A User-Centric Approach to Wireless Sensor Network Programming Languages

Atis Elsts and Leo Selavo
Institute of Electronics and Computer Science
Riga, Latvia
{*atis.elsts,leo.selavo*}*@edi.lv*

***Abstract*—Wireless sensor networks is likely to remain a niche technology until an easy-to-use programming interface for a broad range of users is offered. In this paper we sketch SEAL, a domain-specific language for WSN application description. The target audience of our work is domain experts with limited programming skills. We believe this user group previously has not received enough attention from WSN programming language designers.**

***Keywords*-programming languages, sensor networks**

I. Introduction

Wireless sensor networks (WSN) is a technology that was envisioned as a tool for a broad range of applications and target audiences. This vision has yet to become reality, partially because programming WSN is so difficult.

Consider a specific example: in 2011 our WSN group at the University of Latvia did a small pilot deployment for a precision agriculture application [1]. Once the network was deployed, we discovered that agriculture scientists would like to configure and control it on their own. Unfortunately, at the moment that was not possible, since they had no C programming knowledge. We believe similar situations are encountered frequently, and therefore require a solution.

People who are interested in WSN as a tool include biologists, geologists, agriculture and other scientists, and simply tech-savvy enthusiastic amateurs. These users and potential users are characterized by:

- limited skills in programming and algorithm design;
- no experience with model-driven development (MDD), UML and similar software engineering (SE) tools;
- very limited knowledge of electronics.

On the other hand, they are often technically literate and interested in tweaking various parameters of the WSN. They could function as maintainers of the network, if only the right tools were given to them.

As an analogy, three user groups for a PC operating system such as UNIX can be identified with respect to their technical skills:

- system programmers, who write code for the OS and programs running directly on top of it;
- everyday users, who treat the OS merely as their working environment;
- power users and administrators, who modify configuration files, use advanced control tools, and occasionally write simple scripts.

So far WSN community has largely ignored the existence of this third group of potential WSN users – ones that are in-between the everyday users who view WSN merely as a transparent data source, and WSN programming professionals who have designed and deployed the network. The right kind of software abstractions for this emerging third group of WSN users is a niche that still remains to be filled.

This position paper is organized as follows: in section II we sketch the offered solution, while section III addressed selected points of criticism to our approach.

II. The language

SEAL was designed with the above-described third group of user in mind. Its syntax features three kinds of descriptive statements (for sensors, actuators and system outputs), conditional statements, and syntax for describing state variables. A few SEAL application examples follows.

Listing 1 Blink: toggles a LED every second

```
use RedLed, period 1000ms;
```

Listing 2 Sense-and-send: read light & humidity sensor values and send to radio

```
read Light, period 2s;
read Humidity, period 6s;
output Radio;
```

Listing 3 Event detection with hysteresis: detect a critical temperature range

```
state temperatureCritical false;
when Sensors.Temperature > 50C:
    set temperatureCritical true;
when Sensors.Temperature < 40C:
    set temperatureCritical false;
when temperatureCritical: // blink red LED
    use RedLed, period 100ms;
```

SEAL compiler functions by translating SEAL code to C code, which is then compiled natively for a specific architecture. SEAL features an integrated development environment (IDE) that can be used either to write SEAL code or or generate it by using GUI. The IDE includes easy-to-access application examples. At the moment SEAL is in experimental stage, with the compiler being partially implemented.

III. Analysis

Objection: similar solutions already exist. *Response:* several high-level WSN programming languages and techniques have been proposed [2], too numerous to name them all here. When comparing them to SEAL, we can see that solutions such as *TinyDB* allow to view whole sensor network as relational database, but lack any node-centric programming options, which are useful in the real world. Solutions such as *Regiment* and *Snlog* use concepts from functional programming and Prolog repectively, which might be hard to grasp for people without strong background in computing. Solutions such as *Kairos* and *Pleiades extend* programming langagues (Python and C respectively) for WSN purposes, thus giving *embrass de richess* for novice programmers. None of these have declared ease-to-use for computer science (CS) non-professionals as their primary design goal. Perhaps [3] is the most similar work to ours, but the authors did not mention specific practical applications as their motive, and at the same time are more restrictive in the class of applications they claim to support (e.g. no heterogeneous networks). makeSense [4] is a newer development similar to ours, but their target audience differs: it is users familiar with business process modelling languages.

