WSN Design and Verification using On-board Executable Specifications

S. Gaglio, G. Lo Re, G. Martorella, D. Peri

In IEEE Transactions on Industrial Informatics

WSN Design and Verification Using On-board Executable Specifications

Salvatore Gaglio, *Member, IEEE,* Giuseppe Lo Re, *Senior Member, IEEE,* Gloria Martorella, and Daniele Peri

***Abstract*—The gap between informal functional specifications and the resulting implementation in the chosen programming language is notably a source of errors in embedded systems design. In this paper, we discuss a methodology and a software platform aimed at coping with this issue in programming resource-constrained wireless sensor network nodes (WSNs). Whereas the typical development model for the WSNs is based on cross compilation, the proposed approach supports high-level symbolic coding of abstract models and distributed applications, as well as their test and their execution, directly on the target hardware. As a working example, we discuss the application of our methodology to specify the functional behavior of a radio transceiver chip. The resulting executable specifications are augmented with automatically generated runtime verification code. Our approach is also compared to code development for two prominent WSN general-purpose operating systems.**

***Index Terms*—Embedded systems, resource-constrained devices, symbolic programming, system programming, system specification, wireless sensor networks.**

I. Introduction

TRANSLATING hardware and software specifications into a high-level programming language implementation is notably error prone and embedded systems make no exception [1]. In fact, even if a description of the functional behavior is provided that seems clear in natural language, it is not usually so in the programming language of choice for the mainstream development platforms [2]. Moreover, datasheets and specification documentation may leave out some information or provide details about typical operation cases regardless of the target operating environment, e.g. timing constraints or resources. The incorporation of runtime verification techniques is thus quite desirable to enrich information provided by datasheets and to ensure that the system adheres to its functional specification during execution [3].

The burden posed to the design of embedded systems becomes relevant when the target devices are cooperative units composing Wireless Sensor Networks (WSNs), which are intrinsically characterized by severe hardware limitations [4], [5]. To reduce the decoupling between specification and execution, and the loss of expressivity during the translation process, the adoption of programming environments including languages meaningful enough to specify even low-level hardware operations is needed. The environments should also be lightweight enough to run on resource-constrained sensor nodes. Designers could thus benefit from the possibility of developing code aligned with high-level specifications to make implementations semantically clearer and to speed up code correction when bugs are detected [6].

The authors are with the Dipartimento dell'Innovazione Industriale e Digitale, Università degli Studi di Palermo, Palermo 90128, Italy (e-mail: salvatore.gaglio@unipa.it; giuseppe.lore@unipa.it; gloria.martorella@ unipa.it; daniele.peri@unipa.it). Digital Object Identifier 10.1109/TII.2018.2840534

In WSNs, keeping the semantic content of the target code high is difficult, especially in the context of the interpretation of collected data. Current solutions involve the use of ontologies that allow a high-level knowledge representation while maintaining tight bonds with the domain of interest [7]. Conventionally, the abstract model of the functional description passes through a series of intermediate representations that must then be translated in the target programming language [8], [9], [10], [11]. During this process, either automatic or not, the expressiveness of the high level specifications is progressively lost. In other solutions, abstract models are graphically represented and functionally specified using imperative code and specialized toolkits [12].

We propose, instead, the combination of symbolic computation with an interactive programming methodology to make testing take place during development [13].

The proposed methodology produces executable specifications capturing as much information as possible from the informal specifications provided by datasheets. Executable specifications inherently include verification. An oracle, which is code recording and verifying the runtime behavior of hardware components, is automatically generated from the specifications.

The remainder of the paper is organized as follows. In Section II we present the semantic model. In Section III we describe its basic features and real implementation on WSN nodes. In Section IV we provide guidelines and principles to write executable specifications. Section V describes the step-by-step development of executable specifications for a radio transceiver chip.

In Section VI, we provide experimental results. In Section VII, we discuss advantages and drawbacks of our semantic approach and possible future investigations. Finally, Section VIII reports our conclusions.

II. The Semantic Model

We propose a methodology and a development environment based on a symbolic paradigm that is suitable for resource-constrained devices. Leaving out the formal aspects, as in the concept of denotational semantics [14] the meaning of an expression is formally determined by the meanings of its subexpressions, in our model the task is defined in terms of the words used to describe it.

In our symbolic-based model, words are executable. This implies that a common meaning of a word can be associated with a computation. This approach lends itself to be *compositional* since each symbol has a meaning, and the sequence of words defines the semantics of the entire computation. Therefore, this semantic model is an abstraction that maps symbols, which can be indistinctly numbers, adjectives, objects or actions, to the real world. Moreover, the interpretation of the meaning from the instances themselves permits to express information without the need for conventional metamodeling techniques [15] since it is based on natural language words carrying shared semantics.

In this perspective, we adopted Forth [16] as our foundational tool. Forth is a postfix stack-based programming language that does not have a formally defined syntax, but rather permits the syntax itself to be semantically defined by the order in which the words follow each other. In a Forth-based environment, new words can be easily created and added to the system *dictionary*. The word `:` (colon) is the Forth word to define a new word and `;` (semicolon) ends its definition. A word definition entails the association of a computation that is expressed as a sequence of executable words already in the dictionary. For instance, the following code:

```
: 2*  2 * ;
```

defines the word `2*` that multiplies the value on top of the stack by 2 using the constant `2` and the previously defined word `*`. This concept is similar to the action of explaining the meaning of a word in everyday speech as a sequence of other words with well-known meaning. This means that the programming paradigm is model-oriented, in the sense that the designer incrementally includes new words in the environment, eventually defining a new language made up of words that are aligned with the requirements and the high-level domain. New symbols can be defined as *deferred words*. The computation associated to these symbols can be set to that of any word in the dictionary. This technique is adopted in Section VI to enable or disable the online monitoring feature.

