Wireless Sensor Networks for the Internet of Things: barriers and synergies

Mihai T. Lazarescu

Abstract Wireless sensor networks (WSN) are recognized key enablers for the Internet of Things (IoT) paradigm since its inception. WSNs are a resilient and effective distributed data collection technology, but issues related to reliability, autonomy, cost and accessibility to application domain experts still limit their wide scale use. Commercial solutions can effectively address vertical application domains, but they often lead to technology lock-ins that limit horizontal composability and reuse. We review some important barriers that hinder WSN use in IoT applications and highlight the main effort and cost components. With these elements in mind, we propose an open hardware-software development platform that can optimize the value flow between technologies and actors with stakes in WSN applications. To reach its objectives, the platform fosters reuse, low-effort low-risk fast prototyping accessible to application domain experts, easy integration of new technology and IP blocks, and simplifies the permeation of research results in commercial applications.

Key words: wsn, iot, wsn platform, wsn development, wsn components, wsn application synthesis

1 Introduction

Research and technology advances continuously extend and diversify wireless sensor network (WSN) applicability. As a consequence, WSN designers faced an increasing range of applications and requirements under rising cost and time pressures since the Internet of Things (IoT) paradigm was coined more than 15 years ago [1]. "Typical" requirements for WSN hardware and software are increasingly difficult to define [2] because they continuously adapt to very diverse application

Mihai T. Lazarescu
Politecnico di Torino, Dip. Elettronica e Telecomunicazioni (DET), Corso Duca degli Abruzzi 24, Torino, Italy, e-mail: mihai.lazarescu@polito.it

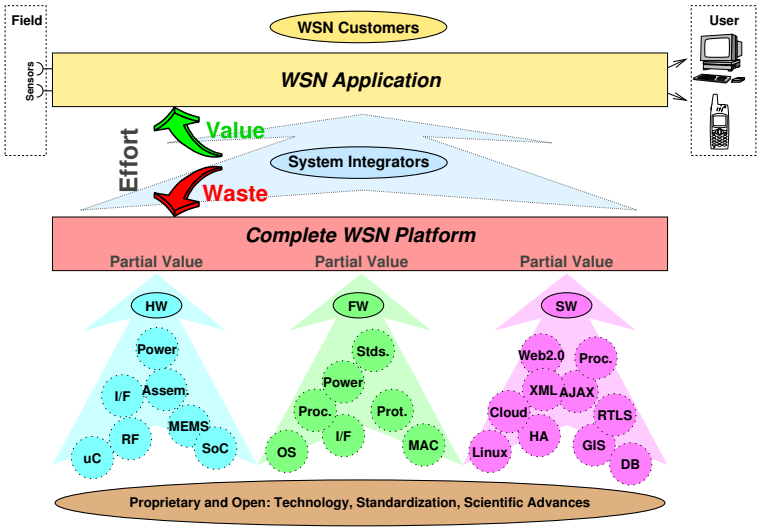

Fig. 1 Value flow for a WSN application and platform.

requirements and operating conditions at a rate which does not seem slowed down by standardization efforts or proprietary API proposals.

Moreover, although WSN solutions are used for numerous applications, the implementations generally differ under various aspects which significantly reduce the economies of scale. Consequently, both hardware and software of WSN solutions are often application-specific prototypes that carry significant non-recurrent engineering (NRE) costs and risks (e.g., reliability, optimization, development time).

Additionally, for various practical reasons WSN deployments are typically developed at lower abstraction levels, which can have two significant undesirable effects. First, this can divert an important development effort from application logic implementation, as shown in Fig. 1, which increases development time and cost, and generally decreases reliability. Second, lower abstraction level development often requires competencies that are seldom found among application domain experts, which can lead to higher development cost and more reluctant adoption of WSN-based solutions.

IoT vision to transform and enrich the way in which we perceive and interact with the reality often assumes capillary distributed devices for which WSNs continue to play an important role as one of the key enabling technologies since IoT paradigm inception. They often need to meet stringent requirements such as long maintenance-free lifetime, low cost, small size, ease of deployment and configuration, adequate processing of collected data, privacy and safety, and, not the least, fast and reliable development cycles that evolve on par with the underlying technologies. These are especially important for environmental monitoring applications, both closer to human day-to-day live (buildings, cities) and remote (e.g., open nature and climate monitoring).

As shown in Fig. 1, at the top of the WSN *value* chain are WSN application customers, whose needs are addressed by system integrators. The latter leverage on the top of WSN *technology* chain, where we find WSN platforms which can fully cover, end-to-end, WSN applications.

WSN platforms and development frameworks play a central role as the primary interface for accessing the underlying technology, standardization, and research state of the art. A complete platform includes field device hardware and firmware, development framework and an application server (e.g., for data storage, monitoring and post-processing). Development framework, IP components and supported hardware need continuous updates because WSN field evolves fast and these are an important factor in the final value delivered to end users. Also, development framework flexibility, reliability, and ease of use are important factors allowing system integrators and application domain experts to direct most of their effort to WSN application development, where it is valued most.

However important, designing and maintaining a complete, flexible, and reliable WSN development framework with extensive hardware support is costly and requires a broad range of expertises, from advanced web and UI design to low level embedded software and IDE, and code generation techniques to support high level application specifications. Quality assurance is also very important, since overall unreliability perception is still a limiting factor to wider WSN adoption.

Most hardware vendors provide various degrees of development support for their own WSN devices. They are usually focused on typical uses of the devices and are often hardware-centric instead of application-centric. As such, the toolsets may require significant extension or adaptation to properly cover a broader range of applications. Additionally, they also tend to significantly lag the state of the art, as they are meant to follow the progress of one producer.

Moreover, the up-front integration effort into existing development flows and the proprietary solutions may lock-in to vendor's hardware, often leading to significant hold-up problems that may hamper business potential. All these aspects finally translate into wasting system integrators' effort, missing or lagging market opportunities, and increasing development costs and risks. For new players it may also add to entry and differentiation barriers, effectively limiting the adoption of WSN solutions.

To best meet expectations and optimize the value, WSN development frameworks need to be based on reuse (both code and tools, since no player can efficiently cover all WSN aspects), to be easy to update or upgrade/replace to keep the pace with the fast evolution of the underlying technologies, and to abstract and automate the flow especially for application domain experts. Besides, the flow should favour design portability between different hardware and software solutions to avoid costly and inefficient vendor lock-ins and, last but not least considering the fast evolution pace in the field, to simplify the permeation of promising research to production.

Development frameworks revolve around the concept of a flexible WSN platform, as shown in Fig. 1, which is the convergence point of multiple WSN hardware and software components and technologies. Effective development tools should start from top-level application-specific requirement descriptions provided by the devel-

oper and automatically find suitable implementations and configurations that support them, based on existing components. At the same time it should provide the developer metrics and tools useful to evaluate solution quality.

This process aims to avoid diverting significant developer effort towards implementation details. This should speed up application development and make the flow more accessible to application domain experts.

At the same time, the flow should simplify the integration and coexistence of tools and technologies from different vendors and projects. Most existing WSN solutions efficiently address specific vertical application domains for various reasons, and not the least because building and maintaining a complete and flexible platform often requires a broad range of competencies and can be very costly. This development flow aims to reduce the effort and cost by:

Being accessible to application domain experts. This helps spreading the use of WSN-based solutions to more application domains, while often reducing development cost and time.

Focusing design effort mostly on application logic. WSN technology is based on several engineering disciplines. WSN development tools and flows should provide a good separation between the underlying technological details and the application developer.

Optimizing implementations for cost, power and reliability. This is especially important for the quality of service of the WSN application over its lifetime. Moreover, they implicitly reduce the recurrent cost for both node production and their field maintenance.

Integrating existing or new tools and technologies. Tool development can be often effort-intensive, hence reusing existing tools, either proprietary or public domain ones, is economical. Besides, the tools may be customized, e.g., for specific hardware or for specific application domains. Effort and cost issues are amplified by the fast evolution of WSN technology.

Maintainability of the complete development flow. This is tightly related to tool integration above. The tools should be easy to integrate in the development platform, e.g., in terms of semantics, interfaces and data formats, in order to simplify the upgrade or replacement of existing tools or the addition of new ones.

Simplifying the comparison of design results. The platform should simplify playing *what if* scenarios, in which elements change, e.g., a tool in the development flow, the target node or the embedded operating system (OS). Since the rest of the platform remains the same, it simplifies the observation of the effects of the change. This should allow a closer reproduction of research results and the comparison between different research tools or approaches. Also, this should allow to select the most effective solution for a given WSN application.

Facilitating research permeation in commercial applications. This is tightly related with the above point. The benefits of new research results can be compared with the existing flows by playing adequate *what if* scenarios. Moreover, research tools are already integrated in the platform simplifying their porting to existing production flows based on the platform.