Objection: WSN are inherently complex and their programming and maintenance should be left to CS professionals. *Response:* so far this has been the taciturn view in WSN community, but we believe it was caused by lack of software abstractions accessible to a broader audience.

WSN programming can be separated in two parts: the more and the less complex. For example, it's easy to describe application logic for sense-and-send or event detection, as well as basic data processing (local data aggregation & filtering), while it's complex to describe distributed algorithms such as multihop routing. SEAL limits itself to the former and lets the OS and middleware handle the latter.

Objection: SEAL departs from best existing SE practices such as MDD. *Response:* the target audience is unaware of MDD, unwilling to learn UML, and is likely to find either writing small code snippets or generating them by button clicking more intuitive.

Objection: SEAL is OS-dependent, since the C code has to rely on a WSN OS for common functionality. *Response*: true, but its easily portable. As proof-of-concept we support code generation for two WSN OS: MansOS [5] and Contiki.

Objection: using SEAL is not as efficient as manually writing C or NesC code. *Response:* while automatically generated code can never be better than the best code written by hand, we note that the C code generated by SEAL compiler uses event-based control flow, and supports small duty-cycles. The sample applications (Blink, Sense-and-send, and Event detection) presented here have 0.037%, 0.627%, and 0.701% duty cycles respectively (estimated on Tmote Sky at 4MHz, without MAC protocol). As for binary code sizes, they are 1360 bytes for Blink and 7628 for Sense & send (evaluated on MansOS).

Objection: the solution is too restrictive, too application-specific. *Response:* while SEAL is developed with a few specific applications in mind, it's suitable for many use cases. As [3] shows, this class of application is prevalent in WSN deployments. Besides, SEAL is not a complete and integrated WSN framework and therefore is easily adaptable to other classes of applications.

Objection: SEAL does not provide a complete framework for WSN programming. *Response:* The objective of SEAL is to be one of blocks in a building-block approach to WSN programming. Together with SEAL IDE and MansOS, SEAL is a complete SE solution for WSN. The application logic is implemented in SEAL; MansOS itself functions as the run-time layer; and MansOS management protocol functions as the macroprogamming layer, utilizing the built-in support for run-time reprogramming.

We believe that a monolithic integrated solution is a bad idea, because it leads to reduced flexibility, therefore integration should be delayed as late as possible. A heavily integrated solution is difficult to maintain and update as well. Case in point is TinyDB: despite its popularity for TinyOS 1, there still is no TinyDB for TinyOS 2.x.

IV. Conclusion

We have identified a new, emerging class of WSN users and speculated on the kind of software they are going to require. We have outlined the main features of our scripting language and argued that it allows these users to create efficient WSN programs for a broad class of typical WSN applications.

Acknowledgment

This work was supported by European Social Fund grant Nr. 2009/0219/1DP/1.1.1.2.0/APIA/VIAA/020.

References

[1] A. Elsts, R. Balass, J. Judvaitis, R. Zviedris, G. Strazdins, A. Mednis, and L. Selavo, "Sadmote: A robust and cost-effective device for environmental monitoring," vol. 7179, pp. 225–237, 2012.

[2] L. Mottola and G. P. Picco, "Programming wireless sensor networks: Fundamental concepts and state of the art," *ACM Comput. Surv.*, vol. 43, no. 3, pp. 19:1–19:51, Apr. 2011.

[3] 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*. IEEE Computer Society, 2009, pp. 85–96.

[4] "makeSense Project," http://www.project-makesense.eu.

[5] G. Strazdins, A. Elsts, and L. Selavo, "MansOS: Easy to Use, Portable and Resource Efficient Operating System for Networked Embedded Devices," in *Proc. SenSys'10*, 2010.