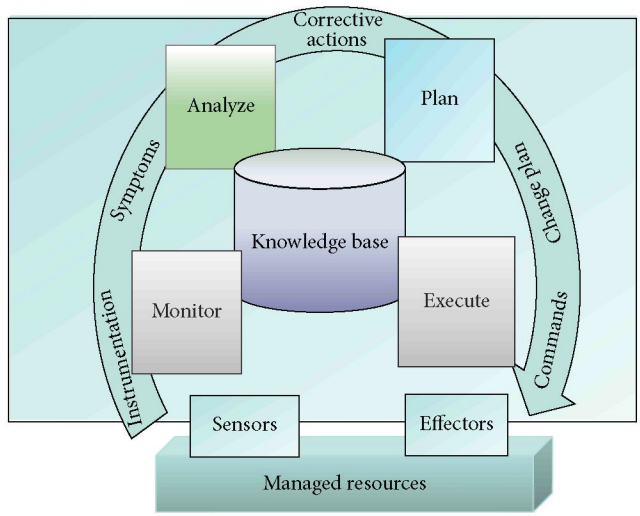

FIGURE 1: MAPE-K loop. Source: [11].

2.2. Autonomic Computing. Autonomic computing (AC) [9, 10] emerged as a solution to deal with the increasing complexity of today's computing systems and human management limitations. Horn [9] defines autonomic systems as software systems that mostly operate without human or external involvement according to a set of rules or policies; in other words, the systems are self-managed. Specifically, IBM proposed the *MAPE-K loop* for supporting autonomic computing (see Figure 1). According to the MAPE-K loop, resources to be managed are composed of a set of sensors that provide information about the current state of the resources. The model implements the following: the monitoring of the information (Monitor); the analysis to detect symptoms that need corrective action (Analyze); the planning of the action required to change the current state of the resource according to a set of goals or policies (Plan); and the execution of the plan through a set of effectors (Execute). These actions are operated over a knowledge base. The MAPE-K loop model offers the advantage of isolating the main concerns that any autonomic process has to provide.

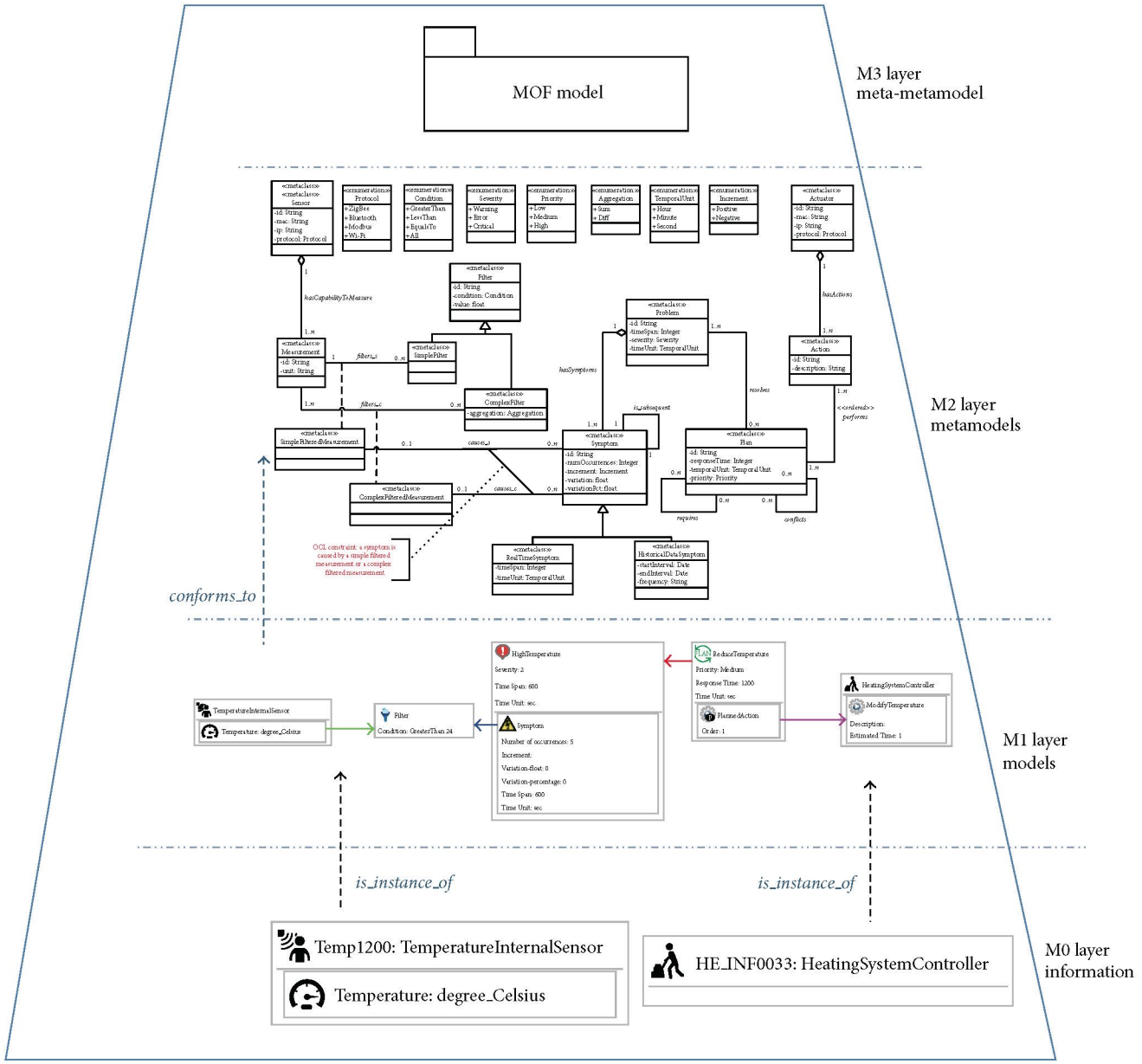

Figure 2: MOF four-layered architecture.

2.3. Model-Driven Development (MDD). MDD is a software development approach in which models can be managed and transformed to facilitate and automate tasks involved in development and evolution by employing high-level abstractions. This approach increases productivity and quality and reduces costs by automating basic activities in software development and evolution [5]. Current quantitative analysis such as the work of Papotti et al. [6] shows that the main advantage of MDD is its code generation, demonstrating that development teams that use code generation are faster than those that applied manual coding.

The OMG metaobject facility (MOF) 2.0 specification [12] defines an architecture to support metamodelling and MDD. Its main purpose is the management of model descriptions at different levels of abstraction. The four-layered metamodel architecture of MOF 2.0 can be described as follows (see Figure 2). The M3 layer (*meta-metamodel layer*) defines the abstract language used to describe the entities of the lower layer (metamodels). The MOF specification proposes the MOF language as the abstract language for defining all types of metamodels, such as the metamodel of UML. The M2 layer (*metamodel layer*) specifies the structure and semantics of the models defined at the lower layer. The M1 layer (*model layer*) comprises the models that describe the data of the lower layer. These models are specified using the primitives and relationships defined in the metamodel layer (M2). Finally, the M0 layer (*information layer*) consists of the instances of the models that are defined at the model layer (M1).