Building business models. Last but not least, the purpose of the platform is to be useful for real WSN applications by providing value through vendor- or application-specific customizations of the general purpose flow. Thus, on the one hand the platform should allow the integration of proprietary tools or protected intellectual property (IP) blocks, e.g., simulation models or functional code. On the other hand, the platform should simplify the contribution of code (custom or general purpose), flows or other developments made by commercial users.

In Section 2 we will briefly review some existing WSN development tools and flows in terms of these objectives. In Section 3 we will review some options for WSN hardware for nodes and server-side software. Then, in Section 4 we will look into a possible development framework that can fulfill most of the above criteria. Section 6 will conclude the work.

2 WSN programming models and tools

Early WSN implementations were manually coded close to hardware or the embedded operating system [3]. Typically this brings smaller code size and higher execution efficiency at the cost of maintenance, portability and design reuse. It also requires a good understanding of various technologies underlying WSN nodes, which is difficult to find among software programmers and seldom found among application domain experts.

Over time, as WSN cost, time to market and operation reliability increase in importance, higher programming abstractions and design automation techniques get increasing attention. The literature is now rich of WSN design aids, both as languages and their compilers as well as support software, such as middleware and real-time embedded OSs.

In the following we are reviewing some relevant categories of design aids, with an eye on their performance along the lines listed towards the end of Section 1.

2.1 Low-level programming

This is a node-centric programming model that ranges from close to hardware and up to some level of abstraction usually provided by an embedded OS or by a virtual machine (VM).

2.1.1 Operating system-level programming

Among the OSs, we can list TinyOS [4], programmable in nesC [5] which offers a good modularization with well-defined interfaces and functional encapsulation that abstracts implementation details. However, low abstraction level and an event-based

programming style without support for blocking operations increase programming complexity.

To reduce the complexity, TinyGALS [6] provides FIFOs for asynchronous message passing on top of the synchronous event-driven model and synchronous method calls. SOS [7] and CoMOS [8] implement priority queues (the former) and preemptive message handling (the latter).

SNACK [9] allows to compose applications by combining services provided by reusable library components. T2 [10] simplifies project reuse and porting on new hardware platforms by combining a horizontal decomposition (to support multiple devices) and vertical decomposition that allows to build the application using hardware-independent functional blocks. OSM [11] extends the TinyOS event-driven model with states and transitions whose actions depend on both events and program state.

Several OSs propose different forms of thread abstraction to simplify programming of event-driven systems, although they are difficult to implement with the limited hardware resources of the target nodes. Fiber [12] provides one blocking context on top of TinyOS. Mantis OS [13] provides preemptive, time-sliced multithreading. TinyThreads [14] provides a stack estimation tool to reduce memory consumption for thread stacks. Protothreads [15] provides a stack-less co-operative multithreading library for Contiki OS [16], another embedded event-based OS. Y-Threads [17] implement efficient preemptive multithreading using a shared stack for the computational parts of the application, while for the control parts are allocated small separate stacks.

2.1.2 Virtual machine or middleware

These programming models have several important features. One of them is efficient dynamic re-programmability of small embedded systems. Maté [18] and ASVM [19] are small application-specific VMs built on top of TinyOS. Melete [20] extends Maté to multiple concurrent applications. VMStar [21] allows dynamic update of is own code or the application. Impala [22] is an example of middleware with efficient over-the-air reprogramming and repairing capabilities.

VMs also provide platform-independent execution models for WSN applications. Token Machine Language (TML) [23] models the network as a distributed token machine in which the nodes are exchanging tokens which trigger matching handlers. This model can be used to implement higher-level semantics that distance themselves from the underlying nesC implementation. t-kernel [24] modifies (“naturalize”) application code at load time to implement OS protection and virtual memory without hardware support.

2.2 High-level programming

High-level programming models typically allow the developers to focus more on application logic, but often at the expense of node performance. For instance, group- or network-level programming can facilitate the collaboration between sensors, which is necessary for a significant part of WSN applications and often challenging to program. However, the designer has few means to check or improve operation efficiency of the nodes within the abstracted units.

In the following we will review a few of the most prominent high-level programming methods and tools.

2.2.1 Model-based development

These methodologies attempt to offer the developers an application entry interface that can be both more productive and also better suited for application domain experts, while preserving lower-level control over design results, which is necessary to control and optimize WSN performance.

An interesting approach uses several Domain-Specific Modeling Languages (DSMLs), one for each of the network-, group- and node-level decomposition of a WSN application [25]. The DSML for data-centric network modeling includes three node types: sources, aggregation/fuse and sink. The DSML for group modeling allows geographical node grouping, definition of network topology (e.g., tree or mesh) and of the amount of in-network processing (aggregation and fusion). The DSML for node modeling offers several types of tasks: for sensor sampling, data aggregation/fusion, networking and sink. However, application specification appears to be limited to parametrization of the models. Hence, the approach can be considered a modular way to customize a parameterized application.

REMORA uses an XML-based modeling abstraction [26] for more advanced component-based designs than TinyOS static composition. Models include services (offered and required) and the triggering events, persistent state and implementation in a C-like language. The event model extends the TinyOS one with attributes, differentiation between application and OS events, configuration and point-to-point or multicast distribution. The framework has a low overhead over Contiki, but offers much improved encapsulation than simple multithreading.

A framework supporting hardware-software co-design is shown in [27]. It is based on tools widely used in industry like Simulink®[1] and Stateflow®[2] and well-known open projects like OMNeT++/MiXiM [28], TinyOS [4] and Contiki [16]. Graphical high-level application entry is supported by the abstract concurrent models provided by Simulink and Stateflow. Application specification can be simulated at node level and can be automatically translated by the framework in network sim-

[1] Mathworks Simulink `http://www.mathworks.com/help/simulink/index.html`

[2] Mathworks Stateflow – Finite State Machine Concepts `http://www.mathworks.com/help/toolbox/stateflow/index.html`

ulation models, including hardware in the loop for better accuracy, as well as implementation models that can run on top of popular embedded OSs.

2.2.2 Group-level programming

This programming model provides constructs to handle multiple nodes collectively as a group abstracting its internal operation into a set of external functions. Groups can be logical or neighborhood-based.

Neighborhood-based groups

Based on physically neighbouring nodes, this type of group is well suited for local collaboration (recurrent in some classes of applications) and the broadcasting nature of WSN communication, improving in-group communication. Abstract Regions [29] and Hood [30] provide programming primitives based on node neighbourhood which often fits the application needs to process local data close to its origin. Hood implements only the one-hop neighbour concept, while Abstract Regions implements both topological and geographical neighbourhood concepts.

Logical groups

These groups can be defined based on logical properties, including static properties like node types, or dynamic, such as sensor inputs, which can lead to volatile membership.

EnviroTrack [31] assigns addresses to environmental events and the sensors that received the event are dynamically grouped together using a sophisticated distributed group management protocol.

SPIDEY language [32] represents logical nodes based on the exported static or dynamic attributes of the physical nodes, and provides communication interfaces with the logical neighbourhood as well as efficient routing.

State-centric groups

These are mostly intended for applications that require collaborative signal and information processing (CSIP) [33]. Around the notion of collaboration group, programmers can specify its scope, define its members, its structure and the member roles within the group. Pattern-based groups, such as neighbourhood- or logical-based can also be defined. Its flexible implementation allows it to be used as building block for higher-level abstractions.

2.2.3 Network-level programming (macroprogramming)

WSN macroprogramming treats the whole network as a single abstract machine which can be programmed without concerns about its low-level inter-node communication.

Database

Database is an intuitive abstraction derived from the main function of the nodes, which is to collect sensing data. Early implementations like Cougar [34] and TInyDB [35] provide support for declarative SQL-like queries. Both attempt energy optimizations. Cougar processes selection operations on sensor nodes to reduce data transfers. TInyDB optimizes the routing tree for query dissemination and result collection by focusing on where, when and how often to sample and provide data.

SINA [36] allows to embed Sensor Querying and Tasking Language (SQTL) scripts with the SQL queries to perform more complex collaborative tasks than those allowed by SQL semantics.

MiLAN [37] and DSWare [38] provide a quality of service (QoS) extension to queries which can be defined based on the level of certainty of an attribute among those that can be measured by the node with a given accuracy. MiLAN converts a query in an energy-optimized execution plan that includes the source nodes and routing tree. DSWare expresses and evaluates the QoS as a compound event made of atomic events, whose presence/absence define the confidence level.

Although database-like interfaces are simple and easy to use, they are not well suited for applications that require continuous sensing or with significant fine-grained control flows.

Macroprogramming

WSN macroprogramming provides more flexibility than database models.

