makeSense: Simplifying the Integration of Wireless Sensor Networks into Business Processes

L. Mottola, G. P. Picco, F. J. Oppermann, J. Eriksson, N. Finne, H. Fuchs, A. Gaglione, S. Karnouskos, P. Moreno Montero, N. Oertel, K. Römer, P. Spieß, S. Tranquillini, and T. Voigt

Abstract—A wide gap exists between the state of the art in developing Wireless Sensor Network (WSN) software and current practices concerning the design, execution, and maintenance of business processes. WSN software is most often developed based on low-level OS abstractions, whereas business process development leverages high-level languages and tools. This state of affairs places WSNs at the fringe of industry. The make*Sense* system addresses this problem by simplifying the integration of WSNs into business processes. Developers use BPMN models extended with WSN-specific constructs to specify the application behavior across both traditional business process execution environments and the WSN itself, which is to be equipped with application-specific software. We compile these models into a high-level intermediate language—also directly usable by WSN developers—and then into OS-specific deployment-ready binaries. Key to this process is the notion of *meta-abstraction*, which we define to capture fundamental patterns of interaction *with* and *within* the WSN. The concrete realization of meta-abstractions is application-specific; developers tailor the system configuration by selecting concrete abstractions out of the existing codebase or by providing their own. Our evaluation of make*Sense* shows that *i)* users perceive our approach as a significant advance over the state of the art, providing evidence of the increased developer productivity when using make*Sense*; *ii)* in large-scale simulations, our prototype exhibits an acceptable system overhead and good scaling properties, demonstrating the general applicability of make*Sense*; and, *iii)* our prototype—including the complete tool-chain and underlying system support—sustains a real-world deployment where estimates by domain specialists indicate the potential for drastic reductions in the total cost of ownership compared to wired and conventional WSN-based solutions.

Index Terms—Business processes, Wireless sensor networks, Embedded software, Internet of Things.

1 Introduction

Wireless Sensor Networks (WSNs) are distributed systems of small battery-powered devices equipped with sensors and actuators. Their key features, such as ease of deployment, make them ideal to harvest data from the environment and to act on it.

Despite the many solutions to WSN-specific challenges, such as energy-efficient communication, industry is reluctant to adopt WSN technology. In fact, WSNs are often perceived as a "foreign body" w.r.t. the established business process technology at the back-end of information systems. This is partly because, despite the several existing proposals [1]–[3], WSNs are still programmed with OS-level constructs [4], [5], which inherently clashes with the high level of abstraction of business process design and execution [6].

- *L. Mottola is with RISE SICS (Sweden) and Politecnico di Milano (Italy);*
- *G. P. Picco is with University of Trento (Italy); A. Gaglione and S. Tranquillini were with University of Trento (Italy) at the time the work was performed;*
- *K. Römer is with Graz University of Technology (Austria); F. J. Oppermann was with Graz University of Technology (Austria) at the time the work was performed;*
- *J. Eriksson, and N. Finne are with RISE SICS (Sweden);*
- *H. Fuchs, S. Karnouskos, N. Örtel and P. Spieß are with SAP (Germany);*
- *P. Moreno Montero is with Acciona Infraestructuras S.A. (Spain);*
- *T. Voigt is with RISE SICS (Sweden) and Uppsala University (Sweden).*

This work was supported by the European Union 7^{th} Framework Programme (FP7-ICT-2009-5) under grant agreement n. 258351 (project makeSense).

Scenarios. Consider for example the goal of improving energy consumption and user comfort in an office environment. Employees book meeting rooms on the Web through a back-end process notifying the expected participants. Inside the rooms, ventilation should be minimal when no meeting is scheduled, whereas knowledge of the booking schedule enables pre-ventilation before a meeting, to ensure proper air quality. Similarly, sensing the actual occupancy, e.g., empty booked rooms and occupied non-booked rooms, helps optimize business operations by prematurely releasing non-occupied rooms or by charging for overuse. To ensure proper air quality, actuators such as ventilation devices, can take local decisions based on the status of surrounding sensors, e.g., CO_2 and motion, and on the booking schedule.

This scenario prominently includes requirements that developers can only address by extending the application logic across business processes and WSNs. Similar traits emerge in a multitude of applications [7], such as predictive maintenance of large machinery and operation of large power plants [8]–[11]. For example, sensors attached to the engines of cargo vessels can monitor their functioning based on vibrations and lubricant temperatures. These data serve at the shipping company's back-end to optimize a cargo's route and to provision spare parts [8]. Differently, sensors and actuators attached to solar panels can steer their operation based on current prices on the energy market, trading off a panel's wear and tear against monetary gains [9].

These scenarios are just further instances of the emerging "Internet of Things" (IoT) [12], whose fundamental challenge is that of extending interactions between humans

and digital machines to the physical world. WSNs provide a low-cost, yet effective means to concretely achieve such integration. As the Internet of Things reaches into industry settings, many predict that the need to implement similar applications flexibly and efficiently is bound to become key [13], [14]. The case of business process integration, in particular, is already frequently observed [15]–[17]. Unfortunately, the current state of affairs lags behind this vision: WSN technology is used only in very specific industry settings, with little confidence in its performance and using handcrafted solutions to achieve simplistic means of business process integration [11], [18], [19].

Challenges. The example applications intrinsically require a tight integration between the business process and the WSN deployed in the environment. Unlike scenarios where sensor devices simply act as data sources, the WSN needs to enact part of the application logic specified in the business process, for efficiency or dependability reasons [19]. As a result, part of the application-specific software is to be deployed right onto the resource-constrained WSN devices.

As we further elaborate in Section 2, this is still a wide open software engineering problem. Despite the many approaches existing to provide higher-level programming abstractions [1], [2] for WSNs, the dominating practice is to rely solely on the low-level constructs of the underlying OS. To make a parallel with conventional platforms, the situation is akin to developing a complex distributed information system without any middleware, by employing only on system calls.

To harmonize the state of the art in WSN software with the more advanced and established practices in business processes, we are to solve two key challenges:

1) The *Integration* challenge refers to the need of coordination across the business back-end and the WSN. Currently, the latter is mostly considered as a stand-alone system: its integration with the back-end is left to application developers. This is relatively simple when such integration consists only in relaying collected sensor data. However, it becomes complex when the WSN must be equipped with application-specific functionality and this must be coded with low-level primitives provided by OSes such as TinyOS [4] or Contiki [5]. The required expertise spans from traditional information systems to low-level embedded systems programming.
2) The *Unification* challenge refers to the need of a single comprehensive WSN programming framework. Existing higher-level WSN programming abstractions [1]–[3] often focus only on one specific problem; their combined use in a single application still requires considerable effort. On the contrary, to drastically simplify programming of WSN-integrated business processes in the diverse scenarios above, developers need to combine several abstractions at once. This requires a unified programming framework where existing and future WSN programming abstractions can blend smoothly.

We maintain, and empirically demonstrate in a real deployment, that these challenges bear great impact on the total cost of ownership, and hence are key in determining the success of WSN technology in this domain.

Contribution and road-map. This paper presents the design, implementation, and evaluation of the makeSense system we develop to address these challenges.

At the core of makeSense is the notion of *meta-abstraction*, described in Section 3. Meta-abstractions capture the essence of existing WSN interaction patterns, and serve to specify the coordination between the business back-end and the WSN. We define the set of meta-abstractions, their abstract semantics, and their relationships in a meta-model that describes these programming concepts independent of their concrete realization. This remains application-dependent, allowing different instantiations of a makeSense system, each tailored to a specific scenario.

To address the *integration challenge*, we treat meta-abstractions as first-class elements in existing business process notations. To demonstrate this, we extend the Business Process Model and Notation (BPMN) [20] with constructs to represent meta-abstractions and to specify their concrete instantiation, as illustrated in Section 4. We choose BPMN because of its standardized nature and wide tool support. We are, however, not strictly tied to BPMN; makeSense is applicable to other workflow-based business process notations [6]. Figure 1 gives a concrete feel of our approach, by showing the model implementing the aforementioned office ventilation scenario. We return to Figure 1 in Section 4, when we describe the specific BPMN extensions we design and how they are used in our example application.

To reconcile the expressiveness of business process notations with the reality of resource-constrained WSN devices, we adopt a fully automated *model-driven approach*, shown in Figure 2. The BPMN model is first compiled into a custom intermediate language, called makeSense macro-programming language (*m*PL), described in Section 5. The language serves a dual purpose: *i)* it encodes the WSN-specific processing at a higher level of abstraction than OS-level WSN programming, thus decoupling the BPMN model from the target WSN platform; and *ii)* it provides a stand-alone language that WSN developers can directly access if and when required, e.g., to fine-tune performance or whenever they consciously decide not to rely on a model-driven approach. The notion of meta-abstractions emerges in *m*PL to provide a unit of language modularity, allowing one to combine multiple programming abstractions, thus addressing the *unification challenge*.

Developers reap the benefits of makeSense by means of the associated toolchain, described in Section 6. The toolchain includes the compilers and run-time layers supporting the above functionality, along with a library of concrete abstractions and a custom editor supporting modeling. A full prototype of the toolchain can be downloaded as a pre-configured virtual machine, along with technical documentation in the form of a tutorial [21].

Our evaluation, in Section 7, is divided in three parts:

- In Section 7.1 we assess our core goal, that is, whether makeSense facilitates the development activities, by conducting two user studies, respectively targeted at the modeling and macro-programming languages. The results confirm that developers perceive makeSense as a significant step forward compared to the state of the art in WSN development. This provides a basis to the argument of increased developer productivity when using makeSense rather than OS-level WSN programming.

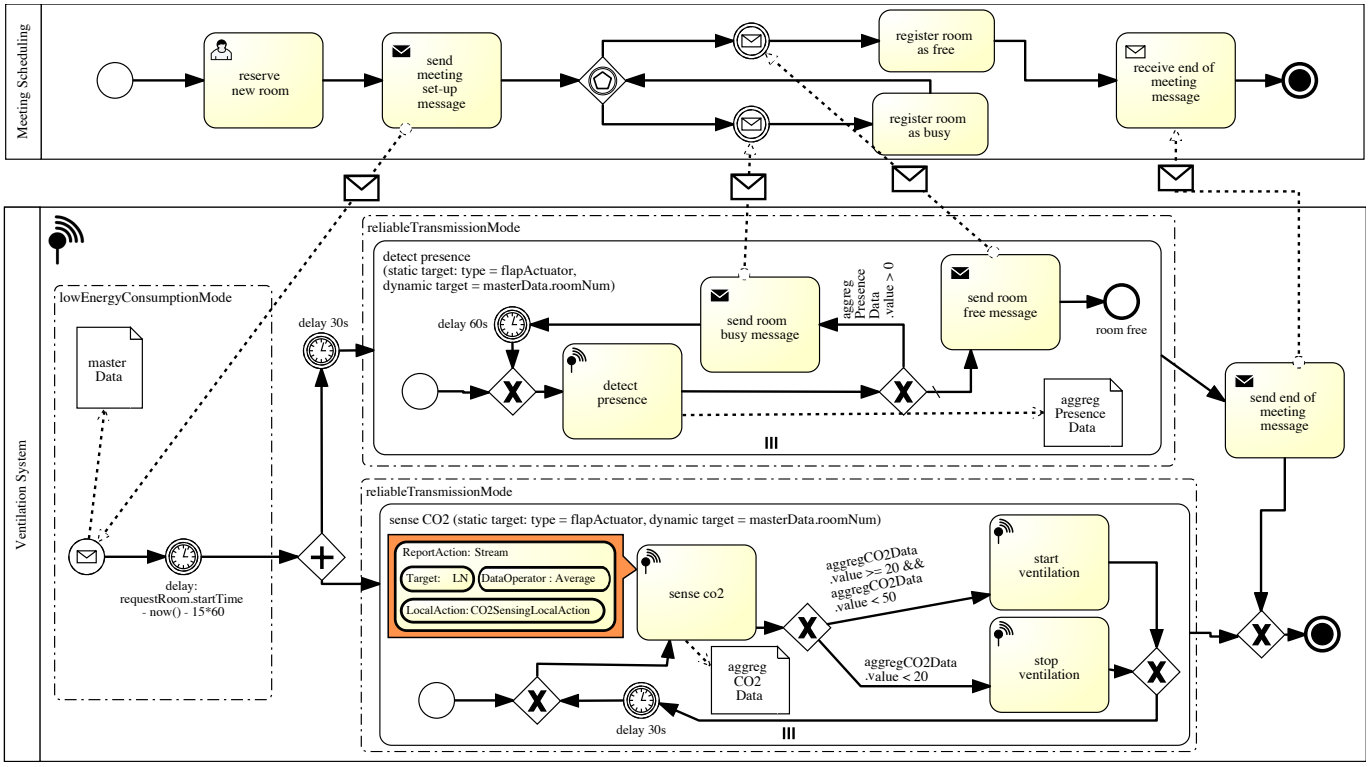


Figure 1. Example make*Sense* BPMN model for adaptive ventilation.

- In Section 7.2, we dissect the performance of make*Sense* from a system standpoint, evaluating the performance overhead induced by the increased level of abstraction and the extent it affects the system's scaling properties. The overhead we measure is acceptable and, most importantly, it remains within the tight limits posed by WSN platforms. Moreover, such overhead does not prevent a make*Sense* system to scale up to hundreds of devices, as we show through time-accurate simulations.
- In Section 7.3 we report on a small-scale real-world deployment of the adaptive ventilation application in Cádiz, Spain. This experience confirms the practical effectiveness of make*Sense*, and is the opportunity to estimate the monetary savings in development, installation, and maintenance compared to an existing wired system and a conventional WSN-based one. Our analysis indicates a 65% reduction in development costs compared to mainstream WSN programming solutions.

We end the paper with a concise survey of related work in Section 8, and with brief concluding remarks in Section 9.

