2013 39th Euromicro Conference Series on Software Engineering and Advanced Applications

SEAL: a Domain-Specific Language for Novice Wireless Sensor Network Programmers

Atis Elsts*†, Janis Judvaitis*† and Leo Selavo*†
*Faculty of Computer Science
University of Latvia
†Institute of Electronics and Computer Science
Riga, Latvia
Email:{atis.elsts,janis.judvaitis,selavo}@edi.lv

***Abstract*—A lot of the prospective wireless sensor network users are novice programmers. Their experience in general-purpose programming languages is either limited or completely nonexistent. There are both financial and scientific incentives to empower these users and allow them to write sensor network applications on their own, rather than having to rely on a qualified computer science professional.**

We present SEAL, a sensor network programming language designed for novice programmers. SEAL manages to avoid computer science concepts that are hard to grasp for novices, while remaining suitable for typical sensor network application scenarios. The language is extensible in application-specific way, has easy-to-learn syntax and allows to implement common sensor network tasks by writing compact, readable code. It is also shown to have high run-time efficiency.

I. Introduction

Wireless sensor networks (WSN) is a technology envisioned as a practical tool for a broad range of applications and target audiences. Nevertheless, sensor network applications at the moment are usually developed by the computer science professional. This approach is both cost-ineffective and restrictive to the sensor network end user, who has limited programming skills. First, it requires participation of a qualified computer scientist in the sensor network development and deployment processes. Second, it requires constant availability of the professional to perform maintenance operations on-site. Finally, it puts the domain expert in the role of a passive user rather than an active contributor.

Up to this date no single novice-friendly programming toolkit, abstraction or methodology has gained significant recognition. There are several possible reasons. First, the early research had theoretical emphasis. From 28 WSN programming abstractions reviewed in a recent survey [17] only approximately half have real hardware implementations, and only one has been used in a real-world deployment [17]. Second, many of the abstractions require prerequisites that are not realistic for application domain experts; for instance, knowledge of functional programming, logical programming, or the C programming language. Third, many are either limited by design to only a particular kind of WSN applications [7] [14], or only solve a particular programming problem.

Inspired by the work done in novice programming research [15] [18] [19] [20], we propose SEAL: a programming language and development environment for WSN application development.

Our contribution is to show that even a very restricted language with no loops or recursion can handle the domain requirements well enough, enabling WSN application programming for people who do not want to learn numerous computer science concepts first.

SEAL features novel declarative syntax that allows to describe typical sensor network applications [22] in compact code with low complexity (Section V-A). SEAL transparently includes common control flow elements of WSN applications, such as setting up event handlers and switching to low-power modes, and allows the programmer to focus on the application logic itself: sensor data reading, processing, and dissemination (Fig. 1).

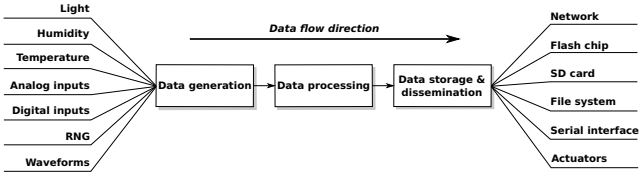


Figure 1: Typical data flow in a sensor node

A SEAL application consists of component use cases and their parameters (Section IV). SEAL allows the domain expert to think in terms that either are already familiar to her, or have concrete real-world counterparts (such as physical sensors). For example, SEAL has support for numeric literals with custom suffixes, allowing the user to specify constants in physical units (such as degrees or luxes) rather than raw sensor readings.

SEAL code is compiled to C and typically adds only a small overhead (Section V-B). Therefore the code written by novice users can be used not only for prototypes and for learning, but also for real world deployments.

978-0-7695-5091-6/13 $26.00 © 2013 IEEE
DOI 10.1109/SEAA.2013.16

II. Approach

This paper implements ideas discussed in Elsts *et al.* [9]. We briefly reiterate them here and present new aspects, such as the need for extensibility, concluding with a set of design decisions.

A. Novice programmers

Many of the prospective sensor network users are not only experts of their respective application domains, but also *novice programmers*. It takes about 10 years to turn into an expert programmer [20]. Therefore, the gap between beginners and experts is huge. However, many of the issues that novices face are related explicitly to imperative languages: "poor grasp of the basic sequential nature of program execution", loops and conditionals, the use of arrays, recursion, and especially "issues relating to flow of control". Most of these issues are not present in a declarative language without recursion.

