`make`*Sense*: Real-world Business Processes through Wireless Sensor Networks

Florian Daniel[3], Joakim Eriksson[1], Niclas Finne[1], Harald Fuchs[4], Andrea Gaglione[3], Stamatis Karnouskos[4], Patricio Moreno Montero[5], Luca Mottola[1], Nina Oertel[4], Felix Jonathan Oppermann[2], Gian Pietro Picco[3], Kay Römer[2], Patrik Spieß[4], Stefano Tranquillini[3], and Thiemo Voigt[1]

[1] SICS Swedish ICT, Kista, Sweden
[2] University of Lübeck, Lübeck, Germany
[3] University of Trento, Italy
[4] SAP AG, Germany
[5] Acciona Infraestructuras S.A., Spain

Abstract Wireless sensor networks (WSNs) have been a promising technology for quite some time. Their success stories are, however, restricted to environmental monitoring. In the industrial domain, their adoption has been hampered by two main factors. First, there is a lack of integration of WSNs with business process modeling languages and back-ends. Second, programming WSNs is still challenging as it is mainly performed at the operating system level. To this end, we provide the `make`*Sense* framework, a unified programming framework and a compilation chain that, from high-level business process specifications, generates code ready for deployment on WSN nodes. In this paper, we present the `make`*Sense* framework and the application scenario for our final deployment.

1 Introduction

Wireless sensor networks (WSN) are small, untethered computing devices equipped with embedded sensors and actuators. WSNs can be deployed much more easily than traditional wired sensors, and are able to coordinate and self-organize so that some high-level application goal is achieved. Many of the early sensor network deployments involved only sensors and realized environmental monitoring applications, that reported aggregated data to a base station [1]. While sensor networks have been successful in this domain, in other domains their adoption has been rather limited.

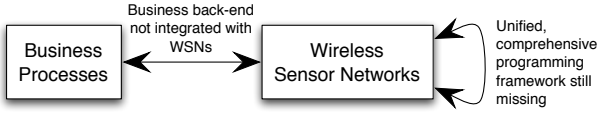

Figure 1. Open problems for using WSNs in business processes.

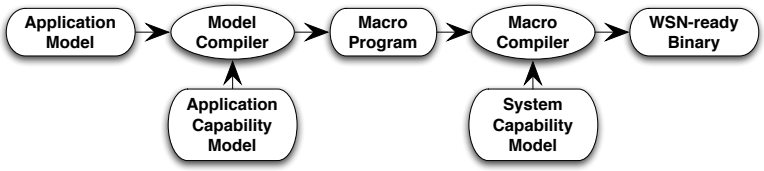

Figure 2. Compiling business process models into WSN-executable code.

As shown in Figure 1, we see two limiting factors that enable widespread adoption of sensor networks: *(i)* there is a lack of *integration* of WSNs with business process modeling languages and back-ends; *(ii)* programming WSNs is still challenging as it is mainly performed at the operating system level since there is no *unifying comprehensive programming* framework.

In the makeSense project [2], we tackle these two issues. We tackle the problem of integration by providing a holistic approach where application developers "think" at the high abstraction level of business processes, but the constructs they use are effectively implemented in the challenging reality of WSNs. Concretely, we let the application developers specify the application in a WSN-specific extension for Business Process Modeling Notation (BPMN). A model compiler transforms the extended BPMN models into traditional and WSN-specific code which allows to distribute process execution over both a WSN and a standard business process engine.

To simplify WSN programming, many programming abstractions have been developed [3], but they are hard to use since they typically focus on one specific problem. To drastically simplify WSN programming, particularly for business scenarios, we provide a broader approach that enables developers to use several abstractions at once. Towards this end, we present a unified comprehensive programming framework into which existing WSN programming abstractions can blend smoothly. These abstractions are "glued" together using a core language, a stripped-down version of Java tailored for WSNs. This macro-programming language is also the target language of the model compiler mentioned above. It can, however, also be used directly by WSN programmers. A macro compiler takes the macro-programming code as input and compiles it down to plain Contiki code that can be executed on WSN nodes or on the gateway between the sensor network and the business process engines.