III. A Semantic Environment running on WSN nodes

The typical WSN programming model, which is supported, for instance, by the widespread TinyOS and Contiki operating systems, is based on hosted cross compilation. Instead, the semantic model we adopted enables interactive development of high-level implementations for the low-level task hardware management. Indeed, we developed a number of words grouped into word sets to deal with the on-board hardware and provide basic functionality for developing high-level task. Our reference hardware platform is the IRIS mote, which is based on an AVR Atmega 1281 microcontroller (MCU) running at nearly 8 MHz and equipped with 128 KiB Flash, 8 KiB RAM and 4 KiB EEPROM memories. Word definitions are stored in Flash memory. Data, stacks and variables are located in RAM. A few pointers and global values are stored in the EEPROM.

The semantic model described in Section II is inherent in the AmForth environment [17] that we ported to our hardware test platform. AmForth is written in Forth and assembly, and permits the interaction with AVR MCUs via a shell in a serial terminal. We also developed words to support the implementation of sensing and actuation tasks as well as networking abstractions. To specify distributed computing tasks, since in the natural description of a cooperative task it can be said that a node tells another node to do something, we defined a syntactic construct that perfectly fits this informal description. The sequence of words between the words `tell:` and `:tell` is sent to a remote node and instantaneously executed on the receiving side.

We thus provide a programming environment in which it is possible to develop code structurally similar to the original specifications, by encapsulating lower level implementations –i.e. to set ports, registers and so on– in expressive words operating directly on the hardware device. This poses the basis to develop code that is fully aligned with its functional description, but in an effective way since it runs directly on the hardware without any other intermediate translation stage.

The on-board interpreter permits to implement tasks on remote nodes and in real circumstances without the need of simulation tools. While syntactically correct statements in most widespread programming languages are formally described by grammars, the syntax of Forth is guided by semantics [18]. Additionally, Forth is not only interpreted but offers on-board compilation as a standard programming method.

Other attempts to bring onboard interpreters to resource-constrained devices target either simple languages as BASIC [19] or higher-level languages, such as Java and Python. Among these, T-RES [20] is a Python-based implementation above Contiki and WiSMote [21], Maté [22] is built above TinyOS and MicaZ[23], and TakaTuka [24] lies above a restricted Java VM running on IRIS mote. These approaches do not offer enough expressiveness to justify their huge resource consumption, though. For instance, VMs of both T-RES and Maté occupy about 15 KiB RAM over 16 KiB and about 3 KiB over 4 KiB of the available RAM, respectively. Considering that the available RAM, which stores the bytecode, is almost saturated, runtime verification of symbolic code is impracticable.

Supporting symbolic computation as well as other programming abstractions on-board these VMs, e.g. aspect-oriented programming, may also require to extend existing programming languages [25]. Differently, in our approach the word dictionary is stored in the relatively abundant Flash, while RAM just holds temporary data, stacks and buffers. Moreover, our methodology does not require cross compilation, as both application and system code can be sent even to deployed nodes [5]. Our methodology also supports runtime evaluations of the verification code and symbolic interchange model generation, both on the target hardware, as shown in Section VI.

IV. Key Steps to Make High-Level Descriptions Executable

In this section we describe the key steps to translate informal specifications into executable ones expressed through symbolic code for the semantic environment previously presented.

- **Step 1: Assessment of functional aspects.** The primary step is grasping and abstracting the main aspects of the system. For distributed computing applications, as our paradigm allows for the exchange of executable code between nodes, the task must be implemented according to an "interaction-oriented" model as a set of computations that take place locally, and interactions in which a node tells another one what to do [5].

- **Step 2: Key concepts identification.** After the whole high-level operation has been figured out, it is necessary to extract the key concepts from the specifications expressed in natural language. A good practice is trying to explain the operation by putting the main concepts into words. This step allows identifying the essential parts and decomposing the problem–e.g. system operation, algorithms and distributed protocols–into smaller computations.
- **Step 3: Translation of concepts into words.** The main concepts already identified in the previous step are quite abstract and can be thought of as the words of the high level code in our semantic model. To maintain a high semantic content, is therefore, good practice to include words whose name refers to the computation associated with it. Therefore, the design proceeds through a top-down approach, from general concepts to more specialized ones. Essentially, a generic concept is the composition of more specific concepts as well as a high-level word is defined on the basis of more specific words.
- **Step 4: Wordset definition.** In this phase coding is carried-out, as words are being defined on real hardware devices. The design process proceeds in a top-down manner, coding follows a bottom-up path. In fact, as the definition of new words is based on words that are already defined, more specific words must be coded before more general ones. The proposed environment is interactive even on deployed nodes through the wireless connection, and this feature makes it possible to define a word and immediately test its operation. Code correction simply involves the redefinition of the word, i.e., its association with another computation.

These guiding principles are applicable even for the development of distributed protocols in the WSN scenario as well as for the development of hardware component drivers. Nevertheless, the design of new protocols and components goes beyond the aim of this discussion. In the next section, we focus instead on practical development on real resource-poor nodes, and on the attainment of executable specifications for hardware components.

V. Case Study: Executable Specifications for Runtime Verification of an On-Board Radio

In this section we apply the outlined methodology showing how to map informal specifications provided by datasheets to a high-level running implementation that drives a hardware component. In particular, we focus on the AT86RF230 transceiver chip [26], which is embedded in our reference hardware platform. For verification purposes, we show the development of an online oracle for monitoring the radio operation during execution and assess that the system operates as expected. The goal is not only to check that the subsystem complies with the specifications but also to discover information omitted by the reference documentation.