Design of SEAL is also influenced by other issues faced by the novice programmers [19], and by programming tools suitable for them [15] [18].

B. Domain specific languages

Three common approaches for domain-specific programming exist: software libraries, embedded domain-specific languages, and domain-specific languages (DSL). It is generally believed that "programs written in a DSL [..] can often be written by non-programmers" [11]. DSL allows direct mapping between domain objects & processes and language elements. The result is the desired: abstractions with real-world counterparts.

In contrast, a general purpose programming language such as C would confuse the novice user by low-level details, for example, pointer semantics or alignment of fields in structures. Higher-level languages like Java or Python still require understanding of computer science concepts, such as loops and recursion.

One common problem from which DSL suffer is that of extensibility [23]. This is especially important because a lot of WSN applications use custom hardware [22]. Therefore the DSL should be designed in away that custom extensions can be added easily (without compiler modifications).

C. Separating user code from system code

Some aspects of WSN system programming are likely to remain complex. Nevertheless, the *application logic* of real-world WSN applications often is trivial [22]. Therefore, there is a need for clear demarcation line between system code and user code. The system code is written by the WSN professional; it describes internal mechanisms of the WSN operating system, device drivers, and distributed algorithms. The user code describes application logic for sense-and-send or event detection, as well as basic data processing, such as aggregation and filtering. It uses the system code to achieve its goals, but abstracts away from low level details of the latter.

D. Declarative or imperative?

Imperative languages (such as BASIC) are common for teaching novice programmers. They allow to have clear mapping between the source code and the execution flow. However, we argue that a declarative language is preferable in the context of our work.

First, many WSN applications are event-based, which means that an imperative language will fail to reflect the execution flow as well. Rather, such a language would require explicit setup of event handling functions. If an imperative language and a polling-based approach is used, there are other issues: in this case a global execution loop is required, and low power mode semantics becomes explicit. A WSN-specific declarative language can hide all of the three: low level details of event handling, the global loop, and low-power mode usage.

Second, there is the question of efficiency. In an imperative language, the behavior of a WSN node (e.g. the radio channel) can be set by modifying a variable or by changing some hardware state. This requires run-time overhead, code memory overhead, and, in the former case, RAM overhead as well. Declarative language allows to *declare* the behavior of the system. An advanced compiler can analyze this declaration and implement it in the compile time, avoiding any run-time overhead, except for hardware initialization. For embedded software, efficiency is often more important than run-time flexibility!

At the same time, several typical WSN applications (for example, event detection based on sensor readings) are naturally expressed using finite state-machine abstraction, which is hard to implement in a declarative fashion. Therefore, a hybrid language that allows to describe global system states is preferable.

E. The design decisions of SEAL

- It should be a DSL. Rationale: minimize client effort.
- Low level concepts should not be available (from the language). Rationale: limit the number of choices.
- Concepts with steep learning curves should be absent or not mandatory. Rationale: the novice user should be able to learn the toolset "on the fly".
- The expressiveness should be limited to application logic, not system logic. Rationale: see Section II-C.
- Make commonly used patterns (the global loop, event handler setup, using low-power modes) implicit. Rationale: shorter, cleaner code. Studies show that novice users are confused by the semantics of low-power modes [16].

- Make examples easy to access. Rationale: see [18].
- Do as much as possible at compile time. Rationale: see Section II-D.
- There should be a choice between textual and visual programming. Rationale: none of the two choices are superlative [19].

III. Related work

There are few sensor network programming languages that take into account novice programming research. From these, SensorBASIC [16] is an imperative language with large interpreter overhead. WASP [7] is limited to a single application archetype.

TinyScript [13] is imperative, event-driven and requires use of multiple source files even for simple applications.

`make`Sense [8] is a new development with objectives similar to ours, but their target audience is different: users familiar with Business Process Modelling Language (BPML), rather than domain scientists & practitioners.

Multiple integrated, higher-level programming and data querying interfaces for sensor networks exists, TinyDB [14] being the most prominent, but they lack the generality we are looking for.

Similar arguments stand against sensor network programming using Web service architectures. (At least the currently existing.) For example, TinySOA [6] is too limiting as it offers no support for in-network data processing.

Sensor networks can be programmed in general purpose high-level languages such as Java [21] or Python [5], but none of these options have gained high acceptance. First, they are heavyweight; second, unsuitable for novice programmers, as they presuppose existing programming language knowledge.