Effectively, this leads to two compilation steps as shown in Figure 2. The model compiler takes as input the application model (in extended BPMN) and an application capability model. The latter is a coarse-grained description of the WSN, providing information such as the type of sensors/actuators available and their operations. The macro compiler takes as input the macro-program generated by the model compiler and a system capability model. The latter provides finer-grained information on the deployment environment (e.g., how many sensors of a given type are deployed at a location). The macro-compiler generates executable code that relies only on the basic functionality provided by the run-time support available on the target nodes. By leveraging the system

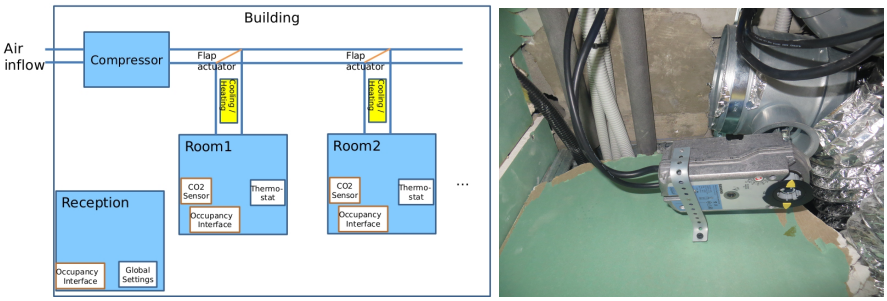

Figure 3. Deployment scenario for `make`*Sense* final deployment. An overview of the scenario (left part), and the actuator with the flap (right part).

capability model, the macro compiler can generate different code for different nodes, based on their application role.

As described in Section 6, the executable code runs atop a dedicated run-time layer, which provides access to low-level functionality such as MAC protocols and sensor devices. The run-time system also contains mechanisms enabling self-optimization of the network functionality, also described in Section 6.

The paper proceeds as follows. In the next section, we briefly present our deployment scenario. In Section 3 we discuss `make`*Sense* application modelling. The subsequent sections present the `make`*Sense* macro-programming language and the macro-compiler. We give an overview on the `make`*Sense* run-time system in Section 6 and the conclude with some final remarks.

2 Deployment

The `make`*Sense* project's deployment is in a student residence in Cadiz, Spain. As shown in Figure 3 we implement a room ventilation scenario, where the actuator (right part of the same figure) opens the flap in the student's bathroom if the measurement of the CO_2 sensor is above a configurable threshold. The external business process is managed by a room reservation system. We use an open source reservation system to manage room reservations that interacts with the sensor network through an occupancy interface. Hence, the sensor network can save energy by not ventilating rooms when they are vacant.

3 Application Modeling

For the integration of WSNs with business processes, we do not just add a service facade to the WSN or deploy middleware components on the gateway as others have done [4]. Instead, we want to enable a process modeler to model processes that are partially executed directly by the WSN itself and partially by traditional business process execution engines. Towards this end, we use and

extend the *Business Process Modeling Notation (BPMN)*. We introduce new attributes that allow the modeler to specify a new *intra-WSN participant* that contains the logic executed by the WSN. Since the latter is resource-constrained we allow only a subset of BPMN elements. Moreover, we introduce a special *WSN activity* type to be used within the *intra-WSN participant*. The WSN activity is (except for the message activity) the only allowed activity type there.

The WSN activity is backed by a meta-model that we describe in the next section. As WSNs are inherently distributed systems, we also introduce a *Target* attribute for lanes and activities within the *intra-WSN participant*, that allows specifying where the respective logic should be executed, based on labels that are relevant at the modeling layer. Finally, we add performance annotations, expressing that the WSN should optimize its operation for a specific goal (e.g., system lifetime or reliability) within a certain subsets of activities. This is used for the self-optimization in the run-time system as described in Section 6.

To assist the process modeler in creating correct, executable models, we use a set of meta-models that describe the WSN in terms of the logical functionality it provides, along with the way it is embedded into the physical set-up (e.g., which sensing or actuation is supported at which logical location). Instances of these meta-models can be created either manually or through dynamic service discovery.

At run-time, the BPMN process is executed in a distributed fashion. To execute the intra-WSN process in the WSN, it is entirely transformed into macro-code, compiled into C, and distributed by the run-time as described in the next sections. For message exchange between the intra-WSN process and the other process, the run-time uses a lightweight protocol, reducing encoded message size by using message structure information on both sides. The compilation step automatically generates process communication endpoints that handle serialization and deserialization of messages and implement process instance correlation.

4 Macro-programming Language

To bridge the gap between business processes and WSNs we defined a high level intermediate macro-programming language where the abstractions contributing to the language are decoupled, leverage on existing implementation, and can be changed or extended easily to suit specific application needs.

The makeSense macro-programming language is based on a core set of *meta-abstractions* which define the fundamental building blocks of the language as units of functionality, reuse, and extensions. They are implemented through different "concrete" abstractions and provide the key concepts enabling interaction with the WSN. The language serves as the "glue" among abstractions, whose composition can be achieved by using common control flow statements. The core language, in our case a stripped-down version of Java we tailored for WSNs, is also the *trait d'union* between the macro-programming abstractions and the BPMN business process model.