Specification of global behaviour

Regiment [39] implements a Haskell-like functional language. By preventing the developer to manipulate directly program states it allows the compiler to extract more parallelism.

Kairos [40] is a language-independent programming abstraction that can be implemented as an extension of existing program languages. As such it defines a reduced set of constructs (node abstractions, one-hop neighbours and remote data access, which includes a weak data consistency model to reduce communication overhead).

Resource naming

Spatial Programming [41] can reference network resources by physical location and other properties and access them using smart messages. These contain the code, data and execution state needed to complete the computation once they reach the nodes of interest.

Using SpatialView [42] the developer can dynamically define a spatialview as a group of nodes that have properties of interest, such as services and location.

Declarative resource naming [43] (DRN) allows both imperative and declarative resource grouping using Boolean expressions. Resource binding can be static or dynamic and the access to resources can be sequential or parallel.

Other metaprogramming abstractions

Semantic Streams [44] supports declarative queries using semantics associated to sensor data by inference units. Both sensors and inference units are built automatically from a Prolog-based specification.

Software Sensor [45] provides a service-oriented architecture in which each hardware sensor is abstracted as a software sensor. The latter can be composed in multiple ways using the SensorJini Java-based middleware in order to define large-scale collaboration within the network.

2.3 Evaluation of existing WSN programming models and tools

We will evaluate how the WSN development tools, tool categories and methodologies reviewed in Section 2 can be used to reduce overall WSN application cost. We will also evaluate their potential to work in synergy with other tools in a comprehensive development platform that is suitable to cover the widening diversity of WSN applications and keep pace with the rapid technological evolution in the field.

2.3.1 Low-level programming evaluation

Low-level programming, be it OS- or VM-based, typically allows a good control over the node and good design optimization, but it often requires in-depth engineering and programming knowledge. This is rarely found among application domain experts and may also divert important design effort from application logic implementation.

Embedded OSs and VMs have typically a layered structure which encapsulates well the hardware-dependent parts in order to facilitate their porting to other hardware nodes, which is important for keeping the pace with technology evolution. They can be maintained with reasonable effort and can be relatively easy integrated in higher-level design flows.

Moreover, embedded OSs usually facilitate the development of library elements that implement specific functions, which can be instantiated in new designs to facilitate design reuse.

The common base offered by the underlying OS or VM can be used as reference to compare the effectiveness of novel solutions. New research results can be included in library elements as well as the OS or VM core design, effectively simplifying their adoption in commercial designs.

Considering the higher complexity of application programming at low levels, commercial services can be offered for, e.g., application development or porting, OS/VM (or library elements) porting to new hardware, and for training and support.

2.3.2 High-level programming evaluation

Abstractions at group- or network-level are meant to hide the inner workings of the abstracted entity, including its internal communications.

On the one hand this is positive because it allows the developer to focus on application logic. On the other hand, the abstractions are often implemented using dedicated or less common (e.g., Haskell-like) programming languages which may be difficult to use by application domain experts.

Also, given that significant parts of the application are handled automatically by the system in order to satisfy higher-level specifications, the developer may not have the means to understand, evaluate or improve the efficiency with which these are implemented by the system.

The tools implementing such abstract flows are typically developed as a close ecosystem. They are unlikely to share models or IP blocks among them, although they may use common lower-level abstractions (e.g., Regiment [39] uses TinyOS) or may be flexible (and simple enough) to allow is implementation as an extension of other programming languages like Kairos. As such, their development and maintenance can be rather costly. Moreover, the application projects are difficult to port between such frameworks, which limits also the permeation of research to commercial applications.

For these flows, in terms of business we can assume training, support and design services.

A distinctive note can be made for model-based design frameworks (see Section 2.2.1). The developer can focus most of the effort on application development, for which the tools allow various degrees of liberty. Application entry interface can be suitable for application domain experts, as demonstrated by the wide adoption of Stateflow® interface by experts in various industrial domains. Automated code generation and good integration of the flow with simulation tools (including hardware-in-the-loop) and target OSs simplify design space exploration for optimizations and also allow manual optimization of the generated projects. Integration with existing projects reduces the cost of framework maintenance. Moreover, they provide an observable development environment where the effects of changes to framework features (e.g., code generation) or to its components (e.g., simulators) can be eas-

ily compared for existing and new designs. Research advances can be evaluated the same way before being included in commercial design flows. Business models that use these flows can enrich their capabilities using custom IP blocks.

3 WSN hardware and server-side support

In the following we will comparatively review some options for WSN node hardware and for server-side software.

3.1 WSN hardware

Its main components are the microcontroller and RF device, either stand-alone or combined in a single integrated circuit, RF antenna, energy sources, transducers, printed circuit board (PCB), and package.

3.1.1 Microcontroller

There are many families of low and very low power microcontrollers, each covering a broad range of capabilities such as storage size for programs (FLASH), data (RAM and EEPROM), and peripherals. Given the diverse and specialized offering, the design flow should assist the developer in selecting the optimal microcontroller for the application, since it can influence significantly node cost and energy consumption.

Microcontroller component package type and size has both a direct impact on the component cost and an indirect one on PCB cost through its size and number of pins. PCB size can further influence the size and cost of the sensor node package.

Additionally, most microcontrollers can operate using an internal clock source. This has no additional component cost or impact on PCB size or complexity, but its frequency is not very accurate nor stable in time. However, if the WSN application (or the communication protocol) does not require accurate timing, the internal oscillator is worth considering.

Thus, microcontroller selection should mainly consider:

- on-board support for hardware interface with the peripherals (e.g., transducer, RF device), application firmware and communication protocol;
- the trade-off between package type and its associated direct and indirect costs at sensor node level;
- adequate support for selected RF communication protocol. Its effects at system level should be carefully weighed. For instance, accurate timers and receive capabilities significantly increase sensor node cost, microcontroller requirements and energy consumption.

3.1.2 RF device

As for the microcontroller, the cost of the RF device significantly depends on its capabilities. E.g., a transceiver will typically cost more and require more external components than a transmit-only radio.

Modern RF devices provide application data interfaces at different levels of abstraction. These can range from digital (or analog) signal levels to advanced data packet management. Although the latter may increase RF device cost, it also sensibly reduce microcontroller computation and memory requirements, working with a cheaper and smaller device, and also reduce energy consumption and cost.

Most RF devices use an external crystal quartz for timing and channel frequency and may optionally output a clock for the microcontroller. However, this implies that the system is active only when the RF device is active, which may increase energy consumption.

Commonly used multihop WSN communication protocols require that all nodes have receive capabilities, which increases the cost of the RF device. Moreover, most of them actively synchronize their clocks at least with the neighbouring nodes to reduce the energy spent for idle channel listening, which requires an accurate time reference constantly running on-board and more microcontroller resources, all translating to higher device cost and energy consumption. However, if the application does not require bidirectional communication, an asynchronous medium access control (MAC) in a star topology may sensibly reduce node cost and energy consumption.

Several producers offer general purpose devices that include a microcontroller and a radio in a single package. These can save PCB space and costs, but the integrated microcontrollers may be oversized for some applications, especially those using a smaller communication protocol stacks. The same may apply for the integrated radio devices.

3.1.3 RF antenna

Antenna influences node cost, size, deployment and performance mainly through its size, cost, and RF characteristics. Most antenna features are correlated with operation frequency, range requirements, and RF properties of the environment.

For instance, star network topologies may require longer RF range to lower the number of repeaters and gateways. In this case, lower RF frequencies (315/433 MHz bands or below) can increase the communication range for a given RF link budget.

WSN application requirements may influence antenna characteristics such as directionality, gain, size, resilience to neighbouring conductive objects (tree trunks, metal bodies). Antenna options can range from omnidirectional $\lambda/2$ dipole, or a $\lambda/4$ whip (both costly to ruggedize), to helical antennas radiating in normal mode (NMHA) with a good size-performance trade-off, to PCB trace antennas (better for higher communication frequencies), and ceramic chip antennas with good performance but higher cost.

PCB components may also influence antenna performance.

3.1.4 Energy supply

Along with node energy consumption, energy supply is very important for the overall reliability and exploitation cost of the network.

A first decision concerns the use of energy harvesting or non-regenerative (e.g., a primary battery).

Environmental energy harvesting may suit applications with access to environmental energy sources (e.g., RF, vibration, fluid flow, temperature gradients, light). Combined energy harvesting solutions can increase the availability [46], although supply reliability is hard to estimate in the general case.

In all other cases, primary batteries should be considered such way to support average energy requirements of the node for the expected lifetime. This cost can be used as the upper bound for the evaluation of energy harvesting-based solutions.

Either way, low node energy consumption is very important for any type of energy source.