3. Model-Driven Engineering Process for Autonomic Sensor-Actuator Networks

This section presents MindCPS solution, a process based on models and model transformations designed to assist and guide the design and (semi)automatic development of CPS. Specifically, we focus on CPS that offer a set of services to users and businesses through *autonomic control loops*—typically composed of *monitoring*, *analysis*, *planning*, and *execution* tasks according to the MAPE-K loop defined in Section 2.2—which allow CPS to autonomously react to a wide range of situations in order to minimize human intervention.

3.1. doMaIN moDel for CPS (MindCPS). CPS are mainly constituted by the *sensors* embedded in devices that continuously collect *measures* from the environment in order to detect *problems* in the system. These *problems* are triggered through *events* in order to *plan actions* to execute them on the physical system through *actuators*. The domain knowledge model *MindCPS* (doMaIN moDel for CPS) consists of a set of modelling primitives. It was designed to support the specification and definition of the main concepts of a CPS, that is, sensors, measurements, events, problems, plans, actions, and actuators. Through this model, one is able to specify the autonomic control loop of CPS from scratch as well as a change in existing CPS, dealing in a higher abstraction level with some issues that make this kind of systems complex. Some of these issues, mentioned above, are the heterogeneity of sensors and actuators and the definition of complex conditions and patterns for problem detection and the respective plans to deal with them. In order to use the modelling primitives of MindCPS, it is necessary to design a domain-specific language (DSL) through the definition of a metamodel, its domain concepts, relationships, and rules (see layer M2 in Figure 2), as well as a graphical language representation. Using this graphical modelling primitives, engineers, even nonsoftware engineers, can model the autonomic behaviour of CPS (see layer M1 in Figure 2) that conforms to the MindCPS metamodel (see layer M2 Figure 2).

The MindCPS metamodel is composed of a set of interrelated metaclasses. These metaclasses define a set of properties and services for each concept considered in the model. On the one hand, metaclasses, their properties, and their relationships describe the structure and information that is necessary to define the domain knowledge of CPS. On the other hand, the methods of metaclasses offer the primitives to develop instances by creating, destroying, adding, or removing elements which are compliant with the constructors of the metamodel (most methods are omitted to gain readability).

Figure 3 shows a fragment of the MindCPS metamodel. Sensors are described by the metaclass *Sensor*. A sensor is characterized by an identifier, an IP address, a MAC address, and a communication protocol (see the attributes *id*, *ip*, *mac*, and *protocol* inside the metaclass *Sensor*). A sensor has the ability to acquire measurements (see the aggregation relationship *acquired* between the metaclasses *Sensor* and *Measurement* in Figure 3). Measurements are defined by the metaclass *Measurement*, which has two attributes: *id* and *unit* (measure unit). The metaclasses *Filter*, *SimpleFilter* and *ComplexFilter*, define the filters of measurements in order to detect when they are indicating a symptom associated with a problem (see these metaclasses and their relationships *isFiltered_bySimpleF* and *isFiltered_byComplexF* in Figure 3). The abstract metaclass *Filter* has three attributes common to simple and complex filters: *id*, *condition*, and *value*, where the condition can be *GreaterThan*, *LessThan*, or *EqualTo* the value. These attributes are inherited by the metaclasses *SimpleFilter* and *ComplexFilter*. Whereas *SimpleFilter* models filters that can be applied only to one type of measurement, *ComplexFilter* defines filters that are applied to more than one measurement. Therefore, the metaclass *ComplexFilter* has the attribute *aggregation* that permits it to specify an aggregation function (e.g., sum, difference, etc.) to be applied to a set of different measurements. Between the metaclasses *Measurement* and *SimpleFilter*, there is an association metaclass called *SimpleFilteredMeasurement*; and between the metaclasses *Measurement* and *ComplexFilter*, there is an association metaclass called *ComplexFilteredMeasurement*. Both metaclasses may indicate symptoms (see the relationships *indicates* and *shows* in Figure 3).

The abstract metaclass *Symptom* defines a symptom of a problem and it is characterized by an identifier (*id*). Four more attributes define when a set of filtered measurements must be considered a symptom in terms of the number of times a measurement must satisfy a filter condition (*numOccurrences*); how much the measurement must vary over time in absolute terms (*variation*) or percentage (*variationPct*); and the type of variation, positive or negative (*increment*). Symptoms are different depending on whether they are detected from real-time filtered measurements or from historical data stored by the system. The metaclass *RealTimeSymptom* defines the real-time symptoms that are detected within a certain period of time, identified by the attribute *timeSpan* (see Figure 3), while the metaclass *HistoricalDataSymptom* historical-data symptoms are detected within an interval defined by a beginning and an end (see the attributes *startInterval* and *endInterval* in Figure 3). Analysis of historical-data symptoms can be triggered when a real-time symptom is detected (see the relationship *triggers* in Figure 3) or periodically according to a frequency (see the attribute *frequency* in Figure 3).

Symptoms are indications of problems (see the aggregation relationship *hasSymptoms* between the metaclasses *Problem* and *Symptom* in Figure 3). Problems are defined by means of the metaclass *Problem*, which has four attributes: *id*, *severity*, *timespan*, and *timeUnit*. The attribute *timespan* refers to the time during which symptoms must be detected to consider that there is a problem. Problems are resolved through a plan (see the association relationship *resolves* between the metaclasses *Problem* and *Plan* Figure 3). The metaclass Plan has three attributes: id, *responseTime*, and *priority*. The attribute *responseTime* is the expected time for the effect of a plan to be noticeable. A plan can include other plans and/or can conflict with other plans (see the association relationships *includes* and *conflictsWith*). Plans perform one or more action/s to solve a problem (see the