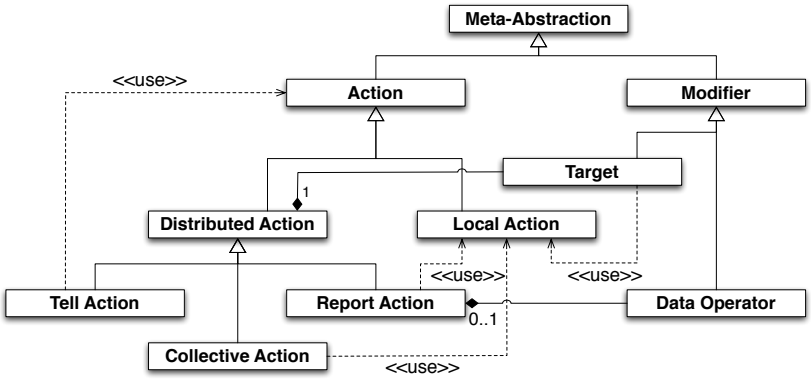

Figure 4. A model for the meta-abstractions of the `make`*Sense* macro-programming language.

Figure 4 shows a UML meta-model for the meta-abstractions provided by the macro-programming language. It focuses on the notion of *action*, a task executed by one or more WSN nodes. Actions are separated into *local*, whose effect is limited to the node where the action is invoked (e.g., acquiring a reading from the on-board temperature sensor), and *distributed*, whose effect instead spans multiple nodes.

Distributed actions may run on several nodes in parallel and are further divided into *tell*, *report*, and *collective* actions. The former two represent the one-to-many and many-to-one interaction patterns commonly used in WSNs to enable communication between the node (the "one") issuing the action and a set of nodes (the "many") where the latter is executed. A tell action enables a node to request the execution of a set of actions on other nodes, e.g., to issue actuation commands or to trigger reconfiguration of system parameters such as the sampling rate. A report action enables a node to gather data from other nodes. Event-based abstractions and periodic, continuous queries both fall in this category. Data acquisition occurring on each target node is specified by a local action given as input to the report action. The output of the local action is returned to the report one. Collective actions, in contrast to tell and report ones, do *not* focus on a special node where the action starts or ends. They enable a global network behavior and are executed cooperatively by the entire WSN through many-to-many communication. An example are distributed assertions [5], where programmers specify a (global) property monitored collectively by the WSN nodes.

The behavior of distributed actions can be customized by a *modifier*. We defined two modifiers, *target* and *data operator*. In our envisioned scenarios the nodes possibly differ along several dimensions, both physical and logical. For example, the ventilation scenario of our deployment in Section 2 requires both CO_2 sensors and flap actuators to be installed in two different rooms. Programmers must be able to map actions to the set of nodes of interest. A target identifies a set of nodes satisfying application constraints, and gives the ability to apply

a distributed action to the nodes in this set. Instead, a report action may have a *data operator*, specifying processing performed on the results after gathering and before they are returned to the caller, e.g., to filter or aggregate the data.

General concepts and operations defined by meta-abstractions are implemented by concrete abstractions, which may then provide different levels of expressiveness and run-time guarantees. To create an instance of a meta-abstraction, a class implementing its interface must be defined in the core language. As abstraction implementations typically closely interact with the operating system, methods of abstraction classes are implemented in C using a native code interface provided by the core language. Some abstractions require extensive configuration, for example, a target needs to define a set of nodes based on their properties [6]. To simplify such configuration, the core language supports the concept of *embedded languages*, code snippets formulated in the declarative configuration language provided by an abstraction. These are efficiently compiled by appropriate compiler plugins, instead of being interpreted at runtime.

The makeSense macro-programming core language provides a framework to integrate the previously described abstractions. In the makeSense framework it mainly serves as an intermediate language for the translation of BPMN models to platform code, but it is also suitable for direct use by programmers. The core language features a Java-like syntax and full support for object-oriented programming. In addition, to make the programmer's task easier, we decided to provide full multi-threading with a Java-like interface based on the Contiki `mt` library [7]. Nevertheless, as we are targeting very resource-constrained microcontrollers, the language needs to be simpler than standard Java. Consequently, some language features had to be removed. For example, the makeSense macro-programming language does not provide garbage collection, but relies on manual memory management. To reduce the resulting burden on the programmer, the language also provides specific constructs to allocate automatic or static objects, for which the memory management is handled by the compiler. In contrast to Java we do not employ a virtual machine approach, but the program is translated to target code that can be directly run on the target platform. The resulting code is predeployed on all nodes, so that it is not necessary to migrate code fragments at run-time.

