Wireless Sensor Networks 38

Mihai Teodor Lazarescu and Luciano Lavagno

Abstract

Versatile and effective, Wireless Sensor Networks (WSNs) witness a continuous expansion of their application domains. Yet, their use is still hindered by issues such as reliability, lifetime, overall cost, design effort and multidisciplinary engineering knowledge, which often prove to be daunting for application domain experts. Several WSN design models, tools and techniques were proposed to solve these contrasting objectives, but no single comprehensive approach has emerged. With these criteria in mind we review several of the most representative ones, then we focus on two of the most effective hardware/software codesign flows. Both offer high-level design entry interfaces based on StateCharts. One allows manual module composition in a full application, and automates its mapping on a user-defined architecture for fast high-level design space exploration. The other flow automates module composition starting from the application specification and by reusing library modules. It can generate the hardware specification and the software to program and configure the WSN nodes. For these we show the typical use for the development of some representative applications, to evaluate their effectiveness.

Acronyms

6LoWPAN	IPv6 over Low Power Wireless Personal Area Network
ADC	Analog-to-Digital Converter
ADM	Abstract Design Module
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
BOM	Bill of Materials
CAN	Controller Area Network

M.T. Lazarescu (✉) • L. Lavagno
Politecnico di Torino, Torino, Italy
e-mail: mihai.lazarescu@polito.it; luciano.lavagno@polito.it

© Springer Science+Business Media Dordrecht 2017
S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_38

CRC	Cyclic Redundancy Check
DMA	Direct Memory Access
DSML	Domain-Specific Modeling Language
EEPROM	Electrically Erasable Programmable Read-Only Memory
EMF	Eclipse Modeling-Framework
FSM	Finite-State Machine
GPIO	General-Purpose Input/Output-pin
GPRS	General Packet Radio Service
GPT	General-Purpose Timer
HAL	Hardware Abstraction Layer
I2C	Inter-Integrated Circuit
ICU	Input Capture Unit
ID	Identifier
I/O	Input/Output
IoT	Internet of Things
IP	Intellectual Property
ISR	Interrupt Service Routine
MAC	Media Access Control
MBD	Model-Based Design
MMC/SD	Multimedia/Secure Digital Card
NVIC	Nested Vectored Interrupt Controller
OS	Operating System
PWM	Pulse-Width Modulation
QoS	Quality of Service
RAM	Random-Access Memory
RC	Resistor-Capacitor
RFID	Radio-Frequency Identification
RF	Radio Frequency
RTC	Real-Time Clock
RTOS	Real-Time Operating System
SDC	Secure Digital Card
SPI	Serial Peripheral Interface
TCP/IP	Transmission Control Protocol/Internet Protocol
TWI	Two Wire Interface
UART	Universal Asynchronous Receiver/Transmitter
UML	Unified Modeling Language
USART	Universal Synchronous/Asynchronous Receiver/Transmitter
USB	Universal Serial Bus
WSDL	Web Service Definition Language
WSN	Wireless Sensor Network
XMI	XML Metadata Interchange
XML	Extensible Markup Language

Contents

38.1 Introduction

Wireless Sensor Networks (WSNs) already covers a broad range of applications in a variety of domains, which is continuously expanding, thanks to advances in research and technology. The range of requirements and problems that WSN designers must address are considerably more diversified today than when the Internet of Things (IoT) paradigm was coined by Kevin Ashton [3] more than 15 years ago. It is increasingly difficult to define "typical" application requirements for WSN hardware and software [31], since both must continuously adapt to very diverse WSN application requirements and operating conditions. Moreover, WSN platform reusability for a wide class of derived applications is becoming more important to lower development effort and time to delivery and to increase reliability.

Existing high-level WSN programming support of any kind is still seldom used for applications deployed in the real world [24]. System and application development and deployment using state-of-the art WSN technologies involve several different and complementary views, yet lacking mature separation of competencies between typical stakeholders and the various engineering disciplines that cover the WSN domain. For various practical reasons, WSN deployments are typically developed at a level very close to the embedded Operating System (OS), which often requires mastering a mix of low-level system and distributed protocol competencies that are seldom found among WSN application domain experts. Figure 38.1 shows how such development flows divert significant development efforts from the application logic, thus contributing to increase development time and cost and lowering overall reliability.

Another factor limiting WSN widespread use are the difficulties faced when porting an implementation to a different hardware platforms. They effectively

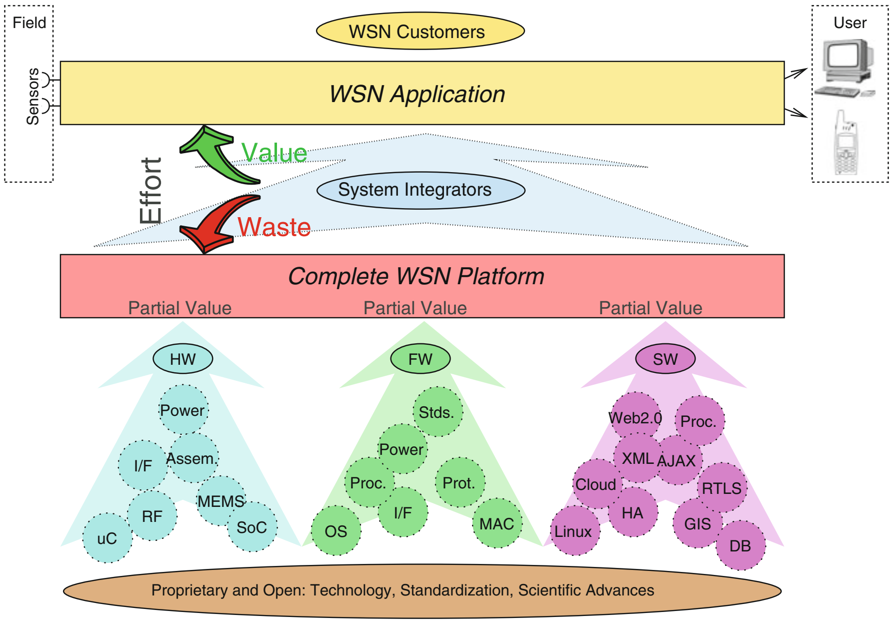

Fig. 38.1 Value flow for a WSN application and platform

reduce programmers' choices in terms of hardware platforms to those that are explicitly supported by each tool, which often is a narrow range.

High-level programming tools for WSN applications generally lack composability and the ability to be reused as building blocks. Code written according to a given programming abstraction can typically be used within a single framework because the collaboration between frameworks is still very limited. Although most frameworks are well suited for specific application domains, they can rarely be extended or composed with others. Consequently, their use is severely limited.

These are important aspects that limit the productivity of WSN application designers; lead to suboptimal or ineffective designs; increase the development time, cost, and risk; and reduce the WSN application reliability which, ultimately, increases maintenance cost.

In this context, we explore in detail two innovative tools for WSN application design. One is a model-based design framework for distributed WSN application development, design space exploration, network simulation (including hardware in the loop), and fast prototyping. It has a MATLAB Stateflow®-based [22] user-friendly interface for architecture-independent application entry, composition, and simulation and is based on an abstract service-based specification model for the application architecture. The framework can automatically map the application on several very different node platforms, and it can generate the simulation and implementation code for popular WSN simulators and operating systems. Hence, it hides most low-level implementation details from the developer in order to increase its usability by application domain experts and to allow fast high-level design space exploration.

The second toolset aims to further accelerate the development cycle and stimulate component reuse by automating the module selection and composition phases based on a high-level application specification. This flow does not make any assumption on the language or format of the behavior specification. Instead, it makes use of metadata that describe the behavior, interface and requirements of each library module, as well as the application specification. Besides simulation and implementation code, the system composition engine can generate nonfunctional requirements, such as a bill of materials and specifications for the target hardware or for the compilation toolchain.

Both frameworks are used to create several representative applications, in order to evaluate their effectiveness.

38.2 Past Work

As we discussed above, WSN design must satisfy two contrasting requirements:

1. Short design time with as little electronics and telecommunications expertise as possible. Many applications today are relatively small-scale and low-margin; hence, the nonrecurrent engineering costs should be kept as low as possible. Moreover, since application-specific aspects dominate both the difficulty and the gains, the most useful design resources are the domain experts.
2. Produce a very optimized implementation, especially for cost, power, and reliability. This is in order to minimize also the recurrent cost, both to procure the nodes, to deploy them, and to keep them running in the field.

The only way to satisfy both requirements is through design automation of some sort. As a result, the literature on WSN design is very rich of design aids, both in the form of design languages and their compilers, and in the form of support software, such as middleware, operating systems and so on. In this section, we review some of this work, in particular by referring to existing survey articles.

38.2.1 Programming Languages and Tools

A first excellent survey of programming languages and tools for WSNs, by Sugihara and Gupta, appeared in [34]. They classify devices and networks according to three orthogonal aspects:

1. Node power consumption, which is of course also related to the computational power, ranging from workstation-class computers, found typically at the “center” of the network, where the most complex elaboration takes place, to small battery-powered microcontrollers that have some ability to compute and route data, to tiny nodes (e.g., Radio-Frequency Identifications (RFIDs)) which scavenge energy from the environment. Most of the programming tools, including those

discussed in this chapter, apply to the middle group, where resources are scarce but computations are nontrivial.

2. Node observables, which range from just the data and an "address" (a node identifier), to the time and to the location of the observation. Satisfying real-time constraints and being able to reason about the location of the node add to the complexity of both the programming language and of the underlying middleware.
3. Size of the network, which ranges from tens to potentially millions of nodes, and which may require significant middleware support to ensure scalability.

At the lowest level of abstraction, namely, the individual node, the article lists several examples of operating systems and programming languages, classifying them according to the paradigm used for modularization: (1) message passing among statically "bound" software components, as in nesC [13] and TinyOS [17]; (2) dynamic association between messages and services, as in SNACK [14]; or (3) lightweight threads, as in Mantis OS [1]. Static binding requires the least resources but goes against flexibility and reconfigurability. Moreover, thread-based programming is more familiar to most developers but requires significantly larger memory resources. Virtual machines have also been considered in this context, although their portability advantages must be carefully weighed against their performance and energy cost.

The next layer of abstraction is group-level programming, in which application development, deployment, and maintenance are eased by the availability of programming constructs or APIs to reason about groups of nodes, based both on physical distance (neighborhood) and on logical properties. Making physical or logical location a first-class citizen of course enables much easier application development, since most WSN data processing must be aware of where the data itself originates.

Finally, the most abstract level considered in that article is the network level, where it describes:

1. database abstractions, where the network is viewed as a huge database, in which nodes and time instants play the role of rows, while kinds of sensed data play the role of columns, as in TinyDB [19]. User-level queries must be decomposed and distributed to physical nodes and radio channels so that Quality of Service (QoS) objectives, such as timeliness and energy consumption, are appropriately optimized.
2. macroprogramming approaches, which provide transparent mechanisms, e.g., to replicate and distribute data, so that the network is viewed as a single distributed computing platform, as in Kairos [15]. This in principle can provide the best optimization opportunities, allowing one to move both computation toward the data and vice versa, depending on the application constraints and requirements. But the huge optimization space makes low-level code generation an extremely challenging task.

Then the article classifies a large number of approaches based on the aspects listed above and evaluates them by their energy efficiency, scalability, failure resilience, and collaboration level (e.g., using centralized or distributed triangulation among nodes to establish node and data location).