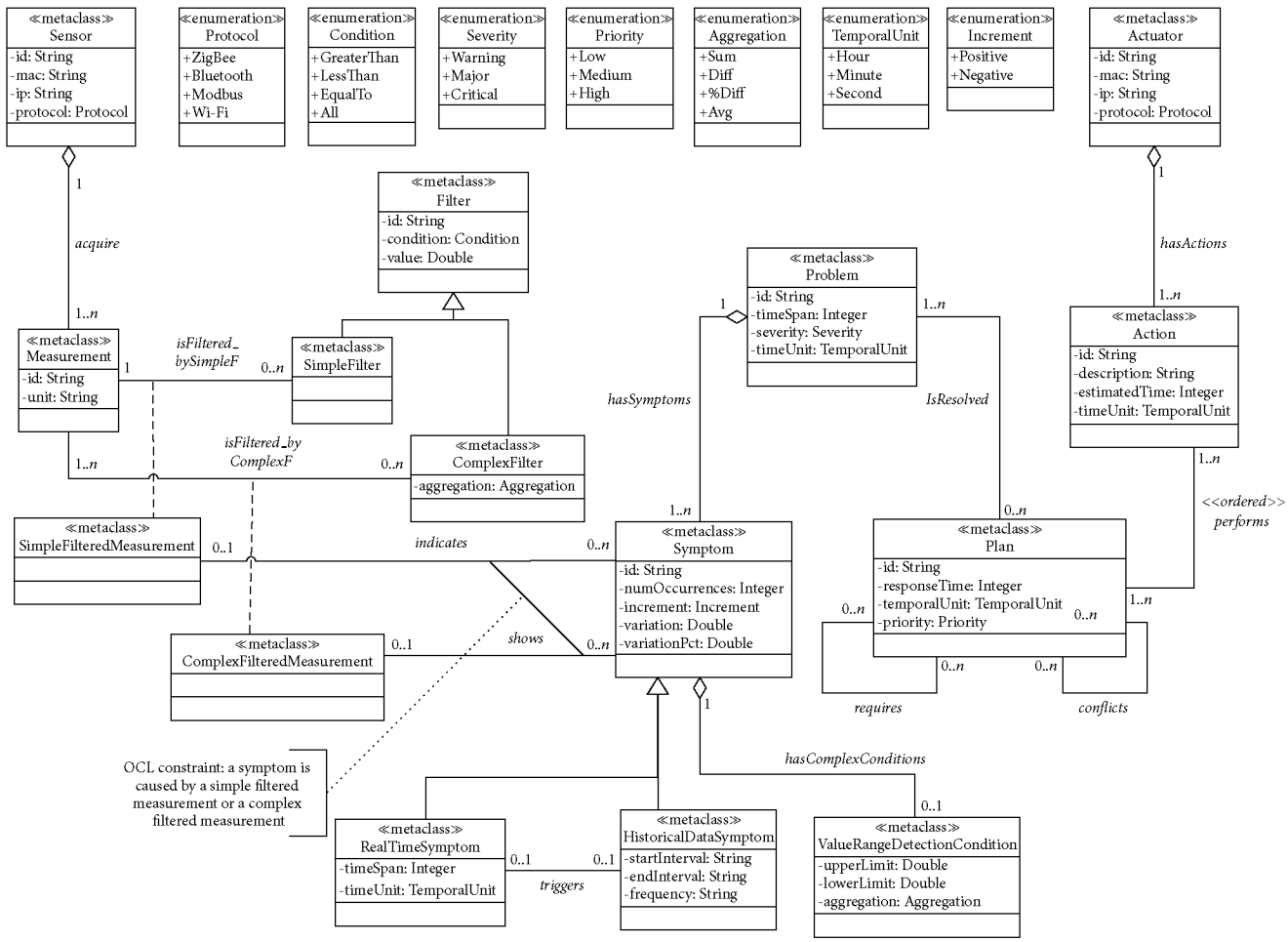

FIGURE 3: MindCPS metamodel.

metaclass *Action* in Figure 3) according to a preestablished order (see the association relationship *performs* that has the stereotype ≪*ordered*≫ in Figure 3). Finally, actuators provide the actions that can be performed (see the aggregation relationship *hasAction* between the metaclass *Actuator* and Figure 3). The metaclass *Actuator* includes the same attributes as the *Sensor* metaclass: *id*, *ip*, *mac*, and *protocol*.

Figure 4 shows the graphical language representation through an illustrative example A sensor that has the capability of measuring three variables—*Measure1*, *Measure2*, and *Measure3*—that are filtered through a set of filters; for example, a filter sets the condition that the value of *Measure1* is *greater than* a constant *X*, while other filters set the condition that the addition (*SUM*) of *Measure2* and *Measure3* is *less than* a constant *X*. These filtered measurements indicate symptoms associated with a problem. For example, a symptom is detected when 3 occurrences of *Measure1 greater than X* are received within 2 seconds with an increment of 2 units between them. This problem can be resolved through one of two possible plans. One of them is composed of two actions—*Action 1* and *Action 2*—and has a time of response of 30 seconds. Finally, Figure 4 shows an actuator that can perform the two actions.

The MindCPS DSL was implemented using the Epsilon generative modelling technologies (GMT) research project and was made available as an Eclipse plugin, the MindCPS tool (see snapshot in Figure 4).

3.2. Model-to-Code Transformation. In MDD, models seek to automate development tasks through model transformations and thereby, reducing the development and/or adaptation time [6, 7]. This is why we define a set of *model-to-code transformations* to generate the code and metadata necessary to support the development of a CPS that is specified through the MindCPS model. To that end, a *model-to-code transformation engine* is the component in charge of automatically transforming a model into the code necessary to implement the core structure and behaviour of CPS. This core behaviour has a common part, that is, the common structure and code that is *common* to a general control loop for any kind of CPS, and a variable part, that is, the code that is *variable* according to information to be monitored, analysed, planned, and executed, which is modelled through a MindCPS model (e.g., the model shown in Figure 4).

To provide a visual of the common structure and code necessary for any kind of CPS according to our definition,

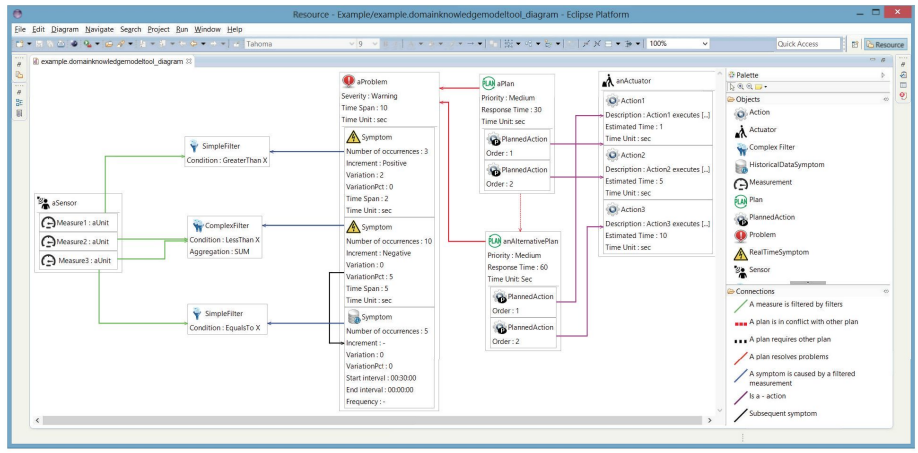

Figure 4: MindCPS graphical language supported by the MindCPS tool.

