A SERVICE-DRIVEN DEVELOPMENT TOOL FOR WIRELESS SENSOR NETWORK

Zhen Yu Song[1], Luciano Lavagno[1], Riccardo Tomasi[2] and Maurizio A. Spirito[2]
[1]*Dipartimento di Elettronica, Politecnico di Torino, Torino, Italy*
[2]*Pervasive Technologies Area, Istituto Superiore Mario Boella, Torino, Italy*

Keywords: WSN Programming, WSN Development Tools.

Abstract: Thanks to advances in the area of embedded low-power microprocessors and short-range wireless communication, pervasive technologies such as Wireless Sensor Networks (WSNs) are easing the collection and integration of real-world data into ICT systems. However, developing and testing application logic for heterogeneous WSN devices remains a challenging and cumbersome task. In order to make prototyping of WSN solutions faster and less error-prone, in this paper we propose a set of development tools for WSNs based on object-oriented and service-driven models. Within these tools, applications are modelled as sets of interconnected blocks, each providing or using a number of services defined at design time. Externally, each block is a self-contained black-box exposing a set of service interfaces and tunable attributes; internally, an event-driven state-chart model represents its logical behaviour. All blocks are automatically created as skeleton templates by the tools, and then can be graphically developed and debugged in different hierarchical depths through widely used Mathworks © tools. Moreover, the developed functional blocks can be automatically converted to platform-specific binaries to ease deployment on actual devices (e.g. on TinyOS-based platforms) and large scale simulation (e.g. in MiXiM) enriched with HIL (Hardware In the Loop) capabilities.

1 INTRODUCTION

Wireless Sensor Networks (WSNs) are communities of tiny, cooperating, resource-constrained objects used to sense and collect physical information.

While in past years WSNs have been considered mostly from the research point of view, nowadays WSN technology has become mature and many different WSN commercial solutions are now available.

Despite the relative maturity of WSN technology, when designing and operating a WSN in real-world scenarios, a number of specific challenges must be taken into account, including: the need to develop application code using low-level languages, often intertwined with system level, platform-specific code; the need to efficiently debug and validate WSN solutions in large-scale simulations, before deploying massive (and expensive) field tests; the need to coequally execute an algorithm on heterogeneous WSN platforms.

The aforementioned challenges, which make developing and testing WSN application tedious and error-prone, directly imply the need for appropriate design and development tools.

Co-design and development-support tools are already gradually responding to these growing needs. For instance a solution to develop WSN applications using a high-level, Stateflow-based approach has been proposed in (Song et al., 2010). The HySim provides code generation and Hardware-In-the-Loop (HIL) co-simulation facilities in order to ease validation of solutions and deployment across multiple WSN platforms. However, HySim had a number of limitations, including the need to embed the entire application logic inside a single, large state-chart block; moreover, HySim simulation capabilities were limited to small-scale networks (up to some tenths of nodes).

This work introduces two major enhancements to reinforce HySim in terms of *scalability* and *modularity*, so as to make it more suitable for the design of large-scale distributed WSN applications. Firstly, it introduces a programming abstraction supporting more complex models, mixing object-oriented and service-driven approaches; secondly, support for large-scale simulation has been added, including the possibility to interface development tools, large-scale simulation environments and actual hardware.

To allow such abstraction, HySim block is introduced as the self-contained building unit, which is

Yu Song Z., Lavagno L., Tomasi R. and A. Spirito M..
A SERVICE-DRIVEN DEVELOPMENT TOOL FOR WIRELESS SENSOR NETWORK.
DOI: 10.5220/0003818100870095
In *Proceedings of the 2nd International Conference on Pervasive Embedded Computing and Communication Systems* (PECCS-2012), pages 87-95
ISBN: 978-989-8565-00-6

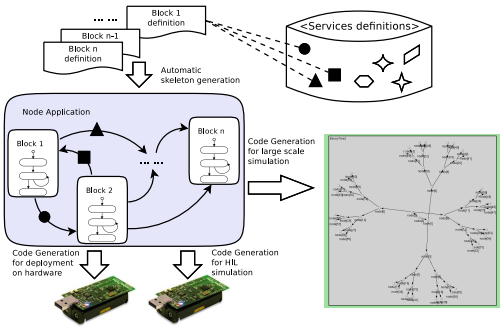

Figure 1: The proposed HySim extension.

externally characterized by the set of services it can consume and expose. As shown in Fig. 1, the proposed solution allows developers to graphically design and debug a WSN application as hierarchically decomposed and loosely coupled "HySim blocks". The block skeleton templates are automatically generated by our tool from services definitions. After the design phase, the tool provides the capability to validate the design via functional small-scale simulation, deploy the solution on different hardware platforms, or verify its performance in large-scale simulation.

