Software Development Support for Shared Sensing Infrastructures: A Generative and Dynamic Approach

Cyril Cecchinel, Sébastien Mosser, Philippe Collet

Software Development Support for Shared Sensing Infrastructures: a Generative and Dynamic Approach

Cyril Cecchinel, Sébastien Mosser, and Philippe Collet

Université Nice Sophia Antipolis, I3S, UMR 7271, 06900 Sophia Antipolis, France
CNRS, I3S, UMR 7271, 06900 Sophia Antipolis, France
{cecchine,mosser,collet}@i3s.unice.fr

Abstract. Sensors networks are the backbone of large sensing infrastructures such as *Smart Cities* or *Smart Buildings*. Classical approaches suffer from several limitations hampering developers' work (*e.g.*, lack of sensor sharing, lack of dynamicity in data collection policies, need to dig inside big data sets, absence of reuse between implementation platforms). This paper presents a tooled approach that tackles these issues. It couples *(i)* an abstract model of developers' requirements in a given infrastructure to *(ii)* timed automata and code generation techniques, to support the efficient deployment of reusable data collection policies on different infrastructures. The approach has been validated on several real-world scenarios and is currently experimented on an academic campus.

Keywords: Sensor Network, Software Composition, Modeling.

1 Introduction

The *Internet of Things* [13] relies on physical objects interconnected between each others, creating a mesh of devices producing information flow. The Gartner group predicts up to 26 billions of things connected to the Internet by 2020. These *things* are organized into sensor networks deployed in *Large-scale Sensing Infrastructure*s (LSIs), *e.g.*, *Smart Cities* or *Smart Buildings*, which continuously collect data about our environment. These LSIs implement *Cyber Physical System*s (CPSs) that monitor ambiant environments.

Facing the problem of managing tremendous amounts of data, a commonly used approach is to rely on sensor pooling [9, 19] and to push data collected by sensors in a central cloud-based platform [15]. Consequently, sensors cannot be exploited at the same time and one needs to rely on data mining solutions to extract and exploit relevant data according to usage scenarios [1, 17]. This approach is adapted for many scenarios where data mining techniques are required, and has the advantage of separating concerns of data collection from data exploitation. Nevertheless, there are many real-life case studies and scenarios where developers need to exploit shared LSIs and implement a diversity of applications that do not need data mining expertise [4]. In this context, the

cloud servers create *de facto* silos that isolate datasets from each others and act as a centralized bottleneck. In addition, the computation capabilities of the other layers of the LSI (*e.g.*, the micro-controllers used to pilot the sensors at the hardware level, or the nano-computers acting as network bridges to connect a local sensor infrastructure to the Internet) are under-exploited [10].

To develop software that fully exploits a LSI, the infrastructure must be considered as a white box. But the developer tasks is then more complex as they have to deal with tedious low-level details of implementation out of their main business concerns. This assumes a deep knowledge of micro-controller (sensor platforms) and nano-computer (bridges) programming [4], while a diversity of technological platforms must be handled. In such a situation, programming at the higher level of abstraction and reusing as much code as possible between scenarios and LSIs is of crucial importance. Moreover developers also have to deal with the sharing aspects. It is hard for them as new requirements must be enacted on the LSI and may easily interfere with the other ones.

In this paper, we propose a tooled approach that tackles all these problems and aims at improving reuse, supporting sharing and dynamic data collection policies. A software framework enables developers to specify and program at an appropriate level their sensor exploitation code. It relies on an abstract model of developers' requirements in a given infrastructure so that timed automata and code generation techniques can be combined to support the efficient deployment of data collection policies into a LSI. As a result, several applications can rely on the same sensors, and thus share them. A given code can be reused, translated and deployed on different infrastructures. The framework also ensures that multiple policies can be dynamically composed, so that the generated code automatically handle all requirements and get only relevant data to each consumer.

The remainder of this paper is organized as follows. In Sec. 2, we describe the motivations of our work by introducing a real-life case study and organizing requirements. We present in Sec. 3 the foundations of our framework. In Sec. 4 we assess our approach, providing an illustration, discussing current applications and validating the identified requirements. Sec. 5 discusses related work while Sec. 6 concludes this paper and describes future work.

2 Motivations

In this section, we motivate our work by first introducing the SmartCampus project, then by defining requirements for a development support for shared LSIs. SmartCampus has been deployed on the SophiaTech campus[1] of the University of Nice, located in the Sophia Antipolis technology park. We also introduce a running example extracted from SmartCampus.

2.1 The SmartCampus Project

The SmartCampus project is a prototypical example of an LSI [4]. It acts as an open platform to enable final users (*i.e.*, students, teaching and administrative staff) to build their own innovative services on top of the collected (open)

[1] `http://campus.sophiatech.fr/en/index.php`

data. This project is exactly the class of LSI this work is addressing as it faces the following issues : *(i)* it is not possible to store all the collected data in a big data approach and *(ii)* even if one can afford to store all these data, the targeted developers do not master data mining techniques to properly exploit it. Typically, pieces of software deployed in SmartCampus are driven by functional requirements (*e.g.*, "where to park my car?") and leverage a subset of the available sensors (here the parking lot occupation sensors) to address these requirements. Contrarily to classical systems that use sensor pooling [9, 19], different applications, such as the parking place locator and an emergency system assessing the availability of fire brigade access in the parking lots, rely on the same set of sensors at the very same time. Moreover, developers do not know the kind of hardware deployed physically in the buildings. To support software reuse across different architectures, there is a need to abstract the complexity of using such heterogeneous sensor networks at the proper level of abstraction.