Figure 5 shows a layered view of the architecture that we have implemented for the construction of CPS' smart nodes. The two-bottom layers of the architecture shown represent the devices of a specific domain and the sensor/actuator network. The *communication middleware* is in charge of acquiring raw data from sensors through a set of drivers that implement the communication protocols of the sensors (e.g., ZigBee, Bluetooth, 802.11, etc.). The central layer implements a *control loop* that will provide low-level services for monitoring, analysis, planning, and execution, as well as for publishing and subscribing events. This layer is composed of controllers that are composed, in turn, of components. Controllers manage the lifecycle and the running context of their respective components. Hence, the *MonitorController* manages the lifecycle and the running context of specific monitors. Each specific *Monitor* implements the *filtering* of raw data in order to identify relevant data, the *translation* of raw data into comprehensible information conforming to the MindCPS model, and the *routing*, that is, the creation of events that are published through an event channel. The *AnalyserController* manages the lifecycle and the running context of specific analysers. Each specific *analyser* processes real-time events—coming from the monitoring—through a CEP (complex event processing) engine and makes use of historical data for detecting anomalies and potential problems. The *Planner&ExecuterController* manages the lifecycle and the running context of specific planners and executers. Each specific *planner* plans corrective actions to change the current state of the resource according to a set of goals and policies, while the *executer* executes the plan through a set of actuators. Finally, the *event driven middleware* implements the channel that interconnects *monitors*, *analysers*, *planners*, and *executers* through events. It can be implemented using different technologies, such as DDS (data distribution service for real-time systems) and JMS (Java message service). These technologies, based on the publish-subscribe paradigm, implement an event driven infrastructure which provide extremely loosely coupled and highly distributed nodes in order to construct scalable solutions and provide an implementation independent from the number of smart nodes of the CPS. This is due to the fact that the publisher of the event has no knowledge of the event's subsequent processing or the interested parties subscribed to the event (asynchronous publish-and-subscribe pattern) [13].

The architecture definition was guided by certain requirements and quality attributes that usually must be addressed when designing CPS. In the context of software architecture, quality attributes are the way to express the qualities we want an architecture to provide to the system or systems that will be built on it [14]. We have focused on interoperability, modifiability, and performance quality attributes. Interoperability and modifiability are achieved thanks to main principles of the event driven SOA paradigm [15], which leverages the interaction between events and services providing flexibility and loose coupling, provides adaptability through service composition and orchestration, and supports encapsulation to integration at highly distributed systems. Performance is addressed by the inclusion of several architectural frameworks for distributed and real-time systems widely tested and used by industry. RTi DDS [16–18] and Esper CEP engine [19] have been proved to be highly efficient in terms of latency, throughput, and memory usage.

As mentioned above, this architecture provides the common structure for the smart nodes of CPS. The behaviour and functionality that are variable depending on the specific CPS are specified through the MindCPS model. As described in Section 3.1, MindCPS supports the specification of the measures to be filtered and monitored, the symptoms to be analysed, the problems to be identified, and the actions to be planned and executed. From the identification of the common and variable parts, it is possible to automatically develop the core behaviour of CPS. From this core control loop, it is possible to provide users and businesses with a wide range of smart applications and high-level services, such as real-time monitoring, alarm management, and decision support (see service layer in Figure 5). Finally, it will be necessary to manually implement the listeners dependent on the drivers of sensors and actuators (see protocol and drivers layer in Figure 5).

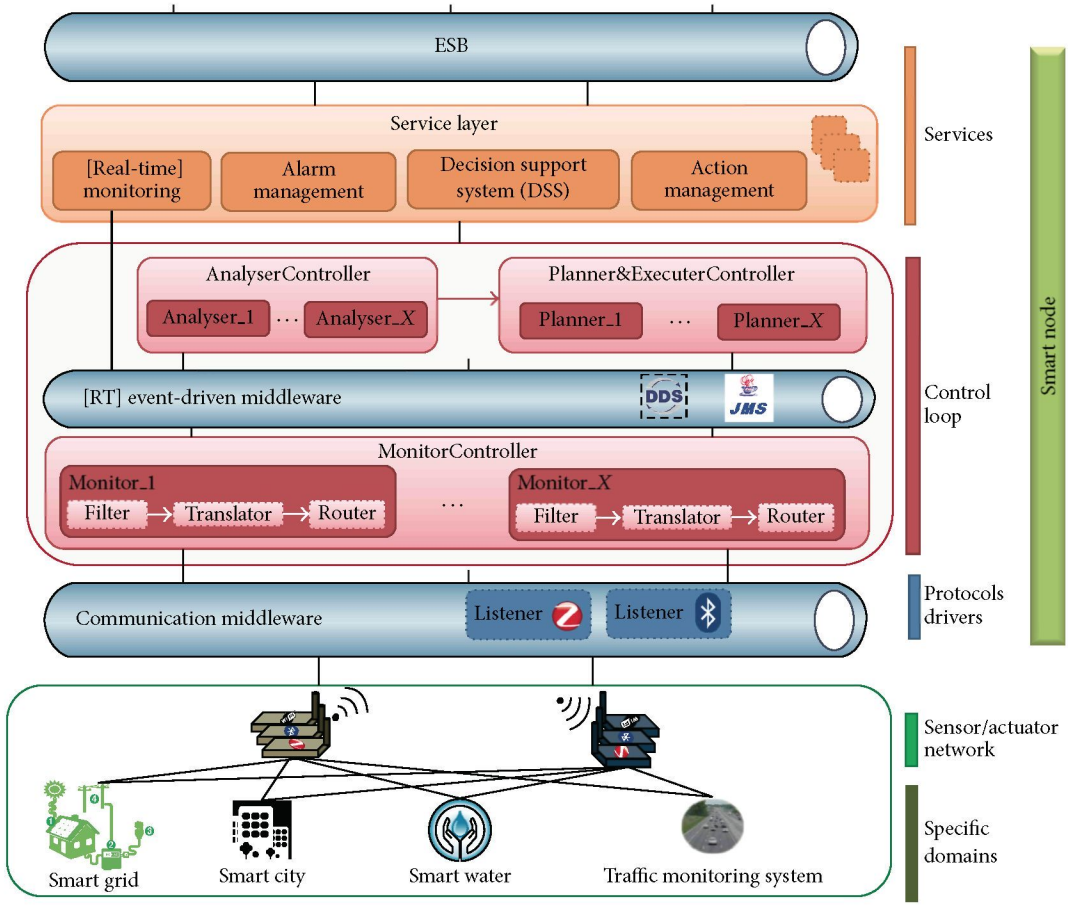

FIGURE 5: CPS architecture.

Once common and variable parts of code to be generated have been identified, model-to-code transformations are defined. Transformations can generate code in any language, such as Java and C#. The task of defining the transformation is complex but can be reused for any CPS based on that language. The difficulty is in defining the first transformation, as subsequent transformations are very similar. We have defined the transformation based on our background in CPS projects. Concretely, the model-to-code transformations were created using previously developed projects, and, therefore, most of the code has been previously tested in production. In detail, the model-to-code transformations we defined generate the following:

(1) the code that implements a control loop to be deployed in the smart nodes of a CPS; specifically, the transformation automatically generates Java code skeletons and composes the common code from a set of common code templates and the variable code from the information modelled in a MindCPS model;

(2) the EPL (Event Processing Language) queries of an Esper CEP engine that implement the advanced analysis and processing of real-time events coming from monitored and filtered measurements of sensors;

(3) the SQL queries of a database manager that implement the advanced analysis and processing of non-real-time data coming from monitored and filtered measurements of external sources, services, or even sensors without real-time restrictions.

3.3. MindCPS into Practice. The design and development of CPS require first modelling the embed software on physical objects (i.e., the smart nodes) that continuously acquire and collect information from the physical environment, which is processed—often in real time—to make decisions and act on the physical world through actuators. To that end, the MindCPS solution provides the modelling primitives for specifying the data to be monitored, the problems to be analysed, and the plans to be executed. Through this specification, it is possible to automatically generate the code that implements the behaviour of the smart nodes of CPS. Figure 6 shows how the MDD process works to support the development process. CPS are specified and configured through the definition of a MindCPS model (see label 1) that conforms to the *MindCPS metamodel* from which, through a model-to-code transformation (see label 2), three artefacts are generated: Java code, EPL, and SQL queries. Finally, the resulting system is executed (see label 3).