Abstractions are represented in the language as ordinary classes with a predefined interface. Some abstractions require extensive configuration, for example in order to specify the set of nodes that form a target. To facilitate such configurations the macro-programming language features an extension mechanism that allows to embed abstraction-specific languages in the macro-programming code. This mechanism relies on specific compiler plug-ins as described in Sec. 5. Listing 1.1 demonstrates the use of embedded code to specify a logical neighborhood [6] to limit the scope of a stream action to this set of nodes. In lines 1 to 6 the logical neighborhood is defined by an abstraction-specific code fragment and the definition is assigned to a code-type variable neighborhoodDef. This variable is used in line 8 to associate the neighborhood definition with a new instance of the logical neighborhood abstraction. Note the use of the newly introduced

Listing 1.1. Use of embedded code in the MPL core language

```
1  code neighborhoodDef = {:
2    neighborhood co2Sensors() {
3      ACM.getFunction() == "sensor"
4          and ACM.getType() == "co2"
5    }
6  :};
7
8  Target co2Sensors = lnew LN(neighborhoodDef);
9
10 Report co2Stream = lnew Stream();
11
12 co2stream.setTarget(co2Sensors);
13 co2Stream.setAction(lnew ReadCO2Level());
14 co2Stream.setDataOperator(lnew MedianOperator());
15 co2Stream.setParameter("period", 5 * 60);
16
17 co2Stream.execute();
18
19 co2Stream.waitResult();
20
21 Object result = co2Stream.getResult();
```

`lnew` operator to create an automatic object instance for which memory management is handled by the compiler. In line 12, the neighborhood is assigned as target scope to a newly created stream action. In the following lines, additional parameters are set and finally the action is executed in line 17. After execution of the action, the program needs to wait until a result and can be fetched.

Another significant feature of the `make`*Sense* macroprogramming language is the provision of a generic object serialization interface. This feature is primarily used by the different abstractions in order to transfer object state between the involved nodes. The object serialization facility is similar to the one provided by Java and allows to write the state of an object to a standardized flat representation. This representation is, for example, suitable to be send over the network and can later be used to recreate an exact copy of the serialized object on the same or a different node. To be applicable for serialization, a class needs to implement the predefined interface `Serializable`. The serialization and deserialization functionality is automatically generated by the macro-compiler, but can be customized by overriding specific methods.

5 Macro-compiler

The `make`*Sense* macro-compiler is responsible for the translation of the macroprogramming language program to Contiki-based C code. The generated C code can than be compiled with the existing Contiki tool chain and can be finally deployed on the nodes.

The basic architecture of the compiler follows the established reference architecture. As shown in Fig. 5, the compilation process consists of four major phases: scanning and parsing, semantic analysis, target code generation, and code partitioning. To support different platforms, like Contiki and TinyOS, it is

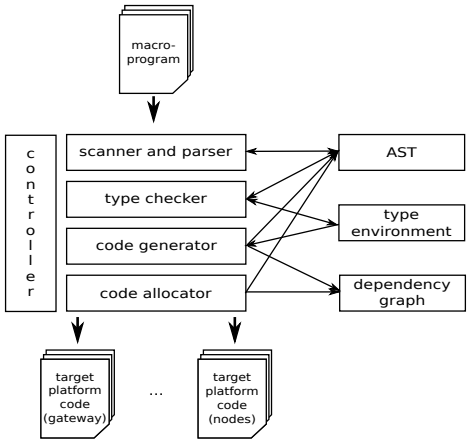

Figure 5. Architecture of the makeSense macro-compiler

possible to replace the generation back end, but the currently implementation only supports Contiki. In the non-standard final code partitioning phase, the compiler determines which translated classes need to be deployed at a specific node class based on a previously established dependency graph and a data flow analysis for the program. The goal of this phase is to remove unneeded code from specific program images. To reduce the size of the deployed program image, the single macro-program specifying the behavior of the whole network is partitioned into node-specific program parts. Each segment only contains those classes that are potentially executed on the nodes belonging to the respective class. For example, it is not necessary to provision program code for actuator control on pure sensor nodes. In the current implementation, we only differentiate between regular nodes and a dedicated gateway, but this concept can be easily extended to a larger number of node classes.

To enable the embedded code introduced in Sec. 4 the macro-compiler exhibits a plug-in interface that allows to integrate small sub-compilers for the abstraction-specific languages. Each of these plug-ins is responsible for parsing, type checking, and translation of the respective code fragments. As shown in Fig. 6, the plug-ins are automatically invoked by the main compiler, if it encounters an embedded code fragment in the macroprogramming code. A return channel allows the plug-ins to inform the compiler about references to macroprogramming language constructs encountered in the embedded code fragments. Like the macro-compiler, the plug-ins are implemented in Java.

