

Towards intelligibility in sensor programming

Doina Bucur

INnovation Centre for Advanced Sensors and Sensor Systems (INCAS³), The Netherlands
doinabucur@incas3.eu

Motivation and Summary

Sensors and sensor-rich embedded systems are eminently applicable for autonomous monitoring and control in non-commercial, home or professional environments, besides their increasing use in medical and ecological settings, or mainstream use in industrial control. Having end users realize, deploy and interact with sensor-based embedded systems is conditioned, among others, by the availability of tools and techniques for *end-user software engineering* [8] specialized for these embedded systems.

At their application logic, embedded sensor applications are inherently *context aware*: they most often implement context awareness by a deterministic *state- and rule-based decision logic* to trigger an output actuation out of a set of inputs from the environment. The logic can thus be seen as a finite-state machine, with rules transitioning between states, given an input from the environment. Since in embedded sensor systems this state- and rule-based decision model is *distributed* (i.e. a particular sensor node triggers an output with taking input from neighbour nodes) this gives that an application's input set may be large, and the decision model complex.

On the other hand, embedded sensors are limited in computational resources. Thus, the programming of these applications is done close to the hardware; microcontroller programmers avoid higher-level languages altogether, as C is perceived as an inherently efficient language due to it being low-level. This efficiency, however, limits expressivity and consequently intelligibility of the application. This last fact also explains why embedded software engineering is still in the domain of professional programmers; research on *end-user embedded software engineering* is yet to fully develop.

In the long term, of interest are “[analysis] tools for end-user programmers which are integrated with the users’ [software development] and are incremental in their feedback” [8]. Such tools may, e.g., automatize the task of collecting relevant information about the program logic from program source, for the purpose of program maintenance (such as

motivated by [9]), including automatically extracting explanations about the context-awareness logic, on the lines of the Intelligibility Toolkit [14].

We state that, currently, embedded systems lack the needed integration between analysis tools and the development process, as well as the expected incremental feedback for end users. Besides this fact however, relevant analysis techniques have been developed for the automatic collection of information about the program logic, including a degree of extracting explanations from embedded program source. In the following:

- (i) We look at the state of the art in programming methods for embedded sensors, in preparation for then over-viewing program analysis tools.
- (ii) Finally, we survey tools for the analysis of these programs, and show that a limited degree of automatically generating *explanations* [13] and other relevant information about the program's behaviour is covered by some program-analysis tools now principally aimed at testing or verifying the correctness of application logic.

Programming for embedded sensor applications

Take a sensor node; when programmed in component-based **nesC** [5] for the mainstream TinyOS operating system [6, 11], an action to be performed at the single occurrence of a contextual event (e.g. an incoming sensor reading or network packet) is straightforwardly coded as a **nesC event**, then possibly continued with a *task* as a deferred procedure call. Both events and tasks are non-blocking—a simple concurrency model which is sufficient for I/O-centric programming. The threaded, synchronous APIs with blocking operations of Contiki protothreaded C [16] and TinyOS TOSThreads [7] improve somewhat on the friendliness of the concurrency model. For all these programming methods:

- Little middleware support exists to express more than a rule-based decision-model type; machine-learning algorithms are rarely, as of yet, implemented for these embedded devices. This fact simplifies the extraction of explanations.
- On the other hand, due to the low-level programming style described above, there can be no intelligible encoding of e.g. decision rules over multi-variate contextual input; these decision rules must be ‘broken’ into individual rules for single-variate input, a fact which results in complicated logic. Given this, exposing application logic post-implementation is a difficult task.

Table 1: Runtime and compile-time tools for program analysis of *rule-based* applications for embedded sensor nodes, with pointers to the types of explanation they can offer