As for the existing WSN macroprogramming and distributed processing abstractions [17]: they can be put on layers below or above SEAL. We aim to complement, rather than to replace them.

IV. Design and implementation

Listing 1: Temperature monitoring application

```
1 read Temperature, period 10s;
2 output Network;
3 when Temperature > 40C:
4     use RedLed, on;
5 end
```

Consider the example application in Listing 1. The code reads temperature with 10 second period and outputs the result to the network. It also monitors whether the temperature is above 40° Celsius and turns the red LED on when this alarm condition is reached. Predefined suffixes, such as 's' and 'C' are used to specify physical units. The code generator automatically converts from these human-readable values to raw sensor readings. (Similar conversions cannot always be done with, for example, the C preprocessor, as it is not Turing-complete.)

A. Overview

The SEAL code is characterized by periodic execution flow. At the high level, the code is structured in *branches*: groups of statements. An *active* branch is a branch whose statements are executing (periodically). An inactive branch *starts* execution whenever all conditions that enclose it are satisfied. An active branch *finishes* execution either when all the statements in the branch have executed fixed number of times (as specified in parameters), or some of the conditions enclosing the branch are not satisfied anymore. *Subbranches* are created and controlled using specific statement types: `when` statements and `do` statements.

There are only two types of executable statements: component use cases and `set` statements. The former are executed periodically by default, the latter: just once. The rest of statements define or structure something.

Table I shows language elements and their usage. The complete grammar of SEAL is available at [4].

There are three component types in SEAL: *sensors* (their "`use`" action is to read and return a value), *actuators* (their action: component-specific activation), and *outputs* (action: print, store or send sensor values, once they become available). Active outputs process values of all currently active sensors (unless specific subset of sensors is listed in parameters of the output's use case).

SEAL is ultimately constrained by the fact that it is not Turing-complete. Predefined functions can be used and parametrized, but no new functions can be defined at the language level.

B. Component library and runtime

The code written in SEAL relies on an operating system services for runtime execution support. The OS behind SEAL at the moment is MansOS, and the C code generated by SEAL compiler uses MansOS networking services, sensor drivers, permanent data storage library etc. However, there are no conceptual obstacles why another WSN OS could not be used, as long as implements at least software timers and a few of SEAL components. There is a proof-of-concept implementation of SEAL subset in Contiki [9].

Between the operating system and the elements of SEAL there is a middle layer: a component library, written in Python. Each SEAL component corresponds to a Python class, and each hardware platform corresponds to a Python module; class inheritance is heavily used. If a new component has to be added, only 5–10 additional lines of Python code are required. Furthermore, the new code is not required to be placed in the main component

Element	Description	Templatized examples

Component use case	Specifies that the named component should be used (read or activated) in the current code branch, using the specific parameters supplied in parameter list. Starts with keyword `use` (can be replaced by `read` as a synonym for sensors, and by `output` as a synonym for system outputs), followed by the name of the component and list of parameters. The comma-separated parameters may include usage period, number of times to execute, and component-specific parameters.	`// light up the default LED`
		`use Led, period 1000ms, once;`
		`// "use" (i.e. read) temperature sensor periodically`
		`read Temperature;`
		`// "use" (i.e. output to) the network;`
		`// enable checksumming and select TDMA MAC protocol`
		`output Network, checksum, protocol TDMA;`

`when` statement	Determines which subbranches to execute depending on conditional expressions. A subbranch of a `when` statement is active whenever the condition that is associated with it (defined immediately after the opening `when`/`elsewhen` keyword) is satisfied, and no conditions associated with previously defined subbranches of this `when` statement are satisfied. There must be exactly one `when` subbranch, and can be zero or more `elsewhen` subbranches as well. The `else` subbranch has no associated condition. If present, it always must be the last.
	SEAL is flexible with regard to what can be used as a conditional expression. It can be a literal, a symbolic constant, system state, a (function of) sensor value, or any syntactically valid combination of the options, constructed using comparison operators and logical connectives. In the end, any nonzero integer value maps to Boolean *true*, zero – to *false*.
		`when <condition_1>:`
		`  // executed when <condition_1> is true`
		`  use <component1>;`
		`elsewhen <condition_2>:`
		`  // executed when <condition_1> is false`
		`  // and <condition_2> true`
		`  use <component2>;`
		`else:`
		`  // executed when both conditions are false`
		`  use <component3>;`
		`end`