Usually, the implementation of an oracle on resource-constrained systems involves considerable overhead [4]. We show instead that the outlined methodology allows for an implementation with low memory footprint. In the remainder of the section we describe the implementation of the executable specifications (Listing 3) as it was actually performed on the target hardware using the symbolic programming methodology and environment described previously. The resulting code is just one of the many possibilities that the designer may choose from. Interestingly, different symbolic formalisms can coexist in the same environment given the plenty of storage space that remains available after the installation of the interpreter.

A. Step 1 in practice

The abstract model used in the data sheet to represent the radio operation is a finite state machine (FSM). Radio key functionalities, such as transmission or frame reception, are enabled by performing state transitions.

Henceforth, we use italics for some keywords in the system functional description that can be found in the final code.

Following the specifications, *running* a certain command, a *state reaches* another state. This occurs either by writing the transition identification number to a predefined radio register, by asyncronous *events*, or by rising or falling the *SLP_TR* pin. *Symbols* performing state transitions are: *trx_off, pll_on, rx_on, sleep, force_trx_off, tx_start*. The *sfd_detected* event indicating the detection of an incoming valid frame, and the *frame_end event* to signal the end of reception/transmission also cause state transitions. The transceiver distinguishes between six events on the same interrupt line that are *dispatched* by reading the *IRQ_STATUS* register. A *transmit* operation can be triggered by writing *tx_start* to the *TRX_STATE* register, if the radio is in the *pll_on* state, whereas frame reception is enabled in the *rx_on* state. A successful state change can be confirmed by reading the somewhat confusingly named *TRX_STATUS* register. The abstract model arising from the first step is illustrated in Figure 1.

B. Step 2 in practice

In this step, non functional requirements, such as state transition timings, which are also provided by the reference documentation, are needed to enrich the FSM description. Therefore, another useful concept that was identified is the *typical time* needed to perform a state transition.

C. Step 3 in practice

Keywords should be aligned with their specific code-implementation. Proceeding from the top downward, as indicated in Figure 2, high-level words are decomposed into more specific concepts. Word names should be chosen as close as possible to those used in the description. This step should not be underestimated, as it is essential to maintain the semantic content of the specifications in the names of words.

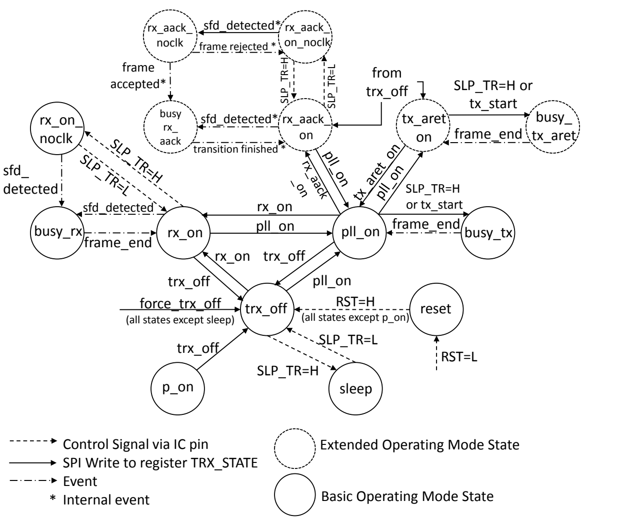


Fig. 1: *Step 1.* The operation of the radio transceiver chip is described by a finite state machine in the datasheets. The `frame_end` event in transmission is actually generated differently, as found with the runtime component verification tool (Sect. VI-B). Internal events are not exposed to the MCU.

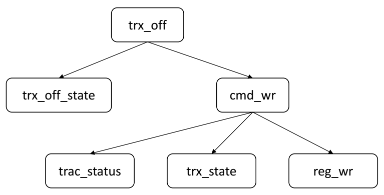


Fig. 2: *Step 3.* High level concepts must be translated into expressive words. The design proceeds following a top-down approach, from general concepts that are progressively specialized.

The top-down approach is reflected in the design of the words that proceed toward a gradually lower level of abstraction until built-in words are reached. As an example, performing `trx_off` involves executing the command associated to the `trx_off` state (Listing 3, line 63). In turn, `cmd_wr` uses the word `reg_wr` that writes to the *TRX_STATE* radio register using the SPI interface (Listing 3, lines 12, 8).

D. Step 4 in practice

Let us translate the high level functional abstract model in Figure 1 into a program to be executed on the target machine for the component verification. The word set to specify FSMs includes words to express concepts, such as events, states, symbols, and state transitions. The words `state:` and `symbol:` define states and symbols of the radio module, as follows:

```
8 state: trx_off
symbol:  trx_off
```

The code captures the fact that the two uses of `trx_off` are semantically different. In the first case, it indicates a state name, preceded by a state number, and in the latter, the symbol that causes the identically named state transition. The word `event:` is used to define high-level events, as follows:

```
event: frame_end
```

The words used to refer to symbols, states and events are rather generic and can be used to specify other hardware components whose functional behavior can be represented with a FSM. To provide a generic and abstract way of specifying FSM state transitions, the word `running:` is used to specify an input symbol or event causing the state transition, whereas the word `reaches:` is used to express the transition among two states caused by the execution of the specified symbol. The underlying idea is to translate the FSM into expressive sentences that bring to mind the graphical representation. For instance, running `trx_off`, the `pll_on` state reaches `trx_off`. This is expressed by the following target code:

```
running: trx_off state: pll_on  reaches: trx_off
doing: nothing
```

The optional word `doing:` is followed by a word whose associated computation is the sequence of actions to be done reaching the arrival state. In this case, the computation associated to the word `nothing` is a no-operation.