3.1.5 Transducers

Transducers are used to sense the properties of interest in the environment surrounding the node. Its performance affects the node in several ways. First, the transducer should have low energy requirements and/or allow (very) low duty cycle operation. Its interface with the node should not increase exceedingly microcontroller requirements. Last but not least, the transducer should not require periodic maintenance, which may significantly increase the operation cost of the network.

3.1.6 Package

Node package protects node components from direct exposure to environment and defines the node external size, mechanical properties and physical aspect. Its cost may increase due to special production requirements and its dimensions. Thus, special requests, such as transparent windows for light energy harvesting, should be carefully considered.

It may also provide the means to mount the node in the field, thus the package should be designed to simplify node deployment and maintenance to reduce the overall cost per node.

3.1.7 Hardware nodes

There are many hardware sensor nodes, both developed for research as well as commercial purposes.[3]

[3] Currently there are over 150 node models listed on Wikipedia `https://en.wikipedia.org/wiki/List_of_wireless_sensor_nodes`

Their characteristics are extremely diverse. E.g., in terms of average current consumption they range from a low end of 30 μA of Metronome Systems NeoMote[4] and 100 μA of Indrion Indriya_DP_01A11[5] [47] up to the high end 100 mA of Nokia Technology Platforms NWSP [48] (wearable) and Massachusetts Institute of Technology ubER-Badge [49].

Considering the wide diversity of features and support, node hardware selection can be daunting, especially for application domain experts. Development tools should provide enough flexibility to map the application on different types of nodes and provide the developer adequate guidance to select a suitable target node.

3.1.8 Server-side support

Servers offer several important functions in a WSN application, such as to receive, store, and provide access to field data. They bridge the low power communication segments, which have important latency-energy trade-offs, with the fast and ubiquitous access to field data needed by humans or applications. Other functions include in-field node configuration and query, as well as software updates.

Global Sensor Networks (GSN) [50] is a middleware that facilitates WSN deployment, programming and data processing. It supports integrated sensor data processing towards a vision of a global "sensor Internet". By abstracting the underlying technology, GSN simplifies, among others, platform additions, combination of sensor data, sensor mobility support and runtime dynamic system configuration adaptation.

Since collaborative aspects can become dominant for IoT applications, they are well supported by projects like Xively[6] (formerly known as Cosm and Pachube) and WikiSensing[7]. These simplify online collaboration over data sets ranging from energy and environmental data to transport services, to generate real-time graphs and widgets for web sites, for historical data analysis and generation of alarms.

4 Semi-automated WSN HW/SW application synthesis

We will now discuss a semiautomated hardware-software application synthesis flow to better understand its benefits in terms of the evaluation criteria presented at the end of Section 1.

The flow can automatically select modules from a previously developed library to perform design composition, both hardware and software, in order to significantly

[4] Metronome Systems NeoMote `http://metronomesystems.com/`

[5] Indrion Indriya_DP_01A11 `http://indrion.co.in/Indriya_DP_01A11.php`

[6] Xively project `http://xively.com/`

[7] WikiSensing project `http://wikisensing.org/`

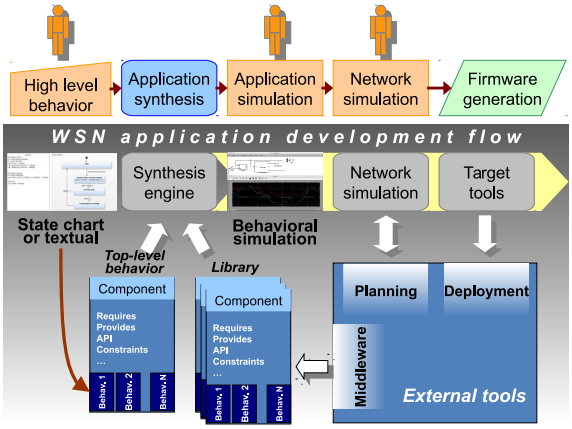

Fig. 2 Main stages of the automated WSN application design flow at node level: input of application behavioral description, automated synthesis of possible solutions (using top-level behavioral description and library components), behavioral simulation, network simulation, node programming code and configuration generation. Developer-assisted phases are tagged using a human body.

increase the productivity of the developers through design reuse, and to allow fast design space exploration for application implementation optimization.

4.1 Semi-automated development flow overview

Fig. 2 shows a typical WSN application development flow at node level. The flow receives a high-level node-centric application specification and is well integrated with external tools, each assisting the developer in specific tasks.

Application-specific behavioral input can range from manual source code input to automated code generation from model-based design (MBD) abstractions (such as Stateflow® used in [27] or similar state charts editors [51]) or from UML-based or ad-hoc high-level modeling flows [52, 53]. Either way, the behavioral input is captured in a specific view of the top-level component, which also includes the metadata needed by the subsequent phase of the flow, namely the automated synthesis.

The top-level component and all library components have the same format, which can include more than one view, e.g., behavioral source code or binary, simulation models for various abstraction levels or simulation environments. All views are handled as black boxes by the synthesis engine regardless their format, the synthesis relying only on component metadata. These can be generated manually or automatically, by MBD flows [27].

The second phase of the flow shown in Fig. 2 is fully automated. A hardware-software system synthesis engine takes the top-level component as input and processes its metadata (such as requires, provides, conflicts). These properties are the starting point for the synthesis process which iteratively attempts to find all subsets of its library of components that do not have any unsatisfied requirements left and, at the same time, satisfy all constraints of the top-level and other instantiated components. These subsets represent possible solutions for application requirements and can be further examined or modified by the developer.

For each solution, the synthesis tool can create simulation projects, as shown in the next steps of the flow in Fig. 2. The simulations are set up to run on external simulators (e.g., OMNeT++ [28]) and can be at various level of abstraction. Basically this is achieved by extracting and configuring the suitable simulation views of the components instantiated in solution into simulation projects.

Besides behavioral models, the components and constraints of the solution can include a bill of materials (e.g., compatible nodes, RF and transducer characteristics, microcontroller requirements) or software dependencies on specific compilation toolchains or underlying OS features.

Finally, the same mechanism is used to generate the projects that can be compiled using the target tools to create the programs for all WSN nodes. These projects are typically generated in the format expected by the target tools (most often a make-based project).

The solutions generated by the synthesis tool can be used as they are, or the developer can optimize them either by changing the specification and rerunning the synthesis, or by manually editing the generated projects. Either way, the developer can use simulations to validate and evaluate the solutions and their improvements.

As mentioned, the benefits of WSN application automated synthesis are compounded by its integration with external tools, such as simulators and target compilation chains, which can provide inputs or assist the developer in other phases of the flow. For instance, Fig. 2 shows some typical interfaces with middleware [54, 55], with WSN planning tools [56] or with deployment and maintenance tools [57].

However, the wide variety of the existing tools and models makes it very difficult to define an exhaustive set of external interfaces. Moreover, any rigidity of tool interfaces or operation models is prone to reduce the value of the tool and hamper its adoption in a context which is characterized by rapid evolution of technology and models, and which does not seem to be slowed down by standardization efforts or proprietary API proposals.

In this context, as we will show later on, an optimal approach for tool integration in the existing and future development flows can be to base its core operation on a model that is expressive enough to encode both high-level abstractions as well as low-level details. Moreover, it is also important to provide well-defined interfaces and semantics to simplify its maintenance, updates, integration with other tools, and extensions to other application domains.

4.2 Automated hardware-software synthesis tool overview

The tool covers the following main functions: application input (provide a suitable interface and processing), automated hardware-software synthesis, and code, configuration and hardware specifications generation.

Application domain experts can benefit most from an interactive user-friendly interface for the description of the WSN application top-level behavior. State charts are well established in this regard for their intuitive use, and they can provide suit-

able high-level models to facilitate the description of application-domain behavior. Alternatively, the synthesis tool can accept application descriptions generated by other tools, such as middleware [58] or metaprogramming [59].

Automated synthesis of hardware-software systems that can support WSN application requirements shields the developer from most time-consuming and error-prone implementation details. At the same time, the synthesis increases the reuse of library components such as: software components (e.g., OS, functional blocks, software configurations, project build setup), hardware components (such as WSN nodes, transducers, radio types or specific devices, hardware configurations), and specifications (e.g., target compilation toolchain, RF requirements).

Incomplete application specifications are also accepted, because the tool can typically infer default parameters based on values provided by library components and heuristics. This allows the developer to refine application specifications over several design iterations using also the results of previous underspecified synthesis runs. Also, the incomplete systems synthesized from incomplete application specifications still satisfy every requirement, and experienced developers can use these incomplete projects as starting points for manual refinements, to save effort.

