An Architecture Framework for Modelling and Simulation of Situational-Aware Cyber-Physical Systems

Conference Paper · September 2017

DOI: 10.1007/978-3-319-65831-5_7

4 authors:

Mohammad Sharaf
Università degli Studi dell'Aquila
18 PUBLICATIONS **50** CITATIONS
SEE PROFILE

Moamin Abughazala
An-Najah National University
3 PUBLICATIONS **5** CITATIONS
SEE PROFILE

Henry Muccini
Università degli Studi dell'Aquila
157 PUBLICATIONS **1,715** CITATIONS
SEE PROFILE

Mai Abusair
Università degli Studi dell'Aquila
7 PUBLICATIONS **11** CITATIONS
SEE PROFILE

An Architecture Framework for Modelling and Simulation of Situational-Aware Cyber-Physical Systems

Mohammad Sharaf[1(✉)], Moamin Abughazala[2], Henry Muccini[1], and Mai Abusair[1]

[1] DISIM Department, University of L'Aquila, L'aquila, Italy
massharaf@yahoo.com, henry.muccini@univaq.it, mai.abusair@gmail.com
[2] Computer Science Department, An-Najah National Univesity, Nablus, Palestine
m.abughazaleh@najah.edu

Abstract. Situational Aware (SiA) Cyber-physical systems (CPS) harmoniously integrate computational and physical components to being aware of what is happening in the surroundings and using this information to decide and act. Architecture description of SiA-CPS can be a valuable tool to reason about the selected solutions, and to enable code generation and simulation. This paper presents an architecture framework that automatically generates from a SiA-CPS architecture description, an executable code used to simulate the architecture model and evaluate it in terms of data traffic load, battery level and energy consumptions. The framework makes use of a model transformation approach where, three SiA-CPS domain-specific modeling views are automatically transformed into the input language of CupCarbon, an open source tool supporting the simulation of sensor network architectures.

1 Introduction

An architecture description is the practice of recording software, system, and enterprise architectures so that architectures can be understood, documented, analyzed and realized [1]. Accordingly, a number of architecture frameworks, architecture description languages, and different views and viewpoints have been proposed in the years [2–5], each one focussing on a specific application domain, set of views and viewpoints, or concerns. As clearly remarked since more than two decades, an architecture description is essential to the analysis of the high-level properties of a complex system [6]. It can expose various kinds of problems that would otherwise go undetected [7].

This paper proposes a means to simulate *situational-aware cyber-physical systems*, based on a multi-view architecture description.

Situational-aware cyber-physical systems (SiA-CPS) are cyber-physical systems that, by transforming sensed data into actionable intelligence, has the ability to observe the (user's) surroundings and make detailed assessments about his environment. SiA-CPS involves being aware of what is happening in the vicinity

A. Lopes and R. de Lemos (Eds.): ECSA 2017, LNCS 10475, pp. 95–111, 2017.
DOI: 10.1007/978-3-319-65831-5_7

to understand how information, events, and one's own actions will impact goals and objectives, both immediately and in the near future [8].

Architecting SiA-CPS requires the description of novel architectural views, were the physical environment, as well as its cyber dimension, plays a key role [9]. Software components, implemented on top of SiA-specific hardware IoT devices (e.g., sensors and actuators, CCTV cameras, beacons), are required to interact in a prescribed open or closed physical space (e.g., a parking lot, a classroom, a hallway) under observation [10]. In previous work [11], some of the co-authors presented an architecture description for SiA-CPS. Designed according to the IEEE/ISO/IEC 42010 standard [1], the CAPS architecture description supports an architecture-driven development of SiA-CPSs, and comprises three (plus two) modeling views, and namely: the software architecture structural and behavioral view (SAML), the hardware view (HWML), and the physical space view (SPML). The latter implements an architecture description for what typically referred as cyber physical space [12–14]. In order to create a combined software, hardware, and space view of SiA-CPS, the three proposed modelling views are linked together via two auxiliary views, denoted as Mapping Modelling Language (MAPML) and Deployment Modelling Language (DEPML).

This work builds on top of [11] by proposing a code generation framework that, by transforming CAPS models into the CupCarbon [15] simulator input, will support the CAPS architecture simulation in terms of data traffic and load and battery consumptions. The framework, through a group of code generators, transforms SAML model (and then, HWML and SPML) into a completely functional code. This paper will focus on the CupCarbon simulator language named Senscript. The CAPS modeling and simulation framework is realized by exploiting advanced Model-Driven Engineering (MDE) techniques, such as metamodelling, model weaving and model transformation.

The main contributions of this paper can be summarized as follows:

- A technical process, comprising four main processes (parsing, analyzing, generating script, and generating project) to manipulate CAPS models;
- A set of model to text transformations, that transforms SAML, HWML, and SPML models into CupCarbon files;
- The initial application to the UnivAq Street Science situational awareness application.

The rest of the paper is organized as follow: Sect. 2 provides background information on the CAPS modeling framework, the CupCarbon Simulator, and the UnivaAq Street Science application scenario. Section 3 details the code generation framework: its process and the transformational approach. Section 4 applies the CAPS simulation approach to the NdR system, while simulation results are presented in Sect. 5. Section 6 concludes the paper.

2 Background

2.1 The CAPS Modeling Framework

The CAPS modeling framework supports the engineer of SiA-CPS. It is based on a multi-view architectural approach designed according to the IEEE/ISO/IEC 42010 standard [1]. The aim of this framework is to support the architecture description, reasoning, and design decision process.