Mottola and Picco provide another excellent survey of programming languages and tools for WSNs in [24]. They also include concrete examples to illustrate the key aspects of each listed approach. In addition to the space and time aspects that were discussed by [34], they also classify applications based on:

1. The goal, i.e., pure sensing or sensing and reacting (or actuation). The former leads naturally to a single or a few sinks, while the latter encourages distributed processing to improve resiliency and reduce communication costs.
2. The interaction pattern: many-to-one, often associated with sensing, one-to-many, often associated with reacting, or many-to-many.
3. The need to support mobility of at least some nodes (e.g., in cattle monitoring applications). This of course requires support for dynamic interaction and reconfiguration, at least at the network level but often also at the application level.

The article then proceeds to classify the *programming languages* based on aspects such as:

1. the scope of communication: (1) physical neighborhood, (2) multihop within a subset of nodes (again based on physical neighborhood or logical grouping), and (3) network-wide.
2. the addressing mechanism: (1) physical, or statically assigned, versus (2) logical, or dynamically assigned based on, e.g., current sensor readings or location.
3. whether communication is explicitly exposed, as in nesC, or implicit, as in TinyDB.
4. the scope of a computation: whether a single statement in the programming language can change the state of: (1) a single node, (2) a group of nodes, or (3) the entire network. Again, the latter offers more scope for optimization, and moves the burden from the programmer to the "compiler," but requires the development of more complex design tools.
5. the data access mode across nodes, via: (1) database abstractions, (2) shared remote variables, (3) code that migrates to find its data, or (4) explicit message passing. Database languages are easy to use, but lead to extremely complex low-level code generation issues when efficiency is important, as is often the case in energy-limited WSNs. Shared variables are also familiar to programmers but are very difficult to synchronize and use correctly. This is especially true in a setting where communication is slow and unreliable, and bandwidth is limited. Code migration can increase the longevity of networks, which can be often expensive to deploy. Code migration is best coupled with energy harvesting, because it can adversely impact battery life. Finally, explicit message passing gives most control to the programmer and is often the preferred choice for real-life deployments.

6. the programming paradigm: (1) imperative, (2) declarative (e.g., SQL-like or functional), or (3) hybrid, where a declarative language, which provides faster application development, can be extended with procedural mechanisms for the most performance- or energy- and power-critical aspects.

Then the article looks briefly at the architectural aspects, e.g.:

1. whether the approach supports only the application programmers (as is often the case with declarative languages and implicit communication) or can be used to build all layers of the software (which normally requires an imperative paradigm and explicit message passing).
2. the ability to access and tune lower layers (e.g., to allow cross-layer optimization of the protocol stack).

Finally, a very large number of different approaches are mapped and classified according to the criteria above.

The article concludes by outlining open areas for further research, such as:

1. tolerance to failures, which is essential for long-term real-life use in an often harsh environment.
2. ease of debugging, which is especially problematic when the nodes are already in the field.
3. real-world deployment, which is always needed to validate new ideas in realistic settings.
4. evaluation methodology, which suffers again from the lack of well-recognized benchmarks and of sensor data coming from real-world deployments.

38.2.2 Middleware and Operating System

Middleware and operating system are essential to support fast development and deployment of WSN software. Hence, they are the foundation (explicitly or implicitly) of all the approaches to WSN design that are outlined in the articles listed above. These two aspects are the direct focus of several survey articles. We will mention only a few of them.

First of all, Mohamed and Al-Jaroodi in [23] classify middleware types according to lines that are very similar to those mentioned above, namely, virtual machine, database, application-driven, and message-oriented. They list and discuss several aspects that still challenge effective deployment and use of middleware for WSNs (and of WSNs in general). These are, namely, (1) scarcity of hardware resources, (2) dynamic changes of network topology and size, (3) heterogeneity, (4) network lifetime, (5) application dependency, (6) security, (7) quality of service, and (8) integration with the broader Internet context.

From this list, they derive several key requirements that should be satisfied by the middleware, e.g.:

1. run-time support for service registration, discovery, and use. This enables dynamic adaptation to changes both of the network topology and of the applications themselves.
2. service transparency to client applications, in particular to hide the heterogeneity of the underlying network.
3. configurability to support a variety of QoS, security, and resource consumption requirements.
4. support for self-organization, in the presence of dynamic network changes due to mobility, addition and retirement of nodes, and so on.
5. interoperability with a variety of underlying devices and network protocols.
6. efficient handling of huge volumes of data.
7. support for security, QoS requirement management, and interoperability with other systems.

The article concludes by classifying and evaluating 15 middleware approaches according to these requirements and by discussing the opportunities for future work.

Along similar lines, Mottola and Picco in [25] provide an outlook into WSN middleware research. One very interesting comment in this article is that most WSN work, including almost all the approaches mentioned in the surveys above, conspicuously ignores the ZigBee industrial standard that specifies how applications can access the network stack and that is supported by several commercial node platforms. While this can be explained by the difficulty to tune and exploit a closed platform, at least the compatibility with its recommendations should be taken into account.

They also analyze the state of the art, including their own TeenyLIME environment [7], and outline open research challenges, such as:

1. supporting one-to-many and many-to-many abstractions, as well as mobility.
2. providing high-level abstractions for application developers, who are often domain experts, rather than electronics or telecommunications engineers, without forgetting network deployers and maintainers.
3. the need for flexibility and expressive power without losing efficiency.
4. support for cross-layer optimizations and interactions within the network stack, which is essential for simultaneous energy and performance optimization, and is seen as a key differentiator between WSNs and telecommunication networks.
5. the need to permit reliable and predictable implementations, since WSNs are embedded systems, which often implement safety-critical applications.
6. support for multiple, concurrent applications, sometimes with very different constraints. These may even have dynamic after-deployment installation and update requirements.
7. integration within broader systems, including of course the Internet, which would require a chapter in itself, especially for its industrial and transportation application areas.

Finally, they again stress the need for all research on WSNs (including the middleware) to be concretely demonstrated in real-world scenarios, not just with simulation results.

Dong et al. [9] provide a good summary of challenges for WSN OSs. The requirements that they pose are similar to those discussed for middleware but are at a lower level: small footprint, energy efficiency, reliability, real-time guarantees, reconfigurability, and programming convenience. First of all, they describe the main components of an OS for WSNs:

1. Task scheduling, which may be event-driven as in TinyOS or thread-based as in Mantis OS. As mentioned above, the former is more efficient, while the latter is more familiar to programmers.
2. Dynamic linking and loading, which adds a lot of flexibility to the network but has a cost in terms of complexity and overhead.
3. Memory management, in particular support for permanent storage, such as flash memory, and for dynamic memory allocation, which may be a problem in resource-constrained nodes.
4. Resource abstraction, to hide details of the underlying hardware and, in some cases, virtualize its access.
5. Sensor interfaces, which provide similar abstraction and virtualization capabilities for the more WSN-specific aspects of the hardware platform.
6. Networking stack, which is an essential part by definition of any WSN OS and may provide higher-level services that cross into the middleware domain.

Then the authors describe and compare several notable examples of WSN OSs, such as TinyOS [17], Contiki [10], SOS [16], Mantis OS [1], Nano-RK [11], RETOS [5], and LiteOS [4].

The classification is based on various aspects, such as (1) static or dynamic resource allocation, (2) event-driven versus multi-threaded scheduling, (3) monolithic or modular architecture, (4) networking support, (5) real-time support, (6), language support, (7) file system support, (8) reprogramming, and (9) remote debugging.

The last part of the article evaluates each approach with respect to the requirements that were defined at the beginning, and provides several recommendations to researchers interested in this domain, which range from keeping the design simple and flexible to considering hardware requirements, application needs, and development costs.

38.2.3 Model-Driven Design

Finally, we will mention three articles that are more specific to the topic of this chapter, namely, model-driven and component-based design of WSN applications.

Shimizu et al. [32] describe a model-driven methodology and tool to speed up design and optimization of WSN applications. Different from the approach described later in this chapter, which focuses only on the Model-Based Design (MBD) of the node code, they define three different Domain-Specific Modeling Languages (DSMLs), respectively, for the network level, the group level, and the node level. Each DSML essentially offers a set of choices for key design parameters at the corresponding layer:

- The DSML for the network considers (1) data source nodes, (2) aggregation and fusion nodes, and (3) sink nodes. At this level, designers can choose how often sensors are sampled and how often data are transmitted toward the sink by each class of nodes.
- The DSML for the group (neighborhood) is similar, but at this level, designers can also choose (1) the network topology (e.g., tree or mesh), (2) the amount of in-network processing (aggregation and fusion), as well as (3) the geographical grouping.
- The DSML for the node considers (1) sampling tasks, (2) aggregation and fusion tasks, (3) sending and receiving tasks, and (4) sink tasks. Here, the designers can make choices on every aspect covered by the approach; thus, they have full customization capabilities.

The use of three DSMLs allows teams with different areas of expertise to hierarchically design and manage a large network, while retaining full control over the result. Automated code generation for simulation completes the flow.

While this approach exploits nicely the advantages of MBD, it is not clear how the user can define an application which cannot be generated simply by choosing appropriate values for the model parameters. In other words, it basically offers a single, albeit very parameterized, WSN "application," which can be customized to cover a broad range of requirements but is not (and most likely can never be) fully general.

In Sect. 38.3, we will present a design framework that is focused on modeling the code of the application tasks themselves and on smartly linking the tasks together at node level.

Taherkordi et al. [35] describe REMORA, a component-based model that is much more advanced than the basic static composition mechanism supported in TinyOS. For example, it includes the ability to dynamically deploy and connect components. Components and their interfaces are described in REMORA using an Extensible Markup Language (XML) format that covers (1) offered and required services that are activated through events, (2) the state that is retained by each component across invocations, and (3) the component (in a C-like language).

The event modeling mechanism in XML is more flexible than its TinyOS counterpart, allowing one to (1) specify event attributes, (2) distinguish between application events and OS events, (3) configure events, and (4) define if they are point-to-point or multicast.

The framework has a very low overhead with respect to Contiki, while providing significantly better encapsulation capabilities, and thus designer productivity, than bare bones multi-threading.

Finally, Compton et al. [6] survey semantics specifications for WSNs, i.e., the ontologies that can be used to describe the requirements of a network and allow a compositional design approach. This is very relevant for our methodology, which is based on an ontology to implement the component search, constrained composition, and parameter value selection capabilities.

The authors describe the capabilities of semantic sensor networks, including the ability to:

1. classify sensors according to functionality, type of output, or method of measurement.
2. find sensors than can perform some measurement.
3. collect data based on various criteria (spatial, temporal, . . .).
4. perform domain-specific inferences on low-level data.
5. react to specific inferred or measured events.

The article then lists 12 different ontologies, both general-purpose and application-specific (e.g., for marine sensors), and compares them in terms of the aspects of a WSN that each of them can describe. These aspects consider:

1. the logical aspects of each node and of the network as a whole, in terms of hierarchy, node identity, node software, deployment, configurations, history, and kind of processes it can support.
2. the physical aspects of each node, such as location, power supply, node platform, physical dimensions, and operating conditions.
3. the observations that each node can make, in terms of accuracy, frequency, response mechanism (periodic or event-triggered), and field of sensing.
4. the sensing domain, considering the measurement units, the features that are measured, and the time.

Finally, the authors summarize how ontologies are supported by various reasoning mechanisms. Later in this chapter, we will discuss an approach where ontology use is extended to describe both the functional and nonfunctional elements that compose WSN nodes in order to allow the automatic synthesis of both node hardware and software needed to support the application functions.

38.3 Model-Based WSN Application Design

The need to improve important metrics of WSN application development such as cost, time to market, lifetime and reliability, as well as its accessibility to domain application experts, can be satisfied using high-level design flows that support some degree of automation.

In Sect. 38.2, we reviewed several models, tools, and techniques that have been proposed in this regard. Although a significant variety of tools was proposed, no single comprehensive approach has emerged.