Code generation can produce simulation or target compilation projects. Network simulations can be configured using the simulation models of the components instantiated in solutions, their parameters and the actual configurations. Realistic communication channels defined by a planning tool [56] can be used, if available. In a similar way, the tool uses the implementation code of the components instantiated in solution to generate and configure the project that compiles the code for WSN nodes programming.

Besides this highly automated process, the system synthesized tool allows experienced developers to manually take over the development flow at any stage: design entry, testing and debug, design synthesis, node application simulation, network simulation, target code generation. Basically, this is achieved by:

- making use of textual data formats that can be edited with general purpose or specialized editors;
- documenting the data formats, their semantics and processing during each phase of the flow;
- allowing one to run manually the individual tools, even outside the integrated flow, e.g., to explore options and operation modes that are not supported by the integrated flow;
- including well-known tools in the flow with clean and well-documented interfaces to simplify their update or replacement for flow specialization.

4.3 Automated synthesis tool input interface

During application specification phase, the developer (or an external tool) provides architectural requirements and top-level behavior as a design component, which becomes the main driver for the subsequent hardware-software system synthesis.

WSN application requirements can be expressed mainly in terms of application-specific behavior and its interfaces, and metadata properties.

As argued above, abstract concurrent state charts are an intuitive and efficient high-level means to simplify top-level application behavior. For design entry within this flow, the state chart tool should also allow to specify the interfaces and metadata for the behavioral part.

Yakindu State Chart Tools[8] is a free source integrated modeling environment based on Eclipse Modeling Framework (EMF) [60] for the specification and development of reactive, event-driven systems based on the concept of state charts. Its features provide significant support for design entry, especially useful for application-domain experts with limited programming experience, such as:

- state chart editing through an intuitive combination of graphical and textual notation. While states, transitions and state hierarchies are graphical elements, all declarations and actions are specified using a textual notation;
- state chart validation that includes syntax and semantic checks of the full model. Examples of built-in validation rules are the detection of unreachable states, dead ends and references to unknown events. These validation constraints are checked live during editing;
- state chart simulation models that allow the check of dynamic semantics. Active states are directly highlighted in the state chart editor and a dedicated simulation perspective features access to execution controls, inspection and setting of variables, as well as raising of events;
- code generation from state charts to Java, C and C++ languages. The code generators follow a code-only approach. The code is stand-alone and does not rely on any additional runtime libraries. The generated code provides well-defined interfaces and can be easily integrated with other target code.

Yakindu was designed for embedded applications with a meta model based on finite state machines (FSMs), either Mealy or Moore. System behavior is defined by the active state, which is determined by FSM inputs and history. Yakindu meta model is similar to UML state chart meta model except for the following differences which are of particular importance for the flow:

- state charts are self contained with interfaces defined by events and variables;
- core execution semantics are cycle-driven instead of event-driven, which allows to process concurrent events and to define event-driven behavior on top;
- time is an abstract concept for the state charts;
- time control is delegated to environment.

The model interpreter and the code generators adhere to core semantics.

Considering the above, Yakindu can be used and extended in order to provide all functions needed for design entry for the flow. Fig. 3 shows the main panes of Yakindu interface. On the right side is a tool pane with elements that can be used to edit the state chart in the central pane (such as transition, state, initial state,

[8] Yakindu SCT project `http://www.statecharts.org/`

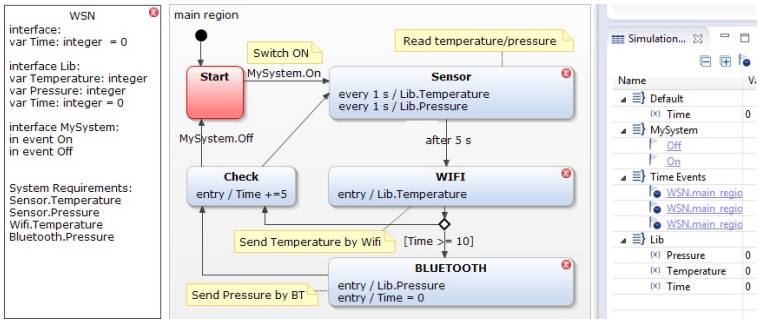

Fig. 3 State chart-based interface for WSN application specification entry and top-level simulation using a customized Yakindu SCT. The right pane can display simulation data or a tool pane for state chart editing. The interface of the state chart can be interactively defined on the left pane. The active state during simulations is highlighted on the state chart in the central pane.

choice, synchronization) or simulation data during chart simulation (as shown in the figure). In the left pane is shown the interface of the state chart. It allows to define variables and events that can be used by chart states, and it was extended to accept the metadata necessary for top-level component for the synthesis tool. The editable state chart in the middle pane describes top-level WSN application behavior, which can be interactively simulated or automatically converted into source code along with the interface and the metadata defined in the left pane, such way to be accepted by the synthesis engine.

To further assist application domain experts in using the interface, wherever is required textual input (such as to fill the properties of the state chart components or the interface), the developer is guided by a context-sensitive editor that lists the legal entries for keywords, names, operators, etc. Also, developer input is checked in real-time and errors are highlighted.

All these are captured in the top-level component of the design that is then used to drive the system synthesis engine. Using library components, the engine attempts to automatically compose a hardware and software system that supports the application-specific behavior and provides all its requirements.

For instance, let us consider a WSN application that collects and send every five minutes the environmental temperature during four intervals of two hours spread evenly during the day. The functional description of this application consists of a periodic check if the temperature collection is enabled, if it is enabled then checks if five minutes have passed from previous reading, and if so then it acquires a new reading and sends it to the communication channel. The whole application behavior can be encoded in just a few condition checks and data transfers, plus some configuration requirements to support them (such as timers, a temperature reading channel, a communication channel). The rest of node application and communications are not application-specific, hence the developer should not spend effort developing or interfacing with them. In this flow (see Fig. 2), these tasks are automatically handled

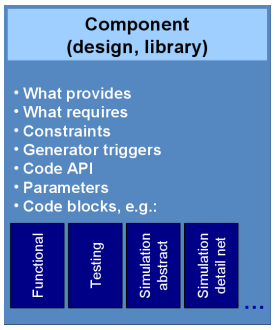

Fig. 4 Top-level application specification component and library components share the same structure: a variable set of views (shown darker on the bottom) that are handed as black boxes by the system synthesis process, and a set of metadata that express the requirements and the capabilities of the component. The components are encoded in XML (Eclipse EMF XMI).

by the synthesis engine, which attempts to build a system that satisfies all specifications by reusing library components, as will be explained later.

The top-level component can include also several types of metadata properties. For instance, if the 6LoWPAN protocol is a compatibility requirement of the WSN application, a requirement for 6LoWPAN can be added to top-level component, regardless if the top-level component functional code interfaces directly with the field communication protocol. This way, the 6LoWPAN requirement directs the application synthesis to instantiate the functional components from the library that provide this communication protocol. However, the synthesis tool will instantiate only those 6LoWPAN components that satisfy other system requirements that are collected from both the top-level and other instantiated components.

4.4 Structure of top-level and library components

Library components are central to the operation of the system synthesis engine (see Fig. 4). They are used for:

- definition by the developer of behavior and requirements of node-level WSN application, modeled as a top-level component;
- definition of library blocks that can be instantiated by the synthesis tool to compose a hardware-software system that satisfies all design specifications;
- interfaces with OS or middleware services, when needed by application behavior;
- provide simulation models, at different levels of abstraction;
- provide target code that is used to build the projects to configure and compile the code for target nodes;
- provide code generators that can be run by the synthesis tool to:
  - either check if the component can be configured to satisfy the requirements derived for the current solution during synthesis, so that it can be instantiated in the solution;

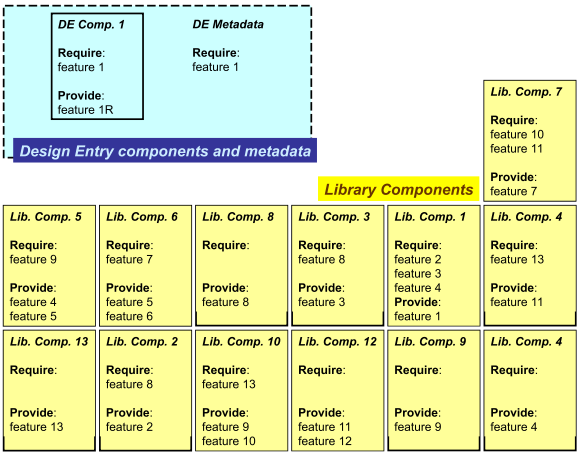

Fig. 5 Simplified example of metadata for design specification component and some library components.

  - or build specialized code stubs, e.g. for API translation and component code configuration that are based on the actual parameters of the solution in which they are instantiated;

- provide specifications for the hardware components that are collected in a bill of materials (BOM);
- provide nonfunctional requirements, such as for a specific compilation toolchain or special RF requirements.