The proposed solution leverages a number of widely used tools from the Mathworks©: Simulink® is used as a functional design and simulation environment, providing functionalities to graphically design blocks at a high-level of abstraction and to support quick wiring and configuration among them; Stateflow® is used to model the internal behavior of blocks; Stateflow Coder®, finally, is used for its code-generation capabilities.

Concerning performance simulation of large-scale networks, the proposed solution includes support for generation of OMNeT++ (Varga, 2000) and MiXiM (Köpke et al., 2008) models.

Concerning deployment on actual hardware, the previous approach has been mantained, supporting generation of code for actual WSN platforms e.g. TinyOS (Levis et al., 2004).

The remaining of the paper is organized as follows: Sec. 1.1 describes relevant related work; Sec. 2 describes the proposed solution, introducing its underlying model and reference work-flow; Sec. 3 presents an example use-case; Sec. 4 draws conclusions.

1.1 Related Work

A wide variety of tools to support WSN deployment is available, focusing on different aspects such as high-level modelling, architectural design, detailed code development, code generation, co-simulation, deployment, validation, debugging, performance monitoring and evaluation (Marron et al., 2009).

Part of these works is focused on programming abstractions suitable for WSN environments (Rubio et al., 2007). WISE-NES and the WSN API (Kuorilehto et al., 2008; Juntunen et al., 2006) provide a unified framework to design, simulate, deploy and use WSN solutions, leveraging the functionalities provided by the Specification and Description Language (SDL). A similar approach has been adopted to support the Insense programming language for WSN (Dearle et al., 2008), which offers the possibility to define and describe a target application, which can be subsequently compiled and mapped to a lightweight process by a dedicated compiler (Dunkels et al., 2004). Solutions such as Virtual Nodes (Ciciriello et al., 2006) and ATaG (Bakshi et al., 2005), supply various levels of programming abstractions to relieve the programmers from addressing low-level WSN mechanisms.

The COMDES framework (Angelov and Sierszecki, 2006; Angelov and Sierszecki, 2004) is a UML-based solution targeting development of Distributed Control Systems (DCS), which are related to WSNs. COMDES enables the definition of applications in terms of interacting subsystems (function units), such as sensor, control unit, actuator, operator station, etc. These subsystems can be soft-wired with one another in order to configure specific applications and their internal behavior can be specified as an operational state machine. Within COMDES, an application can be graphically composed using the tool provided meta-models in the associated graphical Generic Modelling Environment (GME). A number of graphical programming environments are also available for specific platforms (Ghercioiu, 2005; Quantum Leaps, 2011).

The appearance of light-weigth Java virtual machines, like MiniMV (Cota et al., 2010) and Squawk (Shaylor et al., 2003), allows the designers to develop WSN applications using a high level programming language, thus with slightly reduced application code performance due to the the extra resources consumed by the virtual machine. Other techniques based on virtual machines are also available to relieve the designers from concerns due to the low level logic, such as UML technique based virtual machine Matilda (Wada et al., 2007).

Solutions such as Cougar (Bonnet et al., 2001), TinyDB (Madden et al., 2005) and Spine (Gravina et al., 2008) employ a different approach. They leverage distributed database techniques and extensible processing and query mechanisms to abstract the underlying network as an entity. Within these solutions,

a task is described in high-level languages, injected in the network and transformed into low-level procedures running on individual nodes. However, they partially reduce the possibility to obtain a fine-grained control over application logic due to the limited expressiveness of high-level task description languages.

Simulation is widely employed in the first phase of WSN application development, to allow fast and unexpensive verification. For this reason, much effort has been devoted to providing rapid prototyping and graphical debugging capability both for generic (Breslau et al., 2000; Varga, 1999), and specialized simulators (Köpke et al., 2008; Levis et al., 2003; Osterlind et al., 2006). Thanks to these efforts, network protocols and algorithms can be conveniently and repeatedly evaluated and analysed in a scalable manner through virtual deployments. In addition, to increase the realism, simulation techniques can be enriched with Hardware-In-the-Loop (HIL) capabilities, resulting in co-simulative approaches.

In summary, a number of solutions can be employed to ease development of WSN applications (e.g. virtual machines, programming abstractions, code-generation, graphical composition of functionalities, co-simulation, etc).

Nevertheless, a single comprehensive approach has not yet emerged.

Moreover, application scenarios are quickly growing in complexity and scale (Marron et al., 2009), increasing the need to distribute intelligence and support more complex in-network operations, though maintaining reliability and trustworthiness.

As a result, current WSN development tools must evolve to better support composable functionalities and complex distributed scenarios; service-driven models can play a key role in this evolution.

2 THE PROPOSED FRAMEWORK

The proposed framework attempts to combine two different models for structure and behaviour, respectively blocks and hierarchical Finite State Machines (FSMs).

Leveraging the structural model, WSN applications running on the nodes are modelled as a set of interconnected blocks. On the other hand, an event-driven state-chart model is used to represent the internal behaviour of each block.