In this chapter, faithful to the principles of hardware/software codesign that are discussed in the entire book, we present an MBD framework that can speed up and facilitate application development, design space exploration, simulation (including hardware in the loop), and fast prototyping of distributed WSN applications. The framework is based on tools widely used in industry like MATLAB Simulink® [21] and Stateflow® [22]. In addition, the architecture of the application is described using the standard Web Service Definition Language (WSDL).

38.3.1 Development Flow Overview

In the approach presented here, the Simulink® and Stateflow® graphical design tools are used for design entry using high-level abstract concurrent models, which simplify the design, simulation, and prototyping phases.

The abstract model can be automatically translated to simulation models that can be used on widely used network simulators such as OMNeT++/MiXiM [36]. The same model can also be translated for direct implementation on embedded operating systems, like TinyOS and Contiki, for hardware-in-the-loop simulation and deployment.

The framework shown in Fig. 38.2 provides support for target application design using high-level abstract models, without requiring knowledge of the low-level

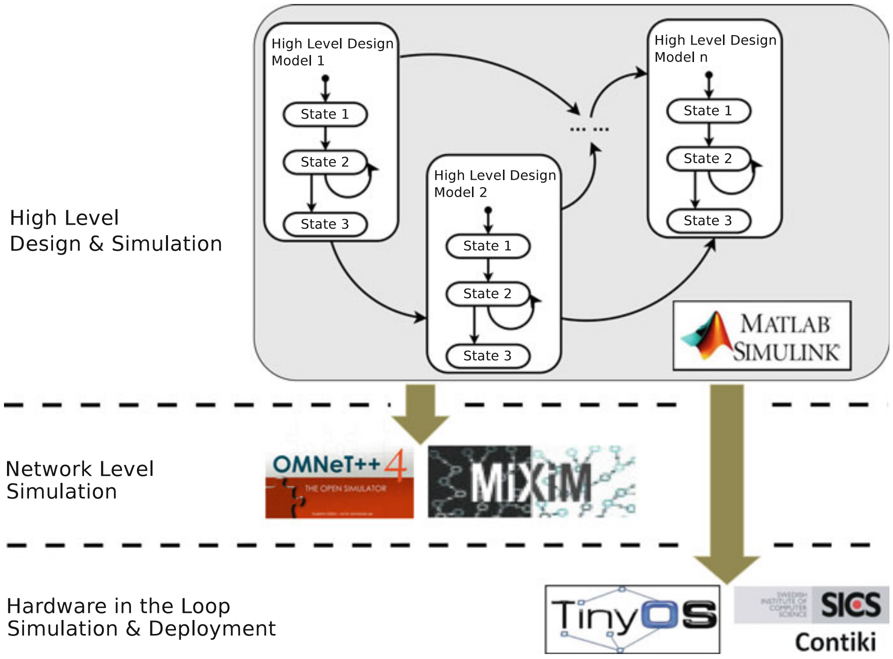
High Level Design Model 1
State 1
State 2
State 3
High Level Design Model 2
State 1
State 2
State 3
High Level Design Model n
State 1
State 2
State 3
... ...
MATLAB SIMULINK
High Level Design & Simulation
Network Level Simulation
OMNeT++ 4
MiXiM
Hardware in the Loop Simulation & Deployment
TinyOS
SICS
Contiki

Fig. 38.2 Overview of the development flow based on Simulink® framework

specifications of the underlying hardware and software platforms or communication protocol stacks. It also allows one to automatically reuse the code generated from the same model for different simulation environments and deployment platforms.

Within the framework, the target application is first decomposed into a set of interconnected high-level object-oriented abstract models. These can exchange messages in a service-driven fashion. The model internal logic is described intuitively using a visual programming language based on Stateflow® StateCharts [20] and Simulink® block diagrams, as shown in Fig. 38.2.

38.3.2 Component Structure

The framework allows the designer to define the structure and the behavior of the WSN application by means of self-contained high-level abstract functional modules (Simulink® blocks) like the one shown in Fig. 38.3. Each module is seen as a "black box" by the other modules, being externally characterized by its tunable attributes and by the used and provided services.

Services used by the module are imported through its inbound service ports, such as InServ_1, …, InServ_Y in Fig. 38.3. Services provided are exported through the outbound service ports of the module, such as OutServ_1, …, OutServ_Z. A module can use also bidirectional service ports (e.g., InOutServ_1, …, InOutServ_X) to connect an inbound service port and an outbound one to represent an instance of a combined service. This service is both used by the module and requires a response by its provider. Each service instance of a module (used, provided, or combined) is associated to an interface, which defines the contents of the messages transmitted by that service.

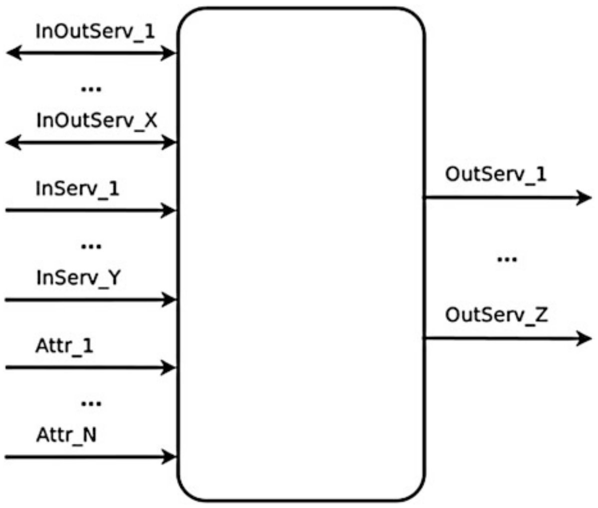

Fig. 38.3 Self-contained high-level abstract functional module (Simulink® block)

A module can also expose tunable attributes (e.g., Attr_1, ..., Attr_N in Fig. 38.3) which are meant to allow the developer to adjust the performance of the module without significant changes to its internal logic. Like the services, attributes are associated to an interface that defines their constituents.

The WSN applications running on nodes can be modeled as an interconnected set of modules. Therefore, the internal details of the modules do not depend on the external entities and they can be loosely interconnected. Each Abstract Design Module (ADM) carries out part of the functions of the target application by exchanging service messages with the connected modules, through its service ports. An outbound service port of a module can be connected to any inbound service port of another module, as long as they share the same interface type. Inbound and outbound service ports can remain disconnected, which means that no incoming service messages are imported by the floating inbound service port and that the outgoing service messages on the floating outbound service port will be discarded (of course the designer must make sure that these missing connections do not impair the overall application functionality). These are similar to unidirectional function calls and can be used to transmit service-specific messages between modules without exposing the internal implementation details.

Module behavior is represented using an event-driven hierarchical Finite-State Machine (FSM) in the form of a StateChart, as shown in Fig. 38.4. The logic flow, i.e., the change of the active state, is determined by either its internal default transitions or by external service messages imported from other modules. These are processed by the FSM based on the values of its tunable attributes, and the computation results are attached to the appropriate outgoing service messages which are sent out through the corresponding outbound service ports.

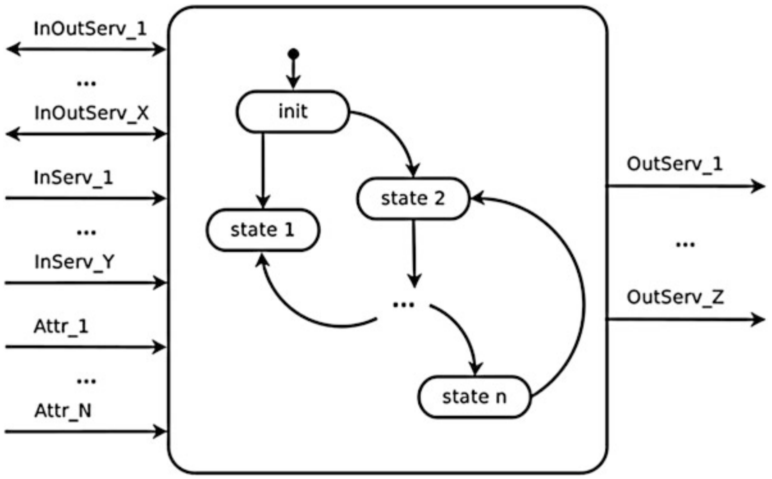

Fig. 38.4 A StateChart implements a finite state machine which defines the behavior of a functional module

User-defined operations can be attached to each state or to state transitions. They will be executed upon the entry, permanence, or the exit phases of each state.

The module has a different representation in the various steps of the workflow, serving different purposes. For instance, it can be viewed as a native Simulink®/ Stateflow® block for modeling and single-node functional simulation, as an OMNeT++/MiXiM module for large network simulations, or as a TinyOS component or Contiki OS process for deployment on the target WSN node.

38.3.3 Design Flow

The workflow illustrates the basic operation of the framework. It uses an iterative V-shape flow that starts with the requirement analysis, as shown in Fig. 38.5. It is made of three development task types:

- Manual tasks include development activities that are not directly supported by the framework and must be manually performed by the designers using other development tools.
- Supported tasks include some activities performed by the designers, with direct tool support from the framework.
- Automatic tasks are fully supported and performed by tools in the framework.

In the following, we describe the steps of the workflow in more detail.

38.3.3.1 Requirement Analysis

Requirement analysis is the first step in the workflow. It is a manual task and it consists in the analysis of the requirements of the target WSN application. It includes a list of the required functions and attributes supported by the application, such as:

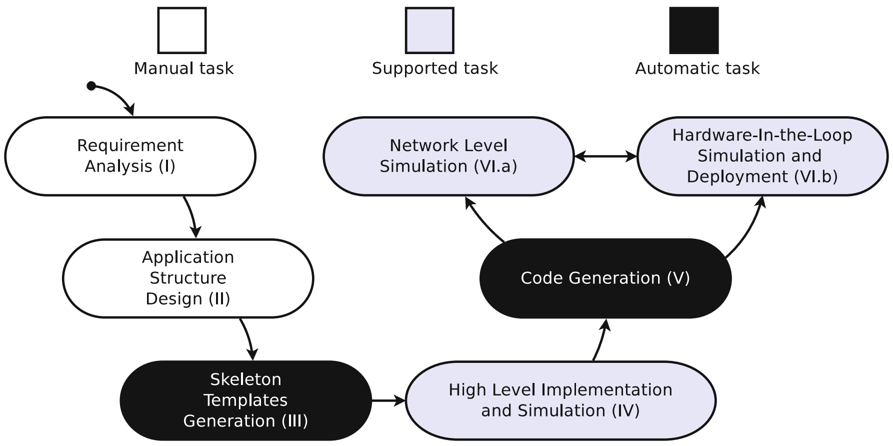
Manual task
Supported task
Automatic task
Requirement Analysis (I)
Application Structure Design (II)
Skeleton Templates Generation (III)
High Level Implementation and Simulation (IV)
Code Generation (V)
Network Level Simulation (VI.a)
Hardware-In-the-Loop Simulation and Deployment (VI.b)

Fig. 38.5 Framework development flow is based on a V-shape iterative model

- what measurements will be performed by the node;
- what operations are expected from the nodes;
- what state variables (attributes) will be exposed as the tunable attributes by the application;
- what kind of criteria will be employed to validate the application and evaluate its performance.

When these requirements are defined, the designer can move to the next step, to describe the modules.

38.3.3.2 Module Description

Module description is based on the results of previous analysis. The developer can decompose the target application into a set of interconnected modules, each implementing a part of the target application functions by exchanging service messages with other connected modules through its service ports.

In this step, the developer lists the services and the attributes that are included in each module or includes them from a library of preexisting descriptions (e.g., defined in previous designs). For each module, the developer defines a description file with all services and tunable attributes of the module, such as the service name, interface, and type of associated service ports.

For instance, for an application that can be decomposed into a set of modules as shown in Fig. 38.6, the developer will define the content of the service messages exchanged among the modules and obtains a model description file for each module. Once defined, the description files are provided to the framework to automatically generate the skeleton templates for the modules in the next step.