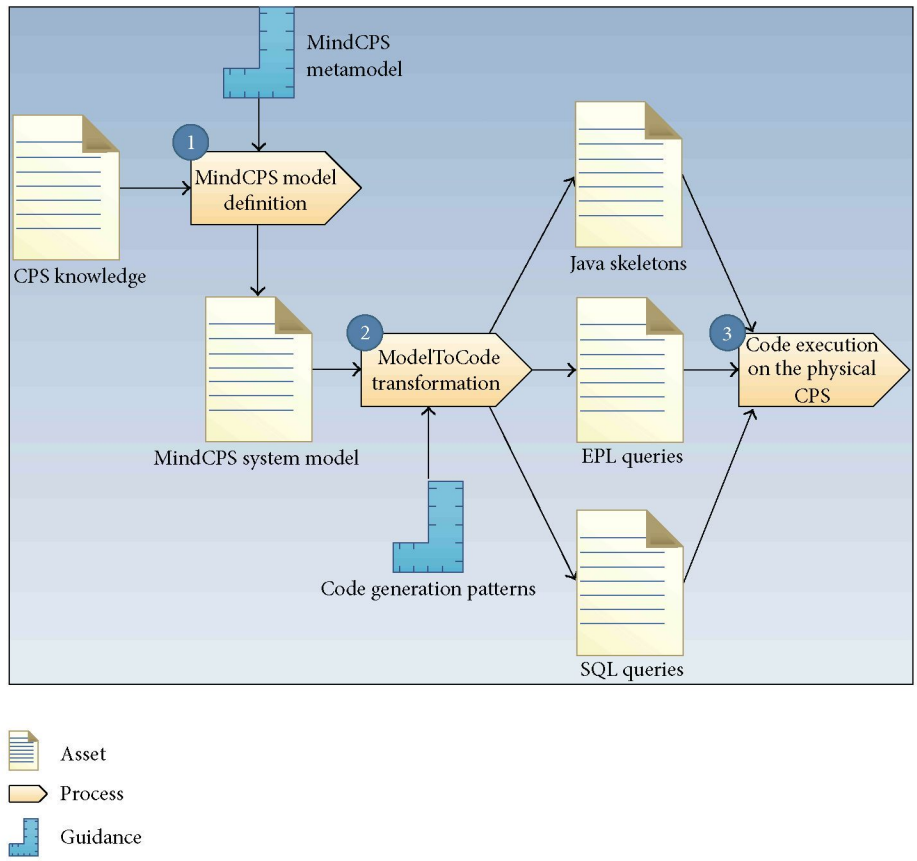

Figure 6: Overview of the MDD process.

The MDD approach supports the automatic generation of the code for new components—for example, a new monitor, analyser, or planner. This code conforms to the core components of the architecture shown in Figure 6. In this way, MindCPS solution simplifies the design and development of CPS, conceptualized as a set of *smart nodes* distributed throughout a WSAN that implement autonomic control loops for smart sensing and actuation. As a result, the effort invested to develop CPS using the MindCPS consists of (i) the time and effort that the engineers invest in defining the conceptual modelling of the CPS and (ii) the time and effort for implementing the code that is specific of each of CPS, that is, the listeners dependent on the drivers of sensors and actuators. Using MindCPS the time and effort of developing the automatically generated code are minimum.

System complexity is also taken under control thanks to the MAPE-K loop model. The MAPE-K loop model offers the advantage of isolating the main concerns that any autonomic process has to provide. The separation of concerns provided by the MAPE-K loop model entails a substantial reduction of interactions between software components that can be more easily handled.

The automation of code generation provided by MindCPS solution also provides higher code quality and faster software design, development, and evolution. Since their components are decoupled and constructed for being reused for other systems, the transformations (code generation patterns) have been iteratively constructed from our experiences in CPS' construction in IMPONET and NEMO projects (see Section 1).

4. Experience Report

The MindCPS MDD process is the result of the research initiated in the projects IMPONET and NEMO&CODED, as it has been previously mentioned. To illustrate the application of the MindCPS solution, we present the implementation of a demonstrator for smart metering with energy efficiency capabilities (see Figure 7).

The demonstrator, which we refer to as *Arboleda*, was deployed in a building located on the south campus of the Technical University of Madrid. The Arboleda demonstrator was equipped with various artefacts; including sensors, *smart grid nodes*, gateways, and actuators. The sensors included power, water, humidity and temperature meters, and a presence detector. The actuators included a HVAC system controller and a photovoltaic generator PLC connected to a solar panel. The Smart Node software components, including the automatically generated code, run on a Raspberry Pi model B (512 MB of RAM) connected through its 100 Mb Ethernet port to the LAN where the rest of the elements are installed. This Raspberry Pi is also equipped with a ZigBee Shield to allow the device communication via ZigBee