2 Problem

Engineering software to collect, process, and store sensor data is straightforward whenever the devices equipped with the relevant sensors act as mere data sources, much like a database. Developers encode the application logic using

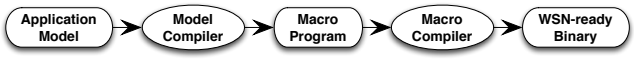


Figure 2. From process models to WSN-executable code.

traditional methodologies and tools, whereas the software on the sensor nodes is application-agnostic. This is the case, for example, when using externally-powered sensor nodes directly connected to mainstream machines. Frameworks enabling various forms of "physical mashups" [22], and programming environments such as Wilyodrin [23] and IBM's Node-red [24] enable this kind of integration.

The application requirements and target platforms we consider, and thus the software engineering problem we tackle, are sharply different. Three key aspects set us apart from the simpler setting above:

1) *WSNs are distributed, wireless systems of battery-powered embedded devices*. The nodes typically route their network traffic through multiple hops to one another or to one or multiple gateways, which provide connectivity to the back-end. The WSN is thus a separate distributed system capable of operating autonomously, and yet subject to severe constraints, especially in terms of energy consumption [7].
2) *Part of the application logic is to be deployed onto the WSN devices*, for efficiency or dependability reasons [19]. For example, sensor data may be processed inside the WSN using domain-specific algorithms, thus reducing network traffic and saving energy. Further, control loops such as the adaptive ventilation in our example application, are most efficiently implemented by limiting the data paths to the portion of physical space under control. Developers need to equip the WSN devices with application-specific software, drastically increasing the complexity of development.
3) *Business processes express complex workflows* through a variety of different constructs expressing ordering of activities, distributed interactions, decisions, and respon-

sibilities. Envisioning part of the resulting workflows to run on a WSN system, thus strikingly different than usual business process execution environments, raises a number of questions Examples are how to properly slice the application logic, how to enable the coordination between WSNs and the back-end, and how to generate efficient WSN code out of a high-level business process specifications.

As we discuss further in Section 8, the current state of the art in this area is limited. Solutions such as "physical mashups" [22], Wilyodrin [23] and IBM's Node-red [24], for example, are unable to relocate part of the application logic onto WSN nodes. Most existing model-driven approaches for WSN software do not address the integration challenge. The few exceptions employ application-specific proxies, incurring in additional development efforts and performance penalties. Here again, no application-specific logic is deployed onto WSN nodes.

The fundamental obstacle, in fact, is to engineer the software running on WSN nodes [1]–[3]. The existing literature [25] already recognizes the distance in methodologies, abstractions, and concrete tools between traditional practices of software engineering and WSN software, regardless of application domains. The lack of a principled approach to developing WSN-integrated business processes is thus probably yet another instance of the "consensual divorce" between software engineering and WSNs [25]. The ambition of `makeSense` is to help reconcile these two domains.

3 Meta-abstractions

To address the integration challenge, we need to facilitate embedding WSN functionality in business process specifications. To this end, we define the notion of *meta-abstraction*: a conceptual model of typical WSN interaction patterns independent of specific implementations. Defining meta-abstractions also requires to formalize their relationships. Knowledge of these is useful to establish how meta-abstractions can be combined, providing a basis to tackle the unification challenge.

Meta-abstractions thus provide: *i) conceptual unity and structure*: coordination among WSN nodes and their integration with the business back-end are expressed with few fundamental concepts and well-defined compositional rules; and *ii) extensibility and flexibility*: by defining the fundamental building blocks abstractly, we only partially constrain their behavior, which is left open for (re)definition by the concrete abstractions in a specific instantiation of `makeSense`. The latter is also instrumental to address the variety of available sensor and actuator technology [26]. None of our design decisions, in fact, is dictated by a specific kind of sensor or actuator. Such details are to be encapsulated in a specific `makeSense` instantiation, based on application requirements and deployment scenario.

Identifying meta-abstractions. To determine the set of useful meta-abstractions, we consider both the existing literature on WSN applications [7] and the emerging programming practices in WSN development [27].

Figure 3 provides an intuition on the key meta-abstractions we define in `makeSense`. *Distributed* actions model WSN interactions involving multiple nodes, as opposed to *local* actions—not shown in Figure 3—that only affect single devices, e.g., the action of sampling a sensor or that of combining the readings of multiple physical sensors on the same device into a higher-level information [28]. Distributed actions are further separated into:

Report actions represent the typical many-to-one interaction pattern used in WSNs to funnel data from multiple nodes (the "many") to a single destination (the "one"). Examples of report actions exist in most WSN applications, notably including those where WSNs are employed to harvest data from the environment using multiple sensor nodes reporting to a data sink, e.g., in habitat, wildlife, or structural monitoring [29]. In this sense, report actions map to the "collection" primitive recognized by Levis et al. [27].

Tell actions are dual w.r.t. report actions and model the one-to-many interaction pattern that enables communication from "one" node to "multiple" others. Examples of tell actions are also found in many WSN applications, especially those employing WSNs to implement control loops [13], [14]. In these applications, actuation commands are typically distributed from a single controller node to multiple actuators in the environment. Thus, tell actions mirror the "dissemination" primitive [27].

Collective actions, different from report and tell actions, do not focus on a specific node, rather enable a global behavior across multiple WSN devices. As such, they model the many-to-many interaction pattern found in WSN-based applications in the health-care, building automation, and vehicular traffic domains [30]. Moreover, collective actions abstract a number of WSN coordination functionality, such as monitoring distributed assertions [31], [32].

Orthogonal to *distributed* actions, *modifiers* represent ways to customize the behavior of a distributed action. A *target* modifier identifies a set of nodes satisfying given application constraints defined over application-level *node attributes*, giving the ability to apply a distributed action solely to these nodes. For example, a target may allow

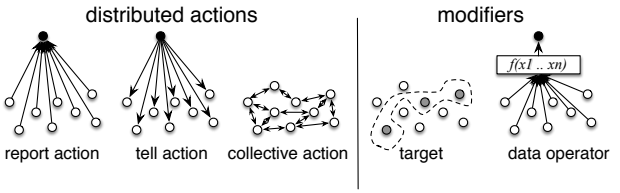


Figure 3. Distributed actions and modifiers.

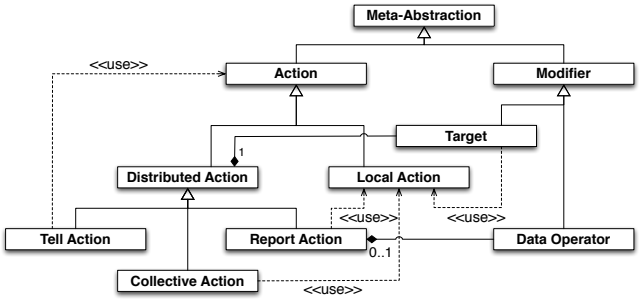


Figure 4. Meta-abstractions in `makeSense`.

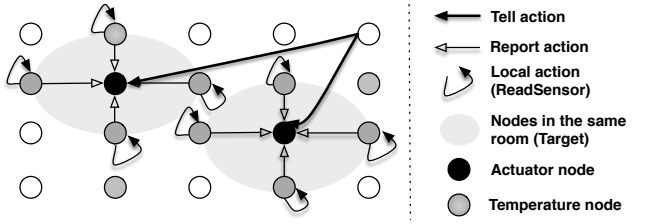


Figure 5. Nesting a *report* action inside a *tell* action to implement distributed control loops.

one to "tell" only the actuators on a specific building floor to take a (local) action. This is particularly useful in the scenarios we focus on, where nodes possibly differ along several dimensions, including their physical placement or the sensors and actuators they are equipped with. Moreover, *data operators* allow one to couple data processing with communication. Examples of these are algorithms to filter data before returning them to the user or forms of in-network processing to reduce the amount of data transferred, thus improving the radio utilization [33].

Composing meta-abstractions. Figure 4 shows a UML meta-model specifying the relationships across meta-abstractions and the allowed combinations.

According to the meta-model, at most one *target* is optionally attached to any distributed action. We choose to enforce this limitation in that applying a distributed action to multiple, possibly intersecting, subsets of nodes may conceal non-trivial semantics issues. Multiple *data operators* may be attached only to *report* actions, according to a programmer-specified order. Data operators, indeed, may be applied in sequence and are most commonly used when collecting sensor data according to a many-to-one pattern [33], whereas they have limited applicability in combination with *tell* or *collective* actions.

As shown in Figure 4, *report* and *collective* actions take as input a *local* action that indicates what the "many" nodes are to execute to produce the input data, e.g., sampling a sensor. *Tell* actions, on the other hand, may use any *action* as input, including *distributed* ones, to communicate what action is to take on the "many" end. This enables a range of sophisticated interaction patterns, such as distributed control loops. For example, developers may write control functionality to run on actuators based on nearby sensor data, as in the case of an actuator controlling a room's ventilation based on CO_2 readings in the same room. To this end, developers can *tell* actuators to execute a *report* action that collects nearby sensor data, as shown in Figure 5, by "nesting" a *report* action inside a *tell* action. The nodes addressed by the *report* action are specified with a proper *target* relative to the actuator, e.g., by binding the sensors' location to that of the corresponding actuator.

To ensure that the set of meta-abstractions and the corresponding meta-model are sufficiently general, we cross-check our design against 48 recent WSN applications [7]. We use the meta-abstraction representation in *m*PL, as explained in Section 5, to perform such analysis, creating the necessary code and comparing the resulting processing with the application requirements. Our analysis indicates that our design would be applicable to all but 3 of these applications, namely, FireWxNet, LOFAR-agro, and MEDiSIN [7]. In essence, the functionality we cannot easily express is that of moving a data processing function across nodes based on real-time sensor readings. Even though we can *tell* a subset of nodes to execute a given action by properly defining a corresponding *target*, making the same *target* definition dependent on remote data is non-trivial. The latter functionality would require the use of an involved combination of *report* actions to move sensor data onto the node where the *target* is defined.

Concrete abstractions. Meta-abstractions do not specify their behavior, which is left to the *concrete abstractions* providing their actual instantiation.

In the current `makeSense` prototype, we use the Logical Neighborhoods (LN) [34] abstraction to provide an instantiation for both *tell* and *target*. LN replaces the physical neighborhood provided by wireless broadcast with an application-defined one. The span of a logical neighborhood is specified declaratively by means of Boolean predicates over application-defined node attributes, using a custom language. LN provides application programmers with a broadcast API that allows one to send data to all nodes belonging to a previously specified logical neighborhood. Besides minor adaptations, LN matches the semantics of both *tell* and *target*.

The DICE [31] system provides an instance of *collective* action by enabling WSN-based distributed monitoring of global invariants. A DICE invariant is expressed by Boolean predicates defined over the state of multiple WSN nodes, e.g., the expected state of actuators based on sensed environmental conditions. The underlying run-time support performs invariant checking in a fully decentralized fashion, thus matching the abstract semantics of *collective* actions.

We build custom implementations to instantiate the *report* and *data operator* meta-abstractions. We essentially wrap a simplified version of an existing data collection protocol [35] with the APIs necessary to make it usable as an instance of *report*. We call it STREAM abstraction, as it primarily supports periodic traffic. Similarly, we create wrappers for a number of common sensor data processing functions to provide a basic library of *data operators*. Further custom local action instances are available for reading sensors and controlling actuators on the target platforms.

Our implementation of LN, DICE, and the STREAM abstraction include mechanisms to transparently adapt to varying underlying wireless topologies. Crucially, the separation between meta- and concrete abstractions shields developers from this complexity, as meta-abstractions conceptually models given WSN interaction patterns *independent of how these are concretely realized*. In the absence of such a separation, developers would be exposed to the intricacies of the corresponding mechanisms [36], ultimately steering their attention away from the application goals.

We export meta- and concrete abstractions both at the modeling layer, as described in Section 4, and at the programming layer, as illustrated in Section 5.

4 WSN-Integrated Business Process Modeling

To tackle the integration challenge, `makeSense` extends business process specification languages to stretch the process

definition and execution to the WSN. To render our ideas concrete, we use BPMN to specify the behavior of WSN-integrated business processes. In addition to the aforementioned standardized nature and wide tool support of BPMN, its expressiveness is already recognized as apt to WSN applications [37]. These aspects motivate our choice of BPMN, even though our approach is generally applicable to workflow-based business process notations [6]. The existing BPMN standard, however, does not suffice in makeSense and requires dedicated extensions, as we discuss next.

Our presentation revolves around the example modeling of the adaptive ventilation application described in the Section 1, shown in Figure 1 using the makeSense extended BPMN. Nonetheless, the design of the BPMN extensions we describe next is rooted in the analysis of a large set of WSN applications [7]. Besides the adaptive ventilation application, we also test the applicability of our BPMN extensions with two additional and diverse application domains, namely, predictive maintenance of large machinery and operation of large power plants [8]–[11]. Albeit not included in the paper for brevity, these results of this analysis are also available [38]–[40].

We begin with a concise primer on BPMN and discuss the limitations of the existing standard that prevent making the concepts in Section 3 emerge in BPMN. Next, we describe our custom extensions to BPMN to specify the behavior of business processes that extend to WSNs.

4.1 The BPMN Standard

The BPMN 2.0 standard [20] defines a notation widely used to define and execute interactions between people, systems, and organizations. Here we focus on the modeling constructs most relevant to our goals.

Key concepts. Central to BPMN is the notion of *process*, which is a behavior pattern with a corresponding run-time instantiation. BPMN *events* define when process instances are created and destroyed, or when their execution is suspended until a condition occurs, e.g., a timeout expires. The execution semantics of BPMN relies on the notion of *tokens* associated to process instances. *Flows* represent the control flow, i.e., the paths available to tokens across the model. These paths can be manipulated at *gateways* acting as decision points. Examples are parallel gateways, used either to synchronize and combine incoming parallel flows or to create new ones. *Tasks* represent the actual operations carried out in a process, e.g., sending a message or entering data by a human. Finally, *data objects* represent data structures, shared within a process instance, and accessible for read/write by all of its elements.