38.3.3.3 Generation of an Application Skeleton

The skeleton template is defined in terms of Stateflow® blocks. In the generated skeleton template, all the services and attributes defined by the developer for that module in the previous step will be interpreted as a port. A combined service will be mapped to a pair of input and output ports in the skeleton template. For instance, the module shown in Fig. 38.4 is instantiated as shown in Fig. 38.6.

Each inbound service will be mapped to an input port associated with the specified data type defined by its interface (e.g., InServ_1 in Fig. 38.3 to InServ_1 in Fig. 38.7), each outbound service is mapped to an output port (e.g., OutServ_1 in Fig. 38.3 to OutServ_1 in Fig. 38.7), and a combined service is mapped to a pair of input and output ports (e.g., InOutServ_1 in Fig. 38.3 to InOutServ_1_IN and InOutServ_1_OUT in Fig. 38.7). For each input and output port, a corresponding driver function is automatically generated, such as those shown in Fig. 38.7:

- driver_InServ_1 for InServ_1
- driver_OutServ_1 for OutServ_1
- driver_InOutServ_1_IN and driver_InOutServ_1_OUT for InOutServ_1

These driver functions are used to detect the incoming service messages and to send out the outgoing service messages for the input ports and output ports,

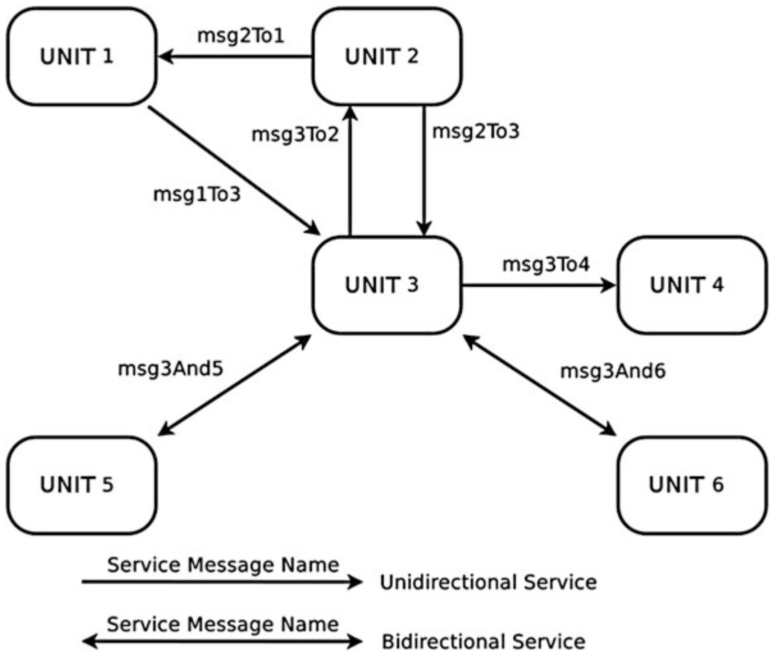

Fig. 38.6 Example of WSN node application decomposition in functional modules

respectively. Similar to the inbound services, for each tunable attribute, the framework will generate an input port and a port state variable (e.g., Attr_1 and var_Attr_1 in Fig. 38.7 for Attr_1 in Fig. 38.3), through which the developer can externally set the desired value for that attribute.

38.3.3.4 Customization of the Application Skeleton

The skeleton template can be customized through a supported task that includes two types of activities in the Simulink®/Stateflow® environment, namely, skeleton template completion, to create a full module, and module composition.

Skeleton template completion is done by the developers by modifying the automatically generated skeleton template with the desired internal functions. This consists in processing the imported service messages based on the values of the exposed tunable attributes and generating the corresponding outgoing service messages.

The internal logic of a module is defined by the developer using StateCharts and block diagrams, without knowing the details of the target platform (WSN node).

As shown in Fig. 38.7, the developer-defined operations implemented within the states are executed upon entry, permanence, or exit phases of each state, while the developer-defined operations implemented between two connected states are executed when the state transfer occurs through that state connection. All these developer-defined operations can execute any developer-defined local functions

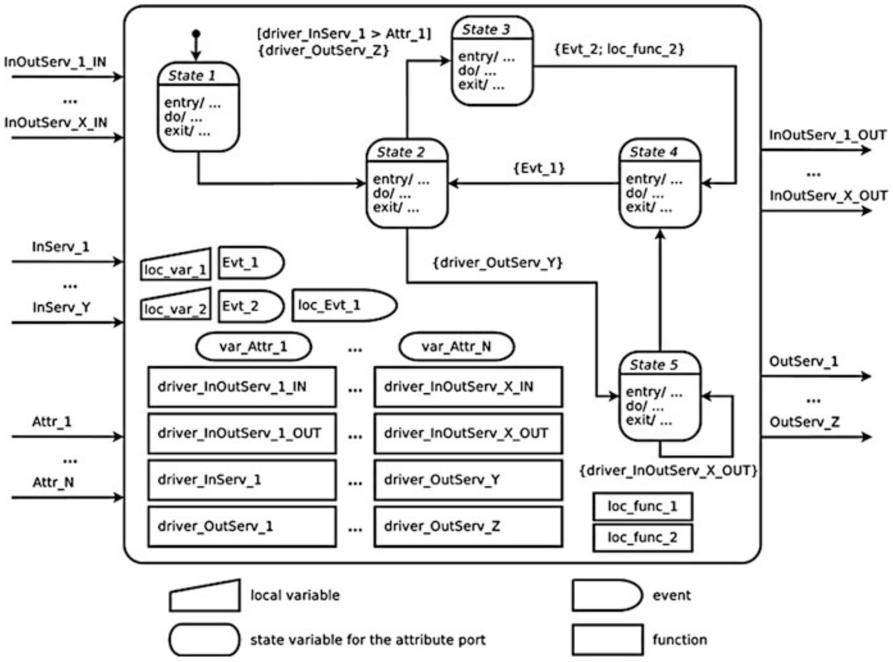

Fig. 38.7 Example of complete WSN node application skeleton

(e.g., loc_func_1 and loc_func_2 in Fig. 38.7) to perform computational tasks, as well as generate outgoing service messages. Besides the externally tunable attributes, additional local variables (e.g., loc_var_1 and loc_var_2 in Fig. 38.7) and local events (e.g., loc_Evt_1 in Fig. 38.7) can be freely defined within each module.

If a single FSM is not sufficient to model the desired function, a single Simulink® block can contain one or more sub-charts which can be integrated into the main FSM in either sequential or parallel execution order, sharing the set of incoming and outgoing messages, attributes, local variables, and local events.

The module composition phase creates the final application. This is done by wiring each outgoing port of each module to the relevant incoming port(s) of other modules and by assigning proper values to the attribute port(s) of each module.

Then the high-level application model can be used in the next step to automatically generate an implementation for different platforms.

38.3.3.5 Code Generation and Deployment

Since the framework helps the developers to automatically port the same high-level design to different platforms, the developers can easily refine their design and explore the hardware/software trade-off space almost without low-level detail knowledge. This is an important benefit, since design porting to a different platform is often effort-intensive and error-prone.

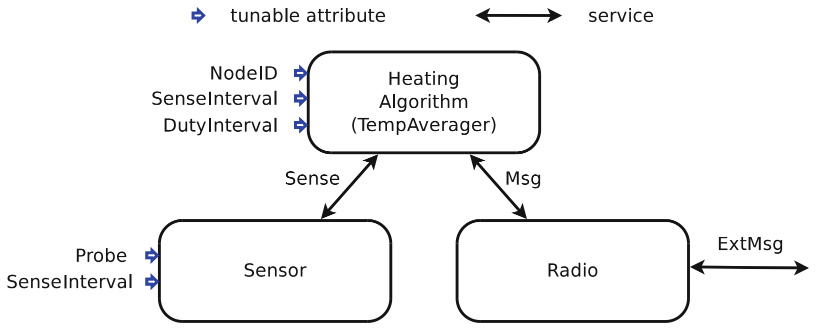

Fig. 38.8 Module structure of example node-level WSN application

We will analyze a use case in which the nodes collect and perform a distributed processing of the data from a temperature sensor. Each WSN node wakes up periodically to carry out the following tasks:

1. sample the temperature values with the desired sampling frequency;
2. collaboratively average these values with those from its one-hop neighbors within a sliding time window;
3. broadcast the calculated average temperature value to the neighbors.

The requirement analysis is used to drive the application architectural design phase, where the developer decomposes the application into a set of interconnected ADMs, each carrying out a part of the entire functionality. Since the ADMs are characterized by services and attributes exposed on their boundary, their internal behavior may not be detailed in this step. The developer just assigns the requirements listed earlier to the constituent ADMs by defining their boundaries. This can be done describing the ADM services and attributes either manually or by importing them from a library (e.g., created in previous designs).

In the proposed use case, the following parameters have been identified as potential tunable attributes for each node:

1. its own node identifier (NodeID);
2. sample refresh interval inside the averaging algorithm (SenseInterval);
3. the sampling period of the temperature sensor (SamplingInterval);
4. the size of the time window to compute averages, which is equal to the node duty interval (DutyInterval).

Based on requirement analysis (first step in Fig. 38.5), the design is split in three interconnected ADMs: a `Sensor` module, a `Radio` module, and an `Algorithm` (`TempAverager`) module, as shown in Fig. 38.8. The Sensor module samples and preprocesses temperature data. The Radio module interfaces with the protocol stack for short-range communication with neighbor nodes. The TempAverager module

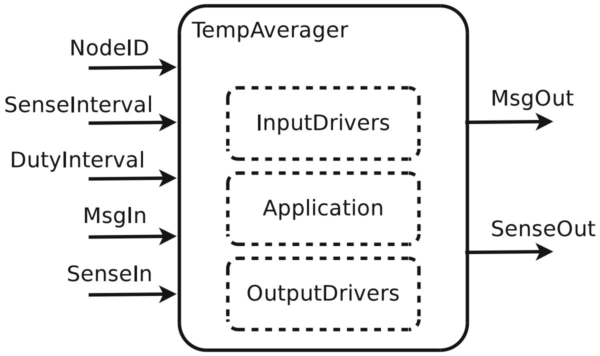

Fig. 38.9 Overview of the generated skeleton template

handles all onboard data processing. Both the Sensor and the Radio modules are connected to the TempAverager module to exchange service messages (e.g., `Sense` and `Msg` in Fig. 38.8).

The developer manually defines a boundary description file for each ADM. These are imported in the framework for the next step, template generation.

ADM description files are then supplied to the framework which maps them automatically to skeleton templates that are defined as Stateflow® blocks. All ADM services and attributes defined in the ADM templates are interpreted as ports or port pairs for a request-response service.

For each port, a driver function is automatically created inside the skeleton template, which hides the low-level Simulink® handling of signals and service messages. These functions handle the service messages exchanges through ports. Like the input services, each tunable attribute has an input port and a state variable allowing to set the attribute value from outside the ADM.

Figure 38.9 shows the skeleton template generated automatically for Stateflow®. Each skeleton template is created with three FSMs that run in parallel: `InputDrivers`, `Application`, and `OutputDrivers` which can be used to implement the application logic.

Once the skeleton template has been filled with functional details, simulated, and debugged, it can be used for automatic code generation. A framework tool converts the high-level and platform-independent design into target code that runs in different network simulation environments or on different target OSs and platforms. These can be:

- Simulink®/Stateflow® can be used for node-level and small-scale network simulation;
- OMNeT++/MiXiM can be used for large-scale network simulation. Each ADM in the WSN application is mapped to a component, which is the programming unit used by all these simulators;

Table 38.1 Code size and the memory usage for the use case application implemented on top of TinyOS using a Telos B node

	ROM [bytes]	RAM [bytes]
Handwritten	17,220	492
Framework-generated	20,562	526

- TinyOS [17] can be used for code deployment on target nodes. Each ADM of the application is automatically converted to a TinyOS module written in nesC [13] containing the ADM internal logic;
- Contiki OS [10] can also be used for deployment. In that case, each ADM is instantiated as a protothread. The generated code can also be run in the COOJA simulator.