The CAPS is designed and implemented taking into account three architectural views: the software architecture structural and behavioral view (SAML), the hardware view (HWML), and the physical space view (SPML). We decided to have all things related to Software in one view (as well for HW and SPML) in order to provide a cohesive modeling environment. In addition, the CAPS has two auxiliary views, denoted as Mapping Modeling Language (MAPML) and Deployment Modeling Language (DEPML), are used to link together the three views. While details are provided in [11], we here summarize the main modeling elements.

The *software architecture view* supports architects in the definition of the software architecture of the SiA-CPS application through the *SAML* modeling language. This view looks exclusively to software elements, with a specific focus on the architecture structure and its behavior [16]. Briefly, the *SAML* view describes how `components` and `connections` exchange messages through `message ports`. Each component in *SAML* model can declare a set of `modes`, and each mode can contain a list of `events`, `conditions`, and `actions`, that all together represent the behavior of the component (an example is provided in Sect. 4). Moreover, `application data` manipulated by actions are defined inside the component.

The *hardware view* describes the hardware characteristics of each `hardware element` to be used within a SiA-CPS. A model in the hardware modeling language (HWML) encompasses specific low-level, node-specific information, like its `memory`, `energy source`, `processor`, installed `sensing units` and `actuating units`. A description of the HWML metamodel is reported in [11].

The *physical space view* describes the physical site, in the real world, where the SiA-CPS equipment will be deployed. The space modeling language (SPML) provides support for developing 3D model editors. The CAPS modeling framework enables engineers to specify 3D syntaxes for SPML in a declarative way which will reduce the amount of effort and need for low-level expertise. It aims to support the standardization of a language for declarative specification of 3D concrete syntaxes for SiA-CPS. This view is especially useful for developers and system engineers when they have to consider the network topology, the presence of possible physical obstacles (e.g., walls, trees) within the network deployment area, and so on. More details are available in [11].

2.2 CupCarbon Simulator

CupCarbon [15] is an open source simulator dedicated for wireless sensor network. It is used for scientific and educational purposes. It assists scientists in

testing their wireless topologies and protocols, while it assists trainers in clarifying how wireless sensors work. Moreover, in CupCarbon, a network can be designed and sensors can be deployed directly onto the map by using OpenStreetMap that is provided through its interface. For more specific sensor nodes configurations, CupCarbon provides a scripting language called *SenScript* that can be used for this purpose. Moreover, its environment allows users to design mobility scenarios through which they generate different events. After deploying sensor nodes, one can use CupCarbon to simulate the energy consumption.

In this paper, the CupCarbon simulator will receive the input files generated by the CAPS code generator. Those files, encoding information coming from CAPS in the SenScript language, will be used to simulate energy consumption, battery level and data traffic of model nodes associated with the CAPS architectural model.

2.3 UnivAq Street Science System

The UnivAq Street Science is the European Researchers' Night (NdR) event organized by the University of L'Aquila. In this event, the research community and public are brought together from the afternoon until late at night to share a combination of entertainment and information. As an exemplification scenario we will take the NdR held in L'Aquila city center, in which performances, lectures, demonstrations, workshops take place in its squares, main streets, and buildings. From our experience in organizing this event in L'Aquila, we captured some source of evidence. First, about 20,000 visitors are coming to the NdR every year. Second, late hours usually have more crowded than early hours. Third, the weather changes the visitor's preferences in what to see and where to stay more. Forth, visitors are unable to easily locate activities and though they miss some of them.

Our research group has been invited to provide new services to improve the quality of the visiting experience. For this purpose, we developed (and we are refining for the 2017 edition) a SiA-CPS with a mobile application as the first step towards a better NdR experience. Through the environmental physical sensors that we are deploying in the NdR area, and the mobile application used by visitors on their smartphones, we are providing the following services: (i) access control to rooms, laboratories, and parking lots, (ii) open and closed spaces monitoring, (iii) balance people crowd among different events and spaces, by using the mobile app to inform visitors about the degree of crowd in a place, (iv) make a planner that creates a tour while minimizing the waiting time and crowd in an area, (v) urban security, specifically in the case of earthquakes, fires and over crowd.

In this paper we focus on two situational-aware services that are planned for the NdR. These services are related to rooms access control and safe airflow. A SAML model will be presented in Sect. 4. We will also describe the related HWML and SPML models. By using the CAPS code generator, the models will be transformed into CupCarbon Scripts. After all, the model generated code

will be sent to the CupCarbon simulator in order to evaluate the architecture's energy consumption, battery level, and data traffic.

3 CAPS Code Generation

CAPS code generation framework is a framework in which CAPS models pass through different interpretations in order to build a code project that is able to be inserted in CupCarbon simulator. By running this code project on CupCarbon, we will be able to evaluate CAPS architecture in terms of data traffic load, battery level and energy consumptions of its nodes. This framework starts from interpreting SAML model and its components into Senscript files. Then, HW specifications and the deployment locations of nodes will be extracted from HWML and SPML models, respectively, and used to set up configuration files. All files result from CAPS models will be used in building CupCarbon project.

As depicted in Fig. 1, the framework has four activities: parsing, analyzing, script generation and project generation. These activities are detailed in the following sections.