Processes belong to *pools*, which represent distinct interacting systems or organizations. Process instances exist independently in their pools, although they can communicate across pools with asynchronous *messages*. For instance, in a trade between organizations, a sales process instance in one organization may communicate with a purchase process instance on another, by means of messages containing the order number. Instead, flows and related elements can be used only inside a single pool.

The need to extend BPMN. Existing work already recognize how the existing BPMN standard is insufficient when integrating ubiquitous embedded technology [17], [41], [42]. In our case, BPMN is unable to express many of the concepts required to export meta- and concrete abstractions at the modeling layer, as we explained in the following.

The slice of the process logic running within the WSN, for example, is difficult to identify unless developers provide appropriate inputs at the modeling layer. Such information is necessary because the WSN-specific part of the process logic requires special handling during compilation, as discussed in Section 6. Similarly, the use of specific WSN interaction patterns, represented in makeSense through the combined use of meta- and concrete abstractions in support of specific tasks, needs to be explicitly indicated by developers. It is in general impossible to infer this information automatically when compiling the process model.

BPMN processes are also typically seen as distinct singletons, whereas in makeSense a single triggering event may need to activate the same behavior on *multiple* WSN nodes, possibly later rejoined when the processing is over. As a result, merely identifying the slice of the process logic germane to the WSN does not suffice. An example is the distributed control functionality of Figure 5. The issues are that: *i)* the (same) control process should be instantiated on multiple actuators, and *ii)* the execution of such control processes must be concurrent. The notion of BPMN process is unable to specify such a behavior.

Finally, WSNs are complex software artifacts, whose performance depends on the deployment environment and is complicated by the scarcity of resources. Optimizing the WSN operation is a non-trivial task, whose burden should not be put on the application developer. BPMN per se cannot express high-level non-functional requirements. Instead, the makeSense toolchain may take these as input to steer the system's configuration towards given performance goals.

4.2 Extending BPMN

We extend BPMN to address these shortcomings. In doing so, we adopt a methodology similar to other domain-specific BPMN extensions [43].

Figure 1 exemplifies the use of the extended BPMN to model the adaptive ventilation application. The bottom pool *VentilationSystem* represent the WSN sub-system, while the top pool concerns meeting schedules. The setup of a meeting from a process instance in the latter triggers a message, shown as an envelope icon, representing the event creating the main process instance in the WSN pool. The information is stored in a data object, shown as a document icon, containing the overall schedule. The newly-created process is immediately suspended by a timeout event, shown as a clock icon, specifying that nothing is to be done until 15 minutes before the meeting. When the timeout expires, the flow encounters a parallel gateway, shown as a diamond with plus sign, which forks two processes concerned with CO_2 sensing and presence detection—the bottom and top rectangles, respectively.

WSN pool. As illustrated in Figure 1, we use a single pool to specify the WSN logic in makeSense. Explicitly distinguishing this pool from others elegantly makes the service contract between the WSN and the rest of the process explicit; both can only communicate through messages. Such

a distinction is anyways necessary as the WSN pool requires a special handling by the makeSense toolchain [38]–[40]. We introduce a new graphical element – the antenna icon in the top-left corner of the *VentilationSystem* pool in Figure 1 – to indicate the WSN pool. This allows the model compiler to treat this pool differently from others: its corresponding BPMN specification must be eventually translated into *m*PL, as shown in Figure 2.

Compared to standard BPMN pools, a different set of modeling constructs is available within a WSN pool. On one hand, only a subset of standard BPMN is supported, as we verify that many of the available constructs, e.g., the variety of gateways and events, are of marginal use in WSNs [7], [39]. On the other hand, we add new BPMN constructs to make meta-abstractions emerge at the BPMN level. Importantly, unlike existing literature aiming to model WSN behaviors in plain BPMN [44], meta-abstractions spare the need to explicitly model communication with and within the WSN, by abstractly capturing possible WSN interaction patterns. This greatly simplifies their integration in business process modeling languages, such as BPMN.

In our example, both pools have terminating end events: once a meeting finishes, any ongoing activities should be terminated. The system can then switch to an idle state with minimal resource usage until the next scheduled meeting takes place.

WSN tasks. Processes in a WSN pool may or may not correspond to concrete interactions with the WSN. In the former case, processes are represented as "service tasks" in BPMN jargon and identified by the same antenna icon of the WSN pool, as for the *sense* CO_2 task in Figure 1.

Every WSN task is mapped onto one *action* meta-abstraction. This mapping, along with the binding of the meta-abstraction with the concrete one, is represented graphically through nesting. For example, in Figure 1 the *sense* CO_2 task is mapped onto a *report* action that acquires values from remote CO_2 sensors. The outer box in the balloon associated to *sense* CO_2 contains a label showing the association between the meta-abstraction (*report*) and the concrete one (STREAM).

The *report* action has associated *modifiers* and *parameters*, shown inside the balloon with a similar pair of meta- and concrete abstractions. In this example, modifiers include a *target* to determine the action scope, bound to the LN concrete abstraction, and a *data aggregator* to specify additional data processing, bound to an AVERAGE concrete abstraction. The only action parameter is, in this case, a *local* action bound to a CO2SENSINGLOCALACTION used to probe the CO_2 sensor. The result of *report* is available as a *data object* that, as in standard BPMN, can be accessed by other elements in the diagram.

Based on the aggregate CO_2 value, an XOR gateway, shown as a diamond with an "X" sign, is used to toggle ventilation in a room. The latter occurs in the *start/stop ventilation* tasks. The processing in the bottom sub-process is thus to collect CO_2 data from a subset of remote nodes, whose values are averaged to perform a local actuation. We find similar patterns to express the decisions based on sensed values also in the other application domains where we apply our BPMN extensions [38], [39].

WSN tasks can be mixed freely with standard BPMN tasks, as in the top process concerned with presence detection. In this case, the result of the *detect presence* WSN task is used, through an XOR gateway, to send different messages to the top process depending on the result of *detect presence*.

Sub-processes. To address the limitation described in Section 4.1 of BPMN processes represented as distinct singletons, and thus unable to express multiple instances of distributed computations within the WSNs, we extend the BPMN sub-process construct. Sub-processes allow one to structure the model hierarchically, like the two sub-processes in Figure 1 concerned with presence detection and CO_2 sensing. In makeSense, sub-processes also determine the node that orchestrates different parts of process execution.

The processing in the WSN pool outside of any sub-process is always orchestrated by a single "bridge" node interconnecting the WSN with a standard business process execution engine. If we applied the same approach also to orchestrate the execution of any sub-process in the WSN pool, the bridge would be involved in every step of process execution. This would generate plenty of network traffic: every node involved in a sub-process would need to inform the bridge of the state of execution at every step, significantly impacting the WSN lifetime. In domains where connectivity between the bridge and the WSN cannot be taken for granted [8], [11], in addition, this approach would simply be unfeasible.

To ameliorate this issue, we make it possible to transfer the execution of individual sub-processes to other arbitrary WSN devices. In Figure 1, for example, the *CO2Sensing* sub-process executes on all actuator nodes of a given room—the one communicated via the message that originally triggered the creation of the outer process. These actuators perform CO_2 sensing remotely through the *report* action bound to the *sense* CO_2 task. We use *target*s to specify where a sub-process executes. In this example, we employ LN to instantiate the *target* meta-abstraction. Therefore, target expressions are represented in Figure 1 next to the name of the sub-process as Boolean predicates over application-defined node attributes, whose definition we illustrate next. Whenever a sub-process terminates, the control flow returns to the bridge node, which continues the execution.

Node attributes. To use *target* meta-abstractions, the node attributes must be available at the BPMN level. To that end, the relevant node attributes are specified in the *Application Capability Model* (ACM), which defines a "schema" of attributes to describe a node's capabilities.

For example, the ACM for our reference application may include a "function" attribute to distinguish nodes equipped with sensors from those equipped with actuators, as well as sensor-specific information such as accuracy or maximum sampling frequency in the former case. Because of its abstract nature, the ACM may also be used to represent existing sensor classification schemes [26]. The actual values taken by attributes on a given node are deployment-specific. In a given deployment of the adaptive ventilation application, a specific CO_2 sensor deployed in room A would be, for example, described by four attributes: ⟨function, sensor⟩, ⟨type, CO_2⟩, ⟨accuracy, 1ppm⟩, and ⟨room, A⟩.

Whenever applicable, node attributes may be denoted

as static, i.e., constants, or dynamic, i.e., values known only at run-time. This distinction enables more efficient compilation. For example, whenever *target*s are used to specify the location of execution for sub-processes, as in Figure 1, we can partition the code during compilation based on the static part of target expressions.

Performance annotations. We tackle the problem of expressing high-level non-functional requirements, described in Section 4.1, by providing performance annotations in BPMN, e.g., to optimize latency vs. energy consumption depending on the application's state. The annotations are conceived as extensions of the BPMN *group* element, and depicted as dotted rounded boxes. In Figure 1, we use this construct to set the WSN nodes into "low-power" mode while they wait for a meeting to begin, and into a "reliable transmission" mode during the meeting.

An underlying run-time adaptation framework would monitor the execution of `makeSense` applications and, if needed, realign the system configuration with the desired performance goals. Section 6 describes a proof-of-concept implementation of such adaptation framework. The goal of this is merely to let us verify that the performance annotations are sufficiently expressive to tune the system's operation towards different performance goals, and that the `makeSense` run-time support provides the necessary hooks.

5 Macro-programming Language

To tackle the unification challenge, the design of *m*PL is centered on two key aspects: *i)* the specification of interfaces to implement the meta-abstractions in Section 3; and *ii)* the definition of a *core language* to express the computation required – at the WSN-level – to coordinate concrete abstractions, according to the meta-model of Figure 4.

The resulting language has a dual role: it provides the target language for compilation of the extended BPMN models, as shown in Figure 2, and can be used directly by developers in a stand-alone fashion[1]. To facilitate the translation of the extended BPMN models, we design *m*PL as an imperative object-oriented (OO) language with static typing. This also fosters code re-use and robust implementations, which are particularly challenging to achieve in WSNs [1], [3]. Although the syntax is reminiscent of Java, the concrete language semantics differs in many respects, because of the need to reconcile the OO paradigm with the resource scarcity of typical WSN nodes.

The remainder of this section focuses on the fundamental features of *m*PL and its role as part of the `makeSense` toolchain. The complete specification of the language is available [45].

5.1 Meta-abstraction Interfaces

The meta-abstraction interfaces, shown in Figure 6, capture the constraints on how meta-abstractions can be composed, as previously illustrated in the meta-model of Figure 4.

1. When *m*PL is used as a stand-alone language, or when *m*PL code is manually modified after the model editor translates the original model into *m*PL, the benefits of the model-driven approach are obviously lost. We discuss this further in Section 6.

```
interface MetaAbstraction extends Serializable {}
interface Action extends MetaAbstraction {
  void execute();
  void stop();
  Boolean isDone();
  void waitDone();
  void setParameter(Object key, Object value);
  Object getParameter(Object key);
  Object checkError();
}
interface LocalAction extends Action {
  Boolean hasResult();
  Object getResult();
}
interface DistributedAction extends Action {
  void setTarget(Target t);
  Target getTarget();
}
interface TellAction extends DistributedAction  {
  void setAction(Action a);
}
interface ReportAction extends DistributedAction {
  void setAction(LocalAction la);
  void setDataOperator(DataOperator dop);
  DataOperator getDataOperator();
  Boolean hasResult();
  Object getResult();
}
interface CollectiveAction extends DistributedAction {
  void setAction(LocalAction la);
}
interface Modifier extends MetaAbstraction {}
interface Target extends Modifier {}
interface DataOperator extends Modifier {
  Object create(Object value);
  Object nextState(Object state, Object value);
  Object evaluate(Object state);
}
```

Figure 6. Meta-abstraction interfaces.

Moreover, they specify the operations associated with meta-abstractions, which concrete abstractions must implement.

The `Action` interface provides an `execute` method to trigger the action, which is limited to the local node in case of a *local* action, or may span multiple nodes for *distributed* actions. For instance, in the case of a *report* action, a call to `execute` starts the process of gathering data from the target nodes. The `stop` method interrupts any processing triggered by `execute`. An action's execution is asynchronous w.r.t. the caller, that is, a call to `execute` does not block. This applies also to *local* actions, as in many cases their duration may be non-negligible, e.g., when operating a mechanical actuator. To enable the caller to synchronize with an action's execution, `waitDone` blocks the caller until the action is terminated, whereas `isDone` performs a non-blocking check on the state of an action's execution.

Actions may also accept parameters to customize their behavior. For instance, a *local* action for opening a vent may accept a parameter specifying the required vent opening angle. The `setParameter` and `getParameter` methods allow developers to specify an action's parameters as key/value pairs. Moreover, the execution of an action may fail; the `checkError` method allows the caller to verify whether an action is successfully executed.

`LocalAction` refines `Action` with the ability to return results to the caller, e.g., a sensor reading. The `hasResult` and `getResult` methods provide an iterator-like interface to access the results. The former is a non-blocking operation

that returns whether results are present; the latter is a blocking operation that returns the first available result. `DistributedAction` introduces the ability to specify a *target* for the action execution, according to Figure 4, therefore limiting its effects to a subset of the WSN nodes. The `Target` interface, although empty, ensures through type-checking that concrete abstractions providing *target* functionality are correctly composed with actions.