`do` statement	Determines the execution order of subbranches depending on their order in source code. The `do` subbranch that is first in the code becomes active immediately after the whole `do` statement itself starts executing. The subsequent `then` subbranches are optional; each of them becomes active after all the previous `do`/`then` subbranches have finished execution.	`do, once:`
		`  use <component1>; // executed just once`
		`then, times 2:`
		`  use <component2>; // executed afterwards, twice`
		`then:`
		`  // executed infinitely, after 1st & 2nd subbranch`
		`  use <component3>;`
		`end`

`define` statement	Defines a new, "virtual" sensor as a function (possibly parametrized) of the currently defined sensors. The new sensor can be from now on used as any other sensor would be.	`// temperature maximum during application lifetime`
		`define MaxTemp max(Temperature);`
		`// a parametrized analog input`
		`define MyInput AnalogInput, port 2, pin 6;`

`const` statement	Defines a symbolic constant. The name of the constant can be used in place of numerical literals.	`const MAX_TEMP 40C;`

`set` statement	Initializes or changes a system state. A state can take any integer or Boolean values. Their value can be set from a literal, a sensor, a function, and a state (the same or another).	`set temperatureCritical False; // assign a constant`
		`set counter add(counter, 1);  // increment`

`load` statement	Loads a component library extension code (a Python file) or runtime extension code (a C file).	`load "ExtensionLibrary.py"; // component definitions`
		`load "ExtensionLibrary.c";  // runtime implementation`

Table I: Selected elements of SEAL

library; it can be put in an application-specific source file. Using `load` statement, the component may be made available from SEAL code. In this way SEAL can be easily extended without modifying the compiler or the default component library.

C. Implementation

SEAL parser, code generator and component library are implemented in Python, using PLY (Python Lex-Yacc [3]) module. The code generator produces OS-specific C source code, which then can be compiled to platform-specific executable.

The generated C code schedules software timers in order to read sensors and perform other actions periodically, and sets up hardware interrupt handlers to read interrupt-based sensors. A special type of network packets is used to exchange information between nodes.

The current state of each conditional expression (from `when` statements) is stored in a Boolean variable. The state is reevaluated whenever it may change; e.g. when a sensor that is part of the conditional expression is read. This allows to implement event-based actions. Whenever a conditional expression changes its value, code subbranches associated with it are started or stopped.

D. Development environments

Two graphical interfaces have been built on top of SEAL. First, there is SEAL integrated development environment (IDE; Fig. 2). The IDE has support for entering code either by keyboard (left side) or by mouse (visual edit in the right side). It allows to upload and debug the applications. Last but not least, it features a menu that allows to access example application code.