Library components are designed to be compatible with the concurrency models provided by the OS or the middleware abstractions which they require at run-time. The same stands for the support of inter-component communication infrastructure that is provided by the OS or the middleware services. However, to achieve a consistent system composition all communications need to go through component interfaces in order to be visible to the synthesis engine, so that it can make the proper decisions. In fact, engine selections are also based on require and provide properties, in addition to other metadata.

4.5 System synthesis process

To exemplify the synthesis process, we show in Fig. 5 a simplified representation of just a few of metadata properties for both library components (bottom) and for top-level specification component (top-left).

At the begin of system synthesis process, the synthesis tool is driven by metadata specifications of the top-level component of the design, then is also guided by library components metadata. As system synthesis progresses and library components are instantiated in the partial solution, instantiated components metadata drive tool search alongside the still unsatisfied specifications of the top-level component.

During the synthesis process, the top-level component and its metadata are considered mandatory, while library components can be instantiated and removed from solution as necessary to satisfy design requirements.

For example, with the top-level specification and library components shown in Fig. 5, the system synthesis engine is able to find several solutions. It starts by loading the top-level component from design entry and the 13 components from the library. Then it explores the possible combinations and reports the following system compositions, each satisfying the specifications and requirements of all instantiated library components:

1. solution using components 1, 2, 3, 4, 8;
2. solution using components 1, 2, 3, 5, 8, 9;
3. solution using components 1, 2, 3, 5, 8, 10, 13.

4.6 Synthesis use for legacy designs

In the following we will briefly present the use of the synthesis tool for a representative legacy WSN application of practical interest, a self-powered WSN gateway designed for long-term event-based environmental monitoring. Source code is rather complex and optimized. It can handle in real time messages and state for up to 1000 sensor nodes with an average current consumption of about 1.6 mA. It can also bidirectionally communicate with the application server over the Internet using TCP/IP sockets through an on-board GPRS modem, and receive remote updates. Regardless, gateway hardware requirements are very low, comparable to those of a typical WSN *sensor* node.

To achieve this performance, it was originally programmed fully by hand written code in C language, without an embedded OS or external libraries (except from some low level standard C libraries).

To convert this legacy project for use with the system synthesis tool, we basically follow these steps:

1. split the project into functional blocks, each suitable for packing as a library component for the synthesis tool;
2. create a synthesis project by defining its specification top-level component;
3. run the synthesis for the project specification component to perform automatic system synthesis;
4. evaluate the solutions found by system synthesis.

It is worth noting that, once created, the library components are reused automatically by the tool whenever they satisfy the requirements of a synthesis project.

The quality of the synthesis process largely depends on the quality of the library components at its disposal. Hence, particular attention should be given to component creation from existing hardware or software IP blocks. One obvious (and easy to automate) operation is to pack the IP code in an appropriate component model (see Fig. 4). But it is important to properly describe its functional elements, such as

interfaces and configuration capabilities, and even more important are the semantics associated to component behavior and data exchanges.

Gateway application software is made of 49 modules, each implementing well-defined functions. These can be generic functions, like task scheduler, oscillator calibration or message queue (which are used by most applications), or specialized functions, such as drivers for specific on-board sensors (which are used only by specific applications).

Besides functional blocks for main gateway behavior, the code includes several modules for safety and error recovery, and drivers and processing modules for sensors and auxiliary devices that can be optionally mounted on node, such as:

`adc` Drivers for the ADC peripherals.
The module captures the ADC interrupt and calls the conversion data processing function.
`anemometer` Weather anemometer sensor handling functions.
Driver and controller for the anemometer transducer.
`battery` Utilities for battery reading processing.
The module provides the battery-specific voltage-to-capacity conversion tables and the functions to perform the conversion.
`cc` Field and mesh radio drivers.
The module handles everything related to the field and mesh radio on board the gateway.
`crc` CRC utilities.
Processing utilities (CRC calculation).
`gw` Node status.
Controls the state and configuration of the node.
`hal` Hardware high level interface.
It processes asynchronous events from the network and on-board switches.
`humidity` Weather humidity sensor handling functions.
Driver and controller for the humidity transducer.
`modem` GPRS modem driver.
Driver for the GPRS modem.
`queue` Message queue.
Storage and processing of the messages queued to be delivered to the server.
`sched` Task scheduler.
Scheduler.
`sensor` Sensor state and data processing.
Maintains the state of the sensors in range based on the contents of their messages (or lack thereof).
`timer` Timer handler.
Provides several timers for use within the node.
`usart` USART drivers.
Drivers for the node USART ports.
`version` Firmware version utilities.
It provides the version of node software.

Fig. 6 shows the metadata of the library component for a very simple gateway module, *version*. The module stores the version of gateway code and provides methods for its access.

At the top level we can see the main categories *properties*, *views*, *resources* and *interfaces*. For this simple component, we have just one property that holds the name of the module. Behavioral views include two files with the source code of the module. Resources include one non-functional requirement that tracks component dependency on a toolchain that supports the C language extensions it uses, and

```
<sgraph:Gss xmi:id="_b2b65395f30689ed09f02e">
    <properties>
        <name>version_component</name><description />
    </properties>
    <views xmi:id="_08f5c2612c510ac5e105e7">
        <behavior>
            <view xmi:id="_5c39ae70c147735f28ad4b" name="version.c"
                type="source" language="C" encoding="base64">
                <description></description>
                <mem>LyoqCiAqIEBmaWxlIHZ [...]</mem>
            </view>
            <view xmi:id="_44e6770ca6e62fc2db54e9" name="version.h"
                type="source" language="C" encoding="base64">
                <description></description>
                <mem>LyoqCiAqIEBmaWxlIHZlc [...]</mem>
            </view>
        </behavior>
    </views>
    <resources>
        <behavior>
            <require><name>avr_libc</name><description /></require>
            <provide><name>version_component</name>
                <description /></provide>
        </behavior>
    </resources>
    <interfaces>
        <behavior>
            <provide>
                <description />
                <function>
                    <name>version_get</name>
                    <return><type>char *</type></return>
                    <port><ord>1</ord><type>char *</type></port>
                </function>
            </provide>
        </behavior>
    </interfaces>
</sgraph:Gss>
```

Fig. 6 Example of a simple library component that includes properties and a code view.

a symbolic provided resource which can be used, for instance, to specifically require this component in design specifications or through the requirements of other components. In terms of interfaces, the component provides a behavioral function which retrieves and returns the code version. Additionally, for most metadata properties one can enter a description that can help the developer understand component semantics when it is displayed in a component or solution editor.

Fig. 7 shows the synthesis result of a minimal gateway system, which requires only the core functions. Moreover, the synthesis tool executed the component configuration helpers to set up their instances in solution with the actual parameters found by the solver (e.g., the scheduler is configured to run only the actual tasks).

Besides the software solution, the synthesis tool collects other requirements of the instantiated components in a list that includes, e.g., hardware node type, radio specifications and target compilation toolchain.

To find a suitable system composition, the solver reached a recursion depth of 888, matched or wired 230 abstract, 472 functional, and two data requirements in less than 0.8 s on an 1.8 GHz Intel® Core™ i7-2677M CPU.

Fig. 7 Result of system synthesis that requires only the main gateway component. It includes 36 out of 49 modules (emphasized), correctly leaving out, e.g., drivers for optional sensors, test suites, interfaces.

adc	hygrometer	**rccal**	**test_tx**
anemometer	**igwc**	rel_mesh	**theft**
battery	**inst**	**rpc**	**timer**
cc	**main**	**run_state**	**twi**
crc	**mesh**	**sched**	**usart**
eeprom	**modem**	**sensor**	**util**
eeprom_ext	**msg_filter**	sensor_ppc	**version**
fc10	**obs**	**service**	**wd**
field	**oc_link**	sio	weather
geophone	**power**	**spi**	zlist
gw	pressure	**sr**	
hal	**queue**	**sw**	
humidity	rain	testing	

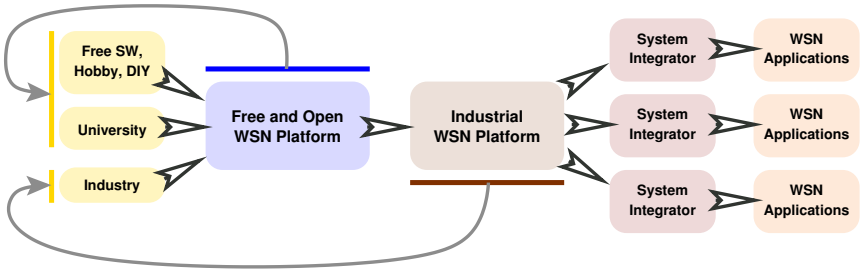

Fig. 8 Ecosystem based on the open WSN platform (for academic research, hobbyists) and its bidirectional synergy with industrialized platform(s) for commercial WSN application development.