In order to exploit these abstractions, described in Sec. 2.1, to effectively support WSN development, a set of tools and a reference work-flow are discussed in Sec. 2.2.

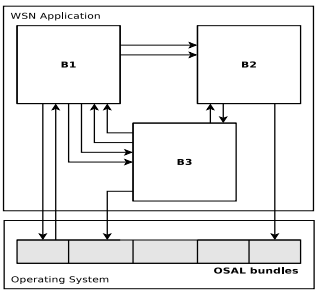

Figure 2: Node abstraction.

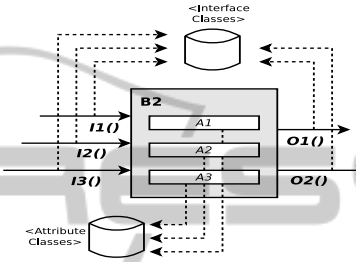

Figure 3: Block abstraction: black-box view.

2.1 Model

Blocks are self-consistent units which are perceived by other blocks as “black-boxes”, only exposing externally a set of known services. Therefore, internal details of blocks do not depend on external entities and thus they can be “wired” with each other in a loosely-coupled fashion.

As described in Fig. 2, the required functionalities of any user-defined WSN application are delivered collectively by a set of blocks (B_1, B_2, …, B_N). All the relevant components and APIs of the host WSN operating system are also exposed as blocks e.g. as a set of OSAL (Operating System Abstraction Layer) blocks. OSAL blocks are platform-dependent, and must thus be provided in the DDK (Device Development Kit) associated with the specific platform.

As described in Fig. 3, a block only exposes a variable number of static attributes (A_1, A_2, …, A_N) and a set of in-bound and out-bound interfaces (respectively I_1, I_2, …, I_N and O_1, O_2, …, O_N). Attributes are static and allow tuning of relevant parameters within each block. Interfaces reflect services definitions, and can thus belong to different service classes, each with a different set of arguments. They are similar to unidirectional function calls and can be used to transmit service-specific events from one block to another, without exposing implementation details.

Within the model, all interfaces and attributes are typed as classes, which are defined during design

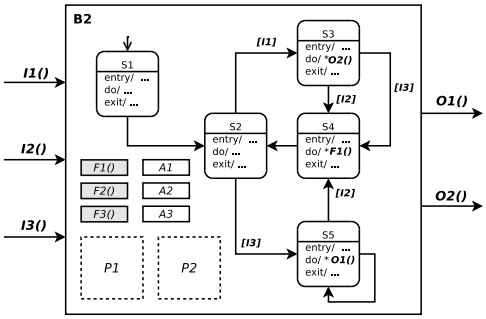

Figure 4: Block abstraction: internal view.

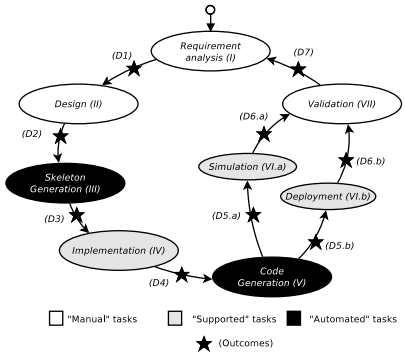

Figure 5: Reference work-flow.

phase. Basic types can support standard numeric arguments (e.g. *uint16_t*, *double*, *char*, ...), while complex types can be derived by composing basic and complex types.

Any out-bound interface of a block can be wired to any in-bound interface of another, as long as they share the same interface type. Events emitted from disconnected out-bound interfaces are simply discarded, while no event can be received from disconnected in-bound interfaces.

The behaviour of each block can be internally modelled as an event-driven hierarchical FSM model (more precisely, a State-chart), as described in Fig. 4. Each FSM has a set of states (S_1, S_2, ..., S_N), one of which (S_I) is the initial state. The logic flow (i.e. the switch from the current active state to the next) can be controlled either by internal default transitions or external events incoming from other blocks (I_1, I_2, ..., I_N). User-defined operations can be implemented inside each state: they will be executed upon entry, permanence or exit phases within each state. Such user-defined operations can execute user-defined local functions (F_1, F_2, ..., F_N) to implement computational tasks, as well as generate out-going events (O_1, O_2, ..., O_N). Besides the externally-tunable attributes, additional local variables can be freely defined within each block. In case a single FSM is not sufficient to model the desired functionalities, a single block can host one or more sub-charts which can be integrated with the main FSM either in sequential or parallel execution order, sharing the set of incoming and outgoing signals, attributes and local variables.

2.2 Work-flow and Tools

The reference work-flow for the proposed solution involves three types of development activities or "tasks" (numbered from *I* to *VII*).

Manual tasks are not directly supported by the proposed solution: they must be manually performed by designers e.g. using other development tools. *Supported* tasks also imply some activity by designers: however, a number of specific tools are provided to support them. *Automated* tasks are instead fully performed by the proposed solution.