2.2 Supporting Shared Sensing Infrastructure

From the functional analysis of the SmartCampus project (ended in December 2013), we highlighted four requirements with respect to software reuse [4]. These requirements are not only project specific, but also do apply to any IoT-based platform needing to share its sensors on a large scale, *e.g.*, LSIs. This class of system include upcoming smart-building or smart cities wishing to aggregate communities of users to leverage sensors-based system and produce innovative services based on citizen needs.

Pooling and Sharing (R_1). Classical systems rely on sensor polling, with corresponding booking policies. Typical examples of such systems are, for example, the IoT lab platform in France, containing 2,700 sensors deployed in 5 research centers across the country for experiments [9], or the Santander smart city with a closed set of IoT based scenarios [19]. Users book a subset of sensors, work with it and release it afterwards. This setup does not match what is expected in the SmartCampus context. Moreover sensors available in an LSI are classically available through pooling mechanisms [12, 21]. On the one hand, this mechanism is useful when sensors are installed to match a particular setup and deliver a single service. But on the other hand, it is completely irrelevant to the set of scenarios defined by our class of application. Providing a system that only support sharing is also irrelevant, as one needs to deploy critical pieces of software to sensors and be assured that such a critical process will be isolated from other processes (this is similar to the virtual machine isolation requirement in cloud computing).

Yield only relevant data (R_2). To support sharing, classical architectures actually collect data at the minimal available frequency, and store the complete dataset in a cloud-based system, like in Xively [15]. Then, data mining techniques are used to recompute the relevant data from this complete dataset [1, 17]. This faked sharing leads to two type of issues: *(i)* application developers must be aware of the data mining paradigm instead of focusing on their system and *(ii)* as their is

no model of what data was expected by the application, it is impossible to reuse the code written for a given LSI into another one. It is worth to note that by yielding only relevant data, developers who want to express mining scenarios will simply define as relevant a wider set of data than classical developers, making the two approaches non-exclusive.

Dynamically support data collection policies (R_3). There is a need to model what kind of data are expected by a given application, in a data collection policy. From a developer perspective, this requirement is critical as it is not reasonable to program specifically for each LSI addressed by the same application, utterly preventing software reuse. In addition applications exploiting the sensors require to change their policies, for example by temporarily increasing some collection frequency during a short period of time. These policies must be enacted dynamically on the LSI to support such changes. Working with a formal definition of what data are expected by the different consumers is the entry point to apply verification and validation techniques on LSIs.

Handling the infrastructure diversity (R_4). As the state of the art relies on artifacts defined at the code level, it is difficult if not impossible to support software reuse across different LSIs. Several approaches leverage operating systems techniques to provide a standard way to program sensors (*e.g.*, TinyOS [14], Contiki [8]). However, these operating systems must be supported by the available sensors. If not, it is up to the developer to manually translate the code from one system to another. In addition, even in the same LSI, hardware obsolescence requires to replace old sensors by new one, often with new hardware due to sensor production life cycle [3]. Thus, it is critical to operate at a code-independent level to express data collection policies.

Summary. Classical approaches are deporting all the sensing intelligence to a cloud-based solution, under-exploiting the computation capabilities of the other layers of the LSI [10]. This also floods the cloud storage with irrelevant data. Even with the help of data mining solutions, the resulting architectures are either centralized with important overhead, or distributed, but still based on an inflexible mining pipeline with identified bottlenecks [17]. Thus, based on the analysis made in the SmartCampus, there is no solution covering all the four requirements expressed by the project. To some extent, this is not surprising. These approaches rely on a strong hypothesis of single consumer and very limited access to the sensors. Thus, only half of R_1 (*i.e.*, pooling) and half of R_2 (*i.e.*, mining) are covered. The two last requirements R_3 & R_4 are covered at the code level, preventing reuse and making maintenance and evolution complex. Considering the emerging class of system that targets communities of developers such as the Fireball project[2], the previous assumptions does not hold anymore.

2.3 Running Example

We introduce here a running example to illustrate the approach. It is a simplification of the use cases identified in the SmartCampus's experimental LSI [4].

[2] `http://www.fireball4smartcities.eu/`

The SmartCampus implementation defines a CPS based on two layers: micro-controllers (sensors and sensor boards) and nano-computer (bridging the sensor network and the Internet). Sensor measurements are sent to a sensor board, which aggregates sensors physically connected to it. The board is usually implemented by a micro-controller that collects data and send them to its associated bridge. A bridge aggregates data coming from multiple sensor boards (thanks to radio or wire-based protocols) and broadcasts on the Internet the received streams to a data collection API, using classical Ethernet connection.