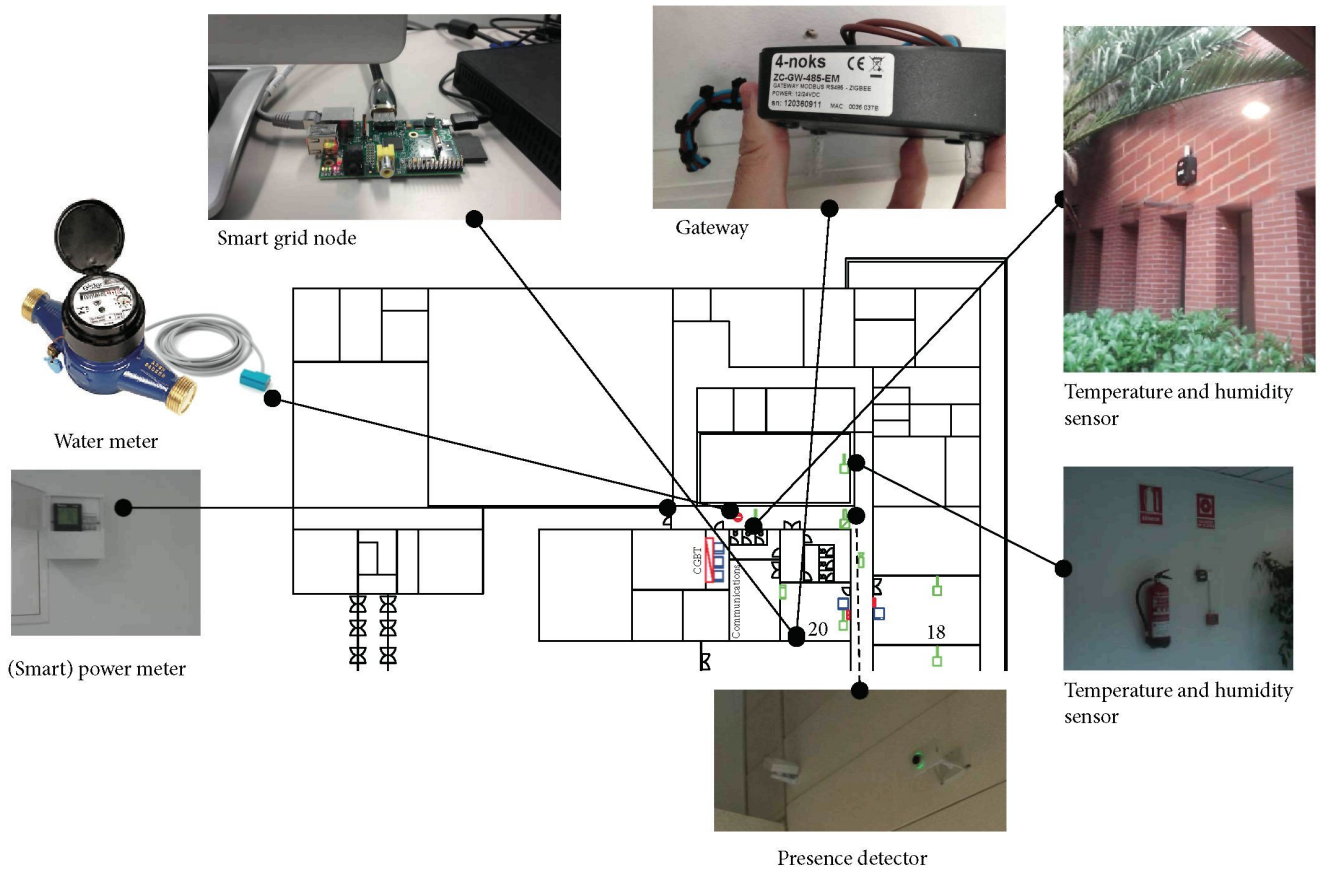

FIGURE 7: Arboleda demonstrator for smart metering.

protocol. It is remarkable that Raspberry Pi model B has a cost per year between 7$ and 15$ (24/7 running), which is fairly good in terms of power consumption.

Figure 8 shows a fragment of a MindCPS that models the domain knowledge of the Arboleda demonstrator as follows. The *PowerMeter_Building* measures and records *power consumption*, *maximum voltage*, *slip frequency,* and *phase angle*, while the *PowerMeter_SolarPanel* measures *power output*, *voltage*, *slip frequency,* and *phase angle*. The resulting measures are filtered to detect symptoms and problems related to unusual consumption levels in the building, as well as synchronization failures (abnormalities of voltage and frequency between the corresponding phases of a solar panel output and grid supply) (see *Unusual Consumption* and *Power Grid Synchronization Failure*). For example, the problem of *Power Grid Synchronization Failure* is detected when the limits for synchronization are exceeded. These limits are as follows: phase angle is ±20 degrees (see *maxPhaseAngleDiff*), maximum voltage difference is 7% (see *voltageDiff*), and maximum slip frequency is 0.44% (see *frequencyDiff*). This problem is solved through a plan that consists of an action that synchronizes—that is, minimizes the difference in voltage, frequency, and phase angle between the corresponding phases of the solar panel output and grid supply—through a PLC connected to the solar panel (see the plan *Repair Sync Failure*).

The *temperature sensor* measures temperature which is filtered (measurements greater than 22°C) to indicate a problem that we call *Rapid Temperature Increase* when 6 consecutive occurrences are detected within 1 minute with an increment of 2°C. The *Cooling Procedure* plan generates an alarm and attempts to solve the problem by launching the *Reduce Temperature* action, provided by the *HVAC System Controller*. Finally, the sensor *Occupancy Sensor* measures the number of people inside the building. The problem of *Unusual Consumption* is detected when two symptoms occur: (1) 5 measures of power consumption *greater than* 21.5 KWh are detected during a period of 30 minutes and (2) during the same 30-minute period, the building occupancy is low (under 15 people). As shown in Figure 8, the historical data analysis for detecting the symptom *Few People in Building* is triggered when an *Over Average Consumption* symptom is detected. The problem *Unusual Consumption* is reported through a service that publishes the problem and associated symptoms to a website (see the plan *Report Unusual Consumption*).

The MindCPS model identifies which measurements a *smart grid node* of the Arboleda demonstrator must be able to monitor and analyse in order to execute plans. Once this is accomplished, the code can be generated. Figure 9 shows an overview of the Java classes that are automatically (see solid rectangles) or semiautomatically (see dashed rectangles) generated for detecting and solving the