As shown in Fig. 5, each task delivers partial outcomes (*D1* ... *D7*), which feed into subsequent tasks.

The considered development work-flow (which can also be viewed as an iterated execution of the classical V-shaped development flow) starts with an analysis of the requirements from the application scenario (Task *I*). The expected outcome of Task *I* is a list of requirements mapped to functionalities (*D1*).

Outcome *D1* can be used to drive the design phase (Task *II*). Within this phase the developer decomposes the global WSN functionality as a network of interconnected blocks (similar to Fig. 2), whose internal behaviour is not yet precisely defined.

The resulting design specifies the functionalities assigned to each block, their main attributes as well as the interfaces which are used to "wire" them together.

Interfaces and attributes can be defined from scratch or taken from a library of pre-existing interfaces and attribute types, defined in previous projects. Since blocks are identified by their "borders", *D2* is a collection of block definitions sharing common types for attributes and interfaces, plus wiring information.

Block interfaces and attributes can be specified using the *WSN Block Definition Language* (WSN-BDL). WSN-BDL files are XML files divided in two parts: the first part specifies all the attribute and interface types (*AClasses* and *IClasses*), and can be shared among multiple WSN-BDL files; the second part lists all the actual attributes and interfaces exposed by a block; for interfaces, also the direction (*dir*) is specified (incoming or outbound). Currently, wiring information is not specified in WSN-BDL, since wiring is performed manually in further steps.

As depicted in Fig. 6, both classes and instances of interfaces and attributes are defined as XML elements. Fig. 6, provides a possible WSN-BDL de-

```
<?xml version="1.0" encoding="UTF-8"?>
<Block name="BlockExample" class="BlockClass">
 <!-- Class Definitions -->
 <AClass name="MYATT1">
  <Arg name="myarray" type="uint8" dim="10"/>
 </AClass>
 <AClass name="MYATT2">
  <Arg name="myvalue" type="uint8" dim="1"/>
 </AClass>
 <IClass name="MYCL1">
  <Arg name="myval1" type="uint8" dim="1"/>
  <Arg name="myval2" type="int16" dim="1"/>
 </IClass>
 <IClass name="MYCL2">
  <Arg name="myval1" type="int16" dim="1"/>
 </IClass>
 <IClass name="MYCL3">
 </IClass>
 <!--  Instances -->
 <Attribute name="A1" class="MYATT1"/>
 <Attribute name="A2" class="MYATT2"/>
 <Attribute name="A3" class="MYATT2"/>
 <Interface dir="in" name="I1" class="MYCL1"/>
 <Interface dir="in" name="I2" class="MYCL2"/>
 <Interface dir="in" name="I3" class="MYCL3"/>
 <Interface dir="out" name="O1" class="MYCL3"/>
 <Interface dir="out" name="O2" class="MYCL3"/>
</Block>
```

Figure 6: WSN-BDL example.

scription for the designs described in Fig. 3 and Fig. 4. The "Class Definition" segment include two attribute classes (*MYATT1* and *MYATT2*) and three interface classes (*MYCL1*, *MYCL2* and *MYCL3*). This section defines the types of attributes and interfaces; for instance the *MYATT1* attribute must be composed of a 10-element array of unsigned 8-bits words, named *myarray*. It is also possible to model interfaces without arguments, e.g. *MYCL3*, which is useful to model events which do not carry any numeric value.

The "Instances" segment specifies the actual attributes (*A1*,*A2*,*A3*) and interfaces (*I1*,*I2*,*I3*,*O1*,*O2*) included in the *BlockExample* block, each with its reference to a specific class; for instance in-bound interface *I1* belongs to class *MYCL1*, and thus will support incoming events transporting two arguments: an unsigned 8-bit word (*myval1*) and a signed 16-bits integer (*myval2*).

D2 (a collection of WSN-BDL files) can be fed into the FSM Skeleton Generator Tool (FSM-SG) to accomplish the Skeleton Generation phase (Task *III*). The FSM-SG is used to parse the WSN-BDL and generate a block template.

The FSM-SG tool has been implemented based on Mathworks© products, namely Matlab®, Simulink® and Stateflow®. The block template generated by the FSM-SG tool (*D3*) is a Simulink® block, providing a bus plug for each external interface and containing a Stateflow® FSM skeleton, which can be used as a starting point for the implementation phase (Task *IV*).

The block template provided by the FSM-SG tool contains a "Main" FSM already populated with some example states (e.g. INIT, START, etc.), whose transitions are controlled by incoming events. It also contains examples of calls to outbound interfaces.

The implementation task includes a "local" version of the implement/simulate/debug cycle, performed using the Simulink® and Stateflow® tools. This simulation can include the interaction of a small set of blocks (modeling a single or a few WSN nodes).