5 Synergies for WSN development tools and platforms

WSN platform development and update is a complex, interdisciplinary, and evolving task. As such, it benefits from allowing all interested parties bring their contribution to its development and extensive use.

WSNs are at the center of a constantly increasing research interest, focused on various functional and technological issues. Research advances the state of the art and most promising results can be made available to many WSN industrial actors through WSN platforms. Reciprocally, the research community would benefit from a common, open, and free WSN platform available for experimentation.

The same platform can be used also by the wider public, as shown by the growing interest recorded for DIY applications, e.g., in home automation and city environmental monitoring. In exchange, the platform would gain from extensive testing, consolidation, porting to popular hardware, improved development tools, and extensions for innovative applications.

Fig. 8 suggest a possible ecosystem that includes non-profit and industrial interested parties, which can help both WSN research as well as spreading the use of WSN technologies to solve real life problems.

The ecosystem revolves around a free and open WSN platform. The platform can include open development tools, like those discussed in Sections 2 and 4, server

software and open node hardware, either as nodes or node components, as discussed in Section 3.

Being open, the platform facilitates contributions from several sources, such as academic research, free software community interested in WSN/IoT projects, as well as industrial partners. The latter may become interested because of the business opportunities that can be opened by a platform that helps reducing the effort, cost and risk of WSN-based solutions.

For this purpose, Fig. 8 suggests a productized version of the open platform, which adds the necessary level of reliability, testing and support forthe development of commercial applications. While the open WSN platform reaches its purpose in a continuous state of flux, the industrial product requires stability, ease of use, proprietary IP blocks support, qualified training and support services.

However, as we argued, developing and maintaining a comprehensive platform is effort-intensive and requires a broad range of engineering skills that are hard to bring together by single entities. But these can be naturally attracted by the open platform, which can serve as foundation to reduce development and maintenance effort for the commercial version(s).

Additionally, the two versions of the platform facilitate the exchanges between academia and industry. Interesting research results can be ported easier to derived commercial platform(s). Conversely is facilitated the contribution of closed-source IPs or improvements to platform infrastructure, library or tools from commercial platforms.

Last but not least, a reference platform simplifies the reproduction of research results as well as collaborative developments.

6 Conclusion

Although WSNs are object of extensive scientific and technological advances and can effectively achieve distributed data collection for IoT applications, their wide scale use is still limited by a perceived low reliability, limited autonomy, high cost and reduced accessibility to application domain experts.

Commercial solutions are often effective in addressing vertical application domains, but they may lead to technology lock-ins that limit horizontal composability, and component or design reuse.

We consider WSN platform an essential enabler for effective application design: fast development with low effort and cost, reliable designs based on extensive reuse, flow accessible to application domain experts, and offering maintainable extensive technology coverage.

After examining how existing WSN development tools and flows satisfy these objectives, we propose a node-level hardware-software development flow. It revolves on automated system synthesis starting from a high-level application specification (both behavior and non-functional requirements) and reuses extensively library components.

We argue that this system can foster synergies between academic research, IoT and WSN developer communities, and system integrators developing commercial WSN application, with the distinct possibility of mutual benefit in an ecosystem that merges open and closed source IPs. We consider synergy important, since effective and reliable WSN development requires a wide range of engineering expertise that are difficult to be covered viably by individual players.

References

1. Kevin Ashton. That 'Internet of Things' Thing. Expert view, RFID Journal, June 2009. http://www.rfidjournal.com/article/view/4986.
2. K. Romer and F. Mattern. The design space of wireless sensor networks. *IEEE Wireless Comm.*, 11(6):54–61, December 2004.
3. Robert Szewczyk, Alan Mainwaring, Joseph Polastre, John Anderson, and David Culler. An Analysis of a Large Scale Habitat Monitoring Application. In *Proceedings of the 2Nd International Conference on Embedded Networked Sensor Systems*, SenSys '04, pages 214–226, New York, NY, USA, 2004. ACM.
4. Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System Architecture Directions for Networked Sensors. *SIGARCH Comput. Archit. News*, 28(5):93–104, November 2000.
5. David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. *SIGPLAN Not.*, 38(5):1–11, May 2003.
6. Elaine Cheong, Judy Liebman, Jie Liu, and Feng Zhao. TinyGALS: A Programming Model for Event-driven Embedded Systems. In *Proceedings of the 2003 ACM Symposium on Applied Computing*, SAC '03, pages 698–704, New York, NY, USA, 2003. ACM.
7. Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. A Dynamic Operating System for Sensor Nodes. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services*, MobiSys '05, pages 163–176, New York, NY, USA, 2005. ACM.
8. Chih-Chieh Han, Michel Goraczko, Johannes Helander, Jie Liu, Bodhi Priyantha, and Feng Zhao. CoMOS: An operating system for heterogeneous multi-processor sensor devices. *Redmond, WA, Microsoft Research Technical Report No MSR-TR-2006-177*, 2006.
9. Ben Greenstein, Eddie Kohler, and Deborah Estrin. A Sensor Network Application Construction Kit (SNACK). In *Proceedings of the 2Nd International Conference on Embedded Networked Sensor Systems*, SenSys '04, pages 69–80, New York, NY, USA, 2004. ACM.
10. Philip Levis, David Gay, Vlado Handziski, Jan-Hinrich Hauer, Ben Greenstein, Martin Turon, Jonathan Hui, Kevin Klues, Cory Sharp, Robert Szewczyk, et al. T2: A second generation OS for embedded sensor networks. *Telecommunication Networks Group, Technische Universität Berlin, Tech. Rep. TKN-05-007*, 2005.
11. Oliver Kasten and Kay Römer. Beyond Event Handlers: Programming Wireless Sensors with Attributed State Machines. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*, IPSN '05, Piscataway, NJ, USA, 2005. IEEE Press.
12. Matt Welsh and Geoffrey Mainland. Programming Sensor Networks Using Abstract Regions. In *NSDI*, volume 4, pages 3–3, 2004.
13. H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han. MANTIS: System Support for multimodAl NeTworks of In-situ Sensors. In *Proceedings of the 2Nd ACM International Conference on Wireless Sensor Networks and Applications*, WSNA '03, pages 50–59, New York, NY, USA, 2003. ACM.
14. William P. McCartney and Nigamanth Sridhar. Abstractions for Safe Concurrent Programming in Networked Embedded Systems. In *Proceedings of the 4th International Conference*

on Embedded Networked Sensor Systems, SenSys '06, pages 167–180, New York, NY, USA, 2006. ACM.
15. Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: Simplifying Event-driven Programming of Memory-constrained Embedded Systems. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, SenSys '06, pages 29–42, New York, NY, USA, 2006. ACM.
16. Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki – A Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, LCN '04, pages 455–462, Washington, DC, USA, 2004. IEEE Computer Society.
17. Christopher Nitta, Raju Pandey, and Yann Ramin. *Distributed Computing in Sensor Systems: Second IEEE International Conference, DCOSS 2006, San Francisco, CA, USA, June 18-20, 2006. Proceedings*, chapter Y-Threads: Supporting Concurrency in Wireless Sensor Networks, pages 169–184. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.
18. Philip Levis and David Culler. Maté: A Tiny Virtual Machine for Sensor Networks. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 85–95, New York, NY, USA, 2002. ACM.
19. Philip Levis, David Gay, and David Culler. Active Sensor Networks. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 343–356, Berkeley, CA, USA, 2005. USENIX Association.
20. Yang Yu, Loren J. Rittle, Vartika Bhandari, and Jason B. LeBrun. Supporting Concurrent Applications in Wireless Sensor Networks. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, SenSys '06, pages 139–152, New York, NY, USA, 2006. ACM.
21. Joel Koshy and Raju Pandey. VMSTAR: Synthesizing Scalable Runtime Environments for Sensor Networks. In *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*, SenSys '05, pages 243–254, New York, NY, USA, 2005. ACM.
22. Ting Liu and Margaret Martonosi. Impala: A Middleware System for Managing Autonomic, Parallel Sensor Systems. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, pages 107–118, New York, NY, USA, 2003. ACM.
23. Ryan Newton, Arvind, and Matt Welsh. Building Up to Macroprogramming: An Intermediate Language for Sensor Networks. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*, IPSN '05, Piscataway, NJ, USA, 2005. IEEE Press.
24. Lin Gu and John A. Stankovic. t-kernel: Providing Reliable OS Support to Wireless Sensor Networks. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, SenSys '06, pages 1–14, New York, NY, USA, 2006. ACM.
25. Ryo Shimizu, Kenji Tei, Yoshiaki Fukazawa, and Shinichi Honiden. Model Driven Development for Rapid Prototyping and Optimization of Wireless Sensor Network Applications. In *Proceedings of the 2Nd Workshop on Software Engineering for Sensor Network Applications*, SESENA '11, pages 31–36, New York, NY, USA, 2011. ACM.
26. Amirhosein Taherkordi, Frédéric Loiret, Azadeh Abdolrazaghi, Romain Rouvoy, Quan Le-Trung, and Frank Eliassen. Programming Sensor Networks Using REMORA Component Model. In *Proceedings of the 6th IEEE International Conference on Distributed Computing in Sensor Systems*, DCOSS'10, pages 45–62, Berlin, Heidelberg, 2010. Springer-Verlag.
27. Zhenyu Song, Mihai T. Lazarescu, Riccardo Tomasi, Luciano Lavagno, and Maurizio A. Spirito. *Internet of Things: Challenges and Opportunities*, chapter High-Level Internet of Things Applications Development Using Wireless Sensor Networks, pages 75–109. Springer International Publishing, Cham, 2014.
28. András Varga and Rudolf Hornig. An Overview of the OMNeT++ Simulation Environment. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, Simutools '08, pages 60:1–60:10, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