For instance, we used the framework code generation function to convert the high-level design ADMs to nesC modules that are suitable for a simple network composed of Memsic Telos rev. B nodes running TinyOS. The generated nesC modules (`Radio`, `Sensor`, and `TempAverager`) are configured, interconnected, and encapsulated in a wrapper nesC module that is then wired in TinyOS to use the existing radio communication services.

Table 38.1 shows the code size and memory usage measured for the binary code generated using the development framework and the same application logic implemented manually. The results show a penalty for the generated code of less than 20% in terms of code size and less than 7% in terms of data Random-Access Memory (RAM) requirements.

38.4 Automated WSN Application Composition

The MBD tool presented in Sect. 38.3 requires the designer to explicitly compose the modules to implement a full application, which is an effort-intensive process.

A different set of tools can automate the application composition phase starting from a high-level application specification and an existing library of reusable modules. The toolset can further speed up WSN application development and the exploration of the design space, as will be discussed next.

38.4.1 Development Flow Using Automated Application Composition

Figure 38.10 compares the automated design flow (shown in the lower part) with a typical node-level WSN application development flow (shown in the upper part). The automated flow accepts a high-level application-centric system description at node level and can be integrated with various external tools, each of them used to assist the developer in specific tasks.

38.4.1.1 Development Flow Overview

The automated flow [2] starts with the input of the application-specific behavior encapsulated in a component format (described later in Sect. 38.4.1.4).

The top-level component and all library components have the same format, with two major sections: a code section and a metadata section. In the first step of the flow in Fig. 38.10, the designer fills both of them for the top-level component, as follows.

The code section can store different types of code (behavioral, simulation models, etc.). These are always considered as (possibly parameterized) black boxes by the system synthesis engine; thus, there are no restrictions on the coding language or the representation format (which can be also binary code for one or more

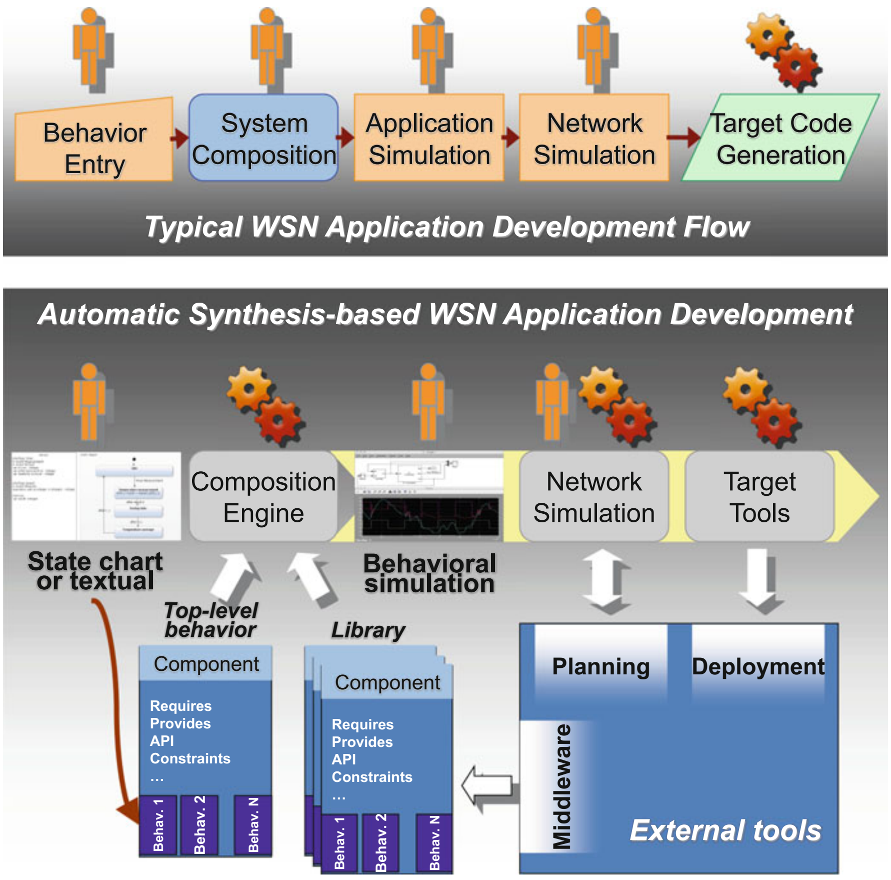

Fig. 38.10 Comparison of the main stages of manual (*top*) and automated (*lower part*) node-level WSN application design flows. A human body tags manual phases while a gear tags automatic ones. The automated flow accelerates mainly the system composition and the preparation of the network simulation

target platforms). Hence, the behavioral code of the component can come from various sources, ranging from manually written source code (e.g., legacy C or nesC code) to code generated by high-level development flows (e.g., metaprogramming approaches [24], Unified Modeling Language (UML)-based or ad hoc high-level modeling flows [8, 29, 32], ▶ Chap. 5, "Modeling Hardware/Software Embedded Systems with UML/MARTE: A Single-Source Design Approach"). MBD tools can also generate suitable behavioral code, for example, the Stateflow® tools used in Sect. 38.3 or the Yakindu Statechart Tools [26] (both provide a state chart-based integrated modeling environment for the specification and development of event-driven systems).

The metadata section of the components is used by the subsequent phase of the flow in Fig. 38.10, namely, the automated system composition. This phase uses only the semantics of the metadata to automatically select the components (more precisely, through parameter-based selection and customization) to compose the node system, both as hardware and software. If the designer uses the flow described in Sect. 38.3, then the component metadata are generated automatically from the module description. Otherwise, the designer should manually enter the metadata.

Figure 38.11 shows one possible integration of the automatic flow with external WSN development tools. The flow shown on the left side of the figure supports WSN network planning using specific tools for input of geographical data (e.g., of a topographic map), selection of node locations in the field, and Radio Frequency (RF) propagation simulation to estimate node connectivity.

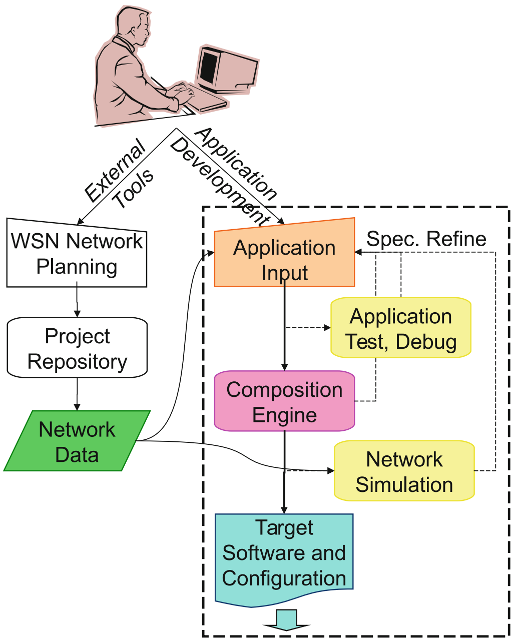

Fig. 38.11 The main phases of the application development flow based on automated system composition and one possible interaction with external tools. The application developer describes both the network layout and composition using external tools (flow on the *left*) as well as the node-level application (flow on the *right*). Where necessary, the application development flow can extract from network description the distinct types of nodes and the network connectivity information. The former is used to create projects for node-level application development, while the latter is needed to prepare the network simulations

The application development flow shown on the right side of Fig. 38.11 can retrieve the network planning data from the project repository. It uses it to extract the number of distinct node types in the network and to create a skeleton project for node-level application development for each type.

For each project, the developer inputs the application in the format of a top-level component, as we mentioned earlier. When the developers use model-based design flows for application input, such as Stateflow® or Yakindu SCT, they can use the features of these design environments to test and refine the application at this high level of abstraction.

The next step, system composition, is fully automated by the composition engine. The engine starts by processing all metadata of the top-level component, such as requires, provides, and conflicts. These encode what is needed by the component in order to operate properly in terms of hardware, interfaces, configurations, etc. (requires), what the component can provide to satisfy the requires of other components (provides), and in which conditions the component cannot operate at all (conflicts).

These properties drive the composition process, which iteratively looks for all subsets of the library of components that do not have any unsatisfied requirements left and, at the same time, satisfy all constraints imposed by the top-level and the other selected components. Each such subset represents a possible system solution that satisfies application specifications. These solutions are automatically saved and can be further examined or manually modified by the developer or used as they are.

For each generated solution, the composition tool can create simulation projects, as shown in the next steps of the flow. The simulations are set up to run on external simulators (e.g., OMNeT++ [36]) and can be at various levels of abstraction. Basically, this consists of the extraction and configuration of the suitable simulation views from the components of the solution and their assembly in simulation projects.

Using a similar mechanism, the composition engine generates the projects that can be compiled with the target tools to create the programs for the WSN nodes. These projects are typically generated in the format expected by the target tools, which often is a make-based project.

Moreover, the components that are instantiated in the solution can include a bill of materials (e.g., compatible hardware nodes, RF and transducer characteristics) or software dependencies on specific compilation toolchains or underlying OS. The composition engine can collect all these, e.g., into a solution-specific Bill of Materials (BOM) and compilation requirements.

As shown in Fig. 38.11, after each step, the developer can analyze the results and attempt to optimize them either by changing the specification (and rerunning the composition) or by manually editing the generated projects.

As mentioned, the benefits of WSN application automated composition are compounded by its integration with external tools, such as simulators, target compilation chains that can provide inputs or assist the developer in other phases of the flow. For instance, Fig. 38.10 shows some typical interfaces with middleware [23, 25], WSN planning tools [30], or deployment and maintenance tools [18]. An example of integration is presented in [2].

However, the wide variety of the existing tools and models makes it very difficult to define an exhaustive set of toolset external interfaces. Moreover, rigid toolset interfaces or operation models can reduce its value and hamper its adoption in the rapidly evolving WSN context, which does not seem to be slowed down by standardization efforts or proprietary Application Programming Interface (API) proposals. Thus, as we will show later on, an optimal tool integration in existing and future development flows would base its core operation on a model expressive enough to encode both high-level abstractions and 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.

38.4.1.2 Automated Composition Tool Overview

The main functions of the tool are application input (interface and processing), automated hardware-software composition, and code and configuration generation.

Application domain experts can benefit most from an interactive user-friendly interface for the description of the WSN application top-level behavior. Stateflow®, as described in Sect. 38.3, are well established in this regard for their intuitive use, and they can also provide suitable high-level models to facilitate the description of the desired application domain behavior. On the other hand, the tool can accept application descriptions generated by other tools, such as middleware [12] or metaprogramming [24].

Automated composition of hardware-software systems able to support WSN application specification shields the developer from most time-consuming and error-prone implementation details. At the same time, the composition increases the reuse of functional components from the library, which can be 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).

While the tool performs some consistency and satisfiability checks of application specifications in order to reject early those that cannot have a solution, other incomplete specifications are accepted because the tool can typically infer default parameters based on the values provided by the library components and heuristics. This allows the developer to refine the specifications during successive design iterations using also the results of previous underspecified composition runs.

Incomplete specifications may lead to the composition of incomplete systems, which nevertheless satisfy every requirement. This can save effort for experienced developers, who can use the resulting incomplete projects as starting points for manual refinements.