Concrete abstractions providing distributed functionality are expected to implement the interfaces `TellAction`, `ReportAction`, or `CollectiveAction`, which extend `DistributedAction`. These provide a `setAction` method that allows developers to specify the action to execute on the "many" side, as illustrated in Section 3. This action is executed in parallel on all nodes targeted by the `DistributedAction`. In the case of a `TellAction`, this can be any `Action` including further `DistributedActions`, which developers can exploit to build schemes with a distributed control flow. For example, the processing of Figure 5 can be specified with *m*PL as:

```
TellAction tell = ...
tell.setTarget(...);      // actuators for ventilation
ReportAction report = ...
report.setTarget(...);    // CO_2 sensors in the same room
LocalAction la = ...      // local action to read CO_2 values
report.setAction(la);
tell.setAction(report);
tell.execute();
```

Unlike `TellAction`, the other distributed actions `ReportAction` and `CollectiveAction` can only execute a `LocalAction`, as in Figure 4. Developers access the results of a `ReportAction` via an iterator-like interface, as in `LocalAction`. Moreover, developers can associate a `ReportAction` to a `DataOperator` instance to perform additional processing on gathered data.

The `DataOperator` interface provides methods to enable in-network processing of gathered data, based on an incremental evaluation of aggregate state [46]. The `create` method initializes the aggregate; the `nextState` method computes a new aggregate starting from the current aggregate and an additional value to merge; the `evaluate` method computes the final aggregate. Numerous data processing algorithms, especially those typically employed in WSNs, can be expressed in this way [33].

Concrete abstractions are integrated in *m*PL as classes implementing the interfaces of Figure 6. These classes differ from regular *m*PL classes in that they are typically not encoded in *m*PL, but they implement the required functionality in low-level C code. The use of C code increases the efficiency and facilitates the reuse of existing WSN implementations. We expect the implementation of concrete abstractions to be conducted by WSN experts already familiar with C programming on embedded devices.

5.2 ScriptActions

The notion of BPMN sub-processes described in Section 4.2 requires to relocate the orchestration of an arbitrarily complex sequence of operations—not just a single `Action`—to a subset of WSN devices, as in the example of Figure 1.

For instance, imagine a specification stating that, on all CO_2 sensor nodes, a reading should be acquired and reported to a nearby actuator only if the battery level is above 20%. The actions to carry out on every CO_2 sensor node are themselves a composition of multiple actions: the actions of acquiring the sensor reading and of checking the battery level are both `LocalActions`, while the action of sending data to an actuator is a `TellAction`. Further, these actions are themselves "glued" together by some application logic. Such a complex behavior cannot be expressed by a single `Action`.

To cater for these needs, we define a modularity construct called `ScriptAction`. The behavior of `ScriptActions` is not predefined, but specified by the application developer using custom *m*PL code. A `ScriptAction` can therefore be used to define arbitrarily complex command sequences to be executed on the nodes targeted by a `DistributedAction` A `ScriptAction` is a sub-interface of `Action`, and can be used wherever the latter is allowed, as in Figure 6.

In the above example of CO_2 sensor nodes, one would write the top-level code for the `TellAction` as

```
ScriptAction a = ...
TellAction t = ...          // the top-level tell action ...
t.setTarget(...);           // ... limited to CO_2 nodes
t.setAction(a);             // a is a ScriptAction
```

where `a` is an instance of a class implementing the `ScriptAction` interface, whose `execute` method would look like:

```
void execute() {
  LocalAction co2 = ... // reading CO_2
  LocalAction battery = ...  // reading battery levels
  TellAction toActuator = ...    // creates a tell action ...
  toActuator.setTarget(...);     // ... towards a nearby actuator

  // reads the CO_2 sensor value
  float p = (float) co2.execute().getResult();
  // checks the battery level
  if ((float) battery.execute().getResult() > 0.2)
    // sends CO_2 reading to the actuator
    toActuator.execute();
}
```

Based on the type rules of the interfaces, `ScriptActions` can be used only in conjunction with `TellAction`, whose `setAction` method accepts a generic `Action`. The name `ScriptAction` refers to the fact that its use in conjunction with a `TellAction` resembles the ability to remotely execute a script. Note, however, that this does *not* imply that the `ScriptAction` is necessarily migrated at run-time. We argue that in the common cases where this technique is used, the "scripts" can be pre-loaded on the target nodes, and invoked at the appropriate time. The *m*PL macro-compiler performs the necessary code allocation as explained in Section 6.

5.3 Core Language

*m*PL provides control flow constructs to coordinate the functionality of concrete abstractions and multithreading support to ease the translation of the extended BPMN diagrams. *m*PL's core language is therefore designed to express actual computation, possibly unfolding through multiple execution units that proceed in parallel. We illustrate here the key concepts; additional details on the design of the core language are also available [47].

Language design. To ease the translation of the extended BPMN models to *m*PL, we provide the key OO abstractions through the familiar Java language syntax. Therefore, the

object model is similar to Java: it supports single inheritance for classes, but a class may implement an arbitrary number of interfaces.

We also need to cater for compiling *m*PL to efficient binary code that fits within the resource constraints of WSN nodes. The memory model thus is different from many OO languages, and supports different allocation schemes. In addition to dynamic allocation of objects on the heap, we also support automatic allocation on the stack and static global allocation. Using these features, programmers can implement code with more predictable memory requirements, as memory overflows are extremely difficult to detect in WSN software [48]. Like in Java, objects are always accessed via references independent of how they are allocated.

To support the translation of parallel executions from BPMN, *m*PL provides basic multithreading primitives, accessed by programmers with an interface closely modeled after the Java `Thread` interface. We implement this interface using the Contiki [5] multi-threading library. In doing so, a potential issue is the high memory overhead, as each thread has its own stack that needs to be constantly kept in memory. To cope with this, *m*PL only supports co-operative multi-threading and a pre-defined maximum number of concurrent threads. If the maximum is reached, attempts to create further threads fail and an error is signaled.

Embedded code. Many of the existing WSN abstractions require extensive configuration, and provide their own language to this end [1]–[3]. These languages are quite diverse along a number of dimensions, ranging from purely declarative [49] to functional designs [50], and from SQL-like [51] to entirely custom syntax [34]. When integrating one such abstraction in `make`*Sense*, the question arises as to how *m*PL programmers may use abstraction-specific languages.

We employ the concept of *embedded code*, represented with a special data type `code`, to address this issue. The concept shares similarities with prepared statements for embedding SQL code in programming languages to interact with DBMSs. However, embedded code fragments in `make`*Sense* are not interpreted at run-time, but they are compiled as part of the macro-program. Whenever the *m*PL compiler encounters an embedded code snippet, it triggers abstraction-specific plug-ins that parse the embedded code and produce C code eventually linked to the rest of the system, as described in Section 6. The C code implements the functionality necessary to configure the concrete abstraction.

For example, the Logical Neighborhood (LN) abstraction that we use to implement the `Target` interface, already provides a custom language [34] to select the nodes belonging to a logical neighborhood, that is, to a *target* in *m*PL. Figure 7 shows an example of the LN language as embedded code in *m*PL. In lines 2 to 6, we use embedded code to identify the set of nodes equipped with a CO_2 sensor and located in a specific room.

The example of Figure 7 also shows how abstraction-specific custom languages often refer to a node's characteristics, such as "function", "type", and "location". This is the case in LN and many other WSN abstractions [31], [49]. In `make`*Sense*, these information are encoded as node attributes in the ACM, as explained in Section 4. To this end, embedded code snippets are automatically given a reference

```
1  code neighborhoodDef = {:
2    neighborhood co2Sensors() {
3      ACM.getFunction() == "sensor" and
4      ACM.getType() == "co2" and
5      ACM.getLocation() == room
6    }
7  :};
8
9  Target co2SensorsRoomA = new LN(neighborhoodDef);
10 co2SensorsRoomA.bind("room", "meeting room A");
11
12 Report co2StreamA = new Stream();
13 co2streamA.setTarget(co2SensorsRoomA);
14 co2streamA.setAction(new ReadCo2Sensor());
15 ...
16
17 Target co2SensorsRoomB = new LN(neighborhoodDef);
18 co2SensorsRoomB.bind("room", "meeting room B");
19
20 Report co2StreamB = new Stream();
21 co2streamB.setTarget(co2SensorsRoomB);
22 co2streamB.setAction(new ReadCo2Sensor());
```

Figure 7. Embedded code example.

to a static *m*PL object `ACM` with pre-defined methods to access a node's attributes, as shown in lines 3 to 5 of Figure 7. In this example, the embedded code snippet accesses the ACM to retrieve the values of node attributes relevant to the definition of the target.

In the same example, the selection of the room is left as a symbol in the embedded code, with no concrete value associated. This allows programmers to reuse the same embedded code fragment for multiple instantiations of the same concrete abstraction. This occurs at every such instantiation by *binding* a symbol in embedded code with a variable or a concrete value in *m*PL using `bind`, as in line 10 and 18. This allows programmers to configure two LN instances targeting different rooms based on the same embedded code snippet. Whenever `bind` is used, during compilation the abstraction-specific compiler plug-in is given a reference to the bound value or variable, so it can generate C code that accesses the actual value whenever needed[2].

Without embedded code, abstraction-specific languages would need to emerge at the meta-abstraction layer, breaking the generality of the latter. In contrast, our solution allows a sharp decoupling between the meta-abstraction and its concrete instance.

6 Toolchain and Run-time Support

We implement a complete toolchain to support the model-driven approach in `make`*Sense*, as illustrated in Figure 2. The toolchain aids developers in writing `make`*Sense* applications, compiling them down to WSN nodes, and managing their execution. A prototype of the entire toolchain, along with an accompanying tutorial, is available online as a pre-configured virtual machine image [21], easing the installation for novice users.

Model editor and compiler. The model editor is a browser-based tool supporting the creation of BPMN diagrams including the extensions described in Section 4, based on

2. This also entails that if such a binding changes because of modifications in either embedded code or the *m*PL program, the *m*PL compiler needs to re-run to generate an updated C code.

the open source editor Signavio Core Components [52]. The editor allows one to create process models with the BPMN extension elements illustrated in Section 4. It does so by supporting *WSN configurations*, that is, workspaces separating the modeling of different WSN installations with different deployment-specific information in the ACM, e.g., about installed sensors and available concrete abstractions. Further details on the model editor are available in [53].

The model editor also creates an extended underlying XML structure model, which we use as a basis to translate the WSN-specific slice of BPMN model into *m*PL. The generated *m*PL code implements a state machine handling execution tokens as per the BPMN standard, but also contains code for exchanging messages via the WSN bridge connecting the WSN devices with the back-end. WSN tasks are transformed into calls to the *m*PL meta-abstraction interfaces. BPMN sub-processes in the WSN pool cause the generation of `ScriptActions`, as explained in Section 5.2.

The *m*PL code output by the model compiler is functionally complete and requires no manual intervention before it can be handed over to the next stage of compilation, unless developers are interested in fine-tuning the code by hand. In this case, the manual changes are not reflected back in the original BPMN model. Techniques exist to enable such back-propagation, and could be integrated in `makeSense` [54].

Macro-compiler. The macro-compiler takes as input an *m*PL program, either generated by the model compiler or, when *m*PL is used in a stand-alone fashion, hand-written by developers. The macro-compiler generates as output executable C code for the target platform [47], the Contiki [5] OS in our case. The resulting C code is then compiled to a deployment-ready binary using the standard Contiki toolchain.

We implement the macro-compiler in Java. An ANTLR-based combined scanner and parser first generates an abstract syntax tree representation of the input program, enriched with further information provided by a semantic analyzer that determines dependencies among the involved classes. Code generation for the target platform is handled by a dedicated back-end, currently supporting Contiki-based C code but designed to be easily replaceable to support other platforms, e.g., TinyOS [4].

The output of the code generator is input to a *code allocator* module, which leverages the dependency information to slice the code for different nodes, based on their role. For example, the WSN bridge most often employs a different set of classes than sensors and actuators, as it does not demand interactions with the environment, but requires the ability to parse external messages. The ability to slice the code accordingly improves the utilization of program memory, precious on WSN nodes.

Embedded code snippets are handed over to abstraction-specific *compiler plug-ins* as the macro-compiler encounters them. Every concrete abstraction employing embedded code is thereby required to provide one such compiler plug-in, responsible for parsing and translating the abstraction-specific embedded code into executable C. The macro-compiler includes specific Java interfaces to enable the integration of such plug-ins.

Run-time adaptation. The extended BPMN diagram may include annotations specifying performance objectives at different stages of the process, as described in Section 4. Moreover, the communication protocols supporting the concrete abstractions need continuous monitoring and optimization to cope with unpredictable environment dynamics [36]. To meet these varying requirements, the protocols must dynamically re-configure.

The kind of run-time adaptation problem we face in `makeSense` differs from existing WSN literature, where the dominating approach is to analytically model a protocol's operation and running optimization algorithms on such model [55]. In `makeSense`, we do not deal with a single protocol whose operation is known in detail. Instead, multiple protocols run simultaneously, each supporting a different concrete abstraction, while their operation may not be precisely known, e.g., whenever a new concrete abstraction is added with a custom run-time support.

Albeit not the central focus of our work, we verify that such an adaptation problem is practically solvable by implementing a proof-of-concept adaptation framework. In doing so, we confirm that the performance annotations provide sufficient information to express given performance objectives, and that the underling `makeSense` run-time support provides the necessary hooks to enact adaptation decisions.

Our proof-of-concept adaptation framework [56] is based on Monte Carlo Q-learning [57], and operates without requiring analytic protocol models. The optimization process takes as input the performance objectives in the BPMN diagram, the current system state, e.g., the link qualities, and the *complete* WSN binary output by the `makeSense` toolchain. Thus, we do not consider a single protocol, rather the same combination of application-level functionality and network protocols in use in the real system. Developers add custom abstractions to *m*PL by simply tagging the "tunable" parameters, and let the adaptation process explore different settings for those parameters as part of the binary image.