3.1 Parsing

In this activity we use as input the SAML model that is typically stored in XMI file. This file encompasses a lot of information that are not needed for code generation. Thus, this information needs to be filtered out in order to acquire a relevant subset of SAML model values that conforms to SAML meta model. These values are used to instantiate templates. Templates and filtering pattern is one of several patterns that are usually followed in implementing code generators, as described in [17].

The parser is in charge of parsing the SAML XMI file to get all the information needed based on templates definitions. Templates are defined based on SAML meta model, and there are different templates defined for different SAML sub models. For example, there are templates for Software Architecture elements (SAML components and connectors), ports, modes and data. This parser, will create an object of software architecture, that in turn carries the SAML model description in java.

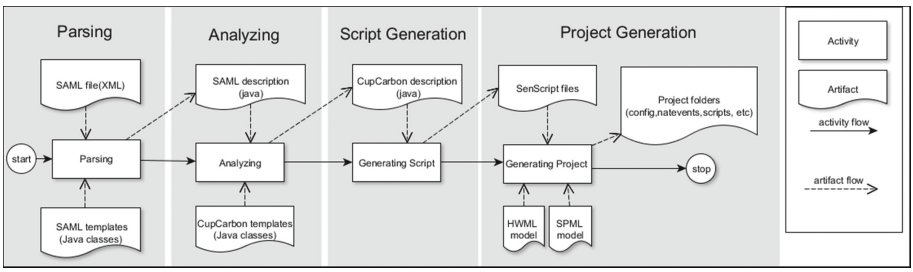


Fig. 1. CAPS automatic code generation framework

Accordingly, the resulting software architecture object contains the structural and behavioral part of the model. For the structural part, it contains the declarations of components, their data, their ports (as defined in the SAML meta model). Moreover, according to the definition, each port can be an IN message port or an OUT message port. The IN port defines APIs for messages to be received from other components. The OUT port defines a set of APIs for messages to be sent for other components [18].

For the behavioral part, each component has a group of modes, or states. The mode is represented as a method that carries the name of the mode. The mode method can contain if statements that perform logical tests. If a test evaluates to true, the mode method performs an action. The action can change variables values defined in the component scope and/or can call another mode method. Exit and entry modes are represented by calling methods of different types. Furthermore, this behavioral part of the code generator provides helpers to navigate through the diverse SAML metaclasses that are responsible for modeling the behavior.

The assumption we have in this work, that, each component in CAPS must be built to have at most one sensing unit (humidity sensor, or temperature sensor, or etc.). The reason of this assumption is, in the current state, CupCarbon is able to manage one sensing unit inside its sensor node. Hence, CupCarbon sensor node can contain many radio modules, a battery and a sensing unit. As it will be shown later, the component in CAPS will be represented as a sensor node in CupCarbon. More complex representation for several sensors in CAPS component will be handled in the future.

3.2 Analyzing

This activity takes as an input the software architecture object and the SAML model description in java, that result from the previous parsing activity. The input must be transformed into CupCarbon compatible information, in order to build CupCarbon Senscript files.

```
public class Component extends CupCarbonElement{

    private ArrayList<Integer> connectedCommponents;
    private SensingUnit  sensingUnit;
    private int numberOfAnalog=0;
    private String name;
    private int device_id;
    private int deviceType;
    private double device_longitude;
    private double device_latitude;
    private double device_elevation;
    private double device_radius;
    private int device_hide;
    private boolean device_draw_battery;
    private double device_sensor_unit_radius;
    private String device_script_file_name;
    private String scriptfile;
    public Component() {
        // TODO Auto-generated constructor stub
        setConnectedCommponents(new ArrayList<Integer>());
        ...
    }
  ......

}
```

(a) Part of CupCarbon template

```
public class ModelAnalyzer {

    public static List<CupCarbonElement> Components = new ArrayList<CupCarbonElement>();
    public static int ComponentId = 0;
    public static int SensorId=0;
    public static int EventId=0;

    public static boolean analyzeModel(SoftwareArchitecture sa) {
        for (SAElement element : sa.getElements()) {
            analyzeElement(element);
        }
        for (CupCarbonElement comp : Components) {
            analyzeConnection(comp);
        }
        return true;
    }
  .......
}
```

(b) Part of CupCarbon Analyzer

Fig. 2. Part of CupCarbon template and analyzer

The Analyzer (ModelAnalyzer) depicted in Fig. 2(b), represents the core section in code generation. The method *analyzeModel* takes the Software Architecture object as a parameter and analyzes each element to create the corresponding objects that will be used later for generating CupCarbon sensors and scripts. The CupCarbon template depicted in Fig. 2(a) represents the structure of CupCarbon element that will be used in this conversion.

The Analyzer is composed of two main parts:

(1) **Element Analyzer:** this part of the code (line 10 in Fig. 2(b)), takes an *SAElement* object as a parameter (that retrieves the parsed SAML software architecture elements) and then checks each element type and creates new objects that will represent the element in CupCarbon. For example if the element is a component, it will create an object from the Component class. If the element is a connection, it will create an object from the Connection class. Another important task of this part, is the mode analyzer. It is responsible for analyzing each mode in a component.