Code generation can produce simulation or target compilation projects. Network simulations can be configured using the simulation models of the components of the solutions, their parameters, and the actual configurations. Realistic communication channels defined by a planning tool [30] can be used, if available. In a similar way, the tool uses the implementation code of the components instantiated in a solution to generate and configure the project that compiles the code for the WSN nodes.

Besides this highly automated process, the tool allows the experienced developers to take over manually the application development at any stage: design entry, testing and debug, system composition, node application simulation, network simulation, and 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 development flow;
- including well-known tools in the flow with clean and well-documented interfaces to simplify their update or replacement for the specialization 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.

38.4.1.3 Automated Composition Tool Input Interface

As argued above, abstract concurrent Stateflow® are an intuitive and efficient high-level means to specify the top-level application behavior. Besides the behavior, the tool should support the specification of interfaces and other requirements of the behavior. These are necessary because the flow does not make any assumptions about the format, the language, or the modeling of the behavioral part.

All these data are captured in the top-level component of the design that is then used to drive the system composition 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 sends 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 it checks if five minutes have elapsed 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 the node application and communications are not application-specific; hence, the developer should not spend effort developing or interfacing with them. In this flow (see Figs. 38.10 and 38.11), these tasks are automatically handled by the composition 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 IPv6 over Low Power Wireless Personal Area Network (6LoWPAN) protocol is a specification of the WSN application, a requirement for 6LoWPAN can be added to the 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 composition to instantiate the functional components from the library that provide this communication protocol. However, the tool will instantiate only those 6LoWPAN components that satisfy other system requirements that are collected from both the top-level and other instantiated components.

38.4.1.4 Structure of Top-Level and Library Components

Library components are central to the operation of the system composition engine (see Fig. 38.12). They are used for:

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

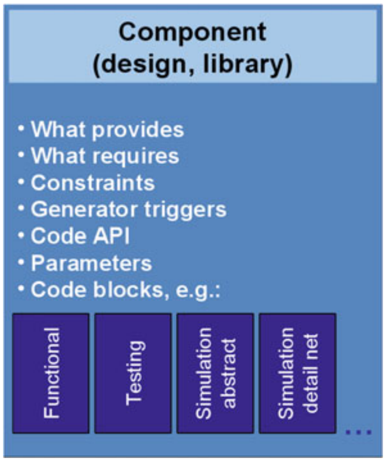

Fig. 38.12 *Top*-level application specification component and library components share the same structure: a variable set of views (shown darker on the *bottom*) that are handed as *black boxes* by the system composition process and a set of metadata that express the requirements and the capabilities of the component. The components are encoded in XML (EMF XMI)

- providing nonfunctional requirements, such as for special radio-frequency requirements or compilation toolchains.

Yakindu SCT was used in this specific case to generate and modify the library components, including their metadata. Hence, the components are encoded using extensions of Yakindu projects, which use the Eclipse Modeling-Framework (EMF) XML Metadata Interchange (XMI) format [27]. XMI is an XML interchange format well supported especially by UML-based tools. Components in other formats can be supported using suitable translators, as long as those formats can adequately represent the meanings of the metadata and the functional models of the Yakindu components.

The library components are designed to be compatible with the concurrency and communication models provided by the underlying OS or middleware abstractions. To achieve a consistent system composition, all external communications among and with the components need to go through their exposed interfaces in order to be visible to the system composition engine.

38.4.1.5 System Composition Process

To exemplify the composition process, we show in Fig. 38.13 a simplified representation of just a few metadata properties for both the library components (bottom) and the top-level specification component (top).

At the begin of the system composition process, the system composition engine is driven by the metadata specifications of the top-level component of the design, and its selections are guided by the metadata of the components in the toolset library. As system composition progresses by instantiating library components in the partial solution, the metadata of the instantiated components will drive the search performed by the engine alongside with the still unsatisfied specifications of the top-level component. During the entire composition process, the top-level component and its metadata are considered mandatory. However, the library components can be instantiated and removed from the solution as necessary, to satisfy the design requirements.

More specifically, at the begin of the system composition process, the engine loads all metadata from the top-level component and all library components. Then the recursive solver of the engine starts to build a partial solution by looking for library components that match the requirements of the top-level specification component. It instantiates these components, one at a time, into the partial solution, and then it repeats the process. This time, it considers the specifications of all the components that are currently instantiated in the partial solution, including the top-level component.

The solution becomes complete when all the requirements of its components are satisfied. Once a complete solution is found, it is saved along with the actual values for all its configuration parameters. Then the solver resumes the search for other solutions by removing components from the current solution and replacing them with alternatives, if any. The solver basically stops when all possible combinations have been tried. As a future development, the composition engine can be coupled

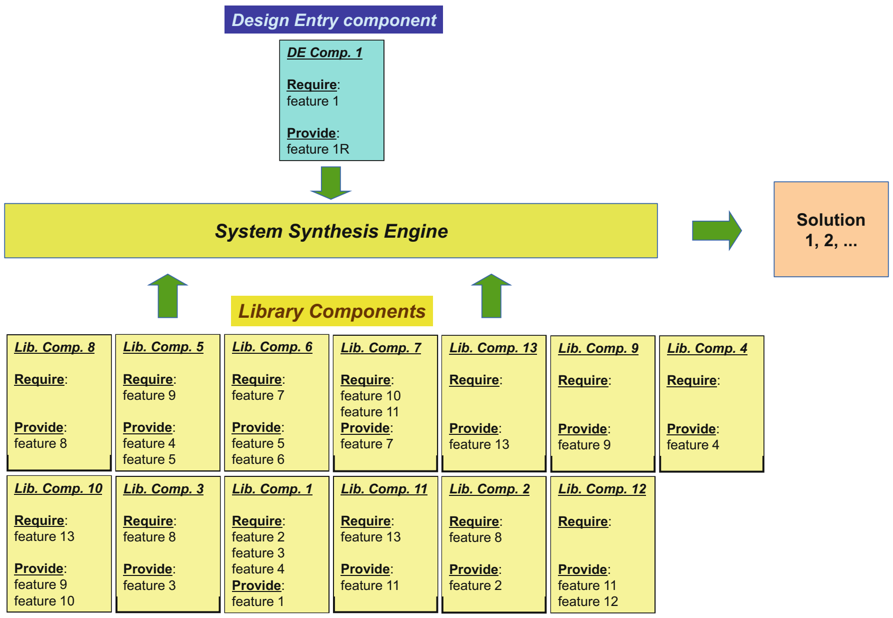

Fig. 38.13 Simplified example of metadata for the design specification component and some library components

with design space exploration tools, e.g., [28], ▸ Chap. 6, "Optimization Strategies in Design Space Exploration" and ▸ Chap. 7, "Hybrid Optimization Techniques for System-Level Design Space Exploration".

For example, considering the top-level specification and the library components shown in Fig. 38.13, the system composition engine loads first the design entry top-level component and all 13 components of the library. Then the engine explores all component combinations that can lead to a complete solution, i.e., a component composition where all mandatory component requirements are satisfied. All complete solutions found are saved. For the example shown in Fig. 38.13, these are:

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.

The format of the saved solutions includes all the elements necessary to instantiate, connect, and configure the selected library components. By revisiting these data and the instantiated components, the engine is able to:

- extract some figures of merit for the solutions from the instantiated components metadata and their actual configuration within the solution, e.g., FLASH and RAM requirements, communication protocol characteristics, etc.;

- generate the BOM and nonfunctional specifications, such as what are the compatible compilation chains;
- generate and configure network simulation models;
- generate make-based projects that can build the programming and configuration code for the target nodes.

The developers can use these data to decide which solution, if any, is suitable. Alternatively, they may decide to change the application specifications in order to improve the solutions or to manually optimize a promising solution.

38.5 Case Studies

In the following, we will present the application of the automated composition flow to two different representative WSN applications of practical interest, which are described in [2].

One application is a self-powered WSN gateway designed for long-term event-based environmental monitoring. It can handle up to 1000 sensor nodes, process, and aggregate their messages, bidirectionally communicate with a server over the Internet using Transmission Control Protocol/Internet Protocol (TCP/IP) through a General Packet Radio Service (GPRS) modem, and receive remote updates. Its hardware requirements are very small, comparable to those commonly used to implement a WSN sensor node. To satisfy these requirements, the original gateway code was handwritten fully in C, without using an embedded OS or external libraries besides the low-level standard C libraries.

The other application is a typical WSN sensor node for remote environmental monitoring. It has transducers for some pollutant gases and it is designed to operate near industrial sites adjacent to urban areas. The node was developed on top of ChibiOS [33], a real-time, preemptive, small, and fast embedded OS.

In both applications, we have started from an existing implementation. However, the flow presented below can be used both for porting legacy code on the toolset and adding toolset support for new hardware:

Create library components. The system composition engine is designed to make extensive use of the components in the library. Hence, the quality of its library strongly determines the quality of the composed systems.
A good quality library should include enough variety of building blocks to support most sensing requirements (e.g., various types of sensor interfaces), processing requirements (e.g., queues, stats, encryption), in-field and out-of-field communication protocols, etc.
The libraries can and should be reused for several designs. Thus, once a library is created, it may receive incremental updates (e.g., support for new sensors or new algorithms) or significant additions (e.g., support for new hardware nodes or embedded OSs).

Create the top-level specification component. This component is application-specific and drives the whole system synthesis process. It needs to include enough requirements to cover all application needs without being over-specified, which would restrict too much the search space of the synthesis engine.

Run the system composition engine. The engine attempts to solve all requirements of the top-level component using the existing components from the library.

Evaluate the solutions. As shown in Sect. 38.4.1.5, the toolset can extract and calculate various figures of merit for each solution which can be used by the developer in order to select a suitable solution. Moreover, the solutions can also be manually analyzed and further tuned.

The applications that we consider here are based on existing projects. In these cases, particular attention should be given to the creation of the components from the existing hardware or software Intellectual Property (IP) blocks in order to allow the toolset to find at least a solution that matches the existing projects. One obvious and easy-to-automate way is to pack the IP code in an appropriate component model (see Fig. 38.12). Then, for each component, it is important to properly describe its functional elements, such as its interfaces and configuration capabilities, and the semantics associated to component behavior and data exchanges.

38.5.1 Full-Custom WSN Gateway

The original gateway project was implemented with limited hardware resources, which are typical for WSN sensor nodes. It included an AVR ATmega1281 microcontroller, two CC1101 radio modems operating in the 433 MHz band using a proprietary communication protocol. These connected the gateway, on separate channels, both with the peer gateways and with much smaller sensor nodes, which were used for high-density environmental monitoring. The gateway included also a GPRS TCP/IP-enabled modem for long-range communication with the server and for remote updates.

The application software of the node is written entirely in C, without an embedded OS. It is made of 49 modules, each of them implementing a well-defined function: generic functions that are used by most applications (like the task scheduler, oscillator calibration, or the message queue) or specialized functions that are used for specific applications (such as drivers for specific onboard sensors).

Instead of an embedded OS, the code uses a module that implements a round-robin scheduler that can periodically run statically assigned tasks. Most tasks are implemented as FSMs using the coroutine approach. Each task executes for a minimum amount of time when started and voluntarily yields the processor whenever it completes its processing or it needs to wait for some reason. Also, each task is responsible for maintaining its own state and persistent data between calls, in order to be able to resume its execution upon its next scheduling slot.

Besides the functional blocks needed to implement the main gateway behavior, the code has several modules that implement safety and error recovery functions.

Also, there are several driver and processing modules for several sensors and auxiliary devices that can be mounted directly onboard the gateway node:

`adc` Drivers for the Analog-to-Digital Converter (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 onboard the gateway.

`crc` Cyclic Redundancy Check (CRC) utilities.
Processing utilities (CRC calculation).

`eeprom` Electrically Erasable Programmable Read-Only Memory (EEPROM) driver.
EEPROM data structure and low-level I/O drivers.