Previous steps also highlighted the high-level concept of transition timing. As the datasheet associates a unique symbol to each transition timing, the word `time:` is used to specify such a symbol. For instance, as found in the chip specifications, `tr5` is the symbol identifying the state transition from the `pll_on` state to `trx_off`. The typical time spent in this state transition is 1 μs. The transition timing, if found in the specifications, can be expressed as follows:

```
1 us time: tr5
```

and incorporated into the executable specifications using the optional prefix `taking:`.

```
running: trx_off state: pll_on  reaches: trx_off
doing: nothing  taking: tr5 time
```

The executable specifications concerning the FSM are provided in Listing 3. For comparison, the code extracted from the Contiki source base that is most related to ours is shown in Listing 4. Although both codes define the FSM state change, they do not overlay perfectly. Indeed, in almost the same lines of code, the former includes all the definitions from the high-level ones defining the FSM to the `reg_wr` word driving the SPI bus signals. The whole transition beginning with the SPI handshaking (`ss_h, ss_l`) and enclosed by critical section markers (`int+, int-`) is compactly represented. In the Contiki code, the hardware driving code is hidden in lower level layers but adds up to the overall source length. In fact, the same abstract model commingles with the language structures, and constraints and debug macros are used. Moreover, Contiki code makes use of busy loops and uses extended states which slightly simplify management of ACK transmission (Fig. 1), whereas ours is totally interrupt driven. Similar considerations apply to the TinyOS radio driver code, which is not shown. The AT86RF230 transceiver distinguishes between six events on the same interrupt line, although only two of them signal state transitions and are reported in the state diagram. The executable specifications bind state transitions, interrupts, and the symbols triggered by the latter, defining an interrupt dispatcher as follows:

```
dispatcher rf230-dispatcher
 conditions: ( -- 0 )
 cond: irq_value trx_end    equals  ;cond --> frame_end
 cond: irq_value rx_start   equals  ;cond --> sfd_detected
 cond: irq_value pll_lock   equals  ;cond --> noop
 cond: irq_value pll_unlock equals  ;cond --> noop
 cond: irq_value trx_ur     equals  ;cond --> noop
 cond: irq_value bat_low    equals  ;cond --> noop
 ;conditions
```

The word `dispatcher` is used to define a dispatcher named `rf230-dispatcher`. Then interrupt conditions are listed, each of which is enclosed in the `cond:` <*code*> `;cond` syntactic construct. The word `equals` compares the signaling bit of the irq register to the bitmask of a state, while `-->` is used to specify the word to be executed once the condition is verified. Events are executable symbols corresponding to interrupt service routines to handle low level interrupts. For instance, a `frame_end` event is triggered once the `irq_value`, i.e. the value in the *IRQ_STATUS* register, has the `trx_end` bit set. Event and bit names are exactly the same reported in the datasheet.

E. A Runtime Component Verification Tool

As specifications are executed on the target hardware platform, implementing a runtime component verification tool only requires to redefine words whose execution causes a state transition. The tool, hereinafter called the oracle, monitors the radio transceiver operation and verifies that it adheres to its state machine model during execution. Actually, the construct `running:` <*symbol*> redefines <*symbol*> as `preamble` <*symbol*> `conclusion`.

The words `preamble` and `conclusion` are *deferred words*. By default `preamble` and `conclusion` are set to a `noop` operation and the online monitoring feature is disabled. To enable it, both words are simply redefined. The redefined `preamble` checks that the current radio state in the `TRX_STATUS` radio register equal the value of the status variable. The redefined `conclusion` records the timer value, the expected reached state from the FSM table, and the corresponding transition timing.

The oracle uses status variables storing the expected radio state as well as some statistics about the expected and actual transition timing, and the current number of right and wrong transitions. Then, the actual radio state is compared to the expected one and statistics are updated.

VI. Experimental Evaluation

In this section we discuss the use of the oracle to gather useful metrics by running the specifications on the target platform. In all but the first test, symbolic code is exchanged by nodes and locally executed to perform the respective task.

To provide values of actual transition timings as accurate as needed, test code was executed repeatedly recording the aggregate time for state transitions and the expected time computed from specifications. Runs with 1, 10, 1000 and 10000 repetitions were performed 5 times each to record the trends as the number of repetitions increased. The average time per repetition along with standard deviation, and minimum and maximum value is reported for each test in Table I. We also performed tests to assess the oracle overhead by performing the runs with 10000 repetitions twice. On the first series of runs the oracle was enabled, on the second one it was disabled. Again, each series of runs was carried out 5 times to obtain average, standard deviation, and minimum and maximum values for the turnaround time. Estimates for oracle overhead were then obtained by comparing the timings of the two runs along with the aggregate transition timings provided by the oracle (see Table II).

A. FSM State Path Test

In the first test the runtime oracle monitored the radio operation of the reference platform during the execution of a task consisting in repeatedly switching OFF and ON the on-board transceiver. From the abstract model perspective, this high-level task involves the transition from the `trx_off` state to the `pll_on` state and back again to the `trx_off` state. As an example of the how close the code can be to the specifications, we provide the complete definition, which is a transposition of the informal description given in the sentence above into a `do-loop`, of the FSM test:

```
: fsm-path   0 do  trx_off pll_on trx_off  loop ;
```

The number of repetitions must be left on the stack before issuing the `fsm-path` command. The oracle response confirmed that, during all the test executions, the radio operation met the specifications as no mismatch between expected and actual states occurred. The runtime verification tool also provided useful information about the transition timing.

Results show that the time required to traverse the state path, which is calculated according to typical transition timing found in the datasheets, always exceeded the actual time needed for this execution on the real target machine (see Table I).

B. Transmission Stress Test

Whereas switching the radio transmitter ON and OFF does not involve dealing with interrupts, the correct transmission of a frame is signaled by a high-level radio event.

The aim of the transmission stress test was to analyze the radio behavior during repeated frame transmissions to verify that events are handled according to specifications.