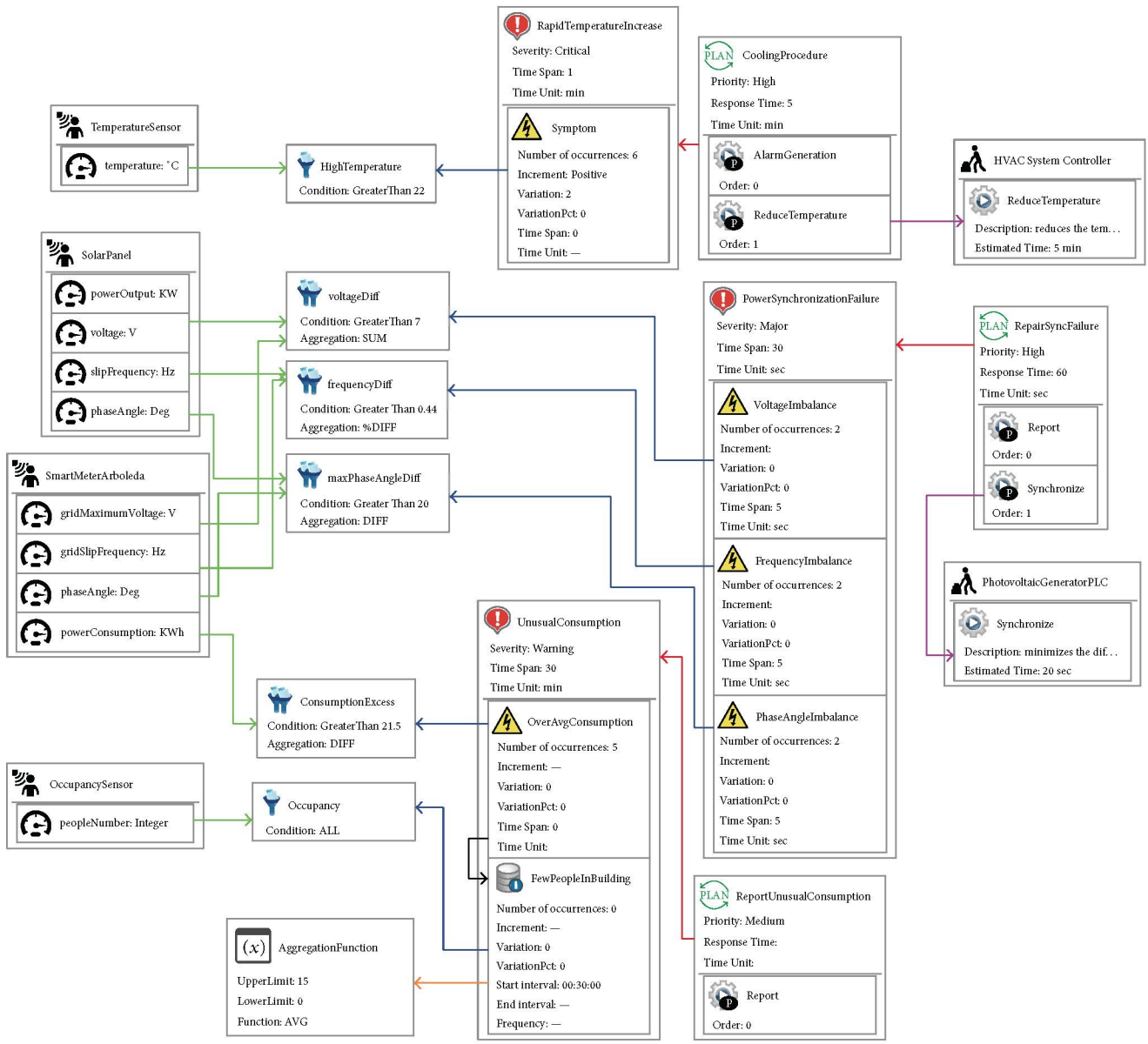

FIGURE 8: A fragment of the MindCPS model of the CPS deployed at the Arboleda demonstrator.

problem *Rapid Temperature Increase*. Three events, three publishers, and three subscribers are created in order to enable interaction between the automatically generated components: *TemperatureMonitor*, *RapidTemperatureIncreaseAnalyzer*, and *CoolingProcedure-Plan*. Event-based interactions between these publishers and subscribers provide the system functionality. The MindCPS process generated 61 classes, 6573 lines of code, 11 running threads, 6 publishers, and 12 subscribers. These numbers could be low, but they are suitable for maintaining loose coupling between components and for implementing design patterns correctly.

In addition to the Java code, a key part of the demonstrator is the analysis and detection of problems—consumption abnormalities. The analysers of this demonstrator are implemented on the Esper CEP engine, and thus EPL queries are also automatically generated. Pseudocode 1 shows the EPL query automatically generated for detecting the *Rapid Temperature Increase* problem.

Code generation can be even more complex in the case of the detection of historical-data symptoms. Although the components for the *Unusual Consumption* problem are generated in a similar way, the main difference lies in how the code for historical-data symptom detection is generated. This is due to the fact that not all database management systems support SQL clauses for pattern recognition. For example, Pseudocode 2 shows the Java method pseudocode for detecting the historical-data symptom *Few People in Building* and Pseudocode 3 shows the generated SQL query. The method *createQuery* creates the query, extracting the interval from which the analysis must be carried out (see *startInterval*)

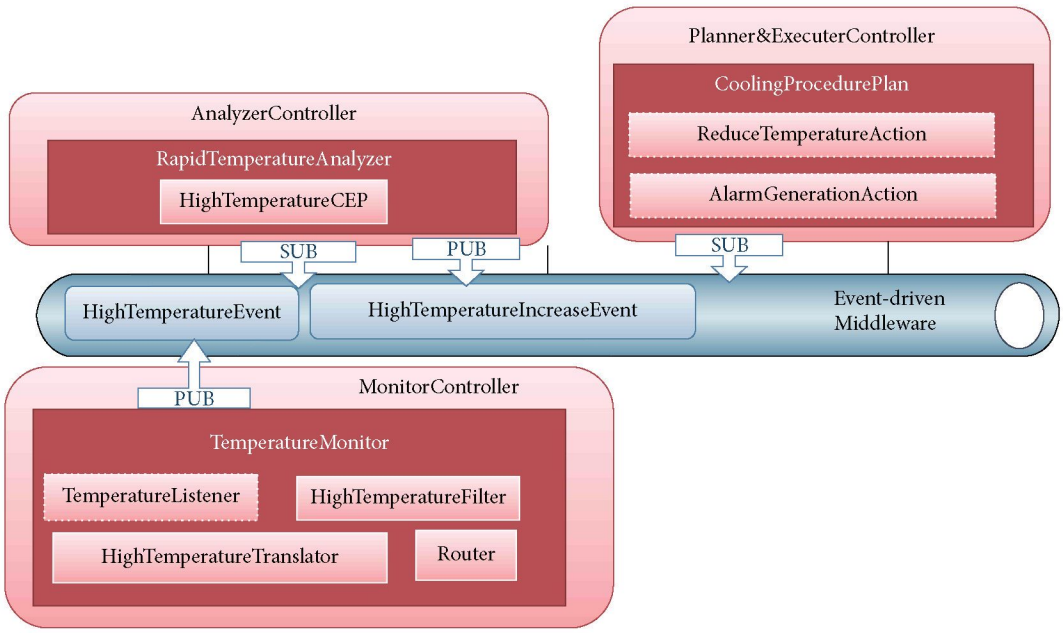

Figure 9: Overview of classes semiautomatically and automatically generated.

```
select * from TemperatureCEP.win:time(1 min)
    match_recognize
   (pattern (A B C D E F G)
         define
         B as B.temperature > A.temperature,
         C as C.temperature > B.temperature,
         D as D.temperature > C.temperature,
         E as E.temperature > D.temperature,
         F as F.temperature > E.temperature,
         G as G.temperature > F.temperature
               and (Math.abs(G.temperature -
                             A.temperature) >= 2))
```

Pseudocode 1: EPL statement.

```
public int fewPeopleInBuildingSymptom(startInterval
String){
    String query = createQuery(startInterval);
    ArrayList<OccupancyEvent> result =
                  OccupancyDAO.executeQuery(query);
return applyAggregationFunction(result, AVG,
                          upperLimit,lowerLimit);
}
public String createQuery(String startInterval){
   String startDateTime= obtainDate(startInterval);
   String query = "...";
   return query;
}
```

Pseudocode 2: Java methods pseudocode for historical-data symptom detection.

```
Select peopleNumber, eventTimeStamp
    from OccupancyHistoricalData
    where eventTimeStamp > @startDateTime
        and eventTimeStamp < current_timestamp
    order by eventTimeStamp asc
```

Pseudocode 3: Generated SQL query *for fewPeopleInBuilding* symptom detection.

based on the current system time. As *endInterval* is not defined, the end of the interval is determined by the current system time. Finally, the method *applyAggregationFunction* returns the number of matches for the searched pattern, applied over the executed query result.

Another issue that affects the time of generation is the number of smart nodes to be generated. Until now, we have modelled small and medium size CPS. However, to cope with the modelling of big CPS using MindCPS, in which the number of nodes undergoes an explosion in number, abstraction and parameterization techniques with different levels of generation will be implemented in the future. These techniques will allow modeling families, subsystems, and types of elements and then instantiating them by refinement in order to scale the creation of nodes as we have demonstrated in previous works [20].

Using the MindCPS model described in Figure 8, we were able to automatically generate most of the code necessary to implement the demonstrator for smart metering. This code will be deployed at the smart grid nodes of the Arboleda demonstrator (see the smart grid node in Figure 7).