The provided template includes Stateflow®-specific input and output drivers, which hide the complexity of handling low-level Simulink® signals. As a result, the internal block implementation can be defined just in terms of high-level events, commands and attributes defined in the initial WSN-BDL file.

Once implementation and debugging have been completed, the resulting block prototypes (*D4*) are ready to be deployed.

In order to provide support for multiple deployment platforms (e.g. simulators or node software platforms), block implementations are processed by the FSM Code Generator tool (FSM-CG, Task *V*). The FSM-CG tool is able to generate platform specific code for either large-scale simulators such as MiXiM (*D5.a*) or actual WSN platforms (*D5.b*) such as TinyOS. Outcome *D5.a* can be used within simulators such as MiXiM (Köpke et al., 2008) to simulate a large number of WSN nodes (while Simulink®-based simulation within Task *IV* could be used only for a limited number of blocks). Outcome *D5.b* can instead be used, after platform-specific compilation, to test the developed application on the actual hardware.

In MiXiM a block is mapped to a *simple module* described by a NED file, while the generated internal implementation is a set of C++ classes, and interfaces are modelled by MiXiM buses transporting messages with the appropriate interface type.

In TinyOS, blocks are mapped to NesC modules connected through TinyOS interfaces, while matching C structures are provided to support interface serialization i.e. the operation of converting a function call representing an event into a format which can be stored or transmitted across a communication link.

After this stage, blocks must be manually wired together, based on information provided in (*D2*).

Both simulation and deployment (Tasks *VI.a* and *VI.b*), can provide experimental results (*D6.a* and *D6.b*) for a final validation phase (Task *VII*).

The resulting validation results (*D7*) can be used to verify requirements and subsequently improve the design, feeding a new loop of the iterative work-flow.

It is important to observe that the service-driven model employed in blocks definition is primal to draw a common line which remains consistent across all design, development and deployment phases. While the model remains the same, the internal application

logic can take different forms: for instance, a block can be initially developed as a “Simulink Model”, then translated to a “MiXiM Module” in a simulation phase, then converted to a “NesC Model” in the deployment stage and finally be compiled as a binary image. Conversion between different forms is automated by the HySim tools.

3 EXAMPLE

For validation purposes, the proposed solution has been tested on a number of actual use cases. While the proposed solution is suitable for development of complex application models, this paper uses a very simple example to explain the main features of the tools and the work-flow. The described use case is a simple application where nodes perform collection and distributed processing of data from a temperature sensor.

In this example, each WSN node:

1. samples temperature values;
2. collaboratively averages the monitored values with those from its one-hop neighbours within a sliding time window;
3. broadcasts its latest calculated average temperature value back to its neighbours.

The following parameters have been identified as potential tunable attributes of each node:

1. its own node identifier (*NodeId*);
2. the sampling period of the temperature sensor (*SamplingPeriod*);
3. the size of the time window to compute averages (*AvgWinSize*);

Based on the above requirement analysis (*task I*), a sketch design is modelled (*task II*). The resulting design divides the application into three blocks: a Sensor block, a Radio block, and a data processing block (*TempAverager*). The Sensor block deals with sampling and pre-processing of temperature data. The Radio block provides support to interface with the radio protocol used to communicate with neighbours. The *TempAverager* block contains all the data processing capabilities. Both the Sensor and the Radio block are connected to the *TempAverager* block.

This preliminary design allows one to concentrate only on the “borders” of the *TempAverager* block, as all the interfaces of other blocks will be shared with it. The resulting WSN-BDL file is reported in Fig. 7.

According to the definition, the *TempAverager* block can request a temperature reading through the *SenseReq* interface, which will cause an asynchronous call to the inbound *SenseRep* when a valid

```
<?xml version="1.0" encoding="UTF-8"?>
<Block name="TempAverager" class="BlockClass">
  <!-- Class Definitions -->
  <AClass name="NodeAddr">
    <Arg name="addr32" type="uint32" dim="1"/>
  </AClass>
  <AClass name="TimeMs">
    <Arg name="t" type="uint16" dim="1"/>
  </AClass>
  <AClass name="NSamplesC">
    <Arg name="n" type="uint16" dim="1"/>
  </AClass>
  <IClass name="Msg">
    <Arg name="Src" type="NodeAddr" dim="1"/>
    <Arg name="Val" type="int16" dim="1"/>
  </IClass>
  <IClass name="SenReq">
  </IClass>
  <IClass name="SenRep">
    <Arg name="SenVal" type="int16" dim="1"/>
  </IClass>
  <!-- Instances -->
  <Attribute name="NodeId" class="NodeAddr"/>
  <Attribute name="SamplingPeriod" class="TimeMs"/>
  <Attribute name="AvgWinSize" class="NSamplesC"/>
  <Interface dir="in"  name="RecvMsg"  class="Msg"/>
  <Interface dir="out" name="SendMsg"  class="Msg"/>
  <Interface dir="out" name="SenseReq" class="SenReq"/>
  <Interface dir="in"  name="SenseRep" class="SenRep"/>
</Block>
```

Figure 7: WSN-BDL of the TempAverager block.