A 26-B IEEE802.15.4-2003 standard compliant frame with a payload consisting in the code to turn ON the green LED of a deployed node was used. The data collected by the oracle show that even in this case, the actual time spent to perform repeated frame transmissions was lower than the value reported in the datasheet.

However, the oracle indicated a violation of the specifications, revealing incorrect state transitions although frame transmission proceeded properly. This seeming incoherence occurred when the oracle found the radio in the `pll_on` state, whereas it expected it to be in the `busy_tx` state on the `frame_end` in transmission (Figure 1). In fact, only having a look at the FSM diagram included in the transceiver specifications, the asynchronous `frame_end` event seemed responsible of the state change from `busy_tx` to `pll_on`. Instead, in another part of the documentation is reported that after a correct transmission the system re-enters the `pll_on`

TABLE I: Results of the FSM State Path, Transmission Stress, and Transmission and Reply Tests, as repetitions increase.

	State transition overall time per repetition [μs]	Repetitions 10	100	1000	10000
FSM State Path	Computed from specifications	181.00	181.00	181.00	181.00
	Actual (Average)	45.76	45.42	45.41	45.53
	Std. Dev.	0.72	0.14	0.03	0.03
	Min Value	44.80	45.28	45.38	45.49
	Max Value	46.50	45.63	45.47	45.55
Transmission Stress	Computed from specifications	48.00	48.00	48.00	48.00
	Actual (Average)	26.06	25.98	25.97	25.98
	Std. Dev.	0.18	0.06	0.02	0.01
	Min Value	25.80	25.91	25.95	25.97
	Max Value	26.30	26.04	25.99	25.98
Transmission and Reply	Computed from specifications	28.80	24.48	24.05	24.01
	Actual (Average)	4245.20	4309.46	4340.15	4344.86
	Std. Dev.	1.45	0.83	0.05	0.02
	Min Value	4243.60	4308.54	4340.11	4344.82
	Max Value	4247.00	4310.32	4340.21	4344,88

state, and only after that the `frame_end` event is generated. To resolve such a discrepancy, as interrupting events are asynchronous and can be triggered by the system at any time, a refinement process of the executable specifications was needed that permitted treating transitions caused by events and symbols differently. In fact, events signal that a state has been reached. Therefore, we slightly changed the executable specifications to take into account events by using specifications like:

```
occurring: frame_end reached: pll_on from: busy_tx
taking: tr11 time
```

This way, a `frame_end` event detection in the `pll_on` state made the oracle consider state transitions as correct, even if it was unable to assess the radio had transitioned through the operatively "invisible" `busy_tx` state.

C. Transmission and Reply Test

The transmission and reply test was aimed at assessing the behavior of the radio component in a distributed computation task. The experimental setup consisted of two WSN nodes. In turn, each node incremented a shared value, stored it and then, after a 10-ms delay accounting for packet collision overhead in larger networks, sent it to the other node along with the sequence of symbols to command it to do the same. The exchange continued until the maximum number of repetitions was reached on a node. The overall state transition time thus aggregates the time for n repetitions of the message exchange task involving both nodes. This test made intensive use of the radio so the symbolic interrupt management code was particularly stressed. Actual time exceeded the expected one as the oracle measured, besides the time needed for state transitions, the time the radio transceiver remained in the reception state until the event signaling reception occurred. Measurements of this kind are often used as metrics to assess energy consumption. In our case, the oracle provided precise results in a real application on real hardware, without needing to resort to simulated environments.

VII. Discussion

To provide an overall comparison of our methodology to the typical WSN development practice we discuss the implementation of application code using the radio subsystem whose executable specifications were defined in the previous

TABLE II: Turnaround timings along with oracle and test code execution time for FSM State Path, Transmission Stress, and Transmission and Reply Tests over 10000 repetitions ($t_4 = t_1 - t_2$, $t_5 = t_2 - t_3$)

Timings per repetition [μs]		FSM State Path	Transmission Stress	Transmission and Reply
Turnaround time	**Oracle on** (t_1)	24623.80	87541.40	2974077.00
	Oracle off (t_2)	4504.40	45879.60	1284048.60
State transition overall time (t_3)		45.53	25.98	4344.86
Execution time	**Oracle** (t_4)	20119.40	41661.80	1690028.40
	Test Code (t_5)	4458.87	45853.62	1279703.74
t_1 **Std. Dev.**		17.40	33.56	52.18
t_1 **Min Value**		24610	87507	2974020
t_1 **Max Value**		24653	87590	2974159
t_2 **Std. Dev.**		0.55	1.14	1.52
t_2 **Min Value**		4504	45878	1284047
t_2 **Max Value**		4505	45881	1284051

TABLE III: Comparison of two WSN programming models based on cross compilation and the interactive one of the experimental setup.

	TinyOS	Contiki	Experimental Setup
Source Code (Driver+Application)			
Lines of code	4200+210	4328+61	649+8
Size in bytes	99775+6501	185720+3838	19266+410
Memory Footprint (RAM+Flash) [B]			
Application	474+11968	3238+20570	1058+5808
Interpreter and compiler	n.a.	n.a.	289+17306

sections. We picked one of the WSN applications provided with TinyOS (v. 2.1.2), which are written in its native C-derived programming language nesC, and wrote versions for our environment (Listing 1) and Contiki. The application consists in sampling the environmental temperature with a given interval between samples (default $1s$). Every 10 samplings the values are transmitted to a base station. Such a simple application is actually disadvantageous for our approach as the benefit of high level symbolic code become generally more evident as the complexity of tasks arise. The three versions of the applications were compared in terms of source lines of code, source size, and memory footprint, with separate figures for driver and application code. Results are reported in Table III. Excluding the overhead of the interpreter and compiler, our driver and application source code turned out to be more compact than the other two, whereas the three are comparable in terms of memory footprint. However, the on-board interpreter and compiler support interactive and experimental programming. Thus, hardware driving and application code can be written and tested definition-by-definition on the target hardware. Moreover, our code is sent via radio, a feature that the other environments lack. This also enables testing interactively distributed applications. For instance, the behavior of the example application (Listing 1) can be changed dynamically on deployed nodes simply by sending them a new symbolic rule. Listing 2 shows the code to broadcast a rule making each node to decrease the sampling interval by 10 ms if the maximum value sampled by the node is below 50.