In our example we consider two users, Alice and Bob, who need to use the same sensors to build their own application. Alice develops an application exploiting the associated LSI by collecting data from a temperature sensor every couple of second. Without any specific support, she has to write *(i)* code to be enacted on the different micro-controllers linked to temperature sensors (an infinite loop measuring the temperature every 2 seconds), *(ii)* code to aggregates these data at the bridge level (reading the data sent by the micro-controllers in proprietary representations, and sending it to the cloud-based collector, after having performed data translation from micro-controller format to the collector one), and finally *(iii)* the code that exploits the collected data to implement her application. We claim here that only the latter should be Alice's concern. On his side, Bob develops an application exploiting a temperature sensor each second and a humidity sensor every three seconds. He needs to perform the same kinds of actions as Alice : *(i)* writing the code that reads temperature sensors each second and humidity sensors every 3 seconds, *(ii)* aggregating these data at the bridge level and *(iii)* implementing his application exploiting the collected data.

As simple as this example is, it illustrates the identified requirements. Both users will need to use the same temperature sensor (R_1), and as they have different usages of this sensor, we do not want them to be flooded with non-desirable data (R_2). As the sensor network evolves and has to support new users (*i.e.*, the arrival of Bob), it needs to dynamically adapt the data collection policies (R_3). Finally, as the sensor networks is going to be heterogeneous and composed of different layers (*i.e.*, collection and network), the produced code must automatically fit the infrastructure (R_4).

3 Contribution: The COSmIC Framework

To address the four identified requirements, we propose the *COSmIC* framework, a set of *Composition Operators for Sensing InfrastruCtures*. This section describes the foundations underlying the framework.

3.1 Data Collection Policies as Timed Automata

In sensor networks, automata are commonly used for protocol modeling, and component model approaches [23] are used to develop embedded applications, focusing on the definition of *Interface Automaton* between each components. These automaton-based interfaces enable the different components. We propose to leverage this representation to *(i)* model the data collection policy expressed

by the developer, implementing what she expects from an LSI through code generation *(ii)* compose and decompose (dynamically) these policies to handle sharing and infrastructure diversity.

We define a data collection policy $p = \langle Q, \delta, q_0 \rangle$ as a simplification of a classical timed automaton. Q is the set of states defined by the automaton, $\delta : Q \to Q$, its (deterministic) transition function from a given state to another one and finally q_0 its initial state. In real LSIs, tick period is rarely lesser than one second, thus our model assumes a single logical clock that triggers a transition each second. As a policy aims to be indefinitely executed on an LSI, it must be cyclic, and the length of the cycle represents the period of p, denoted as P_p. A given state $q \in Q$ contains an ordered set of actions A implementing the way the developer interacts with the LSI.

In our example, the corresponding policy for Alice p_a is represented by an automaton (depicted in FIG. 1) with two states $\{a_1, a_2\}$.

As a policy needs to be enacted on different platforms, user requirements are translated into a set of basic operations:

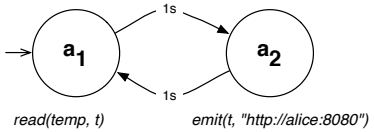
- *read*: Read the value of a *sensor*, *e.g.*, for actions used in our temperature and humidity example.
- *emit*: Send a value to an external *endpoint*, which is usually implemented as a Web service exposing a destination URL for the collected data.

According to this representation and the associated actions, a software developer is able to model what she expects from the LSI for her given use case. The key point is that the developer is completely unaware of the internal implementation of the LSI, and only focuses on reading sensor values and sending data over communication interfaces.

3.2 The Generator Operator (γ)

The designed models are useless if not coupled to code generation algorithms that transform these logical representations into executable code. One of the main issues to tackle here is the variability existing between the different hardware elements that compose an LSI (R_4). For example, at the micro-controller level, a plain Arduino board does not support the *emit* action, whereas an Arduino coupled to an Ethernet communication shield supports it.

We thus define a code generator γ as a couple of functions (pre, do), each consuming as input a policy p. The *pre* function checks a set of preconditions on p to ensure that this policy can be projected to the hardware platform targeted



$$actions(a_1) = \{read(\text{temp}, t)\}$$
$$actions(a_2) = \{emit(t, \text{“http://alice:8080”})\}$$
$$\delta_a(x) \mapsto (x = a_1 \Rightarrow a_2) \vee (x = a_2 \Rightarrow a_1)$$
$$p_a = (\{a_1, a_2\}, \delta_a, a_1)$$

Fig. 1. Excerpt of a data collection policy

by p. The *do* function takes as input the policy to transform, as well as additional parameters given by the environment. These parameters map logical names to physical elements when relevant (*e.g.*, sensors on an Arduino platform are only identified by the pin number they are plugged in). In a production environment, the generators are not executed by the developer herself. She will only express her needs, and will enact them on the LSI which actually knows its internal infrastructure.