Modes (that were represented as methods) and mode transitions (represented as calling methods), will be interpreted as a group of *while* and *if* statements (a Senscript code, that result from a component of two modes, is described in Sect. 4, Fig. 5). The behavioral elements (event-condition-action) contained in the mode will be interpreted into other objects (composite objects) for a component. Thus, the result from this step, is a list of CupCarbon components, each CupCarbon component (CupCarbon element) has a sensing unit, radio module parameters and instructions that will be used in the creation of SenScript file for this CupCarbon component.

(2) **Connection Analyzer:** this part of the code (line 13 in Fig. 2(b)), is used to check the connection between components that already have been generated, in the previous part, and deciding their target and source components (connected components). Moreover, it denotes connections that occur between two components on the same platform or separated by a network, and it can decide how messages are controlled when they are moving between two components on different communication channels. The communication over these connections can be synchronous and asynchronous [19]. Asynchronous communication is acquired through using buffers (declaring communication array), while synchronous communication is acquired through using the buffer and setting its size to 0. For the sake of simplicity, the communication on the connection is made unidirectional and one communication is used for each different exchanged message. The buffer size is set into 1 in asynchronous communication, but during the code generation, the user can change this size of the buffer depending on the requirements of the application to be analyzed.

Finally, the result of analyzing activity is a list of CupCarbon components defined with their connections. The object that carries this information is shown in Line 3 on Fig. 2(b). These objects represent CupCarbon elements description in java, and will be an entry to the script generation activity.

3.3 Generating Script

This activity takes as an input the list of CupCarbon components objects produced from the analyzing phase. This activity is in charge of preparing the Senscripts code. A Senscript file for each component will be produced.

Part of the code responsible of Senscript generation is shown in Fig. 3. Figure 3(a), Lines 5, 6, 7, and 8, show how each component in the list of CubCarbonElement is sent to SenscriptGenerator. Figure 3(b), Line 10, shows the generateSenscript method that receives the component. This method uses several helpers and command instructions to translate the content of component object into Senscript code. In Fig. 3(c) an example of helper methods that translate read and write variables statements are shown.

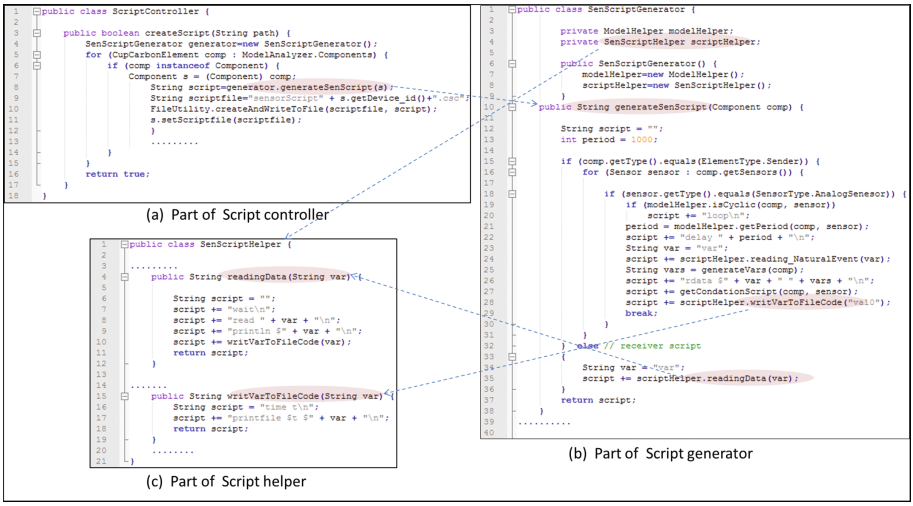


Fig. 3. Part of generating senscripts code

The script controller in Fig. 3(a), is responsible for putting all the translated statements for each component in a distinct Senscript file.

For each component, the script controller depicted in Fig. 3(a), is in charge of storing all the code translated for a component into a senscript file (.csc). The name of the file is distinguished by the device id (component id).

Thus, the result of this activity is a group of SenScripts files that represent all the components in the CupCarbon simulator (the CupCarbon component is represented as a CubCarbon node in the CupCarbon simulator).

3.4 Generating Project

This activity is responsible for creating files readable by the CupCarbon simulator. These files include configuration parameters and radio modules needed

to run a CupCarbon project. Examples of device parameters are: device type, device id, device longitude, device latitude, and device elevation. Examples of device radio modules are: radio standard, radio radius, and radio data rate [15].

The location parameters for each device (device longitude, device latitude, device elevation) will be automatically extracted from the SPML model (XMI file). Other parameters and radio modules will be extracted from the HWML model (XMI file). Then, an automatic generation of parameters and radio modules files are created. Each device has parameters file and a radio module file. An example of parameters and radio modules extracted from these models into CupCarbon files are shown in Figs. 6 and 7 in Sect. 4.

Parameters and radio modules files, along with the generated Senscript files, will form the final project to be loaded in the CupCarbon simulator workspace.

Finally, after loading the project and before running the simulation process on CupCarbon, few more configurations are needed to be done through its interface. First, set some device (node) parameters through its interface (like, setting up the *energy max* value that represents the initial energy of the battery). Second, create natural events for each sensing unit. This in turn will create a natural event file for each sensing unit and store it in the natevent folder in the project workspace.

4 Application of CAPS Models, Code Generation and Simulation to the NdR Case Study

In this section, we will simply represent a partial example of NdR case study introduced in Sect. 2. A simple scenario describes the monitoring of people count and oxygen level in a NdR room will be introduced. We will show its SAML model that plays a major role in SenScript code generation. We will show the needed part of its related HWML and SPML models that will be transfered to CupCarbon configuration files.