6 Run-time System

Figure 7 shows the high-level architecture of the makeSense run-time system. The business process execution engine connects to the sensor network through a dedicated gateway we design. Application performance requirements are specified in

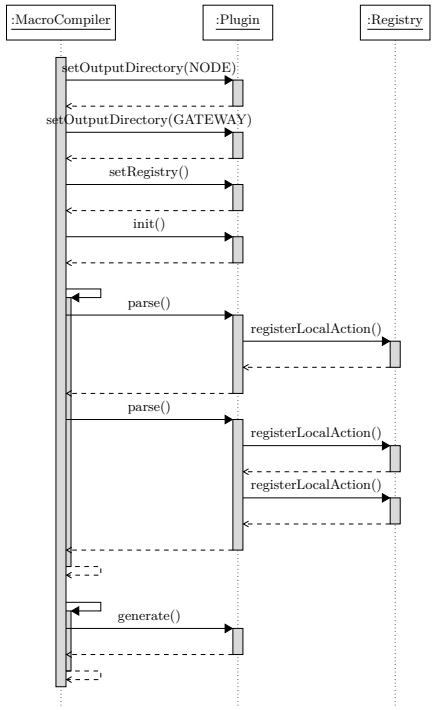

Figure 6. Typical communication sequence of `make`*Sense* macro-compiler plug-in.

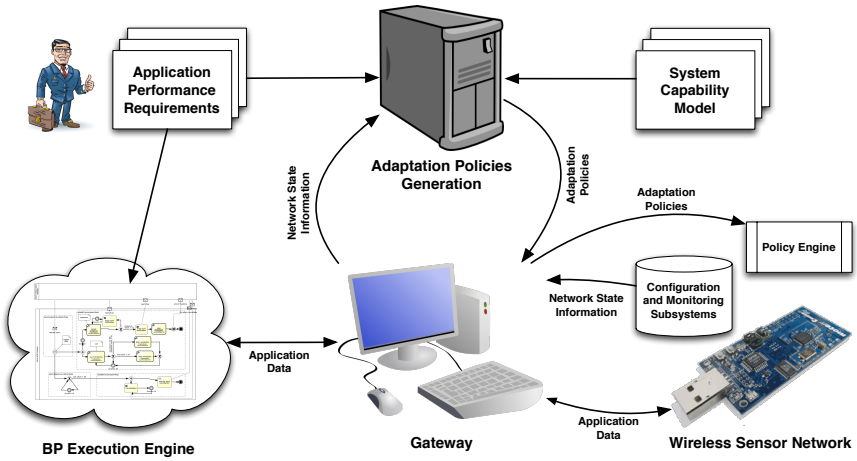

Figure 7. `make`*Sense* run-time architecture.

the extended business processes. These are taken as input by a dedicated opti-

mization engine that generates *self-optimization* policies that allows the network to dynamically tune its behavior. The latter task is carried out based on information from the system capability model and network state information from the deployed network. On the sensor nodes we deploy a dedicated configuration and monitoring subsystem that oversees the application execution inside the sensor network and executes the adaptation policies depending on the observed state.

While the makeSense gateway is implemented with mainstream technology as it is intended to run on a standard machine, the key functionality of the makeSense run-time system lies within the configuration and monitoring subsystem aboard the sensor nodes and in the generation of self-optimization policies. We describe these mechanisms next.

6.1 Monitoring and Configuration

The key design principle of the configuration and monitoring subsystem is to separate protocol logic from configuration [8]. This way, parameters in all parts of the system can be configured through a separate configuration component based on the settings that the self-optimization policies dictate. This makes it simple to handle changes in the objectives of the application, e.g., when the application demands a new objective such as high throughput instead of low energy consumption. Furthermore, we aim at keeping a layered design to make it possible to exchange layers, for example, when a new MAC layer should be used. While researchers have argued that cross-layering is required in wireless sensor networks to achieve high performance, we showed that we can both rely on a layered system and achieve high throughput [8].

In designing the configuration and monitoring functionality, we wish to lessen the burden on developers of configuration policies due to gathering and processing the data input to the self-optimization mechanism. To this end, we opt for a unified tuple space-like API spanning both read and write operations on the local blackboard, and distributed operations to share the configuration and monitoring information across 1-hop neighboring devices [9]. We also aim at a design that has clearer boundaries and hence requires little re-engineering work when new Contiki releases are available. Therefore, we use wrappers between Contiki components, e.g., the MAC protocol, and our configuration run-time.