For example, we consider here the policy p_a defined in the previous section, and an Arduino platform as generation target. The corresponding precondition checker assesses the absence of *emit* actions in the following way:

$$pred_{ard}(p) \mapsto \forall q \in Q_b, \nexists emit(\_,\_) \in actions(q)$$

In this case, the checker detects that this policy cannot be deployed on an infrastructure based solely on Arduino, as this hardware does not match the requirements expressed in the policy, *i.e.*, emiting a data to the Internet.

Considering a policy p valid for the Arduino platform (the decomposition operator described in the next section shows how to make p_a valid), the do_{ard} function then visits p to produce the code to be executed on the board, using the Wiring[3] language as target. It also takes as input a *map* acting as a registry (stored in the LSI environment) binding a sensor name to the physical pin that connects it to the board. Lst. 1.1 shows the resulting code for a board coupled to temperature sensor *temp* on pin 9 and an humidity one *hum* on pin 10. The generator maps actions such as $read(temp, t)$ into Wiring code like `v_t = analogRead(9)`.

```
1  void setup() { // Initialization
2    pinMode(9,INPUT); pinMode(10,INPUT);
3  }
4  void a1() {
5    v_t = analogRead(9); v_h = analogRead(10);
6    delay(1000); return a2();
7  }
8  void a2() {
9    v_t = analogRead(9);
10   delay(1000); return a1();
11 }
12 void loop() { return a1(); } // Entry point
```

Listing 1.1. Generated code example: $do_{ard}(p)$

Code generators also allow users to reuse their policy for different sensing infrastructures as a given COSmIC policy can be translated to many targets.

3.3 The Decomposition Operator (δ)

Considering a given policy, it has to be decomposed into software artifacts that make sense on the different layers of the LSI, *i.e.*, the micro-controller and bridge layers. For each layer, there may be actions that are incompatible with the hardware. Consequently, a given policy p must be decomposed into n layer-specific sub-policies p'_{layer} (where n is the number of layers) that communicate together and where incompatible actions are substituted by internal communication [20].

[3] Wiring is an open-source framework for micro-controllers (`http://wiring.org.co/`).

This decomposition is performed thanks to a compatibility table T. A function $f_T(a, P)$ applied on this table returns a boolean value reflecting the compatibility of an action a on the platform P[4]. This decomposition process is defined as follows:

$$\delta(p) \equiv \forall a \in p, \forall P \in layers, f_T(a, P) \Rightarrow a \in p'_a$$

If we consider micro-controllers implemented by Arduino boards and bridges implemented by Raspberry nano-computers, the micro-controller level will not support the emission of data to external endpoints, due to a lack of proper communication interface. In our example, Alice's policy will be decomposed by the operator into two sub-policies: $\delta(p_a) = \{p'_{mic}, p'_{bri}\}$. Consequently, the p_{mic} will read the sensor value and send it to an internal endpoint (substitution of the *emit* action) thanks to the serial communication that links the sensor board to its bridge. This policy is accepted by the $pred_{ard}$ function defined in the Arduino code generator, meaning that p_{mic} can be deployed on such hardware. The p_{bri} policy is executed on the bridge to read the internal communication port and emit the received data to the external endpoint.

3.4 The Composition Operator ($\oplus$)

On a shared LSI, policies designed by different developers will be executed on the very same piece of hardware COSmIC provides a composition operator denoted as $\oplus$ at the automaton level. It composes two given policies and produces a single policy containing an automaton corresponding strictly to the parallel composition of the two inputs.

The $\oplus$-operator assimilates a timed automaton implementing a policy p with a period P_p as a periodic function. The composition of two periodic functions f_1 and f_2 is a periodic function $f = f_1 \circ f_2$ where its period P_f is the least common multiple of P_{f_1} and P_{f_2}. Applied to policies, this means that the composition of two policies p_1 and p_2, denoted as $p = p_1 \oplus p_2$ is a policy with a period $P_p = lcm(P_{p_1}, P_{p_2})$, where each actions of p_1 (respectively p_2) are executed according to P_{p_1} (respectively P_{p_2}). As this $\oplus$-operator is endogenous, it allows a software developer to dynamically reuse a policy by composing it with new incoming policies.

In our example, the policies defined by Alice in p_a and Bob in p_b will exploit the same temperature sensor on the same micro-controller and use the same bridge to emit their values. More details on the composition and decomposition processes are given in Sec. 4.2.

4 Assessment

In this section, we describe the current implementation and its application to our prototypical LSI. Then, we show how COSmIC can be used to model and deploy the running example. We validate our identified requirements through some acceptance criteria and finally discuss threats to validity.

[4] An example of such a table is given in Sec. 4.2.

4.1 Implementation and Application

The initial prototype of the COSmIC framework is available on GitHub[5]. It is implemented with the Scala language ($\sim$ 3500 lines of code) and covers all the concepts presented in Sec. 3. We are currently experimenting COSmIC on Arduino, Raspberry Pi and Cubieboard platforms as part of the SmartCampus project. We also used the FIT IoT-lab platform[6], featuring a pool of over 2700 sensors nodes spread across France, to experiment on the ARM Cortex M3 platform.