`eeprom_ext` External EEPROM driver.
Driver for external EEPROM module.

`fc10` FC10 sensor handling functions.
It has both the top-level application and drivers for the FC10 transducer.

`field` Communication protocol with sensor nodes.
Processing of messages received from sensor nodes.

`geophone` Geophone sensor driver.
It has both the top-level application and drivers for the geophone transducer.

`gw` Node status.
Controls the state and configuration of the node.

`hal` Hardware high-level interface.
It processes asynchronous events from the network and onboard switches.

`humidity` Weather humidity sensor handling functions.
Driver and controller for the humidity transducer.

`hygrometer` Hygrometer sensor.
It has both the top-level application and drivers for the transducer.

`igwc` Internode communication.
Internode messaging and network formation.

`inst` Node installation mode.
Top-level application that runs during the installation of the node in the field.

`mesh` internode communication protocol.
Processing of node-level messages.

`modem` GPRS modem driver.
Driver for the GPRS modem.

`msg_filter` Messages queue filter.
Configurable application-specific processing of the queued messages.

`obs` Onboard sensor driver.
Drivers for various onboard sensors (not application-specific).

`oc_link` Operating center communication controller.
Controller of the connection with the server and server message preprocessor.

`power` Power module driver.
Driver for the power module.

`pressure` Weather pressure sensor handling functions.
Driver and controller for the pressure transducer.

`queue` Message queue.
Storage and processing of the messages queued to be delivered to the server.

`rain` Weather rain sensor handling functions.
Driver and controller for the rain transducer.

`rccal` Main Resistor-Capacitor (RC) oscillator calibration.
Performs the calibration of the internal RC oscillator.

`rel_mesh` Multihop message queue transfer via mesh, with acknowledge.
Bidirectional internode communication protocol.

`rpc` Remote procedure call.
Processors for remotely setting and querying (monitoring) node data and for sending remote commands.

`run_state` Execution health controller.
Module to monitor the state of the current run, i.e., how far the software execution has progressed since the last boot.

`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).

`sensor_ppc` Path passage counter sensor.
It has both the top-level application and drivers for a passage detector.

`service` Internal service requests handler.
Implements a service request/dispatch controller that can change the state of the node

`sio` Serial link for operating central message transfer.
Communication with the server over a wired serial line.

`spi` Master Serial Peripheral Interface (SPI) driver.
Driver for the SPI port.

`sr` Save-restore of RAM contents across watchdog resets.
Saves the contents of specific RAM areas before a watchdog reset and restores them after the reboot.

`sw` External switch driver.
Driver for the on-node switches.

`testing` Node testbench mode.
Various top-level applications that act as node and sensor node tester.

`test_tx_hw` Sensor testbench mode.
Top-level application for the node in testing mode.

`theft` Node antitheft detector.
Process that detects possible node theft actions.

`timer` Timer handler.
Provides several timers for use within the node.

`twi` Two Wire Interface (TWI) interface.
Driver for the TWI interface.

`usart` Universal Synchronous/Asynchronous Receiver/Transmitter (USART) drivers.
Drivers for the node USART ports.

`util` Utilities.
Various processing functions (e.g., conversion of bin values to American Standard Code for Information Interchange (ASCII) hex).

`version` Firmware version utilities.
It provides the version of node software.

`wd` Watchdog driver.
Driver for various functions attached to the watchdog timer.

`weather` Weather station handlers.
Top-level application that implements a weather station.

`zlist` RAM-efficient mapping of the Identifiers (IDs) of the sensor nodes in range.
Optimized storage and processing of the messages queued to be delivered to the server.

Most of these modules are made of several functions and may include sizable amounts of data. For example, module *queue* includes the data structures for buffering the messages in queues by priority, waiting to be transmitted, and the functions for the operation of the queues (such as query, addition, or removal). Similarly, the *sensor* module maintains the data structure with the status of all sensor nodes in range and provides the functions for their management, such as query or update.

Figure 38.14 shows the metadata of the library component that was generated for a very simple module, *version*. The module implements the function to store the gateway version information and provides methods to access it.

```
<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. 38.14 Example of a simple library component that includes properties and a code view

At the top level, we can see the categories *properties*, *views*, *resources*, and *interfaces*. This simple component has only one property that contains the name of the module. The list of behavioral views includes two files corresponding to the source code of the module. The resources include one nonfunctional requirement to track the dependency of the component on a toolchain that supports the C functions used in the source code and a symbolic resource provided by the component which can be used, for instance, to directly require this component in design specifications or in other components. In terms of interfaces, the component provides a behavioral function, which retrieves and returns the version data. Additionally, for most metadata properties, one can enter a description that can be used, for instance, to help the developer understand the semantics of the component, when it is displayed in a component or solution editor.

Figure 38.15 shows the result of the composition of a minimal gateway system for which the specification was just to include the core gateway functions. Moreover, the composition tool ran the configuration helpers of the components, to set up their instances according to the actual values of their parameters, as found by the solver. For instance, the scheduler is automatically configured to support the actual tasks.

For this minimal requirement, the solver found a suitable composition with a maximum recursion depth of 888, matching 230 abstract requirements, 472 functional requirements, and two data requirements in less than 0.8 s on an 1.8 GHz Intel® Core™ i7-2677M processor.

In addition to software solution composition, the tool collects other requirements of the instantiated components into a BOM list that includes the hardware node type, radio specifications, and the target compilation toolchain.

By changing just the top-level specification component, we used the toolset to automatically compose systems for different application requirements (for different gateway compositions in this application).

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. 38.15 Result of system composition using only the requirements of the main gateway component as specification. Just 36 out of all 49 modules were included by the engine in the final project (emphasized), correctly leaving out, e.g., drivers for optional sensors, test suites, and interfaces

38.5.2 WSN Sensor Node for Air Quality Monitoring

Also for this application, we followed the flow outlined in Sect. 38.5. The existing application software of the node was developed in C for a real-time embedded OS, ChibiOS. This OS has some important features that help increasing the reliability of the applications. For instance, the APIs of the OS are designed to require minimal parameters and to do just one function, with no options and no error conditions.

Since the application was using an Real-Time Operating System (RTOS), we converted several OS modules into library components so that the system composition engine can consider them when searching for a solution to problem specifications. Besides the RTOS modules, we created library components for several application-specific elements, such as the transducer drivers and some special interfaces required by the OS:

`comm` Application layer of the node communication protocol.
`globals` Global definitions and initialization.
`hwcfg/board` Board-specific configurations, e.g., General-Purpose Input/Output-pin (GPIO) and clock setup, and peripherals check.
`crc8` Helper functions for node interface, e.g., CRC calculation.
`if` Interface functions for the node.
`transport` Transport layer for node interface.
`DHT11` Driver for the temperature and humidity sensor.
`GroveDust` Driver for the particulate matter sensor.
`GroveMQ5` Driver for the gas sensor (H2, LPG, CH4, CO, alcohol).
`GroveMQ9` Driver for the gas sensor (CO, coal gas, LPG).
`sensors` Higher-level abstraction of sensor drivers.
`thRdProbes` Periodic reader for sensor data.

Along with these components, we have created library components for an extensive set of OS modules, e.g.:

`can_lld` STM32 Controller Area Network (CAN) subsystem low-level driver source.
`ext_lld` STM32 EXT subsystem low-level driver source.
`adc_lld` STM32F4xx/STM32F2xx ADC subsystem low-level driver source.
`ext_lld_isr` STM32F4xx/STM32F2xx EXT subsystem low-level driver Interrupt Service Routine (ISR) code.
`hal_lld` STM32F4xx/STM32F2xx Hardware Abstraction Layer (HAL) subsystem low-level driver source.
`stm32_dma` Enhanced Direct Memory Access (DMA) helper driver code.
`pal_lld` STM32L1xx/STM32F2xx/STM32F4xx GPIO low-level driver code.
`i2c_lld` STM32 Inter-Integrated Circuit (I2C) subsystem low-level driver source.
`mac_lld` STM32 low-level Media Access Control (MAC) driver code.
`usb_lld` STM32 Universal Serial Bus (USB) subsystem low-level driver source.
`rtc_lld` Real-Time Clock (RTC) low-level driver.
`sdc_lld` STM32 Secure Digital Card (SDC) subsystem low-level driver source.
`spi_lld` STM32 SPI subsystem low-level driver source.
`gpt_lld` STM32 General-Purpose Timer (GPT) subsystem low-level driver source.
`icu_lld` STM32 Input Capture Unit (ICU) subsystem low-level driver header.
`pwm_lld` STM32 Pulse-Width Modulation (PWM) subsystem low-level driver header.
`serial_lld` STM32 low-level serial driver code.
`uart_lld` STM32 low-level Universal Asynchronous Receiver/Transmitter (UART) driver code.
`adc` ADC Driver code.

`can`	CAN Driver code.
`ext`	EXT Driver code.
`gpt`	GPT Driver code.
`hal`	HAL subsystem code.
`i2c`	I2C Driver code.
`icu`	ICU Driver code.
`mac`	MAC Driver code.
`mmcsd`	Multimedia/Secure Digital Card (MMC/SD) cards common code.
`mmc_spi`	MMC/SD over SPI driver code.
`pal`	Input/Output (I/O) Ports Abstraction Layer code.
`pwm`	PWM Driver code.
`rtc`	RTC Driver code.
`sdc`	SDC Driver code.
`serial`	Serial Driver code.
`serial_usb`	Serial over USB Driver code.
`spi`	SPI Driver code.
`tm`	Time Measurement driver code.
`uart`	UART Driver code.
`usb`	USB Driver code.
`chcond`	Condition Variables code.
`chdebug`	ChibiOS/RT Debug code.
`chdynamic`	Dynamic threads code.
`chevents`	Events code.
`chheap`	Heaps code.
`chlists`	Thread queues/lists code.
`chmboxes`	Mailboxes code.
`chmemcore`	Core memory manager code.
`chmempools`	Memory Pools code.
`chmsg`	Messages code.
`chmtx`	Mutexes code.
`chqueues`	I/O Queues code.
`chregistry`	Threads registry code.
`chschd`	Scheduler code.
`chsem`	Semaphores code.
`chsys`	System-related code.
`chthreads`	Threads code.
`chvt`	Time and Virtual Timers related code.
`nvic`	Cortex-Mx Nested Vectored Interrupt Controller (NVIC) support code.
`chcore`	ARM Cortex-Mx port code.
`chcore_v7m`	ARMv7-M architecture port code.
`crt0`	Generic ARMvx-M (Cortex-M0/M1/M3/M4) startup file for ChibiOS/RT.
`vectors`	Interrupt vectors for the STM32F4xx family.
`chprintf`	Mini printf-like functionality.

Figure 38.16 shows an example of a library component that was created for this project. Its structure is similar to the one shown in Fig. 38.14 in terms of dependency and interface data declarations. Besides these, the component includes a parameter definition under the *rpcs* tag. These parameters are made available by the functional code of the component to allow their remote control by a middleware layer or by a monitoring server, using specific protocols. The data associated to these parameters in the library component, which is enclosed in an *rpc* tag, is extracted by the composition tool and is attached to each solution that was generated. These data are later used by external tools for their run-time configuration to properly interface