As shown in Figure 8, the configuration and monitoring subsystem includes a central black-board for storage of configuration parameters, system state, and statistics. The other modules access the blackboard storage via tuple space-like APIs [9] that operate on the relevant data. These APIs can both operate on local data and on the blackboard of the one-hop neighbors. makeSense modules handle their configuration directly via the blackboard while non-makeSense modules, such as Contiki components, are wrapped so that relevant configuration and state can be stored in the blackboard. The monitoring modules are responsible for acquiring information on performance and resource consumption, storing it in the blackboard to make it available to upper layers.

The configuration policy and policy engine are responsible for setting the performance-related parameters. They also provide the interface to the optimizer

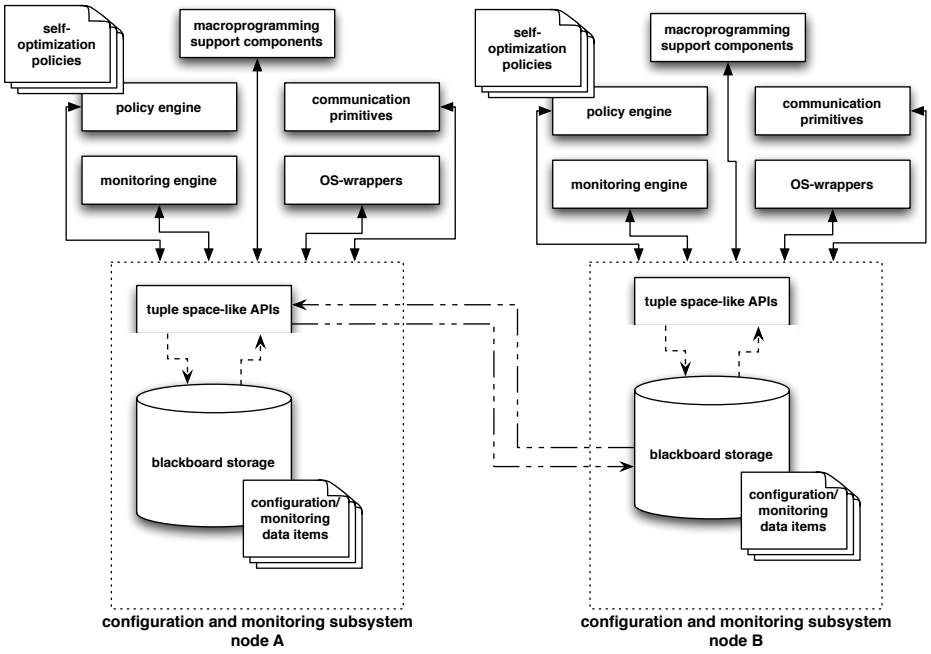

Figure 8. Overview of the configuration and monitoring subsystem

that runs outside the network and is in charge of optimizing the performance, as described next. The policy engine enforces these policies by setting appropriate parameters in the blackboard that determine the corresponding modules' behavior and performance. As any initial configuration is likely to be sub-optimal, the optimizer will dynamically update the configuration. Dynamic updates might also be required when the radio environment changes. For example, when it becomes more difficult to deliver packets due to interference, the optimizer might decide to increase the maximum number of retransmissions.

6.2 Self-optimization

In several real-world deployments the application and operating system code are finely-tuned to achieve a certain performance goal [10]. Most often, this is based on the developers' intimate knowledge of the internal sensor networks mechanisms and a deep understanding of the application requirements. The deployed code is also entirely in the hands of the same developers, who are free to modify and tune the implementations depending on the performance goals.

In general, the approach above is not possible in `make`*Sense*. Two main reasons concur to this: *i)* the executable code is generated from high-level application models, and the mapping from the latter to low-level Contiki C is not

trivial; and *ii)* the programming framework is open to external developers, who may contribute new concrete abstractions along with their supporting run-time. Furthermore, makeSense allows application developers to specify performance objectives that can change at the run-time. This is necessary to support long-lasting business processes subject to real world interactions and rapidly changing requirements. Therefore, the makeSense run-time must be able to *self-optimize* towards the stated performance objectives.

We define *self-optimization* as the property of a system to automatically find near-optimal system configurations whenever application objectives, system parameters, or environmental conditions change. To enable self-optimization, we gather run-time information from the deployed sensor network, e.g., network topology and protocol performance, and feed these to a reinforcement learning algorithm that explores the space of possible configurations using simulations. At the end of each simulation round, the learning process evaluates the performance obtained with a given setting w.r.t. the application's performance goals. Based on this, we derive self-optimization policies that specify which parameters provide better performance as a function of the current application and environment state, including the performance goal. We distribute the policies back to the deployed network where nodes will apply them whenever needed.