To experiment and demonstrate the abstraction of platforms, code generation capabilities, sharing and reuse, we have modeled four identified SmartCampus scenarios and then generated code for each platform type we experiment with:

- *S1 - Late worker detection*: at night, occupied offices are detected by checking if the light is on (*light sensor*) and if there is someone in the office (*presence sensor*);
- *S2 - Fire prevention*: a warning signal on a temperature threshold (*temperature sensor*);
- *S3 - Heat monitoring*: air-conditioning and heating are controlled by checking the ambient air in buildings (*temperature sensor*);
- *S4 - Energy wasting*: To comply with environmental standards, the quality manager wants to monitor light kept on when the building is empty (*light sensor* and *presence sensor*).

Table 1 presents the number of lines of code (LoC) generated for each platform. Every code generator includes a static overhead (template code), specific to the targeted platform. This template provides the implementation of methods called by the COSmIC code generation. The corresponding LoC (italic row on TAB. 1) vary between platforms as some of them are providing more features. The Raspberry Pi template contains only 85 LoC, corresponding to serial reading and value emission on the Internet (a Raspberry Pi cannot read values directly from sensors in the SmartCampus infrastructure). On the other hand, the ARM Cortex M3 template comprises 169 LoC to handle its sensor and network interface. Using a template is efficient as we target low-level platforms, and those functions encapsulate a part of their complexity. For example, a method provided in the ARM Cortex M3 template handles the IPv6 retrieval of sensor measures from a border-router.

We can first observe that without considering the boilerplate code defined in the templates, there is no real difference in terms of LoC between COSmIC and the underlying programming languages. This is not surprising as the design choice of using templates hides low-level details to raise the level of abstraction of each platform. But the key point is that the code written with the COSmIC framework is not a single-target code but actually a model, which can be verified, composed automatically and projected to multiple platforms. Another interesting property is that users do not need to know the underlying platform. For example, if a policy relies only on digital sensors, one can use the Contiki

[5] `http://ace-design.github.io/cosmic/`
[6] `https://www.iot-lab.info/`

Table 1. LoC resulting from scenario generation

	Arduino native	Arduino contiki	Raspberry / Python	ARM Cortex M3 / Python	COSmIC source
Template	*13*	*22*	*85*	*169*	*0*
S_1	6	14	13	11	7
S_2	5	13	11	10	5
S_3	5	13	11	10	7
S_4	6	14	13	11	7
$S = S_1 \oplus S_2 \oplus S_3 \oplus S_4$	63	51	45	39	27
Deployed: S + *Template*	76	73	160	208	N/A

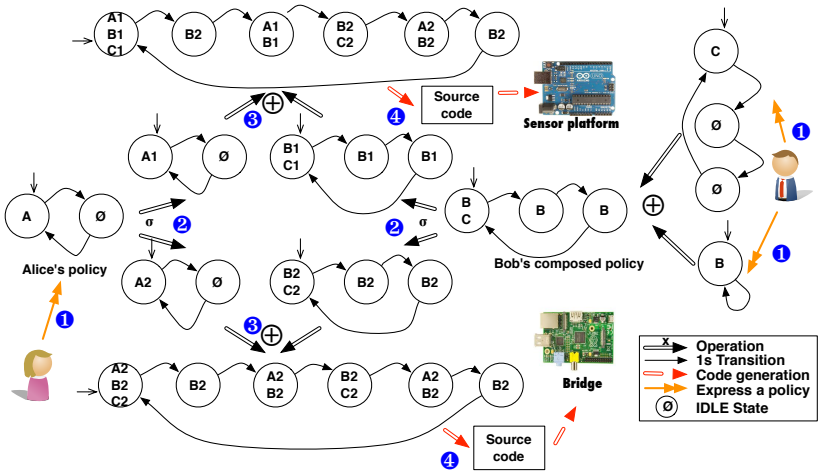


Fig. 2. COSmIC processes on the running example

operating system [8] to use a thread-based implementation of this policy. If a new requirement including values coming from analog sensors needs to be enacted on the same board, Contiki cannot be used anymore and the implementation must be completely rebuilt using only native operations. This is not the case with COSmIC: the two policies will be automatically composed, and it simply implies to change the call to the code generator as the Contiki one will reject the composed policy.

4.2 Illustration

We illustrate the application of the COSmIC operators from policy definition to code generation on the Alice and Bob example (see Sec. 2.3), on the top of the SmartCampus infrastructure[7]. Fig. 2 gives an overview of the different activities.

[7] More details can be found on a companion web page:
`https://github.com/ace-design/cosmic/blob/master/publications/ICSR15.md`

❶ *Policies definition.* In a first step, both users have to define their data collection policies in terms of timed automaton. The Alice's timed automaton p_a is already presented in Sec. 3.1. Bob has to express two policies: *(i)* temperature collection policy (p_{bt}) and *(ii)* humidity collection policy (p_{bh}). Bob uses the $\oplus$-operator to build a single policy p_b containing both temperature collection and humidity collection policies.