Second, we have designed a purely-visual interface of

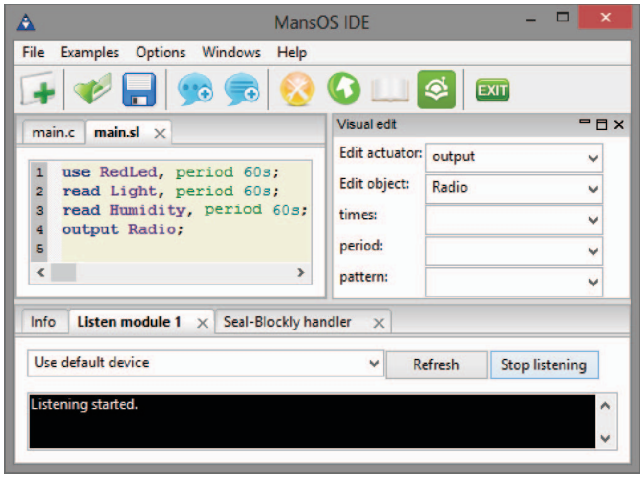


Figure 2: Editing SEAL application in IDE

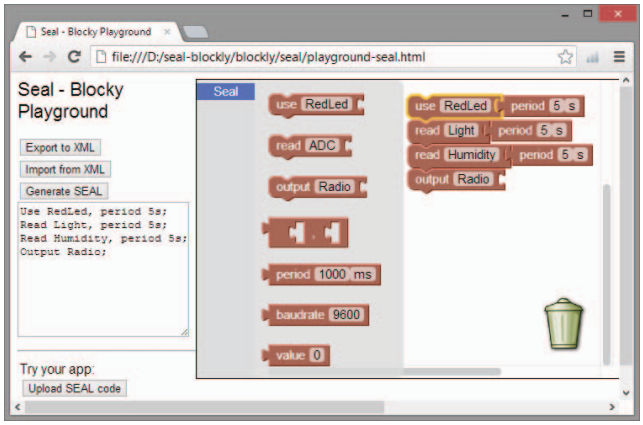


Figure 3: Editing SEAL application in Web browser

SEAL using Google Blockly [2] as a base. Scratch, a similar visual programming language [15] has been successfully used in teaching novice programmers. SEAL-Blockly allows to create applications by putting puzzle-like blocks together and is usable from a web browser (Fig. 3). Visual programming allows to avoid syntax errors and minimize working memory load, since all the possible blocks and their possible combinations are predefined. On the other hand, visual programming is not superior to textual in all contexts, therefore we envision SEAL-Blockly only as one of *several* front-ends.

V. Evaluation

A. Analysis

Compared with single-flow imperative languages like SensorBASIC, SEAL applications put the system in low power mode by default, while SensorBASIC requires that `sleep` command is used. Studies have shown [16] that novice users do not have clear grasp of low-power mode semantics and fail to use the `sleep` keyword correctly. Compared with event-based languages like TinyScript, a SEAL developer does not have to think in terms of event handlers, which are arguably hard for novices to grasp.

	SEAL	TinyScript	C	nesC
Nonterminal symbols	**36**	31	70	24 new
Keywords	**19**	15	31	15 new
Operators	**8**	23	46	1 new
Non-alphanumeric char.	**15**	20	31	0 new

Table II: Metrics of several sensor network programming languages: SEAL, C [1], nesC [10], and TinyScript [13]

The language is small compared to general-purpose languages, even to a parsimonious language like C (Table II), therefore the cognitive effort required to learn the syntax of the language is reduced. The complexity of SEAL syntax is comparable to TinyScript and nesC, although the user of nesC has to know C as well.

Source code metric comparison. Initially, four test applications were implemented:

- *Empty* – an application with no user logic;
- *Sense & print* – periodically sample sensors (light, humidity, and temperature) and send results to serial port;
- *Sense & send* – same as *Sense & print*, except that sensor values are sent to network and stored in external flash memory;
- *SAD* – a real-world application for environmental monitoring, conceptually similar to the third application (Section V-C).

We used two metrics: lines of code and cyclomatic complexity. The latter is essentially the measure of the branches in the control flow of a program [12]. High cyclomatic complexity implies convoluted, hard-to-understand branching. Line count also gives some suggestions how easily the code can be understood, especially if the languages have similar levels of abstraction.

The results are given in Table III and Table IV. As expected, SEAL source code is much shorter (at least 4 times, more for complicated applications) than that of C and nesC (which is especially verbose).

Languages that make event-handing explicit (TinyScript and nesC) unsurprisingly have noticeably higher cyclomatic complexity.

TinyScript and BASIC source line count is similar to SEAL. However, they are more complex, especially TinyScript, which utilizes several additional concepts: variable declarations, array element access, data types, and explicit usage of timers.

In general, *there is no case in which SEAL would score lower than any of the competitors in either of the metrics.*

Data processing support. For data processing SEAL has several dozen built-in functions. The application in Listing 2 demonstrates their usage: it calculates

	SEAL	BASIC	TinyScript	C	nesC
Empty	**0**	0	0	1	1
Sense & print	**1**	1	2	2	5
Sense & send	**1**	1	2	2	23
SAD	**2**	n/a	n/a	4	n/a
Exercise 1	**1**	1	4	2	2
Exercise 2	**2**	2	4	4	5
Exercise 3	**3**	3	4	4	5
Exercise 4	**1**	3	3	4	6

Table III: Cyclomatic complexity comparison

	SEAL	BASIC	TinyScript	C	nesC
Empty	**0**	0	0	3	6
Sense & print	**4**	6	6	15	49
Sense & send	**5**	9	7	42	179
SAD	**14**	n/a	n/a	185	n/a
Exercise 1	**1**	5	8	8	24
Exercise 2	**4**	4	8	13	48
Exercise 3	**5**	5	11	14	34
Exercise 4	**4**	11	16	22	56

Table IV: Lines of code comparison

the absolute difference of two light sensor readings, averages last 10 readings, and outputs the result. The code is understandable by nonprogrammers (Section V-D). Equivalent TinyScript and SensorBASIC applications are 10–20 lines long and contain much more details.