The learning process occurs through extensive simulations performed to infer optimization *policies*, i.e., rules mapping a system state to a set of protocol parameters optimized for a given performance objective. The policies are injected back into the network, where a dedicated sub-system monitors a node's state and possibly executes policies as required. The process relies on the COOJA simulator [58], which allows time-accurate executions of binary code. Because of the binary-level execution, the approach is applicable also to OSes other than Contiki.

The model compiler uses dedicated APIs in *m*PL to translate the performance annotations to executable *m*PL code. In a sense, performance annotations in BPMN act as an "aspect" [59] that percolates through the model- and macro-compiler in Figure 2, down to the executable binary. One may retain the same architecture, but provide different implementations for the same *m*PL APIs to integrate more sophisticated adaptation mechanisms [60], [61].

Execution and maintenance. The Contiki/C code output by the macro-compiler runs on two target WSN platforms: the popular TMote Sky [62] and WisMote [63] devices. As Contiki itself is quite portable, moving to different platforms does not pose excessive challenges.

We provide tools to monitor the system's performance after deployment. Through a dedicated GUI, developers obtain statistics such as packet reliability and energy con-

sumption, which are useful to further tune the performance objectives. An over-the-air reprogramming tool also allows one to replace the code running on a user-specified subset of nodes without manually retrieving them, as is common practice in WSN deployments. The overhead of transferring the whole binary image, as normally required because of the limitations of common WSN operating systems, may be reduced by integrating existing differential re-programming approaches. This is straightforward, as the makeSense toolchain eventually produces C code, which is the target language for many such approaches [64]. These functionalities offer great flexibility after deployment, e.g., to change the application logic or to apply bug fixes.

7 Evaluation

We evaluate makeSense along three dimensions. Section 7.1 assesses the primary goals of makeSense: *i)* facilitating the integration of WSNs with business processes—the *integration* challenge, and *ii)* the provision of a unified WSN programming framework—the *unification* challenge. Section 7.2 studies the performance penalty that the higher-level of abstraction in makeSense imposes. Section 7.3 reports about a real-world deployment in a student dormitory in Cádiz (Spain), which is also the opportunity to estimate the benefits in total cost of ownership that makeSense may enable.

7.1 Benefits to Software Development

Through separate user studies, we assess how makeSense facilitates the *integration* of WSNs with business processes, discussed in the following section, and the effectiveness of the *unified* WSN programming framework, analyzed in Section 7.1.2. We apply different approaches in the two studies to best cater for the different goals and expected user base at the business process or programming level.

7.1.1 Business Process Integration

We assess whether our approach does indeed simplify the development of WSN-integrated business processes.

Approach. We recruit six developers at SAP, with an average age of 29.5 years and no previous knowledge of makeSense. Only one participant is familiar with WSNs; half of them report average knowledge of BPMN; the others have little or no knowledge of either. Beforehand, we give the participants a basic introduction to business process modeling and BPMN, WSNs, and the makeSense model editor. This introduction lasts less than an hour—from a business perspective, an acceptable training effort.

Next, we illustrate a test scenario similar to the Cádiz deployment. The participants are asked to complete different modeling tasks of increasing complexity, varying from "*open the editor and select the correct WSN configuration*" (Task 1) to "*configure outgoing edges in such a way that ventilation is turned on if CO_2 is greater than 1000 ppm; otherwise it is turned off*" (Task 21). After each task, we note down to what extent the participants solve the task, in a "no success", "success with help" and "success" scale. We also collect the encountered problems and offer help to those who do not complete the task, ensuring that all participants are on the same level when starting with the next task.

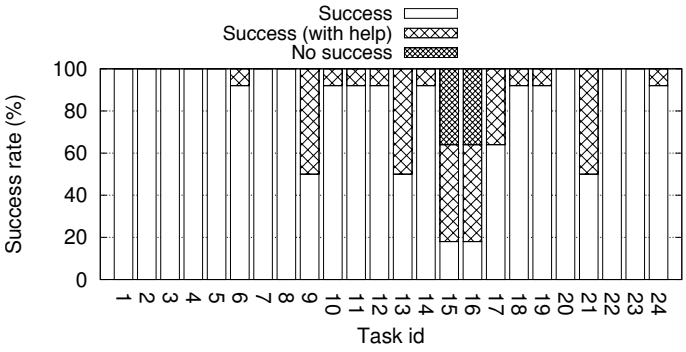


Figure 8. Success rates per modeling task.

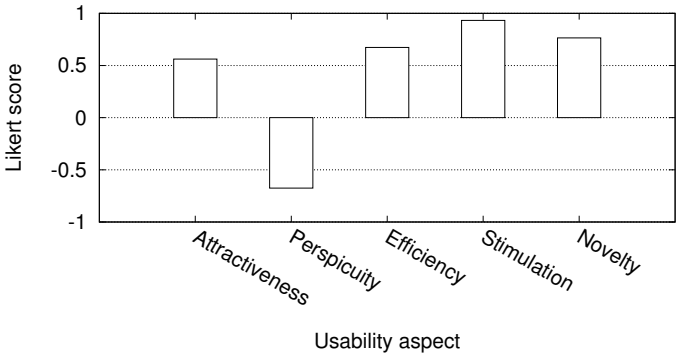


Figure 9. Results from the modeling language questionnaire.

After the exercise, the participants are asked to fill in a standardized User Experience Questionnaire (UEQ) [65], asking for a score on a Likert scale of how the tool appeared both in its pragmatic and hedonic qualities.

Results. Our main indicator for success is the task completion rate, depicted in Figure 8. For 22 out of the 24 tasks, all participants can complete the given task successfully with little or no help. Tasks 15 and 16 prove the most challenging. They are about creating *target* specifications based on Boolean expressions, which the model compiler eventually translates into LN embedded code snippets. As the users only had a brief introduction to the syntax of the expressions, they struggle formulating them correctly. Only one user completes them successfully without support, and two users do not complete them at all. However, all users complete the later Task 23 on their own, which requires similar *target* specifications. Therefore, the skill for creating target expressions can be acquired fast.

These results suggest that in the settings we test, users without expertise in WSNs successfully design and implement WSN-integrated business processes with makeSense. Overall, 4 users are able to complete all tasks with little or no help. In particular, we observe that users are able to familiarize themselves with the presented tool rather quickly. As shown in Figure 9, the usability assessment of the makeSense editor reveals that users finds the system attractive, efficient, interesting (stimulation), and innovative (novelty), as the values above average indicate. However, the questionnaire also reveals that participants sometimes find it hard to find the interface elements for a task (perspicuity).

The feedback we receive generally shows room for improvement w.r.t. the selection of WSN nodes using *target* meta-abstractions. As already noted, the participants

struggled when selecting the relevant nodes using Boolean expressions, which currently have minimal support from the model editor. While the success rates in the second part of the scenario indicate that users did learn this mechanism eventually, the development of additional GUI elements for this feature could help simplify this step. The editor interface could also be improved by merging WSNs elements into the categories of their BPMN counterparts, e.g., listing WSN tasks besides the standard BPMN tasks, by providing hints for data types, and by showing an indication of the created meta-abstraction directly in the BPMN diagram.

Overall, the data collected in the usability study indicates that our approach is effective in facilitating the design and implementation of WSN-integrated business processes.

7.1.2 Macro-programming Language

Besides representing the compilation target for the model compiler, *m*PL may also operate as a stand-alone WSN programming language. We assess its effectiveness in this respect as described next.

Approach. We mainly target novice WSN programmers, namely, persons with average expertise in developing distributed software using conventional programming languages (e.g., Java) and development frameworks (e.g., JEE), yet without specific skills in WSN programming. To this end, we recruit 19 graduate students at Politecnico di Milano, Italy. The students are completely extraneous to makeSense, but successfully attended several general courses on software engineering. Because of their background, the students involved well represent the skills of a junior software developer.

The students are first given a 1.5-hour introduction to WSNs. Next, we teach the students programming WSNs with Contiki/C. The tutorial, lasting 6 hours in total, mirrored previous Contiki tutorials, and included exercises that the participants performed on their own; for example, to setup staple networking functionality such as data collection or to implement control loops based on sensor data. The following day, we teach the students *m*PL programming. We first briefly present makeSense in a 30′ seminar. Next, we walk them through the tutorial [21]. This tutorial lasts about 5 hours, and comprises exercises similar to the Contiki one, including the implementation of decentralized control loops similar to Figure 5.

At the end of the second day, the students are given a link to fill a survey about their experience. Questions in the survey are split in two parts: questions to gauge background and skills, and questions on the use of *m*PL. We report a summary of the results next. The detailed description of the answers is also available [66].

Results. Answers to questions about the participants' background confirm their skills: 79% of the participants claim 5+ years experience with OO languages. On the contrary, they appear fairly new to WSNs: 79% of the participants indicate they learned about WSNs since about three months and about 90% of their knowledge on the subject comes from our general introduction. The participants hence do match the kind of developers we target with makeSense.

Table 1 summarizes the answers on the use of *m*PL. The concept of macroprogramming, known to cause confusion because of the many flavors available [1], appears nonetheless fairly intuitive for makeSense (Q1). The less conventional concept of meta-abstraction seems slightly harder to master; the general agreement, however, is that it is still sufficiently intuitive (Q2). The general feeling is also that makeSense greatly simplifies the implementation of both data collection (Q3) and control applications (Q4). We argue that this especially comes from the clear separation of distributed actions into *report*, *tell*, and *collective* actions. Overall, much of the positive feedback we collect is due to the higher level of abstraction *m*PL offers compared to OS-level WSN programming [3].

Question	Positive · · · Negative				
Q1: *Do you find the concept of macroprogramming in makeSense intuitive?*	47	43	10	0	0
Q2: *Do you find the concepts of meta-abstraction and concrete abstractions intuitive?*	17	73	5	5	0
Q3: *Does makeSense simplify development of data collection applications?*	90	5	5	0	0
Q4: *Does makeSense simplify development of control applications?*	95	5	0	0	0
Q5: *Does object-orientation in makeSense help achieve re-usable implementations?*	90	5	5	0	0
Q6: *How easy is it to adjust and to tune the makeSense tool-chain?*	0	5	32	58	5
Q7: *Is mPL easier to understand and use than Contiki/C?*	48	52	0	0	0
Q8: *How smoother is mPL's learning curve compared to Contiki/C?*	10	52	38	0	0

Table 1
Summary of answers to the questionnaire on *m*PL. Answers are possible in a 5-grade scale, from most positive to most negative. Values are in percentages.

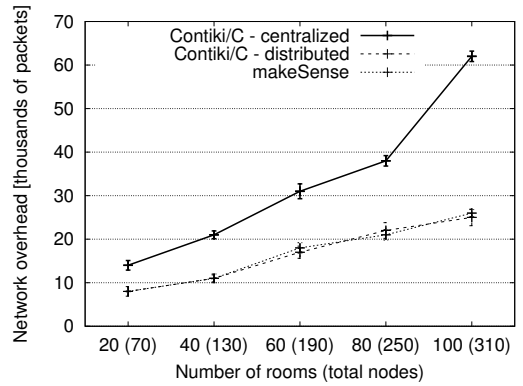
The adoption of a lightweight OO programming model in makeSense also appears well received (Q5). Notwithstanding the system overhead necessary to support an OO programming model, we thus maintain that this design decision pays off. On the other hand, the weakest point is generally identified in the toolchain (Q6). This was somehow expected: we mostly concentrated on designing and implementing the programming model, whereas we limited toolchain support to the bare functionality necessary to obtain running implementations. Improving this aspect, however, "only" entails some additional implementation work with no significant technical challenges involved.

Compared to Contiki/C, makeSense is perceived as easier to understand and to use (Q7). The learning curve is similarly considered smoother for makeSense than for Contiki/C (Q8). This is the key comparison we are interested in, and the results confirm we do achieve our overall goals. Using *m*PL, the participants were indeed able to develop fairly complex applications in a matter of a few hours. Within the same time, using Contiki/C they could not progress farther than a few toy examples.

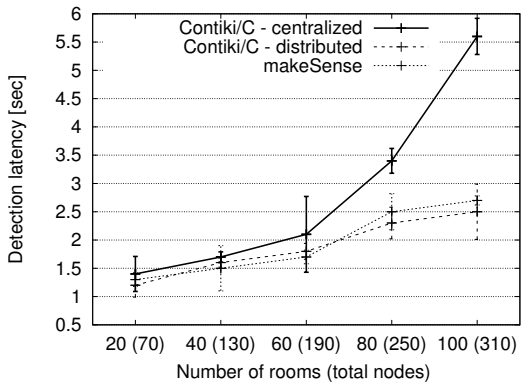
7.2 System Performance

The benefits makeSense enables come at the cost of increased run-time overhead, which we study in the following.

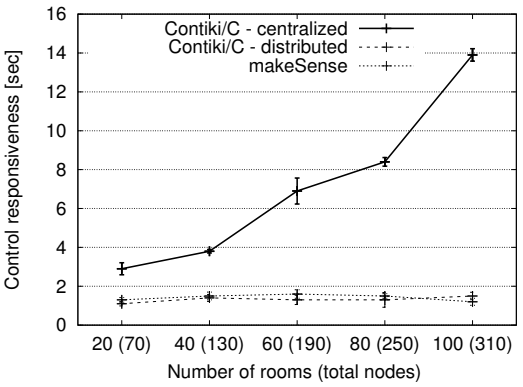
Previous results. In existing work [47], we quantitatively assessed specific overhead figures by comparing representative WSN applications in *m*PL against functionally equivalent Contiki/C implementations. We investigated performance metrics such as *code size*, indicating the size of the binary image to deploy onto the WSN nodes, and *memory*



(a) Communication overhead.



(b) Detection latency.



(c) Control responsiveness.