❷ *Decomposition process.* The next step is related to the decomposition process thanks to the δ-operator. Policies p_a and p_b are global policies that contain incompatible actions for the Arduino micro-controller (e.g. the *emit* action) platform and for the Raspberry nano-computer (e.g. the *read sensor* action). The appropriate compatibility table (Tab. 2) drives the decomposition process, reifying the compatibility of actions per platform. As presented in Sec. 3.3, incompatible actions are substituted by internal communications. After this decomposition process, four layer specific sub-policies are obtained:

$$\delta(p_a) = \{p_{amic}; p_{abri}\} \qquad \delta(p_b) = \{p_{bmic}; p_{bbri}\}$$

❸ *Composition process.* These four sub-policies will be then deployed on the shared infrastructure. The $\oplus$-operator will compose those policies and allow Alice and Bob to exploit the same piece of hardware. p_{amic} is composed with p_{bmic}, and p_{abri} is composed with p_{bbri}:

$$p_{amic} \oplus p_{bmic} = p_{\text{Sensor platform}} \qquad p_{abri} \oplus p_{bbri} = p_{\text{Bridge}}$$

The composition process is an endogenous operation returning a policy that can be reused to be composed, possibly in a dynamic way, with future policies. $p_{\text{Sensor platform}}$ and p_{Bridge} are the policies that will be instantiated on the infrastructure.

❹ *Code generation.* The final step of the deployment process is handled by code generators working directly on the two latter policies. The generated codes are then flashed on the appropriate micro-controllers and bridges using classical LSI deployment tools. At runtime, Alice and Bob will receive sensor values for their application according to their respective needs, although the same sensor is used for both of them.

4.3 Validation

To validate the four requirements presented in Sec. 2, we define an acceptance criterion for each of them and discuss how they are met.

R_1: Pooling and Sharing - More than one application can rely on a given sensor. The illustration in Sec. 4.2 shows that different policies can be enacted on the

Table 2. Excerpt of the COSmIC compatibility table

	Arduino Uno	Raspberry Pi	ARM Cortex M3
read	✓	✗	✓
emit	✗	✓	✓

sensing infrastructure to feed different applications. We performed also this validation on the SmartCampus infrastructure with four scenarios (cf. Sec. 4.1). In this context Table 3 illustrates that the same sensor will be used for different scenarios, validating requirement R_1.

R_2: Yield only relevant data - A given application is only fed with what it expects. As shown in our illustration (Sec. 4.2), a COSmIC user models her data collection policies with timed automaton and triggering of *emit* actions with requested data periodically. The composition operator also maintains this property by construction. It handles two policies p_1 and p_2 respectively T_1 and T_2 periodic, and produces a new lcm(T_1,T_2)-periodic policy. This process is transparent for COSmIC users as her expressed policy will not be modified while she will only receive data as specified in her initial policy. This validates requirement R_2.

R_3: Dynamically support data collection policies - Multiple policies can be dynamically composed. The $\oplus$-operator allows the composition of data collection policies on a sensor network. In the illustration (Sec. 4.2), Bob's policies have been composed into a single one using the $\oplus$-operator. The resulting policy can be used by other operators. Therefore, when a new policy needs to be added to the sensor network, one has just to compose it with the already deployed policy. This endogenous property validates requirement R_3.

R_4: Handling the infrastructure diversity - A given code can be deployed on more than one infrastructure. The infrastructure hardware variability is handled with code generators. These code generators handle a COSmIC DSL input code and produce the code for a given platform. We have successfully modeled and deployed the SmartCampus scenarios on Arduino, Raspberry Pi and ARM Cortex M3 platforms, validating requirement R_4.

4.4 Threats to validity

Scenarios. Our approach is only applied to the SmartCampus context. Even if the corresponding scenarios have been validated through questionnaires and are close to other case studies such as SmartSantander [19], we are aware that we need to step back and introduce more complex scenarios to benchmark COSmIC on a larger scale.

Table 3. Sensor sharing

	Light	Temperature	Presence
Scenario 1 - Late worker detection	✓		✓
Scenario 2 - Fire prevention		✓	
Scenario 3 - Heat monitoring		✓	
Scenario 4 - Energy wasting	✓		✓

Timed automata. Our data collection policies are represented by timed automata. If this approach fits the SmartCampus use case, the combination of different scenarios can lead to a combinatorial explosion, (*e.g.*, collections on a shared sensor at frequencies of one second and one hour would lead to a 3600 states automaton with only two relevant states). The code generation process is impacted by such automata. We currently reduce the size of such automata thanks to a factorization process, but this optimization does not scale with a large number of concurrent scenarios. The use of such automata also impacts the resources. Platforms have to be always powered on, to the detriment of the battery autonomy, to maintain a running clock delivering periodic clock tick. Devising better techniques to handle such cases and providing resource management is part of our future work.

Action execution duration. Our automata represent clocks with a 1 Hz frequency. If the execution of an action is longer than one second, it might be overlapped and aborted by the state transition leading the policy into an inconsistent state. In the future, we plan to use languages based on the formal Clock Constraint Specification Language (CCSL) [6] to determine the duration of action execution and to ensure the temporal correctness of policies.

Deployment of new policies. Our approach handles the dynamic composition of data collection policies and code generation for a given platform. However, we do not support dynamic deployment as some sensor platforms need to be reflashed with a new firmware. When the platform support it, we rely on operating systems (*e.g.*, Contiki) to support this feature.