29. Matt Welsh and Geoffrey Mainland. Programming Sensor Networks Using Abstract Regions. In *NSDI*, volume 4, pages 3–3, 2004.
30. Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler. Hood: A Neighborhood Abstraction for Sensor Networks. In *Proceedings of the 2Nd International Conference on Mobile Systems, Applications, and Services*, MobiSys '04, pages 99–110, New York, NY, USA, 2004. ACM.
31. T. Abdelzaher, B. Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, L. Gu, T. He, S. Krishnamurthy, L. Luo, S. Son, J. Stankovic, R. Stoleru, and A. Wood. EnviroTrack: towards an environmental computing paradigm for distributed sensor networks. In *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on*, pages 582–589, 2004.
32. Luca Mottola and Gian Pietro Picco. *Distributed Computing in Sensor Systems: Second IEEE International Conference, DCOSS 2006, San Francisco, CA, USA, June 18-20, 2006. Proceedings*, chapter Logical Neighborhoods: A Programming Abstraction for Wireless Sensor Networks, pages 150–168. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.
33. J. Liu, M. Chu, J. Liu, J. Reich, and F. Zhao. State-centric programming for sensor-actuator network systems. *Pervasive Computing, IEEE*, 2(4):50–62, October 2003.
34. P. Bonnet, J. Gehrke, and P. Seshadri. Querying the physical world. *Personal Communications, IEEE*, 7(5):10–15, Oct 2000.
35. Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *ACM Trans. Database Syst.*, 30(1):122–173, March 2005.
36. C. Srisathapornphat, C. Jaikaeo, and Chien-Chung Shen. Sensor Information Networking Architecture. In *Parallel Processing, 2000. Proceedings. 2000 International Workshops on*, pages 23–30, 2000.
37. W.B. Heinzelman, A.L. Murphy, H.S. Carvalho, and M.A. Perillo. Middleware to support sensor network applications. *Network, IEEE*, 18(1):6–14, Jan 2004.
38. Shuoqi Li, Sang H. Son, and John A. Stankovic. *Information Processing in Sensor Networks: Second International Workshop, IPSN 2003, Palo Alto, CA, USA, April 22–23, 2003 Proceedings*, chapter Event Detection Services Using Data Service Middleware in Distributed Sensor Networks, pages 502–517. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.
39. Ryan Newton, Greg Morrisett, and Matt Welsh. The Regiment Macroprogramming System. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, IPSN '07, pages 489–498, New York, NY, USA, 2007. ACM.
40. Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan. Macro-programming Wireless Sensor Networks Using Kairos. In *Proceedings of the First IEEE International Conference on Distributed Computing in Sensor Systems*, DCOSS'05, pages 126–140, Berlin, Heidelberg, 2005. Springer-Verlag.
41. C. Borcea, C. Intanagonwiwat, P. Kang, U. Kremer, and L. Iftode. Spatial programming using smart messages: design and implementation. In *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on*, pages 690–699, 2004.
42. Yang Ni, Ulrich Kremer, Adrian Stere, and Liviu Iftode. Programming ad-hoc networks of mobile and resource-constrained devices. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 249–260, New York, NY, USA, 2005. ACM.
43. Chalermek Intanagonwiwat, Rajesh Gupta, and Amin Vahdat. *Algorithmic Aspects of Wireless Sensor Networks: Second International Workshop, ALGOSENSORS 2006, Venice, Italy, July 15, 2006, Revised Selected Papers*, chapter Declarative Resource Naming for Macroprogramming Wireless Networks of Embedded Systems, pages 192–199. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.
44. Kamin Whitehouse, Feng Zhao, and Jie Liu. *Wireless Sensor Networks: Third European Workshop, EWSN 2006, Zurich, Switzerland, February 13-15, 2006. Proceedings*, chapter Semantic Streams: A Framework for Composable Semantic Interpretation of Sensor Data, pages 5–20. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

45. Eric Wei-Chee Lin. *Software Sensors: Design and Implementation of a Programming Model and Middleware for Sensor Networks*. University of California, San Diego, 2004.
46. S. Bandyopadhyay and A.P. Chandrakasan. Platform architecture for solar, thermal and vibration energy combining with MPPT and single inductor. In *VLSI Circuits, VLSIC*, pages 238–239, June 2011.
47. Manjunath Doddavenkatappa, Mun Choon Chan, and A. L. Ananda. *Testbeds and Research Infrastructure. Development of Networks and Communities*, chapter Indriya: A Low-Cost, 3D Wireless Sensor Network Testbed, pages 302–316. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
48. T. Ahola, P. Korpinen, J. Rakkola, T. Ramo, J. Salminen, and J. Savolainen. Wearable FPGA Based Wireless Sensor Platform. In *Engineering in Medicine and Biology Society, 2007. EMBS 2007. 29th Annual International Conference of the IEEE*, pages 2288–2291, August 2007.
49. Mathew Laibowitz, Jonathan Gips, Ryan Aylward, Alex Pentland, and Joseph A. Paradiso. A Sensor Network for Social Dynamics. In *Proceedings of the 5th International Conference on Information Processing in Sensor Networks*, IPSN '06, pages 483–491, New York, NY, USA, 2006. ACM.
50. K. Aberer, M. Hauswirth, and A. Salehi. Infrastructure for Data Processing in Large-Scale Interconnected Sensor Networks. In *Mobile Data Management*, pages 198–205, May 2007.
51. A Mülder and A Nyßen. TMF meets GMF. *Eclipse Magazin*, 3:74–78, 2011. `https://svn.codespot.com/a/eclipselabs.org/yakindu/media/slides/TMF\_meets\_GMF\_FINAL.pdf`.
52. Krishna Doddapaneni, Enver Ever, Orhan Gemikonakli, Ivano Malavolta, Leonardo Mostarda, and Henry Muccini. A Model-driven Engineering Framework for Architecting and Analysing Wireless Sensor Networks. In *Proceedings of the Third International Workshop on Software Engineering for Sensor Network Applications*, SESENA '12, pages 1–7, Piscataway, NJ, USA, 2012. IEEE Press.
53. A.R. Paulon, A.A. Fröhlich, L.B. Becker, and F.P. Basso. Model-Driven Development of WSN Applications. In *Computing Systems Engineering (SBESC), 2013 III Brazilian Symposium on*, pages 161–166, December 2013.
54. Nader Mohamed and Jameela Al-Jaroodi. A Survey on Service-oriented Middleware for Wireless Sensor Networks. *Serv. Oriented Comput. Appl.*, 5(2):71–85, June 2011.
55. Luca Mottola and Gian Pietro Picco. Middleware for wireless sensor networks: an outlook. *J. Internet Services and Applications*, 3(1):31–39, 2012.
56. Apala Ray. Planning and analysis tool for large scale deployment of wireless sensor network. *International Journal of Next-Generation Networks (IJNGN)*, 1(1):29–36, 2009.
57. Mihai T. Lazarescu. Design of a WSN Platform for Long-Term Environmental Monitoring for IoT Applications. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 3(1):45–54, March 2013.
58. Nadia Gámez, Javier Cubo, Lidia Fuentes, and Ernesto Pimentel. Configuring a context-aware middleware for wireless sensor networks. *Sensors*, 12(7):8544–8570, 2012.
59. Luca Mottola and Gian Pietro Picco. Programming Wireless Sensor Networks: Fundamental Concepts and State of the Art. *ACM Comput. Surv.*, 43(3):19:1–19:51, April 2011.
60. Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.