To conclude this section, it is necessary to underline some limitations of the presented case study. Our testing scenario pursues to check the MDD generation and its feasibility in small/medium size CPS, although it is not able to measure performance in large-scale systems. We plan to measure performance and other quality attributes impact in complex CPS, providing metrics and accurate statistics, but this specific work is out of the scope of this paper.

5. Related Work

The main contribution of MindCPS is the adoption of MDD for the (semi)automatic construction of CPS. A few works were found regarding the modelling of this kind of system, particularly autonomic CPS that follow an MDD process. Prakash et al. [21] combine MDD and several autonomic computing principles in a different domain. Specifically, they present a model-driven methodology for designing and verifying autonomic behaviours of future network architectures. Zein et al. [22] present a metamodelling approach to define a DSL for smart sensor description for a deep-sea observatory. This approach is focused on defining a metamodel, while the work presented here goes two steps further in the MDD process by (1) defining a DSL by means of a graphical modelling tool and (2) performing model-to-code transformation. The MindCPS metamodel defines higher-level WSAN functionalities than the metamodel presented by Zein et al. [22].

6. Conclusions and Further Work

The current work presents the MindCPS solution that supports and automates the design and development of CPS. *Design* is guided by a DSL for specifying the autonomic behaviour of CPS. *(Semi)automatic development* is supported by an MDD approach, specifically by a model-to-code transformation whereby models that conform to the MindCPS DSL are transformed into (i) Java code that implements the autonomic control loop for smart sensing and actuation and communication through an event publish-subscribe middleware; (ii) the EPL queries of an Esper CEP engine that implement the advanced analysis and processing of real-time events coming from monitored and filtered measurements of sensors; and (iii) the SQL queries of a database manager that implement the advanced analysis and processing of non-real-time data coming from monitored and filtered measurements of external sources, services, or even sensors without real-time restrictions.

The MindCPS solution was put into practice in a demonstrator for smart metering located in a building on the south campus of the Technical University of Madrid. With the implementation of this demonstrator, we were able to check the feasibility of the MindCPS solution and also refine and improve the model.

As further work, we are planning to provide a CPS framework with the capability of dynamic reconfiguration, or even *self*-reconfiguration, by adding mechanisms for safe-stopping in order to guarantee that the elements to be evolved at runtime are placed in safe state, avoiding the possibility that changes in these elements would introduce inconsistencies into the running system.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

The work reported here has been partially sponsored by the Spanish fund: IMPONET (ITEA 2 09030, TSI-02400-2010-103), NEMO&CODED (ITEA2 08022, IDI-20110864), and MESC DPI2013-47450-C2-2-R. It has also been funded by the

UPM (Technical University of Madrid) through its researcher training program.

References

[1] E. A. Lee, "Cyber physical systems: design challenges," in *Proceedings of the 11th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC '08)*, pp. 363–369, May 2008.

[2] R. Rajkumar, I. Lee, L. Sha, and J. Stankovic, "Cyber-physical systems: The next computing revolution," in *Proceedings of the 47th Design Automation Conference (DAC '10)*, pp. 731–736, June 2010.

[3] J. A. García and J. J. Rodríguez, "Open CPS platforms," in *Proceedings of the CPS 20 Years from Now—2nd Experts Workshop Cyphers*, pp. 22–26, 2014.

[4] J. Yick, B. Mukherjee, and D. Ghosal, "Wireless sensor network survey," *Computer Networks*, vol. 52, no. 12, pp. 2292–2330, 2008.

[5] S. Beydeda, M. Book, and V. Gruhn, *Model-Driven Software Development*, Springer, 2005.

[6] P. E. Papotti, A. F. Do Prado, W. L. De Souza, C. E. Cirilo, and L. F. Pires, "A quantitative analysis of model-driven code generation through software experimentation," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7908, pp. 321–337, 2013.

[7] A. G. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*, Addison-Wesley, Longman, Boston, Mass, USA, 2003.

[8] V. C. Güngör, D. Sahin, T. Kocak et al., "Smart grid technologies: communication technologies and standards," *IEEE Transactions on Industrial Informatics*, vol. 7, no. 4, pp. 529–539, 2011.

[9] IBM White Paper, "An architectural blueprint for autonomic computing," 2005, http://www-03.ibm.com/autonomic/pdfs/AC%20Blueprint%20White%20Paper%20V7.pdf.

[10] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 4–50, 2003.

[11] V. Tewari and M. Milenkovic, "Standards for autonomic computing," *Intel Technology Journal*, vol. 10, no. 4, pp. 275–284, 2006.

[12] Object Management Group, "Meta-object facility (MOF) specification 2.0," Tech. Rep. formal-06-01-01, Object Management Group, 2006, http://www.omg.org/spec/MOF/2.0/PDF/.

[13] G. Hohpe and W. Bobby, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*, Addison-Wesley Longman Publishing, Boston, Mass, USA, 2003.

[14] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, Addison-Wesley Professional, 3rd edition, 2012.

[15] S. Overbeek, B. Klievink, and M. Janssen, "A flexible, event-driven, service-oriented architecture for orchestrating service delivery," *IEEE Intelligent Systems*, vol. 24, no. 5, pp. 31–41, 2009.

[16] P. Bellavista, A. Corradi, L. Foschini, and A. Pernafini, "Data distribution service (DDS): a performance comparison of OpenSplice and RTI implementations," in *Proceedings of the 18th IEEE Symposium on Computers and Communications (ISCC '13)*, pp. 377–383, July 2013.

[17] C. Esposito, S. Russo, and D. di Crescenzo, "Performance assessment of OMG compliant data distribution middleware," in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS '08)*, pp. 1–8, IEEE, Miami, Fla, USA, April 2008.

[18] M. Xiong, J. Parsons, J. Edmondson, H. Nguyen, and D. C. Schmidt, *Evaluating the Performance of Publish/Subscribe Platforms for Information Management in Distributed Real-Time and Embedded Systems*, 2010, http://www.omgwiki.org/dds/, http://portals.omg.org/dds/sites/default/files/Evaluating_Performance_Publish_Subscribe_Platforms.pdf.

[19] M. Dayarathna and S. Toyotaro, "A performance analysis of system s, s4, and Esper via two level benchmarking," in *Quantitative Evaluation of Systems*, vol. 8054 of *Lecture Notes in Computer Science*, pp. 225–240, Springer, Berlin, Germany, 2013.

[20] J. Pérez, I. Ramos, J. A. Carsí, and C. Costa-Soria, "Model-driven development of aspect-oriented software architectures," *Journal of Universal Computer Science*, vol. 19, no. 10, pp. 1433–1473, 2013.

[21] A. Prakash, R. Chaparadza, and A. Starschenko, "A Model-driven approach to design and verify autonomic network behaviors," in *Proceedings of the 3rd IEEE International Workshop on Management of Emerging Networks and Services*, pp. 701–706, IEEE, December 2011.

[22] O. K. Zein, J. Champeau, D. Kerjean, and Y. Auffret, "Smart sensor metamodel for deep sea observatory," in *Proceedings of IEEE OCEANS EUROPE*, pp. 1–6, May 2009.