Listing 2: Calculating the average difference

```
1 define Diff difference(TotalSolarRadiation,
      PhotosyntheticRadiation);
2 define AverageDifference average(take(Diff, 10));
3 read AverageDifference, period 500ms;
4 output Serial;
```

Network-programming support. SEAL also facilitates WSN programming by automatically generating code for different roles in the network. In particular, SAD application requires the user to write the code not only for the mote, but also for the base station role and two data-forwarder roles. The code is short, but has to be placed in 6 additional files. The SEAL compiler generates source code for each of these roles automatically.

B. Efficiency and feasibility evaluation

We measured binary code size and static RAM usage of the test applications for TelosB and SM3 platforms (SM3 is a custom MSP430-based hardware platform used in SAD application). MansOS SVN revision 918 and *msp430-gcc* 4.5 was used.

Results. Compared with native implementations in MansOS, SEAL has less than 40 % resource usage overhead, excluding the empty application (Fig. 4, Table V). Compared with TinyOS on TelosB, SEAL actually demonstrates better results. One reason lies in its declarative nature: SEAL allows to declare system's behavior and policies at *compile-time*, avoiding any runtime overhead. In sum, SEAL is certainly feasible on very low-power microcontrollers.

	Code memory		RAM	
	TelosB	SM3	TelosB	SM3
Empty	75.8 %	26.0 %	7.0 %	2.1 %
Sense & print	21.1 %	7.1 %	29.9 %	15.0 %
Sense & send	23.8 %	16.7 %	39.1 %	9.8 %
SAD	n/a	11.6 %	n/a	4.8 %

Table V: SEAL code memory and RAM usage overhead

C. Applicability evaluation

SEAL is tuned specifically for sense-and-send type of applications. However, is can be used for other application archetypes a well (though it may not be the best choice for all of them!). The versatility of SEAL is demonstrated by the fact that it supports both periodic and event driven sampling and data transmission. It has support for actuation, interactivity (queries can be defined and handled), data interpretation (using predefined functions and their combinations), and data aggregation across nodes.

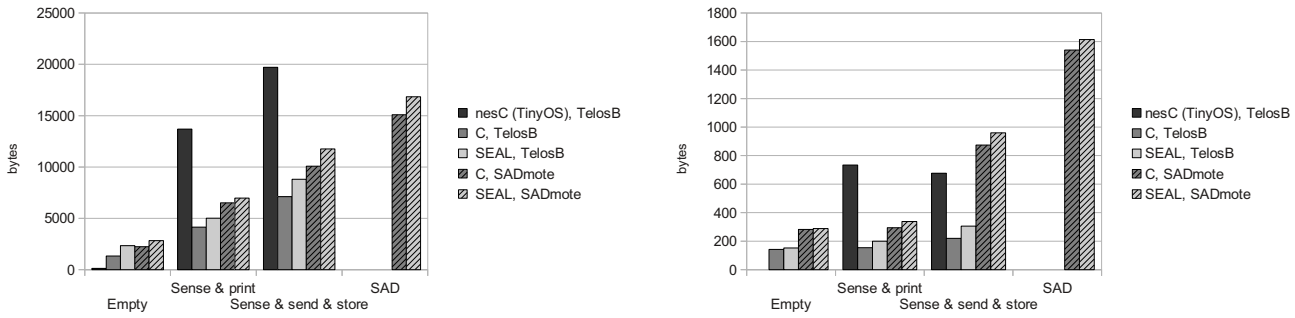
In this paper we present a single application example. More examples and documentation are available at [4].

A well-known WSN scenario. We now show how SEAL can be applied to a scenario described in Werner-Allen *et al.* [24]: sensor network deployed on an active volcano. The network performs high-rate (100 Hz, multiple channel) seismoacoustic monitoring and detects events of interest (e.g. small earthquakes). Since the sampling rate is so high, it is not feasible to transmit all the data to the base station. Instead, triggered data collection is used: data is downloaded from each sensor device only after a significant earthquake or eruption.