Figure 4 shows the SAML model of the CAPS. It is important to note that this figure is actually a screen-shot of our CAPS tool [20]. From a structural point of view, the shown SiA-CPS model is composed of five main components; OxygenSensor, RoomPeopleCounter, RoomController, EntranceLockActuator, WindowsLockActuator, and Server.

The OxygenSensor component is responsible for monitoring the Oxygen breathing percentage in a room. It includes two modes: (1) Normal mode: in this mode the oxygen sensor reads Oxygen breathing percentage (O2) in a room every 100 s. A timer is set in this mode to schedule the reading from the oxygen sensor. A message carrying the O2 value is sent from the output message port of the OxygenSensor component to the in port of the RoomController component. Moreover, if the reading of O2 is less than 0.19 that means the state of the room will enter the critical mode. (2) Critical mode: in this mode the oxygen sensor reads Oxygen breathing percentage (O2) in a room every 1 s, since this mode indicates the unsafe level of oxygen. Also in this mode, a timer is set to schedule the reading from the oxygen sensor. A message carrying the O2 value is sent

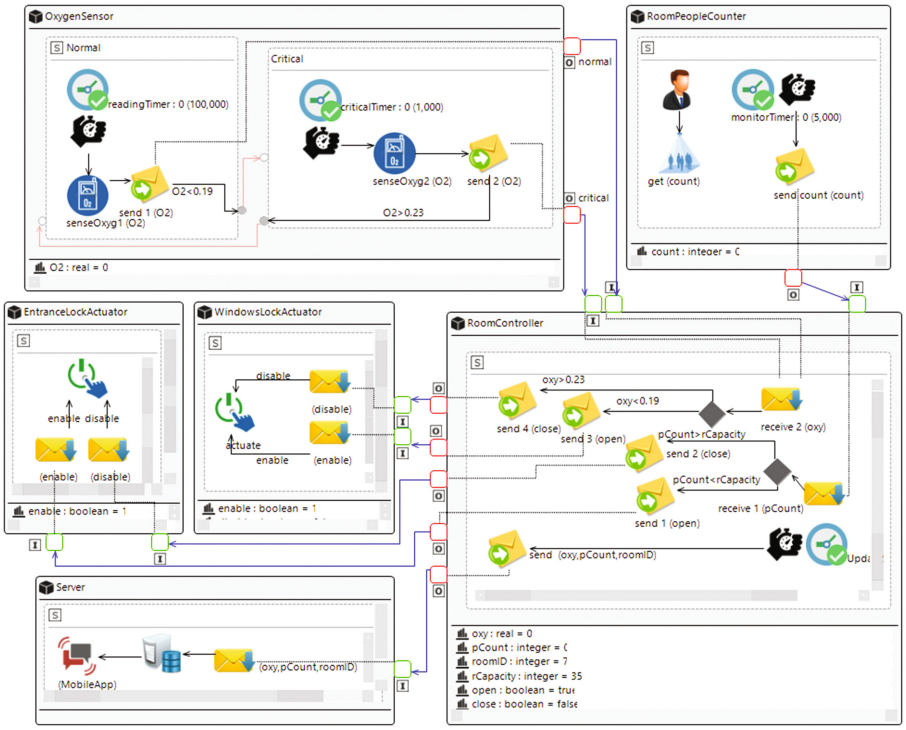


Fig. 4. The Software architecture of simple scenario in NdR case study

from the output message port of the OxygenSensor component to the in port of the RoomController component. Moreover, if the reading of O2 is more than 0.23 that means the state of the room will go back to the normal mode.

The RoomPeopleCounter component is responsible for counting the people in a room. This counter work all the time to count people in a room. But, it updates this information to the RoomController only every 5 s. When a timer is fired, it sends a message containing the people count in a room (count) from the out port of the RoomPeopleCounter component to the in port of the RoomController component.

The RoomController component is responsible for receiving sensors data in a room and take decisions based on its values. The decisions in this example are related to send a control messages to the actuators to open and/or close, windows and doors. The description shown in RoomController component is a simplified version of its supposed work. In this example, RoomController component receives two types of messages from its out ports:

(1) Message contains Oxygen breathing percentage: if the received value is less than 0.19, this component sends through its out port a message to the in port of the WindowsLockActuator component. This message contains a Boolean variable (open), the value of this variable is set to true. Otherwise, if the

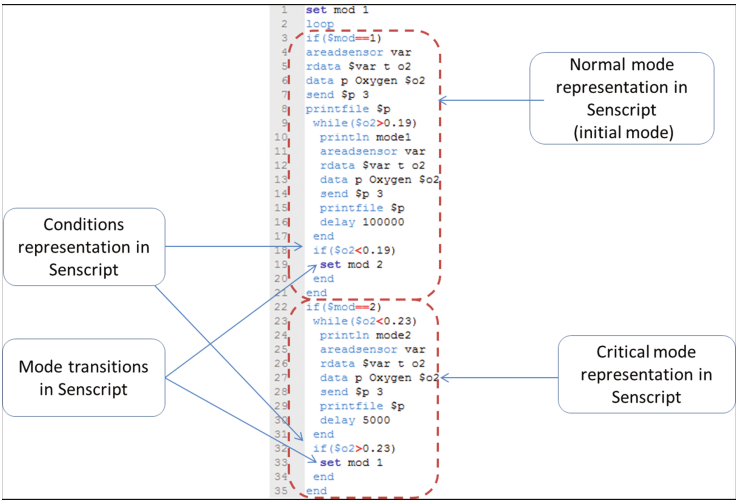


Fig. 5. SenScript generated by CAPS code generator for OxygenSensor component

received value is more than 0.23, this component sends through its out port a message to the in port of the WindowsLockActuator component. This message contains a Boolean variable (close), the value of this variable is set to false.