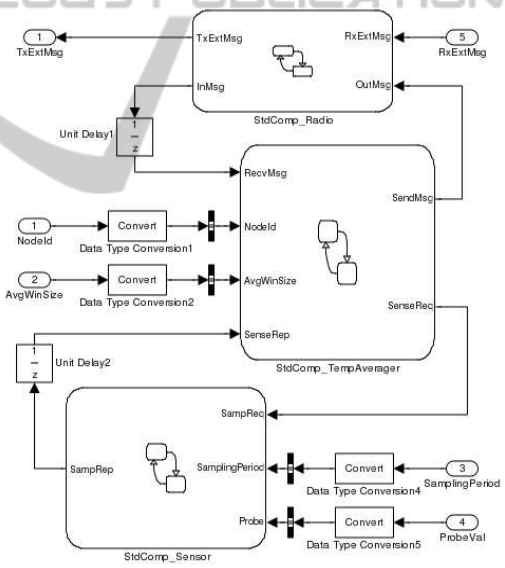

Figure 8: Node applications composed of linked blocks.

value is ready. When a processing phase is complete, the *TempAverager* block can request a data broadcast to the radio through the *SendMsg* command, which will cause a *RecvMsg* event in the 1-hop neighbours.

When these definition files are fed to the FSM-SG, the corresponding Simulink block templates are automatically generated (*task III*) and can be manually wired as described in Fig. 8.

Within the Simulink®, in-bound and out-bound interfaces are mapped to input and output ports respectively, to exchange service-specific events with

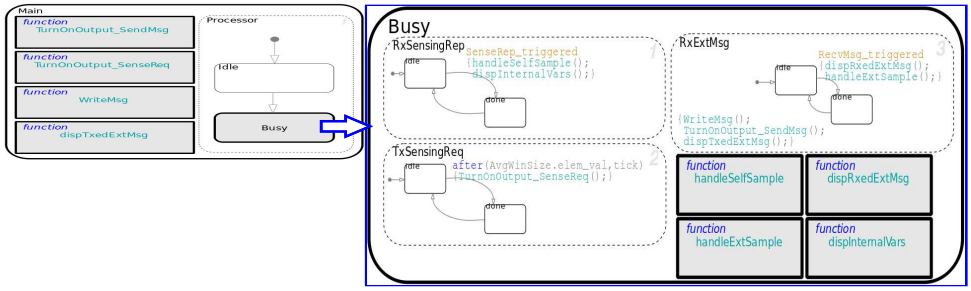

Figure 9: Implementation of TempAverager block.

other blocks, such as *RecvMsg*, *SendMsg*, *SenseRep* and *SenseReq* in Fig. 8. The attribute instances are mapped to input ports in order to import the tunable parameter values into the block, i.e. *NodeId* and *AvgWinSize* for *StdComp_TempAverager*.

As shown in Fig. 9, the internal logic of the *TempAverager* block is implemented using an event-driven FSM paradigm, supported by APIs available inside the template and reflecting external interfaces and attributes.

In the example, a two-state (*Idle*/*Busy*) implementation is proposed in the *Main* portion. Inside the *Busy* state, three parallel state machines (*RxSensingRep*, *TxSensingReq* and *RxExtMsg*) are in charge of receiving sensing samples from the *SenseRep* port, sending sensing request to the *SenseReq* port and processing the received external message from the *RecvMsg* port respectively. The developer can rely on both APIs to model incoming events, e.g. the *RecvMsg_triggered* event, and also to model outgoing commands e.g. *TurnOnOutput_SendMsg*. As previously mentioned, except for the APIs that are exposed in the template, the internal behaviour in Fig. 9 is independent from the external context shown in Fig. 8.

To verify this part of the design in isolation, the Radio and the Sensor have been modelled as simple traffic generators and the whole system has been graphically debugged as a normal Simulink model.

Afterwards, the developed Simulink blocks have been fed into FSM-CG (*task V*) for large-scale simulation (*task VI.a*) and real deployment (*task VI.b*).

In the MiXiM simulation domain, those functional blocks are transformed to “Simple modules” (*Sensor*, *Radio*, *TempAverager*) as displayed in Fig. 10, and manually interconnected and configured in the initialization file (e.g. to assign to each node a *NodeId*, *AvgWinSize* and *SamplingPeriod* values). In the example, an “adaptor” object, provided with the DDK, is used as an adaptation layer simply to accommodate the messages to send and receive through the standard ports between the Appl layer and the NIC layer.