This approach sharply differentiates from existing solutions. Rather than requiring detailed modeling of the individual protocols, as done for example with great effort for MAC protocols [11], we treat the entire application as a black-box. This may lead to sub-optimal solutions, but also enjoys greater flexibility as it lets users add programming abstractions to the framework along with their supporting protocols and have the latter "implicitly" optimized.

We describe next the key aspects of the self-optimization functionality

Off-line learning. A typical makeSense application will have several modes of operation, along with different performance objectives. The same application can, for example, have energy efficiency as the major objective for most of the time, but in some emergency situations switch to latency. This means that there is no static system configuration that is optimal at all times. The configuration needs to be adapted as soon as the objective changes.

The approach taken for adapting configurations is to have a simulation framework that can simulate the set-up of a specific makeSense application. The simulation is fed with the applications network topology, the sensor network nodes firmwares being used, and network state information from the deployed system. The simulation is then run together with learning mechanisms that tune the configuration while simulating the application scenario. During the simulation, the learning mechanism will evaluate the performance given the application objectives. We choose to use a reinforcement learning based approach for the learning mechanism. As shown later, initial results demonstrate that this is a promising approach.

State monitoring. The nodes need to monitor their internal state to adapt their configuration. This state is an important part of the input to the configu-

ration policies and includes, as a function of the needed policy, information such as density, network congestion, and energy levels. The decision on what is needed is partly set by what information is relevant to the performance objectives. Monitoring the local state is needed to allow the performance goals to change over time because the nodes only adapt their configuration based on what they know in their local state. To change the performance objectives during run-time, the relevant parameters need to be updated in the nodes local state.

The selection of what should be included in the nodes' local states is important. Including too many parameters increases the time required to learn configuration policies and also increases energy consumption for parameters requiring active monitoring. Including too few parameters, on the other hand, makes it difficult to find reliable configuration policies because they might not have enough information.

Learning. In its simplest form, a policy is a mapping between a state and a set of actions that should be performed when the application is in this state. An action in this case can be a value to update in the blackboard that triggers a reconfiguration.

We are using a reinforcement learning based approach for the process of learning policies. The learning is performed during simulation using a plug-in for the Cooja simulator. A utility function based on the performance objectives provides the reinforcement learning with the needed rewards to implement the learning process. The specific learning mechanism that we use is First Visit Monte Carlo Policy Iteration [12]. We use the Cooja simulator as it allows to to accurately emulate sensor nodes such as TMote Sky and Wismote. This makes it possible to reuse the firmwares that are executed on the real sensor network in the simulator, making the simulation behavior as realistic as possible.

To automate the learning process in Cooja, we design and implement a new extension for the simulator that is able to run multiple simulation rounds of the same scenario. This extension uses the same simulation configuration files as Cooja and after the regular simulation it restarts the scenario at fixed time intervals. Before resetting the scenario the learning process takes place.

Initial results. We run experiments to assess the ability of the self-optimization framework to dynamically identify policies that improve the resulting system performance. We consider as example the following performance objective, formulated as a linear combination of desired reliability, goodput, and energy consumption:

$$utility = \frac{received}{sent} * 25.0 + received * 100.0 - 0.07 * energy \qquad (1)$$

Figure 9 reports a screenshot of the reinforcement learning simulation framework while optimizing for the performance objective above. In the figure, stream denotes the goodput in received packets per learning period. Throughout different simulation runs, the learning algorithms understands that a way to maximize the value of (1) is to favor packet transmissions (denoted as stream in the figure) even though they lead to slightly higher energy consumption. This is a