Figure 10. Large-scale performance in the building ventilation scenario.

consumption. These metrics, rarely significant for conventional platforms, are key for WSNs because of resource constraints. For example, with only 48 KB of program memory on TMote Sky nodes, an increase in code size of a few KB may determine whether the resulting binary does fit at all.

The results indicated that although `makeSense` introduces a non-negligible system overhead, the absolute values make it still applicable to typical WSN platforms. Further, the Contiki/C implementations are strictly tied to the considered applications, whereas the `makeSense` counterparts are more generic. Based on our analysis, a Contiki/C implementation providing a level of flexibility similar to `makeSense` would likely show a similar resource consumption [47].

In contrast, here we concentrate on how such system overhead may possibly impact the ability of a `makeSense` system to *scale* to large scenarios. This serves to demonstrate the general applicability of `makeSense`.

Setup and metrics. We use the COOJA time-accurate simulator [58] to re-create a scenario akin to the one in the Section 1. A custom Java program runs the business process in Figure 1 by fictitiously generating meeting schedules. The readings for CO_2 and presence detection sensors are randomly generated from a uniform distribution.

We simulate a variable number of meeting rooms, each including one CO_2 sensor, one ventilation actuator, and one "controller" device that gathers data from the sensor and accordingly operates the corresponding actuator. We also simulate ten "backbone" WSN devices acting as wireless routers to ensure connectivity. The wireless connectivity among nodes is determined by enforcing all devices in a room or in the backbone to be within each other's wireless range, and by randomly deciding what node in the backbone is within the communication range of one in a room. Even though the setup is that of the adaptive ventilation application, the network topology, the traffic patterns, and the data paths are quite common in industrial WSN applications [11], [18], which gives our results wider validity.

We consider both system-level and application-specific metrics. As for the former, we measure: *i)* the *processing overhead*, indicating the microcontroller usage on the WSN devices; and *ii)* the *network overhead* as the number of packets transmitted over the air. These metrics are fundamental for WSNs; for example, the communication overhead is often regarded as a proxy for energy consumption, which ultimately determines the system's lifetime. To study the application's delivered quality of service, we measure: *iii)* the *detection latency* at the back-end to register a room as busy or free, depending on the readings of presence sensors; and *iv)* the *responsiveness of the control loop*, determined as the time the system leaves a room without ventilation when the CO_2 readings are above threshold, or vice-versa[3].

We compare the `makeSense` implementation against two functionally-equivalent Contiki/C implementations, based on different architectures. One is completely *centralized*, that is, all sensor data is funneled to a single device that acts as a bridge to the business process, and the controller nodes in every room are not utilized for application-level processing. Actuation commands are centrally decided as well, and then distributed to the actuators. This architecture mirrors approaches where the integration with external business processes occurs by means of custom-developed application-specific proxies [69]–[71].

The other Contiki/C implementation is *distributed* in that it allocates the CO_2 sensing and presence detection tasks in Figure 1 to controller nodes in every room. This requires a significant development effort using Contiki/C: the application code must be *manually customized* and *tuned* depending on the type of node. With `makeSense`, the use of sub-processes—eventually translated into `ScriptActions`—and of properly defined `Targets`, together with the code allocation functionality in the macro-compiler make this process automatic.

3. Sensing room occupancy by comparing CO_2 readings against a threshold may appear trivial. In fact, much more sophisticated approaches exist [67], [68]. The specific way to process sensor readings is, however, orthogonal to our work with `makeSense`, which imposes no restrictions on the application logic.

We ask seasoned Contiki programmers to implement either of the baseline applications, and to do so in the most optimized fashion, to provide a challenging baseline to compare makeSense performance against. Each run lasts 24 simulated hours. The results we present next are averages over at least 10 repetitions of the same parameter settings with varying wireless topologies.

Results. We measure a similar processing overhead across all implementations we test, with a *worst-case* 9.2% overhead for makeSense compared to the distributed implementation with Contiki/C. The application we consider does not include any particular heavy processing, as most WSN applications we target [7]. The added MCU usage in makeSense is mostly due to OO support in *m*PL. This requires additional functionality compared to a hand-written C implementation that can be finely optimized for the task at hand. In absolute terms, however, the overhead is limited, and not expected to impact the general applicability of makeSense.

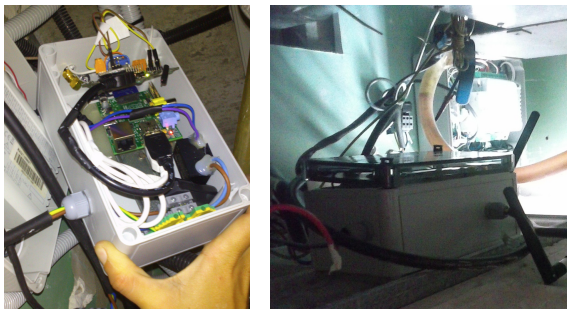
Figure 10 reports the results we obtain for the remaining performance figures. Figure 10(a) shows that the network overhead is approximately the same using makeSense or the distributed Contiki/C implementation, essentially because the data paths within the network and the underlying protocols are largely the same. This is the most important aspect determining a system's lifetime, which is thus minimally affected by makeSense when compared with a Contiki/C implementation that, however, requires significantly greater development effort.

In contrast, the centralized Contiki/C implementation in Figure 10(a) scales much worse, culminating at more than twice the network overhead than the distributed implementations with 100 rooms. Being entirely centralized, additional rooms cause added network traffic that eventually causes collisions and re-transmissions on the wireless channel. The system's lifetime thus suffers as a result. However, this kind of centralized implementations are mandatory when the back-end integration is realized with application-specific proxies.

The detection latency measures shown in Figure 10(b) confirm these observations. The performance for makeSense and the distributed Contiki/C implementation is roughly the same. The centralized Contiki/C implementation, however, suffers from a steep increase in the latency to signal room occupancy to the business process. This is due to the increasing number of re-transmissions, caused by greater network overhead. In fact, the chart does *not* account for executions where the message to the business process is lost on the way, e.g., because a maximum number of re-transmissions is reached. This almost never happens in the distributed implementations, whereas it becomes increasingly likely with the centralized one. With 100 rooms, about one message every three is lost this way. Whenever this happens, the back-end is notified only at the next iteration of the sensing loop; for presence detection, this occurs after an entire minute.

Figure 10(c), showing the control responsiveness we measure in our experiments, again demonstrates that makeSense incurs little overhead compared to a distributed Contiki/C implementation, which however, requires specialized skills to develop. The performance for both implementations is constant, as the data paths are limited to individual rooms. Therefore, their number becomes immaterial. In contrast, the centralized Contiki/C implementation suffers similarly to Figure 10(b). The sensor readings must (unnecessarily) travel all the way up to the business process, where the actuation decisions are taken and then distributed in the opposite direction. This causes significant network traffic, which impacts this implementation's ability to scale.

Both the trends and the absolute values in Figure 10 provide evidence of makeSense general applicability, especially w.r.t. large-scale settings. In summary, makeSense can reconcile the simplicity of programming provided by application-specific proxies [69]–[71] with the efficiency of hand-coded decentralized architectures.



(a) CO_2 sensor. (b) Ventilation actuator.

Figure 11. Node installation in Cádiz.

7.3 Real-world Deployment and Total Cost of Ownership

We deploy a simplified version of the adaptive ventilation application in Section 1 in a student dormitory in Cádiz (Spain). This effort is instrumental to both test the functioning of a makeSense-based implementation in a real setting and to estimate the reduction in total cost of ownership makeSense may enable.

Installation and operation. The application adjusts the ventilation in a student's room based on CO_2 levels. The system automatically shuts down at night to prevent noise disturbing the student's sleep. We deploy three WSN nodes in each of two student rooms: one to monitor the CO_2 levels, shown in Figure 11(a); one to operate a motor hooked to a ventilation duct, shown in Figure 11(b); one to act as a controller that gathers data from the sensor and commands the actuator based on a simple threshold-based control law. We also install a WSN bridge in a nearby electrical cabinet, and an additional WSN node in the ceiling to ensure overall multi-hop connectivity. We assess the functioning of the system through the logs collected at additional devices we deploy attached to the WSN nodes. These monitor the application execution through a WSN node's serial interface.

We design and implement the software by using *tell* actions to re-locate the control law on the controller nodes, similar to Figure 5. The corresponding logic is encapsulated in a `ScriptAction` including a *report* action to gather CO_2 readings and a further *tell* action to command the actuators. A third *tell* action is used at the WSN bridge to shut down the system at night.

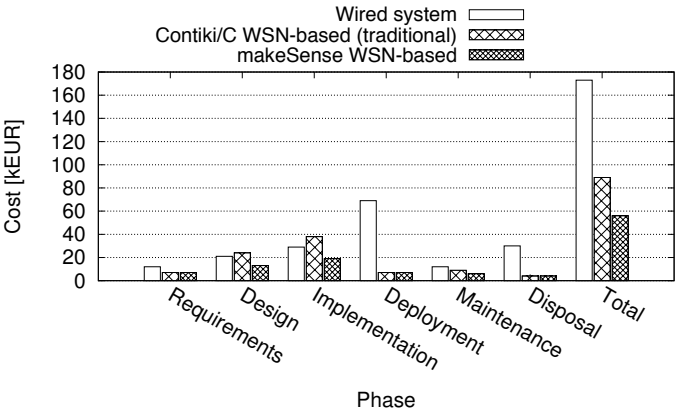


Figure 12. Estimates of total cost of ownership in the real deployment of Cádiz according to the development phase.

We run the system in Cádiz for about a week. Based on the logs, we confirm that the system worked correctly for the whole deployment duration, including night-time shutdowns. During day-time, the actuator operates the air duct according to two patterns. When a person is in a room, the CO_2 readings slowly increase until the threshold is passed. This triggers the actuator to open the duct, causing a decrease in CO_2 values until they are below threshold. At this point, the actuator closes the duct and the behavior repeats, on average every 15', as long as a person is in the room. Instead, when a person is not in the room, CO_2 readings are stable and always below threshold.

Total cost of ownership. Our installation in Cádiz replaces a conventional wired system [72]. Moreover, our industrial partners previously developed multiple WSN-based installations [73]–[75] by only employing traditional WSN programming systems. We can therefore, compare the development, installation, and operating costs of both a wired and a traditional WSN-based installation with those of makeSense. The estimates we obtain, although certainly not absolutely accurate in a quantitative sense, provide a rough indication of the savings enabled by makeSense.

For the conventional wired system, we calculate the costs using the task and material list of the earlier installation [72], which was based on a proprietary system as in most similar cases [76]. The hardware necessary for both the traditional WSN-based installation and the makeSense one is the same. For this, we refer to market prices at the time of the Cádiz deployment. Considering the costs of components, assembly, and customization at a major embedded hardware distributor [77], a CO_2 sensor node costs 200 € and an actuator node costs 250 € in quantities up to 50 items. To estimate the software development costs for a WSN-based installation, using either traditional programming systems or makeSense, we consider the hourly development cost incurred by our industrial partners in a previous similar WSN project [75]. These cover all development phases, including design, implementation, and testing. For the makeSense installation, we consider the same cost figures and scale them down to the reduction in development time observed in the user studies, as in Section 7.1.

Note that the approach above is likely to overestimate the development costs for makeSense. The specialized embedded programmers necessary for the traditional WSN-based installation are highly skilled persons [78]. Their unit cost is probably higher than that of personnel with average expertise in distributed software, whom makeSense empowers with a level of abstraction and tools sufficient to effectively develop WSN-integrated business processes.

Figure 12 reports the results of our analysis split according to the development phase. We consider a deployment of 30 nodes lasting 5 years, which is in line with existing installations in smart buildings [76]. Also note that the different phases in Figure 12 concern not only the development of the software, but also of the hardware and the costs of the actual physical installation. The savings shown for different systems in different phases are explained as follows:

- During requirement analysis, both WSN-based systems equally incur lower costs because of the increased flexibility due to wireless communications, which spares much of the effort required to consider the wiring requirements.
- During design, the traditional WSN-based approach corresponds to the highest costs because of the effort required to design custom functionality to integrate with the business process out of the WSN.
- During implementation and testing, the conventional approach pays the cost of dealing with a proprietary system, which may either require specialized training or outsourcing the implementation to the system's provider.
- In the same phases, the traditional WSN-based approach suffers the complexity of current low-level WSN programming systems. makeSense, however, speeds up this task by hiding most low-level details.
- At deployment time, again the wireless systems incur lower costs because of the flexibility of wireless communications, which avoids cabling costs and facilitates optimizing sensing and actuation points, unlike with a wired system.
- For maintenance, the conventional system possibly requires the intervention of the system's provider. The simpler WSN hardware makes maintenance easier, while the reprogramming facility in makeSense further simplifies the task.
- Disposing a wired system is extremely costly, as construction works might be needed to remove wires [73], [74]. This is not the case for both WSN-based systems.

Overall, we estimate that using a WSN-based system in place of a wired one corresponds to savings ranging from 52% (Contiki/C) to 67% (makeSense) [66]. Using WSNs, the requirement analysis, design, and implementation phases account for a significant part of the whole figure. Using WSN technology with traditional development approaches turns out costly compared to makeSense: our estimates indicate makeSense allows one to save 43% of these costs compared to traditional WSN programming systems, such as Contiki/C [66]. This is largely due to the ease of programming that makeSense enables, grounded in the specific support for integrating WSNs with business processes.

8 Related Work

We concisely survey related efforts according to their scope.

Business processes and ubiquitous technology. Integrating or augmenting business processes with ubiquitous embedded technology is often a challenge.