(2) Message contains people count: if the received value is less than room capacity (rCapacity = 35), this component sends through its out port a message to the in port of the EntranceLockActuator component. This message contains a Boolean variable (open), the value of this variable is true. Otherwise, if the received value is more than rCapacity, this component sends through its out port a message to the in port of the EntranceLockActuator component. This message contains a Boolean variable (close), the value of this variable is false.

The RoomController component is in charge of sending a message contains the values of the Oxygen (oxy), people counter (pCounter), and ID of a room (roomID) to the Server, in port, from the out port of the RoomController.

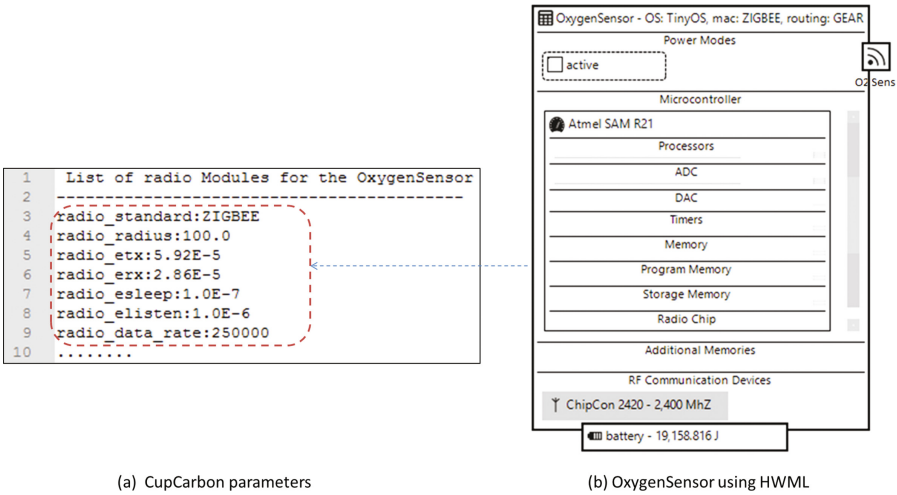
WindowsLockActuator component is the one responsible for performing the correct action of opening or closing the windows. If it receives from its in port a message containing a true value coming from the out port of the RoomController component, it enables the actuator and thus it opens the windows. Otherwise, if the value received from the out port of the RoomController component, to its in port, containing a false value, the actuator closes the windows.

EntranceLockActuator component is liable for performing the correct action of opening or closing the doors. If it receives from its in port a message containing o true value coming from the out port of the RoomController

component, it enables the actuator and thus it opens the doors. Otherwise, if the value received from the out port of the RoomController component, to its in port, containing a false value, the actuator closes the doors.

Finally, Server component is liable of processing the different data received through its in ports from other components. In this example, we restrict the Server component responsibility in sending updates to the users running NdR mobile application. This update indicates the rooms state if they are full or not, depending on the roomID and pCount received from out port of RoomControllers.

For the sake of space, we show in figure 5 the code generated by CAPS generator for OxygenSensor component. Line 3 and Line 20 represent the normal and critical mode respectively. Line 14 and Line 27 represent the timers in the normal and critical modes, respectively. Line 4, 10 and 23 represent the instructions of reading the current oxygen value from the oxygen sensor.



(a) CupCarbon parameters (b) OxygenSensor using HWML

Fig. 6. An Example of HWML model and its representation in CupCarbon

According to the HWML model, we take the OxygenSensor component as an example to show the configuration information needed for it in CupCarbon simulator. Figure 6(b), shows part of HWML model that represents hardware specifications for OxygenSensor component in CAPS. This OxygenSensor is equipped with O2 sensor for sensing the percentage of oxygen in a room. It uses Texas Instruments ChipCon 2420 RF transceiver and it uses batteries of two AA with up to 19159 Joules. The sensor radio standard is 802.15.4 and radio radius is 20. Figure 6(a), shows a screenshot for an Oxygen Sensor radio modules configuration file.

Referring to SPML model. The example, depicted in Fig. 7, represents part of the physical environment of our NdR scenario. Figure 7(b), shows part of the

SPML model that describes the deployment physical position of OxygenSensor device. Figure 7(a), shows CupCarbon parameters needed from the OxygenSensor SPML model, these parameters are: device longitude (x), device latitude (y) and device elevation (Elevation).