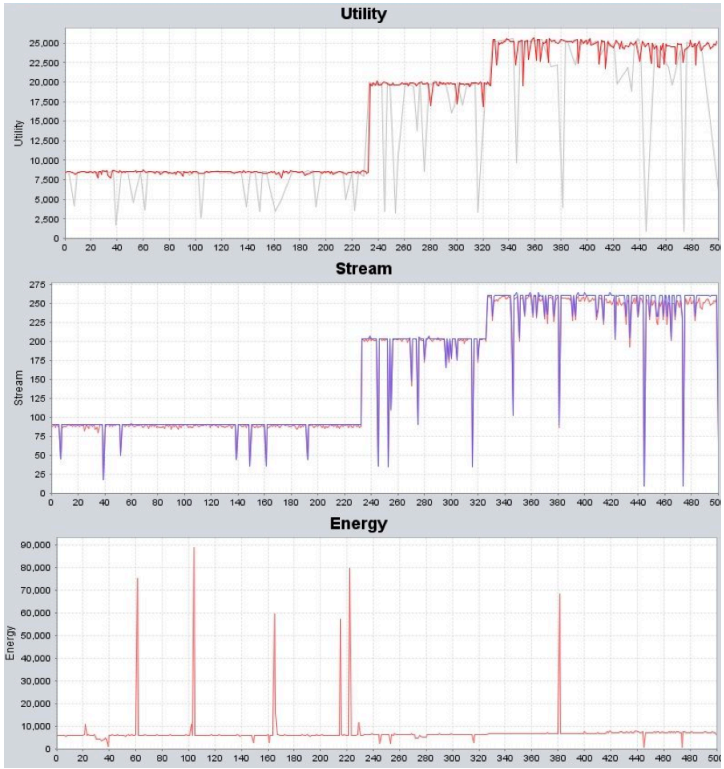

Figure 9. The utility improves significantly over time (top); the learning algorithm detects that it is better to send many messages, improving goodput (middle), even if the energy cost slightly increases (bottom).

direct result of the objective formulation, which poses the largest weight on the goodput.

7 Conclusions

In this paper, we have presented the makeSense approach for generating sensor networking code from business process models. Our approach integrates business processes with sensor networks in a novel way. Through a compilation chain an application models specified in slightly extended BPMN is transformed to both code that runs in the sensor network and code that is executed by traditional business process engines. We have also presented our final application scenario we are currently deploying in a student residence in Spain.

Acknowledgments. The work leading to these results has received funding from the European Union Seventh Framework Programme (FP7-ICT-2009-5) under grant agreement n° 258351 (make*Sense*).

References

1. A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless Sensor Networks for Habitat Monitoring. In *First ACM Workshop on Wireless Sensor Networks and Applications (WSNA 2002)*, Atlanta, GA, USA, September 2002.
2. F. Casati and F. Daniel and G. Dantchev and and J. Eriksson and N. Finne and S. Karnouskos and P. Moreno Montera and L. Mottola and F. Oppermann and G.P. Picco and A. Quartulliz and K. Römer and P. Spiess and S. Tranquilliniz, and T. Voigt. Towards business processes orchestrating the physical enterprise with wireless sensor networks. In *NIER track, 34th International Conference on Software Engineering (ICSE)*, pages 1357–1360, Zurich, Switzerland, June 2012.
3. L. Mottola and G.P. Picco. Programming Wireless Sensor Networks: Fundamental Concepts and State of the Art. *ACM Computing Surveys*, 43(3), 2011.
4. D. Guinard, V. Trifa, S. Karnouskos, P. Spiess, and D. Savio. Interacting with the SOA-based Internet of Things: Discovery, query, selection, and on-demand provisioning of Web services. *IEEE Trans. on Service Computing*, 3(3), 2010.
5. Kay Römer and Junyan Ma. PDA: Passive distributed assertions for sensor networks. In *Proc. of the Int. Conf. on Information Processing in Sensor Networks (IPSN)*, 2009.
6. L. Mottola and G. Picco. Logical Neighborhoods: A Programming Abstraction for Wireless Sensor Networks. In *Proc. of the Int. Conf. on Distributed Computing in Sensor Systems (DCOSS)*, 2006.
7. A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Proc. of the Workshop on Embedded Networked Sensor Systems (Emnets)*, 2004.
8. N. Finne, J. Eriksson, N. Tsiftes, A. Dunkels, and T. Voigt. Improving sensornet performance by separating system configuration from system logic. In *Proceedings of the Sixth European Conference on Wireless Sensor Networks (EWSN2010)*, Coimbra, Portugal, 2010.
9. P. Costa, L. Mottola, A. L. Murphy, and G. P. Picco. Programming Wireless Sensor Networks with the TeenyLime Middleware. In *Proc. of the* 8^{th} *ACM/USENIX Int. Middleware Conf.*, 2007.
10. M. Ceriotti, L. Mottola, G. P. Picco, A. Murphy, S. Guna, M. Corra, M. Pozzi, D. Zonta, and P. Zanon. Monitoring heritage buildings with wireless sensor networks: The Torre Aquila deployment. In *Proceedings of the International Conference on Information Processing in Sensor Networks (ACM/IEEE IPSN)*, pages 277–288, Washington, DC, USA, 2009. IEEE Computer Society.
11. M. Zimmerling, Federico Ferrari, Luca Mottola, Thiemo Voigt, and Lothar Thiele. pTunes: runtime parameter adaptation for low-power MAC protocols. In *ACM/IEEE Int. Conference on Information Processing in Sensor Networks (IPSN)*, 2012.
12. R.S. Sutton and A.G. Barto. *Reinforcement learning: An introduction*. The MIT press, 1998.