The application logic (Listing 3) consists of three parts. First, the application samples acoustic and seismic sensors and stores the data locally (lines 8–11). Second, event detection algorithm is continuously run, as described in [24] (lines 13–20). Node detects an event when the ratio between two exponentially weighted moving average (EWMA) functions becomes large enough. In this case the node sends notification to the base station. If high-enough number of nodes has detected an event, the base station issues data collection query. Third, the node handles data collection query command (lines 22–28). The command includes a timestamp. Data from the interval [timestamp; timestamp + 60 seconds] is sent back to the base station.

Listing 3: Seismoacoustic monitoring

```
1 const COMMAND_EVENT_DETECTED 1;
2 const COMMAND_REQUEST_DATA 2;
3 const DATA_COLLECTION_INTERVAL 60s;
4 const EWMA_COEFF_1 0.15;
5 const EWMA_COEFF_2 0.2;
6 const EVENT_DETECTION_THRESHOLD 0x1234;
7
8 read AcousticSensor, period 10ms;
```



(a) Code memory usage

(b) RAM usage

Figure 4: Application binary code size and RAM usage comparison

```
 9 read SeismicSensor, period 10ms;
10 output File (AcousticSensor, SeismicSensor,
11        Timestamp), name "SensorData.bin";
12
13 define CombinedSensor sum(AcousticSensor,
       SeismicSensor);
14 define EventDetectionFunction difference(
15     EWMA(CombinedSensor, EWMA_COEFF_1),
16     EWMA(CombinedSensor, EWMA_COEFF_2));
17 read EventDetectionFunction;
18 when EventDetectionFunction >
       EVENT_DETECTION_THRESHOLD:
19     output Network (Command), command
           COMMAND_EVENT_DETECTED;
20 end;
21
22 read RemoteCommand;
23 read RemoteTimestamp;
24 when RemoteCommand == COMMAND_REQUEST_DATA:
25     output Network, file "SensorData.bin",
26         where RemoteTimestamp >= Timestamp
27         and RemoteTimestamp <= add(Timestamp,
               DATA_COLLECTION_INTERVAL);
28 end
```

A real-world use case. For environmental monitoring in precision agriculture use case (referred as "SAD") we deployed a 19 node multi-hop network with motes in multiple roles, programmed exclusively with SEAL. The total code size is 14 lines (see also Section V-A). The application is conceptually similar to the *Sense & send* application, but with several practical extensions including: blinking a LED while sensors ar read (so a quick visual indication shows that the mote is still working); selecting a specific light sensor based on mote's address, as not all motes have the same sensors; and tweaking several low-level options and optimizations.

D. Empirical usability evaluation

In order to evaluate the usability of our software we conducted a preliminary user study. We adopted methodology and exercises from the SensorBASIC user study [16]. For evaluation we recruited computer science undergraduate students with no WSN programming experience (the "intermediate programmers"), as well as people from other domains with little or no programming experience (the "novice programmers"). The participants were given access to computers with IDE installed

	Ex. 1	Ex.2	Ex.3	Ex.4	Total, %
TinyScript, intermed.	3/3	3/3	3/3	3/3	100 %
SEAL, intermed.	3/3	3/3	3/3	3/3	100 %
SEAL, novice 1	3/3	0/3	2/3	n/a	55.6 %
SEAL, novice 2	3/3	2/3	2/3	2/3	75 %

Table VI: Exercise completion success rate

(SEAL or TinyScript, depending on their group) and with attached sensor devices that were equipped with light, temperature and humidity sensors.

The first three exercises asked to write code for specific applications (adopted from [16]):

1) Blink a LED with 2 second period (Listing 4);
2) Send a message to the base station if the light sensor is covered (Listing 5);
3) Turn on the LED if and only if the light sensor is covered (Listing 6).

The fourth exercise asked to "describe the meaning of the code in Listing 2".

Listing 4: Solution to exercise 1

```
1 use RedLed, period 2s;
```

Listing 5: Solution to exercise 2

```
1 read Light;
2 when Light < 100:
3    use Print, format "No␣light", out Radio;
4 end
```

Listing 6: Solution to exercise 3

```
1 when Light < 100:
2     use RedLed, on;
3 else:
4     use RedLed, off;
5 end
```

In total there were 12 participants: six computer science students, three physicists, two agricultural scientists, and one electrical engineer; seven were undergraduate students, five – graduate students or scientific staff. The results of the study are given in Table VI.

The study demonstrates that novice programmers can use SEAL and succeed in more than half of cases. Our results are comparable with outcomes from [16] (54 %,

45 %, and 46 % in the group with no programming experience, 100 %, 89 % and 67 % with limited programming experience). The user study from [7] (three different exercises) similarly reported 40.6 % average success rate (although WASP and WASP2 had 80.6 % average).