Although the interactivity of the testing phase makes it fast, it cannot prevent every malfunction and error. It is impossible, in fact, for the designer to enumerate all the potentially occurring situations as well as the behavior of the system for all the possible inputs. Future work will concern on-board formal verification and automatic translation of specifications.

Listing 1: Code of the test application. In-line comments follow a backslash.

```
variable base-station     \ A variable for the receiver address
10 buffer samples         \ A buffer for 10 samples
variable interval 1000 interval !   \ The default interval is 1000 ms
```

```
: send-samples default-pkt dest addr base-station @ s_addr! payld
      addr samples buffer-ptr samples buffer-count  a>pkt outbound
      transmit ;
: oscilloscope.isr 1 timer3.tick +! temperature samples buffer-add
      samples buffer-full? if send-samples samples buffer-reset then ;
: oscilloscope ['] oscilloscope.isr interval @ timer3.init
      timer3.start ;  \ Init time3 to trigger the interrupt service
                      \ routine with the interval value, and start it.
```

Listing 2: Code enclosed between `tell:` and `:tell` is broadcast to make each node adjust its sampling interval by processing its last readings.

```
bcst tell: samples buffer-max 50 < [if] -10 interval +! [then] :tell
```

VIII. Conclusions

In this paper, we proposed a development methodology for distributed computing based on a very high-level and interactive programming approach that is feasible for resource-constrained devices such as those usually found in WSNs. We applied the proposed approach showing how to proceed from the descriptions provided by datasheets to a high-level implementation of a driver for a WSN radio transceiver chip. Moreover, the executable specifications also provide an oracle for the runtime verification of the hardware module. The final code is compact and coherent with the FSM functional model in the datasheets. We also discussed the implementation of a WSN application using the radio driver in our interactive programming environment and compared it to versions for two prominent WSN programming environment. We showed how our application code, while high-level and compiled on-board, has direct access to the hardware low-level details. Moreover, the behavior of our implementation can be changed at runtime by sending a symbolic rule to deployed nodes.

References

[1] E. Estevez and M. Marcos, "Model-Based Validation of Industrial Control Systems," *IEEE Transactions on Industrial Informatics*, vol. 8, no. 2, pp. 302–310, May 2012.

[2] R. Drechsler, M. Soeken, and R. Wille, "Formal Specification Level: Towards Verification-driven Design Based on Natural Language Processing," in *Forum on Specification and Design Languages (FDL), 2012*, Sept 2012, pp. 53–58.

[3] S. Fischmeister and P. Lam, "Time-Aware Instrumentation of Embedded Software," *IEEE Transactions on Industrial Informatics*, vol. 6, no. 4, pp. 652–663, Nov 2010.

[4] P. Iyenghar, E. Pulvermueller, M. Spieker, J. Wuebbelmann, and C. Westerkamp, "Time and Memory-aware Runtime Monitoring for Executing Model-based Test Cases in Embedded Systems," in *11th IEEE International Conference on Industrial Informatics (INDIN), 2013*, July 2013, pp. 506–512.

[5] S. Gaglio, G. Lo Re, G. Martorella, and D. Peri, "DC4CD: A Platform for Distributed Computing on Constrained Devices," *ACM Trans. Embed. Comput. Syst.*, vol. 17, no. 1, pp. 27:1–27:25, Dec. 2017. [Online]. Available: http://doi.acm.org/10.1145/3105923

[6] N. Moha, Y.-G. Guéhéneuc, A.-F. Meur, L. Duchien, and A. Tiberghien, "From a Domain Analysis to the Specification and Detection of Code and Design Smells," *Formal Aspects of Computing*, vol. 22, no. 3-4, pp. 345–361, 2010.

[7] J. Serrano, J. Serrat, and J. Strassner, "Ontology-Based Reasoning for Supporting Context-Aware Services on Autonomic Networks," in *IEEE International Conference on Communications, 2007. ICC '07*, June 2007, pp. 2097–2102.

[8] A. F. Martins and R. de Almeida Falbo, "Models for Representing Task Ontologies," in *Proceedings of the 3rd Workshops on Ontologies and their Application*, 2008.

[9] H. Wada, P. Boonma, J. Suzuki, and K. Oba, "Modeling and Executing Adaptive Sensor Network Applications with the Matilda UML Virtual Machine," in *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications*. ACTA Press, 2007, pp. 216–225.

[10] M. Shahbaz, K. C. Shashidhar, and R. Eschbach, "Iterative Refinement of Specification for Component Based Embedded Systems," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11. New York, NY, USA: ACM, 2011, pp. 276–286.

[11] M. A. Wehrmeister, C. E. Pereira, and F. J. Rammig, "Aspect-Oriented Model-Driven Engineering for Embedded Systems Applied to Automation Systems," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 4, pp. 2373–2386, Nov 2013.

[12] A. Bakshi, J. Ou, and V. K. Prasanna, "Towards Automatic Synthesis of a Class of Application-specific Sensor Networks," in *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, ser. CASES '02. New York, NY, USA: ACM, 2002, pp. 50–58.