5 Related Work

Programming sensor networks with specific OS. Several operating systems have been specifically designed for sensing infrastructures, *e.g.*, TinyOS [14] or Contiki [8]. TinyOS is based on a component architecture and comes with its programming language NesC. A developer can create new components or reuse components from the TinyOS's component library to build her own application. Contiki is adapted for networked and resource-constrained devices. Contiki applications can be written and compiled using a specific C compiler. Those OS abstract some complexities of application development, such as memory or energy optimization, but the developer has to be aware of what kind of sensor platforms she is using, directly dealing with their implementation details at a lower level. This leads to a lack of reusability, whereas our approach introduces a generic way to program sensor network. The COSmIC code is written independently from a sensing infrastructure and code generators handle the transformation to a targeted platform.

Sensor network as a database. On top of operating systems deployed on sensing infrastructure, several approaches consider the sensor network itself as a database [7]. Storing the data as close as possible to the sensor producing it instead of pushing everything to the Cloud was demonstrated as cost-efficient and energy saving [22]. The TinyDB system [16] (not maintained since 2005) provides processing mechanisms for sensor querying and data retrieval. It considers

a sensor as a micro-database storing their collected data, and allows developers to query sensors according to different criteria (*e.g.*, location). The Cougar system [24] also considers data collected by sensors, and supports users by only expressing queries that are automatically propagated to the sensors. This system does not support sharing (as queries cannot be composed easily), and relies on a centralized engine that computes a collection planning and collects data. On the contrary, COSmIC fully distributes the policies to the different sensors and the infrastructure layers, and supports multiple endpoints for each application.

Model-driven and generative approaches. The model-driven development paradigm has been notably used to design dynamically adaptive systems and to evolve them at runtime [18]. In this approach, the current context model is analyzed at runtime and, if an adaptation needs to be performed, a suitable configuration is built thanks to reference models. The approach can fit lightweight nodes in a sensor network [11]. Our work differs as we do not perform adaptiveness according to the context but design sensor network applications with policies based on a composition equation that can be reused for other compositions or for verification purposes. Exploiting runtime composition is a perspective of our work. The way we generate code is close to the Scalaness approach [5]. It is a type-safe language used to wirelessly program embedded networks running under TinyOS. Two stages are required to program these networks: *(i)* one writes a Scalaness program, which is then *(ii)* translated into Java bytecode. We differ from this approach as we use behavior models to generate code and we do not always have the same destination platforms as we target heterogeneous sensor networks.

6 Conclusions & Perspectives

In this paper we have presented the *COSmIC* Framework used for supporting different developers' collect policies on a shared LSI, generating code deployed at the approriate layer of the LSI. It addresses several limitations of classical approaches, focusing on the sharing of the infrastructure and the production of relevant-only datasets, and allowing software developers to focus on their concerns instead of LSI implementation details. The framework is implemented using the Scala language and preliminary experiments have been conducted on top of the SmartCampus platform [4]. The *COSmIC* framework is a first step for composing policies on an LSI, with a focus on policy definition and composition operators.

Future work aims at extending the approach and making it scale to very large LSIs. First, we will extend this set of operators to build a complete composition algebra, with a formal definition of operator properties (*e.g.*, commutativity, associativity, idempotency), conflict detection mechanisms to prevent inconsistent states (*i.e.*, sending a value before reading it) and a formal support to attach constraints to actions. We also plan to extend these constraints to timed ones, using the TimeSquare toolkit [6] to specify and analyze constraints based on its logical time model and to check them also at runtime. We also plan to enlarge the set of interactions with the LSI by introducing new actions allowing

a developer to perform some data computation within the sensor network (*e.g.*, Compute the average value of data coming from different sensors.

For those developers, we will improve the available abstractions by providing a higher level DSL. It will notably hide the creation and management of states and transitions, providing a real focus on what data are collected, processed and used in applications.

The decomposition operator also triggers interesting challenges with respect to the variability of hardware (*i.e.*, Arduino, Phidgets platforms) and facilities (*i.e.*, Supported programming language, resources available) available in the context of LSIs. We plan to use a feature modeling [2] approach to capture this variability, and to bind these models to the generation mechanisms, providing a variable code generation according to the available hardware in a given LSI. Finally, we also plan to support policy composition and variability reasoning at runtime to handle dynamic adaptiveness. We expect the resulting tooled approach to provide an end-to-end support for developers of the massively under-deployment sensing infrastructures.