VI. Conclusion

We address the problems faced by novice programmers as sensor network application developers. Using lessons from novice programming theory, we build SEAL, a WSN application description language. SEAL avoids several typical, but hard-to-learn options, allows to write compact and simple code, is compiled to efficient binary code (only up to 24 % code memory and up to 40 % RAM usage overhead), and is suitable for typical WSN application scenarios.

Acknowledgments

This work was supported by European Social Fund, project No. 2011/0054/1DP/1.1.2.1.2/11/IPIA/VIAA/-002. We would like also to thank all the test subjects for cooperation.

References

[1] ISO/IEC 9899:1999 – Programming languages – C, 1999.

[2] Blockly: A visual programming editor. http://code.google.com/p/blockly, 2012.

[3] PLY (Python Lex-Yacc). http://www.dabeaz.com/ply/, 2012.

[4] The SEAL programming language. http://open-sci.net/wiki/seal, 2012.

[5] PySense: A language to program wireless sensor network at once. http://code.google.com/p/pysense/, 2013.

[6] E. Avilés-López and J. García-Macías. TinySOA: a service-oriented architecture for wireless sensor networks. *Service Oriented Computing and Applications*, 3(2):99–108, 2009.

[7] L. Bai, R. Dick, and P. Dinda. Archetype-based design: Sensor network programming for application experts, not just programming experts. In *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*, pages 85–96. IEEE Computer Society, 2009.

[8] F. Casati, F. Daniel, G. Dantchev, J. Eriksson, N. Finne, S. Karnouskos, P. Montera, L. Mottola, F. Oppermann, G. Picco, et al. Towards business processes orchestrating the physical enterprise with wireless sensor networks. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 1357–1360. IEEE, 2012.

[9] A. Elsts and L. Selavo. A User-Centric Approach to Wireless Sensor Network Programming Languages. In *SESENA '12: Proceedings of the 3rd Workshop on Software Engineering for Sensor Network Applications*, pages 29–30, New York, NY, USA, 2012.

[10] D. Gay, P. Levis, D. Culler, and E. Brewer. nesC 1.3 Language Reference Manual, 2009.

[11] P. Hudak. Domain-specific languages. *Handbook of Programming Languages*, 3:39–60, 1997.

[12] C. Jones. Software metrics: good, bad and missing. *Computer*, 27(9):98–100, 1994.

[13] P. Levis. The TinyScript language: A Reference Manual. http://www.cs.berkeley.edu/~pal/mate-web/files/tinyscript-manual.pdf, 2004.

[14] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. TinyDB: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30:122–173, March 2005.

[15] J. Maloney, K. Peppler, Y. Kafai, M. Resnick, and N. Rusk. Programming by choice: Urban youth learning programming with Scratch. *ACM SIGCSE Bulletin*, 40(1):367–371, 2008.

[16] J. Miller, P. Dinda, and R. Dick. Evaluating a BASIC approach to sensor network node programming. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 155–168. ACM, 2009.

[17] L. Mottola and G. Picco. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Computing Surveys (CSUR)*, 43(3):19, 2011.

[18] L. Neal. A system for example-based programming. In *ACM SIGCHI Bulletin*, volume 20, pages 63–68. ACM, 1989.

[19] J. Pane and B. Myers. Usability issues in the design of novice programming systems,. Human-Computer Interaction Institute Technical Report CMU-HCII-96-101, 1996.

[20] A. Robins, J. Rountree, and N. Rountree. Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2):137–172, 2003.

[21] R. B. Smith. SPOTWorld and the Sun SPOT. In *Proceedings of the 6th international conference on Information processing in sensor networks*, IPSN '07, pages 565–566, New York, NY, USA, 2007. ACM.

[22] G. Strazdins, A. Elsts, K. Nesenbergs, and L. Selavo. Wireless sensor network operating system design rules based on real world deployment survey. *Accepted for publication in Journal of Sensor and Actuator Networks*, 2013. ISSN 2224-2708.

[23] A. Van Deursen and P. Klint. Little languages: little maintenance? In *SIGPLAN Workshop on Domain-Specific Languages*. Citeseer, 1997.

[24] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh. Fidelity and yield in a volcano monitoring sensor network. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 381–396. USENIX Association, 2006.