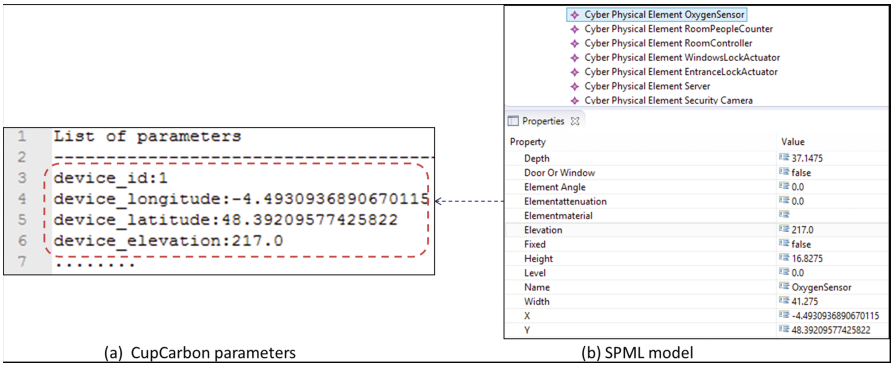


Fig. 7. An Example of SPML model and its representation in CupCarbon

In addition to the described models, there are two auxiliary models: (i) MAPML is Mapping Modeling Language used to link SAML and HWML, (ii) DEPML is Deployment Modeling Language (DEPML) used to link SAML and SPML. For the sake of simplicity, they are not described here. For better details about these auxiliary models, please see [11].

5 Results

In this section we will describe the results of running our project in CupCarbon simulator. We applied three different behaviors for the NdR case study example. These behaviors are, first, the OxygenSensor runs with normal mode only. Second, the OxygenSensor runs with critical mode only. Third, the OxygenSensor runs in normal and critical modes, that is originally described in Sect. 4. These three different behaviors are applied in our CAPS modeling framework tool. Then, we used our CAPS automatic code generation framework tool to produce three CupCarbon projects. After all, we ran the simulator under the three behaviors and compare data traffic, energy consumption and battery level of it nodes.

For all simulation experimentations, we fixed simulation time to 6000 s, and energy max for all nodes to 19159 J. For OxygenSensor natural events, we selected random generation between 0.10–0.35. For people counter natural events, we selected random generation between 15–45. The Fig. 8, shows a screen shot of running the NdR project (applying third behavior) on CupCarbon simulator.

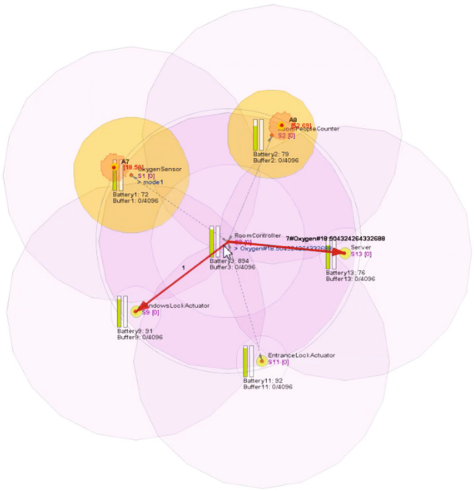


Fig. 8. Running NdR project on CupCarbon simulator

Table 1. Messages exchanged in components during simulation

Component ID	Component Name	# of sent messages			# of received messages			Data traffic in KB		
		Normal mode	Critical mode	Normal+ Critical	Normal mode	Critical mode	Normal+ Critical	Normal mode	Critical mode	Normal+ Critical
S1	OxygenSensor	81	1459	123	0	0	0	3	40	4
S2	RoomPeopleCounter	151	151	151	0	0	0	5	5	5
S3	RoomController	403	2628	460	232	1610	274	17	97	19
S9	WindowsLockActuator	0	0	0	55	902	70	1	4	1
S11	EntranceLockActuator	0	0	0	117	117	117	1	1	1
S13	Server	0	0	0	231	1609	273	7	47	8

In all the behaviors applied, the OxygenSensor and RoomPeopleCounter components are always sending messages and they don't receive any. While, WindowsLockActuator, EntranceLockActuator and server components are always receiving messages and they don't send any. But, In RoomController, the traffic is in both direction, it sends and receives messages. This explains some zero values appear in the table depicted in Table 1, this table shows the exchanged messages through the IN and OUT ports of the components during running the three behaviors in CupCarbon, it also shows the data traffic in Kilo Bytes that occur at each component.

Further, from Table 1, we conclude that the data traffic of OxygenSensor, RoomController, and server node receive the highest traffic when we run the critical mode behavior. This is due to the high messages they exchange during this mode. The normal mode receives a low amount of traffic but it could be not safe enough for detecting Oxygen level in a room. Thus, we can notice that using

the critical and normal modes give also a low range of data traffic compared to using only critical mode. Moreover, using critical and normal modes together is still a safe behavior. Thus, this proves how small changes in the architecture can affect on the efficiency concern.

According to the battery levels and power consumption. The Figs. 9(a), (b) and (c), show the battery level when running the simulator under the three behaviors. The Figs. 9(e), (d) and (f), show the energy consumption when running the simulator under the three behaviors. From these figures, if we want to consider for example the two nodes, OxygenSensor (S1 in blue) and RoomController (S3 in red), we notice that S3 received the highest battery level drain when running the three modes. That is expected since the controller receives the highest data traffic among them. A minor improvement is noticed on RoomController when running the two modes together. While, for OxygenSensor we can notice how it experience the lowest battery level drain when it runs only the normal mode, the highest drain is when it runs critical mode, but we can also notice that running the two modes together gain better battery level improvements than critical mode. If we notice also the same nodes in the energy consumption charts, we can see how running critical and normal modes together shows a good improvements comparing to running only critical mode. Moreover, we notice how the energy consumption in normal mode is close to energy consumption in normal and critical modes together.