For example, Graja et al. [17] present extensions to BPMN to model the interactions between a cyber-physical system and the environment it is immersed in, along with the corresponding physical and control dynamics. Several works integrate IoT devices with business processes by presenting them as a form of sensing resource to the workflow definition [42], [69], [71], [79]. Yousfi et al. [41] revisit the the XML schema definition of BPMN as well as the notation, to represent business process where ubiquitous technologies are employed to generate triggering events. Baresi et al. [80] present an extension to the Guard-Stage-Milestone framework to integrate the functionality of smart objects in cross-organization business processes. Finally, Kim et al. [81] describe a method to develop adapters between BPEL documents and IoT services.

Unlike in makeSense, in these works no application-specific logic is deployed on the embedded devices. Differently, application-specific proxies are created on intermediate machines, such as gateways or home routers [70], [82]. These mediate the interactions between a standard business process engine and the embedded devices. Such an approach presents several limitations, in that it involves a significant per-application development effort. Moreover, it is likely inefficient whenever, for example, the application logic involves coordination of co-located sensors and actuators. In these cases, data that should be processed in-situ must travel across a complex network infrastructure to the business process engine and back, causing increased latency and energy consumption. We quantitatively verify this claim in Section 7.2 with the *centralized* baseline.

Model-driven development (MDD). The high-level of abstraction of MDD approaches provides an asset to move developers away from the intricacies of low-level embedded programming [83]. In makeSense, we leverage this feature as well, through the model-driven approach shown in Figure 2.

In other works, FRASAD [84] defines a node-centric software architecture and a rule-based programming model to let developers describe applications. Executable code is generated from the initial models through an automatic model transformation process. Patel et al. [85] present a development methodology that separates IoT application development into different concerns, and a supporting development framework that integrates code generation, task-mapping, and linking techniques. Vidal et al. [86] offer modeling primitives for explicitly specifying the autonomic behaviour of a cyber-physical system and model transformations for automatically generating part of the embedded code. None of these works, however, explicitly targets the integration of the embedded system with an existing back-end, as we do in makeSense.

Works also exist that use business process notations as a WSN programming language, hence not explicitly considering integration with the back-end [87]. In these cases, the business process notation merely represents a graphical alternative to traditional programming languages. For example, the work by Caracas and Bernauer [44] relies on a subset of BPMN to generate WSN code from business process specifications. Glombitza et al. [88] use BPEL to a similar end. Common to these works is is the *direct* generation of platform-specific low-level code out of high-level models. In contrast, our model-driven approach, shown in Figure 2, along with the use of *m*PL as intermediate language, caters for better platform-independence and increased extensibility. Nevertheless, we demonstrate in Section 7 that the system overhead of makeSense is comparable to existing works, and does not appreciably impact scalability.

WSN programming. Closest to our work is Baobab [89], a framework whose objective is to strike a balance between generality and applicability of the meta-model layer. In Baobab, users can extend the meta-model layer by defining their own domain-specific or platform-specific meta-models. This is similar to the developers' ability in makeSense to add new concrete abstractions to the meta-model, depending on application requirements. However, Baobab exclusively targets functionality running within the WSN, unlike the back-end integration we target in makeSense.

In the general field of WSN programming, several solutions exist to raise the abstraction level [1], [2]. For example, some systems represent the WSN as a distributed database [51], or provide sophisticated macroprogramming frameworks atop C [90] or using custom languages [50]. Database-like interfaces are usually limited to data collection applications while more complex frameworks may require learning a new language. Systems of both categories are usually monolithic and do not provide a well-defined interface to integrate application-specific abstractions. Instead, *m*PL provides an extensible programming framework.

The syntax of *m*PL's resembles Java, a language most business-oriented programmers are already familiar with. Although several Java virtual machines exist that are targeted at low-power embedded systems [91]–[93], they still require a significant amount of resources. Our approach differs by generating customized C code, which is in turn compiled into optimized binary code.

9 Conclusion

makeSense aims at simplifying the integration of WSNs into business processes. Central to makeSense is the notion of meta-abstraction, which captures fundamental WSN interaction patterns. We use meta-abstractions to extend a mainstream business process notation, and as the cornerstone of a new WSN programming language. The latter serves as intermediate language in a compilation toolchain that transforms process models to deployment-ready WSN binaries. Our evaluation shows that the performance penalty due to abstraction is well compensated by the advantages it bears, both at the modeling and programming levels. This ultimately results in reduced total cost of ownership, as we demonstrated based on real-world experience.

References

[1] L. Mottola and G. P. Picco, "Programming wireless sensor networks: Fundamental concepts and state of the art," *ACM Computing Surveys*, vol. 43, no. 3, pp. 1–51, Apr. 2011.

[2] T. B. Chandra and A. K. Dwivedi, "Programming languages for wireless sensor networks: A comparative study," in *Computing for Sustainable Global Development (INDIACom), 2015 2nd International Conference on*. IEEE, 2015, pp. 1702–1708.

[3] L. S. Bai, R. P. Dick, and P. A. Dinda, "Archetype-based design: Sensor network programming for application experts, not just programming experts," in *Information Processing in Sensor Networks, 2009. IPSN 2009. International Conference on*. IEEE, 2009, pp. 85–96.

[4] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," *ACM SIGARCH Computer Architecture News*, vol. 28, no. 5, pp. 93–104, Dec. 2000.

[5] "The Contiki Operating System," Web page—www.contiki-os.org.

[6] A. H. M. Hofstede, W. M. P. Aalst, M. Adams, and N. Russell, Eds., *Modern Business Process Automation: YAWL and Its Support Environment*. Springer, 2010.

[7] F. J. Oppermann, C. A. Boano, and K. Römer, "A decade of wireless sensing applications: Survey and taxonomy," in *The Art of Wireless Sensor Networks*. Springer, Dec. 2013, pp. 11–50.

[8] V. C. Gungor and G. P. Hancke, *Industrial Wireless Sensor Networks: Applications, Protocols, and Standards*, 1st ed. Boca Raton, FL, USA: CRC Press, Apr. 2013.

[9] J. Cabezas, E. Aguilar, A. Pastor, A. Mendez, A. Torralba, R. Carvajal, E. Aranda, and J. Viguera, "On the design of wireless sensor networks for autonomous heliostats in solar tower power plants," in *IECON 2012 - 38th Annual Conference on IEEE Industrial Electronics Society*, Oct. 2012.

[10] L. Krishnamurthy, R. Adler, P. Buonadonna, J. Chhabra, M. Flanigan, N. Kushalnagar, L. Nachman, and M. Yarvis, "Design and deployment of industrial sensor networks: experiences from a semiconductor plant and the north sea," in *Proceedings of the 3rd international conference on Embedded networked sensor systems - SenSys '05*, 2005.

[11] J. Åkerberg, M. Gidlund, and M. Björkman, "Future research challenges in wireless sensor and actuator networks targeting industrial automation," in *2011 9th IEEE International Conference on Industrial Informatics*, Jul. 2011.

[12] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer Networks*, vol. 54, no. 15, pp. 2787–2805, Oct. 2010.

[13] A. Gilchrist, "Introducing industry 4.0," in *Industry 4.0*. Apress, 2016, pp. 195–215.

[14] J. Lee, B. Bagheri, and H.-A. Kao, "A cyber-physical systems architecture for industry 4.0-based manufacturing systems," *Manufacturing Letters*, vol. 3, pp. 18–23, Jan. 2015.

[15] A. Yousfi, A. de Freitas, A. K. Dey, and R. Saidi, "The use of ubiquitous computing for business process improvement," *IEEE Transactions on Services Computing*, vol. 9, no. 4, pp. 621–632, Jul. 2016.

[16] W.-T. Lee and S.-P. Ma, "Process modeling and analysis of service-oriented architecture-based wireless sensor network applications using multiple-domain matrix," *International Journal of Distributed Sensor Networks*, vol. 12, no. 11, Nov. 2016.

[17] I. Graja, S. Kallel, N. Guermouche, and A. H. Kacem, "BPMN4cps: A BPMN extension for modeling cyber-physical systems," in *2016 IEEE 25th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, Jun. 2016.

[18] J. Åkerberg, F. Reichenbach, M. Gidlund, and M. Björkman, "Measurements on an industrial WirelessHART network supporting PROFIsafe: A case study," in *ETFA2011*, Sep. 2011.

[19] M. Zimmerling, L. Mottola, P. Kumar, F. Ferrari, and L. Thiele, "Adaptive real-time communication for wireless cyber-physical systems," *ACM Transactions on Cyber-Physical Systems*, vol. 1, no. 2, pp. 1–29, Feb. 2017.

[20] "OMG Standard: Business Process Model and Notation (BPMN) 2.0," Object Management Group (OMG), 2011. [Online]. Available: www.omg.org/spec/BPMN/2.0/

[21] "`make`*Sense* tutorial and virtual machine," Web page—makesense.sics.se/tutorial/makeSense-tutorial.zip.

[22] D. Guinard, V. Trifa, and E. Wilde, "A resource oriented architecture for the Web of Things," in *2010 Internet of Things (IOT)*, Nov. 2010.

[23] "Wyliodrin: The IDE for the IoT," Web page—www.wyliodrin.com.

[24] "IBM Node-RED," Web page—www.nodered.org.

[25] G. P. Picco, "Software engineering and wireless sensor networks: Happy marriage or consensual divorce?" in *Proceedings of the 2010 ICSE Workshop on Software Engineering for Sensor Network Applications - SESENA '10*, 2010.

[26] R. White, "A sensor classification scheme," *IEEE Transactions on Ultrasonics, Ferroelectrics and Frequency Control*, vol. 34, no. 2, pp. 124–126, Mar. 1987.

[27] P. Levis, A. Woo, E. Brewer, D. Culler, D. Gay, S. Madden, N. Patel, J. Polastre, S. Shenker, and R. Szewczyk, "The emergence of a networking primitive in wireless sensor networks," *Communications of the ACM*, vol. 51, no. 7, p. 99, Jul. 2008.

[28] A. Wilson, "Sensor- and recognition-based input for interaction," in *Human Factors and Ergonomics*. CRC Press, Mar. 2009, pp. 153–175.

[29] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis, "Collection tree protocol," in *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems - SenSys '09*, 2009.

[30] J. A. Stankovic, I. Lee, A. Mok, and R. Rajkumar, "Opportunities and obligations for physical computing systems," *IEEE Computer*, vol. 38, no. 11, 2005.

[31] Ş. Gună, L. Mottola, and G. P. Picco, "DICE: Monitoring Global Invariants with Wireless Sensor Networks," *ACM Transactions on Sensor Networks*, vol. 10, no. 4, pp. 1–34, Jun. 2014.

[32] K. Römer and J. Ma, "Pda: Passive distributed assertions for sensor networks," in *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*, ser. IPSN '09, 2009, pp. 337–348.

[33] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TAG: A tiny aggregation service for ad-hoc sensor networks," in *Proceedings of the 5th symposium on Operating systems design and implementation - OSDI '02*, 2002.

[34] L. Mottola and G. Picco, "Logical Neighborhoods: A programming abstraction for wireless sensor networks," in *Distributed Computing in Sensor Systems*. Springer, 2006, pp. 150–168.

[35] L. Mottola and G. P. Picco, "MUSTER: Adaptive energy-aware multisink routing in wireless sensor networks," *IEEE Transactions on Mobile Computing*, vol. 10, no. 12, pp. 1694–1709, Dec. 2011.

[36] N. Baccour, A. Koubâa, L. Mottola, M. A. Zúñiga, H. Youssef, C. A. Boano, and M. Alves, "Radio link quality estimation in wireless sensor networks," *ACM Transactions on Sensor Networks*, vol. 8, no. 4, pp. 1–33, Sep. 2012.

[37] A. Caracaş and T. Kramp, "On the expressiveness of BPMN for modeling wireless sensor networks applications," in *Lecture Notes in Business Information Processing*. Springer, 2011, pp. 16–30.

[38] A. Dunkels *et al.*, "D-1.1—Application and programming survey," `make`*Sense* consortium, Tech. Rep., 2010, available at goo.gl/njXuP5.

[39] L. Mottola *et al.*, "D-1.2—Requirement specification," `make`*Sense* consortium, Tech. Rep., 2011, available at goo.gl/fW88qT.

[40] P. Spieß *et al.*, "D-4.3—Final application and system capability model," `make`*Sense* consortium, Tech. Rep., 2011, available at goo.gl/NrNuYt.

[41] A. Yousfi, C. Bauer, R. Saidi, and A. K. Dey, "uBPMN: A BPMN extension for modeling ubiquitous business processes," *Information and Software Technology*, vol. 74, pp. 55–68, Jun. 2016.

[42] S. Meyer, A. Ruppen, and C. Magerkurth, "Internet of things-aware process modeling: Integrating IoT devices as business process resources," in *Advanced Information Systems Engineering*. Springer Berlin Heidelberg, 2013, pp. 84–98.

[43] R. Braun and W. Esswein, "Classification of domain-specific BPMN extensions," in *Lecture Notes in Business Information Processing*. Springer Berlin Heidelberg, 2014, pp. 42–57.

[44] A. Caracas and A. Bernauer, "Compiling business process models for sensor networks," in *2011 International Conference on Distributed Computing in Sensor Systems and Workshops (DCOSS)*, Jun. 2011.

[45] L. Mottola *et al.*, "D-3.1—The `make`*sense* programming model," `make`*Sense* consortium, Tech. Rep., 2011, available at goo.gl/QiEW0o.

[46] J. Gray, A. Bosworth, A. Lyaman, and H. Pirahesh, "Data cube: a relational aggregation operator generalizing GROUP-BY, CROSS-TAB, and SUB-TOTALS," in *Proceedings of the Twelfth International Conference on Data Engineering*, 1996.

[47] F. J. Oppermann, K. Romer, L. Mottola, G. P. Picco, and A. Gaglione, "Design and compilation of an object-oriented macroprogramming language for wireless sensor networks," in *39th Annual IEEE Conference on Local Computer Networks Workshops*, Sep. 2014.