Similar to the network in Fig. 10, any application

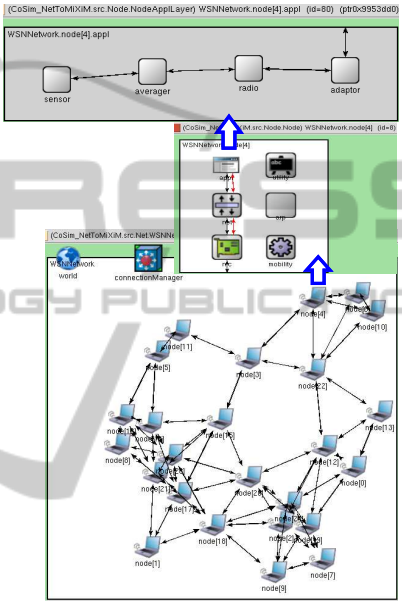

Figure 10: Simulation in MiXiM.

logic can be easily constructed with the obtained compound node module (the automatically transformed modules can be connected and placed in the Appl layer of the MiXiM node stack) so as to evaluate the performance of the algorithm in a large-scale simulation, while the configurations like size of the playground, sensitivity of transceivers, initial position and mobility of each node can be simply set in the initialization file and managed by MiXiM.

To validate the deployment capabilities, a simple test-bed has been set up using a set of Memsic Telos rev. B nodes running TinyOS.

In this case the FSM-CG tool transforms blocks in the form of NesC modules. For ease of integration, the automatically generated NesC modules (*TOS_StdComp_RadioC*, *TOS_StdComp_SensorC* and *TOS_StdComp_TempAveragerC*) are firstly configured, interconnected and encapsulated into a NesC module, ManualTOSGenedAppC.

The generated blocks are consequently manually

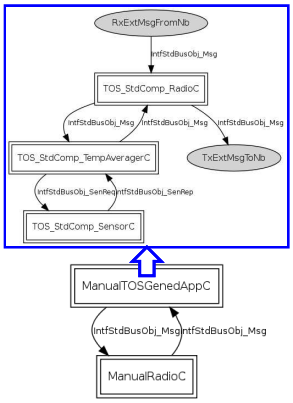

Figure 11: Deployment in TinyOS.

Table 1: Code size and the memory usage for the example application.

	ROM (bytes)	RAM (bytes)
Hand-written	17220	492
Auto-generated	20562	526

wired in TinyOS to adapt the existing radio communication services as presented in Fig. 11. Finally, the resulting project has been compiled and downloaded into real nodes. Code size and memory usage have been compared for both the binary version generated with the proposed solution and the same application logic implemented manually. As shown in Table 1, in the described example the proposed solution does not introduce significant overhead.

4 CONCLUSIONS AND FUTURE WORKS

In this paper, we presented an object-oriented and service-oriented application development tool for WSNs, which decomposes an application into a set of blocks that can be graphically developed and easily debugged separately or integrated in a network setup. In order to relieve the developers from the cumbersome and error-prone re-implementations task, the blocks can be seamlessly ported to different platforms, with minor wiring and adaptation for simulation and deployment.

The presented use-case provides a demonstration about how in-network processing capabilities can be developed with the proposed solution. The use-case can be easily extended to more complex applications e.g. by connecting with more functional blocks or adding more parallel operational FSMs.

Short-term future directions include the extension of the FSM-CG tool to support more actual platforms (e.g. through the Contiki O.S. (Dunkels et al., 2004)) and validation of the proposed solution on more realistic usage scenarios.

Longer-term future works will investigate how to ease discovery and distribution of intelligence to support composition of WSN applications over multiple nodes. These works will leverage the ability of the proposed solution to turn a WSN application into a set of composable blocks exposing services, which can be eventually be discovered and executed remotely.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge partial funding of this work by the regional project “Piattaforma Tecnologica Innovativa per l’Internet of Things”.

REFERENCES

Angelov, C. and Sierszecki, K. (2004). A Software Framework for Component-Based Embedded Applications. *11th Asia-Pacific Software Engineering Conference*, pages 655–662.

Angelov, C. and Sierszecki, K. (2006). A Component-Based Framework for Distributed Control Systems. *32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO’06)*, pages 20–27.

Bakshi, A., Prasanna, V. K., Reich, J., and Larner, D. (2005). The abstract task graph: a methodology for architecture-independent programming of networked sensor systems. In *Proceedings of the 2005 workshop on End-to-end, sense-and-respond systems, applications and services*, EESR ’05, pages 19–24, Berkeley, CA, USA. USENIX Association.

Bonnet, P., Gehrke, J., and Seshadri, P. (2001). Towards sensor database systems. In *Proceedings of the Second International Conference on Mobile Data Management*, MDM’01, pages 3–14, London, UK. Springer-Verlag.