Therefore, to recognize the tradeoff between safety, energy and data traffic concerns. We notice that using only normal mode achieves energy efficiency by

Fig. 9. CupCarbon results for battery level and power consumption in different modes (Color figure online)

having the least data traffic. The energy efficiency is inversely proportional to the data traffic size (number of messages exchanged). However, in normal mode, the safety concern will be less than it in critical mode. Therefore, using normal mode with critical mode provide a better compromise between those concerns.

Our simulation results show that by using CAPS modeling framework, CAPS code generation framework, and CupCarbon simulation for SiA-CPS, we can get the early evaluation for energy and data traffic savings while developing SiA-CPS. This vision of testing the architecture in the early stage will improve the process of architecting such systems.

According to the real experimentation, it is a work that we are planning to do. It requires a long process of deploying sensors and actuators. We will use the arduino code, that result from cubcarbon, to be installed in real sensors. The Ndr event runs in L'Aquila every year in September, so our first experimentation is supposed to be in September 2017.

6 Conclusions and Future Work

In this paper we proposed a modelling platform supported by a code generation framework, and integrated with CupCarbon simulation for engineering situational-aware cyber-physical systems.

This frameworks allows engineers to run a trade-off analysis between energy consumption and performance indices like sensor nodes throughput, reliability and network latency. The modeling framework, automatic code generation, and simulations represent only the starting point of a series of goals we are willing to achieve in the mid-term.

The code generation framework functioning and simulation has been tested by applying an energy and data traffic-related simulation of the NdR SiA-CPS.

In future work, we plan to make these frameworks used by practitioners involved in the development of SiA-CPS. We wish to record and analyze their usage patterns and collect their feedback for further improvements. We are also working to realize an analysis that, while getting in input a series of environmental configurations options, can tell us which configuration can increase the network life-time.

Moreover, we will expand in the near future the Ndr case study to real experimentation that includes different architectures of massive size and real events. Thus, the real evaluation will be compared to the result from the simulator and this will clearly state the efficiency of our work.

References

1. ISO/IEC/IEEE: ISO/IEC/IEEE 42010:2011 Systems and software engineering - Architecture description (2011)
2. Kruchten, P.B.: The 4+1 view model of architecture. IEEE Software **12**(6), 42–50 (1995)

3. Rozanski, N., Woods, E.: Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives. Addison-Wesley Professional, Boston (2005)
4. Kruchten, P., Capilla, R., Dueñas, J.C.: The decision view's role in software architecture practice. IEEE Softw. **26**(2), 36–42 (2009)
5. Emery, D., Hilliard, R.: Every architecture description needs a framework: expressing architecture frameworks using ISO/IEC 42010. In: WICSA/ECSA 2009 (2009)
6. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, Upper Saddle River (1996)
7. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. SIGSOFT Softw. Eng. Notes **17**(4), 40–52 (1992)
8. Jajodia, S., Liu, P., Swarup, V., Wang, C.: Cyber Situational Awareness, vol. 14. Springer, Heidelberg (2010)
9. Malavolta, I., Muccini, H., Sharaf, M.: A preliminary study on architecting cyber-physical systems. In: Proceedings of the 2015 European Conference on Software Architecture Workshops, vol. 20. ACM (2015)
10. Muccini, H., Sharaf, M., Weyns, D.: Self-adaptation for cyber-physical systems: a systematic literature review. In: Proceedings of the 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, pp. 75–81. ACM (2016)
11. Muccini, H., Sharaf, M.: Caps: architecture description of situational aware cyber physical systems. In: 2017 IEEE International Conference on Software Architecture (ICSA), pp. 211–220. IEEE (2017)
12. Menon, V., Jayaraman, B., Govindaraju, V.: The three rs of cyberphysical spaces. Computer **44**(9), 73–79 (2011)
13. Malavolta, I., Muccini, H.: A survey on the specification of the physical environment of wireless sensor networks. In: 2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications, pp. 245–253 (2014)
14. Tsigkanos, C., Kehrer, T., Ghezzi, C.: Architecting dynamic cyber-physical spaces. Computing **98**(10), 1011–1040 (2016)
15. Bounceur, A.: Cupcarbon: a new platform for designing and simulating smart-city and IOT wireless sensor networks (SCI-WSN). In: Proceedings of the International Conference on Internet of things and Cloud Computing, p. 1. ACM (2016)
16. Crnkovic, I., Malavolta, I., Muccini, H., Sharaf, M.: On the use of component-based principles and practices for architecting cyber-physical systems. In: 2016 19th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE), pp. 23–32 (2016)
17. Voelter, M.: A catalog of patterns for program generation. In: EuroPLoP, pp. 285–320 (2003)
18. Bucchiarone, A., Di Ruscio, D., Muccini, H., Pelliccione, P.: From requirements to code: an architecture-centric approach for producing quality systems. arXiv preprint arXiv:0910.0493 (2009)
19. Pelliccione, P., Inverardi, P., Muccini, H.: Charmy: a framework for designing and verifying architectural specifications. IEEE Trans. Softw. Eng. **35**(3), 325–346 (2009)
20. Muccini, H., Sharaf, M.: Caps: a tool for architecting situational-aware cyber-physical systems. In: 2017 IEEE International Conference on Software Architecture (ICSA). IEEE (2017)