[48] Y. Chen, O. Gnawali, M. Kazandjieva, P. Levis, and J. Regehr, "Surviving sensor network software faults," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles - SOSP '09*, 2009.

[49] C. Frank and K. Römer, "Algorithms for generic role assignment in wireless sensor networks," in *Proceedings of the 3rd international conference on Embedded networked sensor systems - SenSys '05*, 2005.

[50] R. Newton, G. Morrisett, and M. Welsh, "The Regiment macroprogramming system," in *Proceedings of the 6th international conference on Information processing in sensor networks - IPSN '07*, 2007.

[51] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TinyDB: an acquisitional query processing system for sensor networks," *ACM Transactions on Database Systems*, vol. 30, no. 1, pp. 122–173, Mar. 2005.

[52] "Signavio Core Components: Creating BPMN 2.0 diagrams," Web page—code.google.com/p/signavio-core-components.

[53] S. Tranquillini, P. Spieß, F. Daniel, S. Karnouskos, F. Casati, N. Oertel, L. Mottola, F. J. Oppermann, G. P. Picco, K. Römer, and T. Voigt, "Process-based design and integration of wireless sensor network applications," in *Lecture Notes in Computer Science*. Springer, 2012, pp. 134–149.

[54] Y. Yu, Y. Lin, Z. Hu, S. Hidaka, H. Kato, and L. Montrieux, "Maintaining invariant traceability through bidirectional transformations," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2012.

[55] M. Zimmerling, F. Ferrari, L. Mottola, T. Voigt, and L. Thiele, "pTunes: Runtime parameter adaptation for low-power MAC protocols," in *Proceedings of the 11th international conference on Information Processing in Sensor Networks - IPSN '12*, 2012.

[56] F. J. Oppermann *et al.*, "D-3.1—Report on protocol selection, parameterization, and runtime adaptation evaluation of system," RelyOnIT consortium, Tech. Rep., 2015, available at www.relyonit.eu/fileadmin/user_upload/D-3.2.pdf.

[57] R. Sutton and A. Barto, "Reinforcement learning: An introduction," *IEEE Transactions on Neural Networks*, vol. 9, no. 5, pp. 1054–1054, Sep. 1998.

[58] J. Eriksson, F. Österlind, N. Finne, N. Tsiftes, A. Dunkels, T. Voigt, R. Sauter, and P. J. Marrón, "COOJA/MSPSim: Interoperability testing for wireless sensor networks," in *Proceedings of the Second International ICST Conference on Simulation Tools and Techniques*. ICST, 2009.

[59] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP'97 — Object-Oriented Programming*. Springer, 1997, pp. 220–242.

[60] F. Fleurey, B. Morin, and A. Solberg, "A model-driven approach to develop adaptive firmwares," in *Proceeding of the 6th international symposium on Software engineering for adaptive and self-managing systems - SEAMS '11*, 2011.

[61] T. Bourdenas, D. Wood, P. Zerfos, F. Bergamaschi, and M. Sloman, "Self-adaptive routing in multi-hop sensor networks," in *2011 7th International Conference on Network and Service Management*, Oct. 2011, pp. 1–9.

[62] J. Polastre, R. Szewczyk, and D. Culler, "Telos: Enabling ultra low-power wireless research," in *IPSN 2005. Fourth International Symposium on Information Processing in Sensor Networks, 2005.*, 2005.

[63] "WisMote Sensor Node," Web page—wismote.org/doku.php.

[64] Q. Wang, Y. Zhu, and L. Cheng, "Reprogramming wireless sensor networks: challenges and approaches," *IEEE Network*, vol. 20, no. 3, pp. 48–55, May 2006.

[65] B. Laugwitz, T. Held, and M. Schrepp, "Construction and evaluation of a user experience questionnaire," in *Lecture Notes in Computer Science*. Springer, 2008, pp. 63–76.

[66] J. Eriksson *et al.*, "D-5.4—Final application implementations and evaluation of system," `makeSense` consortium, Tech. Rep., 2013, available at goo.gl/GWJP0A.

[67] S. Meyn, A. Surana, Y. Lin, S. M. Oggianu, S. Narayanan, and T. A. Frewen, "A sensor-utility-network method for estimation of occupancy in buildings," in *Proceedings of the 48th IEEE Conference on Decision and Control.*, 2009.

[68] C. Basu, C. Koehler, K. Das, and A. K. Dey, "Perccs: person-count from carbon dioxide using sparse non-negative matrix factorization," in *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, 2015.

[69] I. Amundson, M. Kushwaha, X. Koutsoukos, S. Neema, and J. Sztipanovits, "Efficient integration of web services in ambient-aware sensor network applications," in *2006 3rd International Conference on Broadband Communications, Networks and Systems*, Oct. 2006.

[70] N. Glombitza, D. Pfisterer, and S. Fischer, "Integrating wireless sensor networks into web service-based business processes," in *Proceedings of the 4th International Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks - MidSens '09*, 2009.

[71] K. Pandey and S. Patel, "A novel design of service oriented and message driven middleware for ambient aware wireless sensor network," *International Journal of Recent Trends in Engineering*, vol. 1, no. 1, 2009.

[72] "Clear-Up: Clean and resource-efficient buildings for real life," Web page—www.clear-up.eu.

[73] "FIEMSER: Friendly intelligent energy management systems in residential buildings," Web page—www.fiemser.eu.

[74] "Arrowhead: Addressing efficiency and flexibility at the global scale," Web page—www.arrowhead.eu.

[75] "TEACH: Tecnologies and tools to prioritize assessment and diagnosis of air pollution impact on inmovable and movable cultural heritage," Web page—goo.gl/B0hTAy.

[76] K. Whitehouse, "The rise of the intelligent building," *IQT Quarterly*, no. 3, 2013.

[77] "RS Components," Web page—uk.rs-online.com.

[78] P. Koopman, *Better Embedded System Software*. Drumnadrochit Education, 2010.

[79] K. Dar, A. Taherkordi, H. Baraki, F. Eliassen, and K. Geihs, "A resource oriented integration architecture for the internet of things: A business process perspective," *Pervasive and Mobile Computing*, vol. 20, pp. 145–159, Jul. 2015.

[80] L. Baresi, G. Meroni, and P. Plebani, "A GSM-based approach for monitoring cross-organization business processes using smart objects," in *Business Process Management Workshops*. Springer International Publishing, 2016, pp. 389–400.

[81] S. D. Kim, J. Y. Lee, D. Y. Kim, C. W. Park, and H. J. La, "Modeling BPEL-based collaborations with heterogeneous IoT devices," in *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*, Aug. 2014.

[82] S. Nastic, H.-L. Truong, and S. Dustdar, "SDG-pro: a programming framework for software-defined IoT cloud gateways," *Journal of Internet Services and Applications*, vol. 6, no. 1, p. 21, Aug. 2015.

[83] I. Malavolta and H. Muccini, "A study on MDE approaches for engineering wireless sensor networks," in *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, Aug. 2014.

[84] X. T. Nguyen, H. T. Tran, H. Baraki, and K. Geihs, "FRASAD: A framework for model-driven IoT application development," in *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*. IEEE, Dec. 2015.

[85] P. Patel and D. Cassou, "Enabling high-level application development for the internet of things," *Journal of Systems and Software*, vol. 103, pp. 62–84, May 2015.

[86] C. Vidal, C. Fernández-Sánchez, J. Díaz, and J. Pérez, "A model-driven engineering process for autonomic sensor-actuator networks," *International Journal of Distributed Sensor Networks*, vol. 11, no. 3, Mar. 2015.

[87] A. Caracas, "From business process models to pervasive applications: Synchronization and optimization," in *2012 IEEE International Conference on Pervasive Computing and Communications Workshops*, Mar. 2012.

[88] N. Glombitza, S. Ebers, D. Pfisterer, and S. Fischer, "Using BPEL to realize business processes for an Internet of Things," in *Ad-hoc, Mobile, and Wireless Networks*. Springer, 2011, pp. 294–307.

[89] B. Akbal-Delibas, P. Boonma, and J. Suzuki, "Extensible and precise modeling for wireless sensor networks," in *Lecture Notes in Business Information Processing*. Springer, 2009, pp. 551–562.

[90] N. Kothari, R. Gummadi, T. Millstein, and R. Govindan, "Reliable and efficient programming abstractions for wireless sensor networks," *ACM SIGPLAN Notices*, vol. 42, no. 6, p. 200, Jun. 2007.

[91] N. Shaylor, D. N. Simon, and W. R. Bush, "A Java virtual machine architecture for very small devices," *ACM SIGPLAN Notices*, vol. 38, no. 7, p. 34, Jul. 2003.

[92] F. Aslam, L. Fennell, C. Schindelhauer, P. Thiemann, G. Ernst, E. Haussmann, S. Rührup, and Z. A. Uzmi, "Optimized java binary and virtual machine for tiny motes," in *Distributed Computing in Sensor Systems*. Springer, 2010, pp. 15–30.

[93] N. Brouwers, P. Corke, and K. Langendoen, "Darjeeling, a Java compatible virtual machine for microcontrollers," in *Proceedings of the ACM/IFIP/USENIX international middleware conference companion on Middleware '08 Companion - Companion '08*, 2008.

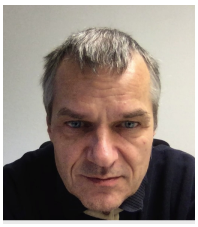
L. Mottola is Associate Professor at Politecnico di Milano (Italy) and Senior Researcher at RISE SICS. His research interests focus on networked embedded systems. He obtained the Google Faculty Award and received the ACM SigMobile Research Highlight and multiple Best Paper Awards at ACM MOBISYS and ACM/IEEE IPSN 2011 and 2009. He was PC co-chair for IEEE DCOSS 2015, ACM EWSN 2017, and ACM SENSYS 2017. He is Associate Editor for ACM Transactions on Sensor Networks.

G. P. Picco is a professor at the Univ. of Trento, Italy. His research spans software engineering, distributed systems, and networking, with a focus on wireless sensor networks, Internet of Things and Cyber-Physical Systems. His scientific awards include a "Most Influential Paper" at ICSE (2007) and best papers at IPSN (2009, 2011, 2015) and PerCom (2012). He is an associate editor for ACM Transactions on Sensor Networks, and has served in the same role for IEEE Transactions on Software Engineering.

F. J. Oppermann received his MSc in Computer Science from University of Oldenburg (Germany) in 2007. He obtained his PhD from Graz University of Technology (Austria) in 2016 based on a thesis on programming and configuration of wireless sensor networks. Currently, he is working as a systems engineer for trust provisioning at NXP Semiconductors Austria GmbH. His research interests include wireless sensor networks, the Internet of Things, and security aspects of networked embedded systems.

J. Eriksson is a senior researcher at RISE SICS where he is a member of the Networked Embedded Systems group. His current research focuses on system software and scalable routing for networked systems and the Internet of Things. He is also working closely with commercial IoT companies to help them implement reliable networking and robust software for IoT devices.

N. Finne is with RISE SICS, working with networked embedded devices. His current research focuses on system software for resource-constrained networked devices and he is an active developer of the Contiki OS.



H. Fuchs is with SAP, and has been developing innovative solutions coupling Internet of Things and business processes. Nowadays he designs, develops and tests cloud-based applications and services.

A. Gaglione is a Technologist at Digital Catapult, UK, where he is responsible for initiating and developing new innovation projects in the area of the Internet of Things (IoT). His current work focuses on fostering and driving business adoption of IoT and low-power wide-area networks, and establishing digital marketplaces for IoT-enabled smart cities. He has over 10 years' experience in academia and industry, researching networked embedded systems and their integration into the IoT. He is a member of the IEEE.

S. Karnouskos is with SAP, investigating the added-value of integrating networked embedded devices and enterprise systems. For the last 20 years Stamatis leads efforts in several European Commission and industry funded projects related to Cyber-Physical Systems, Internet of Things, Industrial Informatics, Smart Grids, Security and Mobility.

P. Moreno Montero has a degree in Physics science by the Universidad Autonoma de Madrid. At present he is part of the Eco-efficiency group in ACCIONA Construccion but he also has an ICT background, both roles developed during ten years working in ACCIONA. During this period he is combining research in-house with research in European research programs.

N. Oertel is with SAP and investigates the integration of smart items into SAP products. Nina has worked on several EU projects dealing with RFID, Internet of things, sensor networks and business process integration. Nina's research interests include data mining, business process modeling, cloud infrastructures and the business case for ubiquitous computing.

K. Römer is professor at and director of the Institute for Technical Informatics at Graz University of Technology. He held positions of Professor at the University of Lübeck in Germany, and senior researcher at ETH Zurich in Switzerland. His research interests encompass wireless networking, fundamental services, operating systems, programming models, dependability, and deployment methodology of networked embedded systems, in particular Internet of Things, Cyber-Physical Systems, and sensor networks.

P. Spieß is with SAP SE as a Design Technologist. He used to work as a researcher aiming at creating a more direct coupling of physical processes (monitored and influenced e.g. by Wireless Sensor and Actuator Networks, embedded systems, manufacturing and automotive equipment, smart meters) with business processes as they are orchestrated by modern enterprise software.

S. Tranquillini is a former postdoc researcher in the Department of Information Engineering and Computer science at the University of Trento, Italy, where he received his PhD. His main research interests were in the area of business process management and integration with network of workers, namely crowdsourcing and Wireless Sensor Networks. Today Stefano is the CTO of Chino.io, a startup that develops a solution for developers to store sensitive data in compliance with EU and US laws and regulations.

T. Voigt is a Professor of computer science at the Department of Information Technology, Uppsala University. He also leads the Networked Embedded Systems group at RISE SICS. His current research focuses on system software for networked systems and the Internet of Things. His work has been cited more than 10000 times. He is a member of the editorial board for the IEEE Internet of Things newsletter and ACM Transactions on Sensor Networks.