Breslau, L., Estrin, D., Fall, K., Floyd, S., Heidemann, J., Helmy, A., Huang, P., McCanne, S., Varadhan, K., Xu, Y., and Yu, H. (2000). Advances in network simulation. *Computer*, 33(5):59–67.

Ciciriello, P., Mottola, L., and Picco, G. P. (2006). Building virtual sensors and actuators over logical neighborhoods. In *Proceedings of the international workshop on Middleware for sensor networks*, MidSens ’06, pages 19–24, New York, NY, USA. ACM.

Cota, C., Aguilar, L., and Licea, G. (2010). A java compatible virtual machine as an embedded middleware for wireless sensor networks. In *Electronics, Robotics and Automotive Mechanics Conference (CERMA), 2010*, pages 265 –270.

Dearle, A., Balasubramaniam, D., Lewis, J., and Morrison, R. (2008). A component-based model and language for wireless sensor network applications. In *Proceedings of the 2008 32nd Annual IEEE International Computer Software and Applications Conference*, pages 1303–1308, Washington, DC, USA. IEEE Computer Society.

Dunkels, A., Grönvall, B., and Voigt, T. (2004). Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Workshop on Embedded Networked Sensors*, Tampa, Florida, USA.

Ghercioiu, M. (2005). A graphical programming approach to wireless sensor network nodes. In *Sensors for Industry Conference, 2005*, pages 118 –121.

Gravina, R., Guerrieri, A., Fortino, G., Bellifemine, F., Giannantonio, R., and Sgroi, M. (2008). Development of body sensor network applications using spine. In *Systems, Man and Cybernetics, 2008. SMC 2008. IEEE International Conference on*, pages 2810 –2815.

Juntunen, J., Kuorilehto, M., Kohvakka, M., Kaseva, V., Hannikainen, M., and Hamalainen, T. (2006). Wsn api: Application programming interface for wireless sensor networks. In *Personal, Indoor and Mobile Radio Communications, 2006 IEEE 17th International Symposium on*, pages 1–5.

Köpke, A., Swigulski, M., Wessel, K., Willkomm, D., Haneveld, P. T. K., Parker, T. E. V., Visser, O. W., Lichte, H. S., and Valentin, S. (2008). Simulating wireless and mobile networks in omnet++ the mixim vision. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems&workshops*, Simutools'08, pages 71:1–71:8, ICST, Brussels, Belgium, Belgium.

Kuorilehto, M., Hännikäinen, M., and Hämäläinen, T. D. (2008). Rapid design and evaluation framework for wireless sensor networks. *Ad Hoc Netw.*, 6(6):909–935.

Levis, P., Lee, N., Welsh, M., and Culler, D. (2003). Tossim: accurate and scalable simulation of entire tinyos applications. In *Proceedings of the 1st international conference on Embedded networked sensor systems*, SenSys'03, pages 126–137, New York, NY, USA. ACM.

Levis, P., Madden, S., and Gay, D. (2004). *TinyOS: An Operating System for Sensor Networks*. Ambient Intelligence edited by W. Weber, J.Rabaey, and E. Aarts.

Madden, S. R., Franklin, M. J., Hellerstein, J. M., and Hong, W. (2005). Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30:122–173.

Marron, P. J., Karnouskos, S., Minder, D., and the CONET consortium (2009). *Roadmap on Cooperating Objects*. Kluwer Academic Publishers, Luxembourg, EU.

Osterlind, F., Dunkels, A., Eriksson, J., Finne, N., and Voigt, T. (2006). Cross-level sensor network simulation with cooja. In *Local Computer Networks, Proceedings 2006 31st IEEE Conference on*, pages 641–648.

Quantum Leaps (2011). The qp framework. http://www.state-machine.com/.

Rubio, B., Diaz, M., and Troya, J. (2007). Programming approaches and challenges for wireless sensor networks. In *Systems and Networks Communications, 2007. ICSNC 2007. Second International Conference on*, page 36.

Shaylor, N., Simon, D. N., and Bush, W. R. (2003). A java virtual machine architecture for very small devices. *SIGPLAN Not.*, 38:34–41.

Song, Z. Y., Mostafizur, M., Mozumdar, R., Tranchero, M., Lavagno, L., Tomasi, R., and Olivieri, S. (2010). Hy-sim: model based hybrid simulation framework for wsn application development. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*, SIMUTools '10, pages 87:1–87:8, ICST, Brussels, Belgium, Belgium.

Varga, A. (1999). Using the omnet++ discrete event simulation system in education. In *IEEE Transactions on Education*, volume 42, page 11 pp.

Varga, A. (2000). The OMNET ++ Discrete Event Simulation Event. *Proceedings of the European Simulation Multiconference (ESM'2001)*.

Wada, H., Boonma, P., Suzuki, J., and Oba, K. (2007). Modeling and executing adaptive sensor network applications with the matilda uml virtual machine. In *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications*, pages 216–225, Anaheim, CA, USA. ACTA Press.