[13] S. Gaglio, G. Lo Re, G. Martorella, and D. Peri, "A Fast and Interactive Approach to Application Development on Wireless Sensor and Actuator Networks," in *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, Sept 2014, pp. 1–8.

[14] H. F. Guo, L. Cao, Y. Song, and Z. Qiu, "Automated Test Oracle Generation via Denotational Semantics," in *2014 14th International Conference on Quality Software*, Oct 2014, pp. 139–144.

[15] V. Vyatkin, "Software Engineering in Industrial Automation: State-of-the-Art Review," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 3, pp. 1234–1249, Aug 2013.

[16] D. M. Hanna, B. Jones, L. Lorenz, and S. Porthun, "An Embedded Forth Core with Floating Point and Branch Prediction," in *2013 IEEE 56th International Midwest Symposium on Circuits and Systems (MWSCAS)*, Aug 2013, pp. 1055–1058.

[17] "AmForth Documentation," 2013, available online at http://amforth.sourceforge.net/amforth.pdf.

[18] B. Stoddart, C. Ritchie, and S. Dunne, "Forth Semantics for Compiler Verification," in *Proceedings of the 28th EuroForth Conference*, 2012, pp. 45–58.

[19] J. S. Miller, P. A. Dinda, and R. P. Dick, "Evaluating a BASIC Approach to Sensor Network Node Programming," in *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, ser. SenSys '09. New York, NY, USA: ACM, 2009, pp. 155–168.

[20] D. Alessandrelli, M. Petracca, and P. Pagano, "T-res: Enabling reconfigurable In-network Processing in IoT-based WSNs," in *2013 IEEE International Conference on Distributed Computing in Sensor Systems*, May 2013, pp. 337–344.

[21] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *29th Annual IEEE International Conference on Local Computer Networks*, Nov 2004, pp. 455–462.

[22] P. Levis and D. Culler, "Maté: A Tiny Virtual Machine for Sensor Networks," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS X. New York, NY, USA: ACM, 2002, pp. 85–95. [Online]. Available: http://doi.acm.org/10.1145/605397.605407

[23] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, *TinyOS: An Operating System for Sensor Networks*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 115–148. [Online]. Available: http://dx.doi.org/10.1007/3-540-27139-2_7

[24] F. Aslam, C. Schindelhauer, G. Ernst, D. Spyra, J. Meyer, and M. Zalloom, "Introducing TakaTuka: A Java Virtualmachine for Motes," in *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, ser. SenSys '08. New York, NY, USA: ACM, 2008, pp. 399–400. [Online]. Available: http://doi.acm.org/10.1145/1460412.1460472

[25] E. Lakshika, C. Keppitiyagama, and D. Wathugala, "AOnesC: An Aspect-Oriented Extension to nesC," in *2008 New Technologies, Mobility and Security*, Nov 2008, pp. 1–5.

[26] "AT86RF230 Datasheet," 2016, available online at http://www.atmel.com/images/doc5131.pdf.

Listing 3: Executable specifications for the radio subsystem

```
$7F constant reg_addr_mask   $E0 constant trac_status_mask
$80 constant reg_rd_command  $C0 constant reg_wr_command
$08 constant trx_off_state   $02 constant trx_state

: reg_rd
      reg_addr_mask and reg_rd_command or
      -int  ss_h ss_l spi! spi@ ss_h  +int ;
: reg_wr
      reg_addr_mask and reg_wr_command or
      -int  ss_h ss_l spi! spi! ss_h  +int ;
: trac_status   trx_state reg_rd trac_status_mask and 5 rshift ;
: cmd_wr   trac_status 5 lshift or trx_state reg_wr ;

14 states   10 symbols   1 events

00 state: p_on    01 state: busy_rx   02 state: busy_tx
06 state: rx_on   08 state: trx_off   09 state: pll_on
15 state: sleep   17 state: busy_rx_aack
18 state: busy_tx_aret       22 state: rx_aack_on
25 state: tx_aret_on         28 state: rx_on_noclk
29 state: rx_aack_on_noclk   30 state: busy_rx_aack_noclk
start: rx_aack_on

symbol: trx_off     symbol: pll_on     symbol: tx_aret_on
symbol: rx_aack_on  symbol: tx_start   symbol: rx_on
symbol: SLP_TR=H    symbol: SLP_TR=L   symbol: RST=L
symbol: RST=H

event: frame_end

880 time: tr1   880 time: tr2   35 time: tr3   180 time: tr4
  1 time: tr5   180 time: tr6    1 time: tr7     1 time: tr8
  1 time: tr9    16 time: tr10  32 time: tr11    1 time: tr12
120 time: tr13

running: trx_off state: p_on reaches: trx_off taking tr1 time
running: trx_off state: pll_on reaches: trx_off taking: tr5 time
running: trx_off state: rx_on reaches: trx_off taking: tr7 time
running: pll_on state: trx_off reaches: pll_on taking: tr4 time
running: pll_on state: rx_on reaches: pll_on taking: tr9 time
running: rx_on state: trx_off reaches: rx_on taking: tr6 time
running: rx_on state: pll_on reaches: rx_on taking: tr8 time
running: tx_start state: pll_on reaches: busy_tx taking: tr10 time
running: tx_aret_on state: pll_on reaches: tx_aret_on
running: pll_on state: tx_aret_on reaches: pll_on
running: rx_aack_on state: pll_on reaches: rx_aack_on
running: pll_on state: rx_aack_on reaches: pll_on
running: tx_aret_on state: trx_off reaches: tx_aret_on
running: rx_aack_on state: trx_off reaches: rx_aack_on
running: tx_start state: tx_aret_on reaches: busy_tx_aret
running: SLP_TR=L state: rx_aack_on_noclk reaches: rx_aack_on
running: SLP_TR=H state: rx_aack_on reaches: rx_aack_on_noclk
running: SLP_TR=L state: rx_on_noclk  reaches: rx_on
running: SLP_TR=H state: rx_on reaches: rx_on_noclk
running: SLP_TR=H state: pll_on reaches: busy_tx
running: SLP_TR=H state: tx_aret_on reaches: busy_tx_aret
running: SLP_TR=H state: trx_off reaches: sleep
running: SLP_TR=L state: sleep reaches: trx_off
occurring: frame_end reached: pll_on from: busy_tx taking: tr11 time
occurring: frame_end reached: tx_aret_on from: busy_tx_aret
unmeasurable busy_tx   unmeasurable busy_tx_aret

: trx_off   trx_off_state cmd_wr ;
```