References

1. Aggarwal, C.C. (ed.): Managing and Mining Sensor Data. Springer (2013)
2. Apel, S., Batory, D.S., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines - Concepts and Implementation. Springer (2013)
3. Buratti, C., Conti, A., Dardari, D., Verdone, R.: An overview on wireless sensor networks technology and evolution. Sensors 9(9), 6869–6896 (2009), `http://www.mdpi.com/1424-8220/9/9/6869`
4. Cecchinel, C., Jimenez, M., Mosser, S., Riveill, M.: An Architecture to Support the Collection of Big Data in the Internet of Things. In: International Workshop on Ubiquitous Mobile Cloud (UMC'14, co-located with SERVICES'14). pp. 1–8. IEEE, Anchorage, Alaska, USA (Jun 2014)
5. Chapin, P.C., Skalka, C., Smith, S.F., Watson, M.: Scalaness/nesT: Type Specialized Staged Programming for Sensor Networks. In: Jrvi, J., Kstner, C. (eds.) GPCE. pp. 135–144. ACM (2013)
6. Deantoni, J., Mallet, F.: TimeSquare: Treat your Models with Logical Time. In: Carlo A. Furia, S.N. (ed.) TOOLS - 50th International Conference on Objects, Models, Components, Patterns - 2012. Lecture Notes in Computer Science - LNCS, vol. 7304, pp. 34–41. Czech Technical University in Prague, in co-operation with ETH Zurich, Springer, Prague, Tchèque, République (May 2012)
7. Diao, Y., Ganesan, D., Mathur, G., Shenoy, P.J.: Rethinking data management for storage-centric sensor networks. In: CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings. pp. 22–31 (2007)
8. Dunkels, A., Gronvall, B., Voigt, T.: Contiki - a lightweight and flexible operating system for tiny networked sensors. In: Local Computer Networks, 2004. 29th Annual IEEE International Conference on. pp. 455–462 (Nov 2004)
9. Fambon, O., Fleury, E., Harter, G., Pissard-Gibollet, R., Saint-Marcel, F.: Fit iot-lab tutorial: hands-on practice with a very large scale testbed tool for the internet of things. In: 10èmes journées francophones Mobilité et Ubiquité (UbiMob). pp. 1–5 (June 2014)
10. Fleurey, F., Morin, B., Solberg, A.: A Model-Driven Approach to Develop Adaptive Firmwares. In: Giese, H., Cheng, B.H.C. (eds.) SEAMS. pp. 168–177. ACM (2011)

11. Fouquet, F., Morin, B., Fleurey, F., Barais, O., Plouzeau, N., Jezequel, J.M.: A Dynamic Component Model for Cyber Physical Systems. In: Proceedings of the 15th ACM SIGSOFT Symposium on Component Based Software Engineering. pp. 135–144. CBSE '12, ACM, New York, NY, USA (2012)
12. Gluhak, A., Krco, S., Nati, M., Pfisterer, D., Mitton, N., Razafindralambo, T.: A Survey on Facilities for Experimental Internet of Things Research. IEEE Communications Magazine 49(11), 58–67 (Dec 2011), http://hal.inria.fr/inria-00630092
13. Gubbi, J., Buyya, R., Marusic, S., Palaniswami, M.: Internet of Things (IoT): A Vision, Architectural Elements, and Future Directions. Future Generation Comp. Syst. 29(7), 1645–1660 (2013)
14. Levis, P., Madden, S., Polastre, J., Szewczyk, R., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E., Culler, D.: Tinyos: An operating system for sensor networks. In: in Ambient Intelligence. Springer Verlag (2004)
15. LogMeIn: Xively (May 2014), http://xively.com/
16. Madden, S.R., Franklin, M.J., Hellerstein, J.M., Hong, W.: Tinydb: An acquisitional query processing system for sensor networks. ACM Trans. Database Syst. 30(1), 122–173 (Mar 2005), http://doi.acm.org/10.1145/1061318.1061322
17. Mahmood, A., Ke, S., Khatoon, S., Xiao, M.: Data mining techniques for wireless sensor networks: A survey. IJDSN 2013 (2013)
18. Morin, B., Barais, O., Jezequel, J., Fleurey, F., Solberg, A.: Models@run.time to Support Dynamic Adaptation. Computer 42(10), 44–51 (2009)
19. Sanchez, L., Galache, J., Gutierrez, V., Hernandez, J., Bernat, J., Gluhak, A., Garcia, T.: Smartsantander: The meeting point between future internet research and experimentation and the smart cities. In: Future Network Mobile Summit (FutureNetw), 2011. pp. 1–8 (June 2011)
20. Stickel, M.E.: A Unification Algorithm for Associative-Commutative Functions. J. ACM 28(3), 423–434 (Jul 1981)
21. Tonneau, A.S., Mitton, N., Vandaele, J.: A Survey on (mobile) wireless sensor network experimentation testbeds. In: DCOSS - IEEE International Conference on Distributed Computing in Sensor Systems. Marina Del Rey, California, États-Unis (May 2014), http://hal.inria.fr/hal-00988776
22. Tsiftes, N., Dunkels, A.: A database in every sensor. In: Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems. pp. 316–332. SenSys '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/2070942.2070974
23. Völgyesi, P., Maróti, M., Dóra, S., Osses, E., Lédeczi, Á.: Software Composition and Verification for Sensor Networks. Sci. Comput. Program. 56(1-2), 191–210 (2005)
24. Yao, Y., Gehrke, J.: The cougar approach to in-network query processing in sensor networks. SIGMOD Rec. 31(3), 9–18 (Sep 2002), http://doi.acm.org/10.1145/601858.601861