```
<sgraph:Gss xmi:id="_c3cd36f777bdea1a5ae079">
    <properties>
        <name>comm_component</name>
        <cmt><rpcs><rpc>
        <description>Set sensor sampling frequency.</description>
        <name value="RATE" /><values><set type="integer" /></values>
        </rpc></rpcs></cmt>
    </properties>
    <views xmi:id="_190b8090159699e0b68bce">
        <behavior>
            <view xmi:id="_b860c0e4a20a75489a5f76" name="comm.c"
            type="source" language="C" encoding="base64">
                <mem>LyoNCiAqIENvbW [...]</mem></view>
            <view xmi:id="_37a76b4ed27649239a0554" name="comm.h"
            type="source" language="C" encoding="base64">
                <mem>LyoNCiAqIENvbW0uaA0KICo [...]</mem></view>
        </behavior>
    </views>
    <interfaces>
        <behavior>
            <provide><data>
                <name>m_sPacket</name><base>t_PktHeader</base><size>6</size>
            </data></provide>
            <provide><function>
                <name>Comm_Init</name><return><type>void</type></return>
                <port><ord>1</ord><type>void</type></port>
            </function></provide>
            <provide><function>
                <name>Comm_Write</name><return><type>void</type></return>
                <port><ord>1</ord><type>t_u8 *</type></port>
                <port><ord>2</ord><type>t_u8</type></port>
            </function></provide>
            <require><function>
                <name>Crc8</name><return><type>unsigned char</type></return>
                <port><ord>1</ord><type>void *</type></port>
                <port><ord>2</ord><type>int</type></port>
                <port><ord>3</ord><type>unsigned char</type></port>
            </function></require>
            <require><data>
                <name>g_Kau8Sync</name><base>t_u8</base><size>4</size>
            </data></require>
            <require><data>
                <name>g_pSApp</name><base>SerialDriver *</base><size>4</size>
            </data></require>
        </behavior>
    </interfaces>
</sgraph:Gss>
```

Fig. 38.16 Example of a library component used for the air quality monitoring application. It includes a parameter that can be remotely accessed at run time

with the component. The data can also include human readable descriptions for the developers or the beneficiaries of the WSN application.

These library components were added to the same library that was used for the first application. In this way, the synthesis engine is able to compose systems for both hardware node types by selecting suitable compatible components to match the specification requirements.

For this application, the solver found a suitable system composition with a maximum recursion depth of 109, matching 22 abstract requirements, 50 functional requirements, and 12 data requirements, in less than 0.2 s on an 1.8 GHz Intel® Core$^{\text{TM}}$ i7-2677M processor.

We used the toolset to compose nodes with different sensors, sensor combinations, sensing periods, and remote monitoring interfaces (as the one mentioned above). The synthesis engine used the high-level requirements in the top-level component (which were provided by the developer) to automatically select and compose suitable hardware, software, and configuration for the node.

38.6 Conclusion

Wireless sensor networks can be used for many applications in a variety of domains, but their reliability, lifetime, overall cost, and design effort limit their actual use. Moreover, WSN design flows often lack a well-defined separation between the application designers and the multidisciplinary engineering knowledge needed to cover the operation of the underlying technology. This considerably reduces the use of WSN solutions by the application domain experts, even though WSNs would provide very effective solutions for their applications.

We briefly overviewed some of the most important existing WSN development techniques, abstractions, and tool categories to evaluate how well they respond to these requirements. From the review, the importance of the trade-off between implementation optimization and accessibility to application domain experts became apparent. On the one hand, the development flows that allow significant design optimizations imply a level of hardware, software, and network design knowledge that is seldom found among application domain experts. On the other hand, highly abstracted design flows may often lead to poorly optimized WSN designs and are difficult to port to target platforms outside the (often) narrow range supported by the tool. Also, most of the tools themselves generally lack composability and the ability to be used as building blocks within new development flows.

Model-based design flows seem to provide effective trade-offs between the manual effort that is required to optimize the designs and the availability of a high-level development flow that increases designer productivity. In this context, we presented in more detail two innovative toolsets that offer user-friendly high-level design entry interfaces as well as various degrees of automation to hide the low-level implementation details from the developer. Both flows allow design optimization to various degrees and also manual optimization for skilled developers to increase the performance of the resulting WSN designs. To evaluate their effectiveness, we have illustrated the use of both tools for the development of some typical applications.

References

1. Abrach H, Bhatti S, Carlson J, Dai H, Rose J, Sheth A, Shucker B, Deng J, Han R (2003) 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. ACM, New York, pp 50–59. doi:10.1145/941350.941358

2. Antonopoulos C, Asimogloy K, Chiti S, D'Onofrio L, Gianfranceschi S, He D, Iodice A, Koubias S, Koulamas C, Lavagno L, Lazarescu MT, Mujica G, Papadopoulos G, Portilla J, Redondo L, Riccio D, Riesgo T, Rodriguez D, Ruello G, Samoladas V, Stoyanova T, Touliatos G, Valvo A, Vlahoy G (2016) Integrated toolset for WSN application planning, development, commissioning and maintenance: the WSN-DPCM ARTEMIS-JU project. Sensors 16(6):804. doi:10.3390/s16060804
3. Ashton K (2009) That 'Internet of Things' thing. Expert view RFID J http://www.rfidjournal.com/article/view/4986
4. Cao Q, Abdelzaher T, Stankovic J, He T (2008) The LiteOS operating system: towards Unix-like abstractions for wireless sensor networks. In: Proceedings of the 7th international conference on information processing in sensor networks, IPSN '08. IEEE Computer Society, Washington, DC, pp 233–244. doi:10.1109/IPSN.2008.54
5. Cha H, Choi S, Jung I, Kim H, Shin H, Yoo J, Yoon C (2007) RETOS: resilient, expandable, and threaded operating system for wireless sensor networks. In: Proceedings of the 6th international conference on information processing in sensor networks, IPSN '07. ACM, New York, pp 148–157. doi:10.1145/1236360.1236381
6. Compton M, Henson C, Lefort L, Neuhaus H, Sheth A (2009) A survey of the semantic specification of sensors. In: 2nd international semantic sensor networks workshop
7. Costa P, Mottola L, Murphy AL, Picco GP (2007) Programming wireless sensor networks with the TeenyLime middleware. In: Proceedings of the ACM/IFIP/USENIX 2007 international conference on middleware, middleware '07. Springer, New York, pp 429–449.
8. Doddapaneni K, Ever E, Gemikonakli O, Malavolta I, Mostarda L, Muccini H (2012) 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. IEEE Press, Piscataway, pp 1–7
9. Dong W, Chen C, Liu X, Bu J (2010) Providing OS support for wireless sensor networks: challenges and approaches. Commun Surv Tuts 12(4):519–530. doi:10.1109/SURV.2010.032610.00045
10. Dunkels A, Gronvall B, Voigt T (2004) 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. IEEE Computer Society, Washington, DC, pp 455–462. doi:10.1109/LCN.2004.38
11. Eswaran A, Rowe A, Rajkumar R (2005) Nano-RK: an energy-aware resource-centric RTOS for sensor networks. In: Proceedings of the 26th IEEE international real-time systems symposium, RTSS '05. IEEE Computer Society, Washington, DC, pp 256–265. doi:10.1109/RTSS.2005.30
12. Gámez N, Cubo J, Fuentes L, Pimentel E (2012) Configuring a context-aware middleware for wireless sensor networks. Sensors 12(7):8544–8570
13. Gay D, Levis P, von Behren R, Welsh M, Brewer E, Culler D (2003) The nesC language: a holistic approach to networked embedded systems. SIGPLAN Not 38(5):1–11. doi:10.1145/780822.781133
14. Greenstein B, Kohler E, Estrin D (2004) A sensor network application construction kit (SNACK). In: Proceedings of the 2nd international conference on embedded networked sensor systems, SenSys '04. ACM, New York, pp 69–80. doi:10.1145/1031495.1031505
15. Gummadi R, Gnawali O, Govindan R (2005) Macro-programming wireless sensor networks using Kairos. In: Proceedings of the first IEEE international conference on distributed computing in sensor systems, DCOSS'05. Springer, Berlin/Heidelberg, pp 126–140. doi:10.1007/11502593_12
16. Han CC, Kumar R, Shea R, Kohler E, Srivastava M (2005) A dynamic operating system for sensor nodes. In: Proceedings of the 3rd international conference on mobile systems, applications, and services, MobiSys '05. ACM, New York, pp 163–176. doi:10.1145/1067170.1067188
17. Hill J, Szewczyk R, Woo A, Hollar S, Culler D, Pister K (2000) System architecture directions for networked sensors. SIGARCH Comput Archit News 28(5):93–104. doi:10.1145/378995.379006

18. Lazarescu MT (2013) Design of a WSN platform for long-term environmental monitoring for IoT applications. IEEE J Emerg Sel Top Circuits Syst 3(1):45–54. doi:10.1109/JETCAS.2013.2243032
19. Madden SR, Franklin MJ, Hellerstein JM, Hong W (2005) TinyDB: an acquisitional query processing system for sensor networks. ACM Trans Database Syst 30(1):122–173. doi:10.1145/1061318.1061322
20. Mathworks (2013) Generate C and C++ code from simulink and stateflow models. The MathWorks. https://it.mathworks.com/products/simulink-coder/
21. MATLAB and Simulink Release 2010a (2010) The MathWorks, Inc., Natick, Massachusetts, United States
22. MATLAB and Stateflow Release 2010a (2010) The MathWorks, Inc., Natick, Massachusetts, United States
23. Mohamed N, Al-Jaroodi J (2011) A survey on service-oriented middleware for wireless sensor networks. Serv Oriented Comput Appl 5(2):71–85. doi:10.1007/s11761-011-0083-x
24. Mottola L, Picco GP (2011) Programming wireless sensor networks: fundamental concepts and state of the art. ACM Comput Surv 43(3):19:1–19:51. doi:10.1145/1922649.1922656
25. Mottola L, Picco GP (2012) Middleware for wireless sensor networks: an outlook. J Internet Serv Appl 3(1):31–39. doi:10.1007/s13174-011-0046-7
26. Mülder A, Nyßen A (2011) TMF meets GMF. Eclipse Mag 3:74–78. https://svn.codespot.com/a/eclipselabs.org/yakindu/media/slides/TMF_meets_GMF_FINAL.pdf
27. OMG, XML (2007) Metadata Interchange (XMI) Specification. http://www.omg.org/spec/XMI/2.1.1/PDF/index.htm. (Accessed 4 June 2016)
28. Palermo G, Silvano C, Valsecchi S, Zaccaria V (2003) A system-level methodology for fast multi-objective design space exploration. In: Proceedings of the 13th ACM great lakes symposium on VLSI, GLSVLSI '03. ACM, New York, pp 92–95. doi:10.1145/764808.764833
29. Paulon A, Fröhlich A, Becker L, Basso F (2013) Model-driven development of WSN applications. In: 2013 III Brazilian symposium on computing systems engineering (SBESC), pp 161–166. doi:10.1109/SBESC.2013.27
30. Ray A (2009) Planning and analysis tool for large scale deployment of wireless sensor network. Int J Next-Gener Netw (IJNGN) 1(1):29–36
31. Romer K, Mattern F (2004) The design space of wireless sensor networks. IEEE Wirel Commun 11(6):54–61. doi:10.1109/MWC.2004.1368897
32. Shimizu R, Tei K, Fukazawa Y, Honiden S (2011) 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. ACM, New York, pp 31–36. doi:10.1145/1988051.1988058
33. Sirio G (2013) ChibiOS/RT. http://www.chibios.org/ (Accessed 4 June 2016)
34. Sugihara R, Gupta RK (2008) Programming models for sensor networks: a survey. ACM Trans Sen Netw 4(2):8:1–8:29. doi:10.1145/1340771.1340774
35. Taherkordi A, Loiret F, Abdolrazaghi A, Rouvoy R, Le-Trung Q, Eliassen F (2010) Programming sensor networks using REMORA component model. In: Proceedings of the 6th IEEE international conference on distributed computing in sensor systems, DCOSS'10. Springer, Berlin/Heidelberg, pp 45–62. doi:10.1007/978-3-642-13651-1_4
36. Varga A, Hornig R (2008) 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. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, pp 60:1–60:10