Listing 4: Contiki source code of the radio state change function

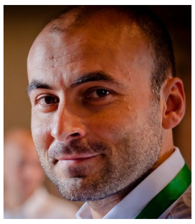
```
radio_set_trx_state(uint8_t new_state)
{
   uint8_t current_state;
   if (!((new_state == TRX_OFF) ||
        (new_state == RX_ON) ||
        (new_state == PLL_ON) ||
        (new_state == RX_AACK_ON) ||
        (new_state == TX_ARET_ON))){
      return RADIO_INVALID_ARGUMENT; }

    if (hal_get_slptr()) {
       DEBUGFLOW('W');
      return RADIO_WRONG_STATE;
   }

   rf230_waitidle();
   current_state = radio_get_trx_state();

   if (new_state == current_state){
      return RADIO_SUCCESS;
   }

   if(new_state == TRX_OFF){
      if (hal_get_slptr()) DEBUGFLOW('K');DEBUGFLOW('K');
      DEBUGFLOW('A'+hal_subregister_read(SR_TRX_STATUS));
      radio_reset_state_machine();
   } else {
         if (((new_state == TX_ARET_ON) && (current_state == RX_AACK_ON)) ||
         ((new_state == RX_AACK_ON) && (current_state == TX_ARET_ON))){
         hal_subregister_write(SR_TRX_CMD, PLL_ON);
         delay_us(TIME_STATE_TRANSITION_PLL_ACTIVE);
      }
      hal_subregister_write(SR_TRX_CMD, new_state);

        if (current_state == TRX_OFF){

#if defined(__AVR_ATmega128RFR2__) || defined(__AVR_ATmega256RFR2__)
    hal_subregister_write(SR_TRX_RPC, rpc);
#endif
         delay_us(TIME_TRX_OFF_TO_PLL_ACTIVE);
      } else {
         delay_us(TIME_STATE_TRANSITION_PLL_ACTIVE);
      }
   }

    current_state = radio_get_trx_state();
   if (current_state != new_state) {
      if (((new_state == RX_ON) && (current_state == BUSY_RX)) ||
         ((new_state == RX_AACK_ON)&&(current_state == BUSY_RX_AACK))){
      } else {
         DEBUGFLOW('N');DEBUGFLOW('A'+new_state);
         DEBUGFLOW('A'+radio_get_trx_state());DEBUGFLOW('N');
         return RADIO_TIMED_OUT;
      }
   }

   return RADIO_SUCCESS;
}
```

Side-by-side comparison between the executable specifications for the AT86RF230 transceiver and part of the code managing the same device in Contiki. The executable specifications fully describe the operation of the transceiver and include the low-level code driving the SPI interface to which it is connected. Definitions for states, timings and transitions are clearly separated. The few lines of the IRQ dispatcher shown in Section V-D and of the state-changing words other than `trx_off` were omitted here.

Salvatore Gaglio (M'76) received the Laurea degree in electronic engineering from the University of Genoa, Italy, in 1977, and the degree of M.S.E.E. from the Georgia Institute of Technology, USA, in 1979. He has been a Full Professor in computer science and artificial intelligence at the University of Palermo, Palermo, Italy, since 1986. From 1998 to 2002, he was the Director of the Study Center on Computer Networks of the National Research Council (CNR), Italy, and from 2002 to 2016, he was the Director of the Branch of Palermo of the High Performance Computing and Networks Institute of CNR. From 2005 to 2012, he was member of the Scientific Council of the ICT Department of CNR, Italy, and from 2015 to 2018, he was the President of the Accademia Nazionale di Scienze, Lettere ed Arti di Palermo, Italy. His current research interests include artificial intelligence and robotics. He is a member of IEEE, ACM, and AAAI.

Giuseppe Lo Re (SM'11) received the Laurea degree in computer science from the University of Pisa, Pisa, Italy, in 1990, and the Ph.D. degree in computer engineering from the University of Palermo, Palermo, Italy, in 1999. He has been an Associate Professor in computer engineering with the University of Palermo, since 2004. In 1991, he joined the Italian National Research Council (CNR), where he achieved the Senior Researcher position. His current research interests include computer networks and distributed systems, broadly focusing on wireless sensor networks, ambient intelligence, Internet of Things. Dr. Lo Re is a senior member of IEEE and of its Communication Society and ACM.

Gloria Martorella received the Master's and Ph.D. degrees in computer engineering from the University of Palermo, Palermo, Italy, in 2013 and 2017 respectively. She is currently a Postdoctoral Research Fellow in computer engineering with the University of Palermo. Her current research interests include ambient intelligence, distributed systems, logic and symbolic programming, and formal methods for specification and verification of resource-constrained embedded systems.



Daniele Peri received the Master's and Ph.D. degrees in computer engineering from the University of Palermo, Palermo, Italy, in 1998 and 2004, respectively. He is cuurently an Assistant Professor in computer engineering with the University of Palermo. His current research interests include intelligent and distributed embedded systems, embedded architectures, system software design, symbolic and logic programming, as well as specification and verification of embedded systems.