	Explanation types (from [13])	Analysis tool (language or platform)	Application scope
At compile-time	<i>Input, Output</i> , Boolean <i>What if, Why</i> , e.g. If the temperature reading is <i>X</i> , will the LED state become <i>Y</i> ? (Yes/No; if No, with program trace showing why)	<i>TOS2CProver</i> [2] (gcc for TelosB sensor platforms)	a single sensor node
	<i>Input, Output</i> , Boolean <i>What if, Why</i> , e.g. Can the sensor ever send message <i>Z</i> on the radio, regardless of the state of the network? (Yes/No, with program trace showing why)	<i>T-Check</i> [12] (nesC for TinyOS) and <i>KleeNet</i> [15] (Contiki C)	a network of sensor nodes
	Limited <i>How to</i> (see Fig. 1), e.g. What is the sequence of environmental events needed to put this node into a particular state?	<i>FSMGen</i> , finite-state machine extraction [10] (nesC for TinyOS)	a single sensor node
At runtime	<i>What</i> , e.g. The program encountered an operational exception; what are the truth values of relevant assertions over relevant program variables now?	<i>Safe TinyOS</i> , <i>Neutron</i> [4, 3] (nesC for TinyOS)	a network of sensor nodes
	<i>Why</i> , e.g. It is reported that a driver API has been used incorrectly; how did this happen?	<i>Interface contracts</i> [1] (nesC for TinyOS)	a single sensor node

1. REFERENCES

- [1] W. Archer, P. Levis, and J. Regehr. Interface contracts for TinyOS. In *Proceedings of the international conference on Information Processing in Sensor Networks (IPSN)*, pages 158–165. ACM, 2007.
- [2] D. Bucur and M. Z. Kwiatkowska. Software verification for TinyOS. In *IPSN*, pages 400–401, 2010.
- [3] Y. Chen, O. Gnawali, M. Kazandjieva, P. Levis, and J. Regehr. Surviving sensor network software faults. In *Proceedings of the Symposium on Operating System Principles (SOSP)*. ACM, 2009.
- [4] N. Coopriider, W. Archer, E. Eide, D. Gay, and J. Regehr. Efficient memory safety for TinyOS. In *Proceedings of the conference on Embedded Networked Sensor Systems (SenSys)*, pages 205–218. ACM, 2007.
- [5] D. Gay, P. Levis, and D. Culler. Software design patterns for TinyOS. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on languages, compilers, and tools for embedded systems (LCTES)*, pages 40–49. ACM, 2005.
- [6] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. *SIGPLAN Not.*, 35(11):93–104, 2000.
- [7] K. Klues, C.-J. Liang, J. Paek, R. Musäloiu, R. Govindan, A. Terzis, and P. Levis. TOSThreads: Safe and Non-Invasive Preemption in TinyOS. In *Proceedings of the ACM conference on Embedded Networked Sensor Systems (SenSys)*. ACM, 2009.
- [8] A. J. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, B. Myers, M. B. Rosson, G. Rothermel, M. Shaw, and S. Wiedenbeck. The state of the art in end-user software engineering, 2011. Accepted for publication in ACM Computing Surveys.
- [9] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Trans. Softw. Eng.*, 32:971–987, December 2006.
- [10] N. Kothari, T. Millstein, and R. Govindan. Deriving state machines from TinyOS programs using symbolic execution. In *Proceedings of the international conference on Information Processing in Sensor Networks (IPSN)*, pages 271–282. IEEE, 2008.
- [11] P. Levis, D. Gay, V. Handziski, J.-H. Hauer, B. Greenstein, M. Turon, J. Hui, K. Klues, C. Sharp, R. Szewczyk, J. Polastre, P. Buonadonna, L. Nachman, G. Tolle, D. Culler, and A. Wolisz. T2: A second generation OS for embedded sensor networks. Technical Report TKN-05-007, Technische Universität Berlin, 2005.
- [12] P. Li and J. Regehr. T-Check: bug finding for sensor networks. In *Proceedings of the 9th International Conference on Information Processing in Sensor Networks (IPSN)*, pages 174–185. ACM, 2010.
- [13] B. Y. Lim and A. K. Dey. Assessing demand for intelligibility in context-aware applications. In *Proceedings of the 11th international conference on Ubiquitous computing, Ubicomp '09*, pages 195–204, New York, NY, USA, 2009. ACM.
- [14] B. Y. Lim and A. K. Dey. Toolkit to support intelligibility in context-aware applications. In *Proceedings of the 12th ACM international conference on Ubiquitous computing, Ubicomp '10*, pages 13–22, New York, NY, USA, 2010. ACM.
- [15] R. Sasnauskas, O. Landsiedel, M. H. Alizai, C. Weise, S. Kowalewski, and K. Wehrle. KleeNet: discovering insidious interaction bugs in wireless sensor networks before deployment. In *Proceedings of the 9th International Conference on Information Processing in Sensor Networks (IPSN)*, pages 186–196, 2010.
- [16] The Contiki Operating System. <http://www.sics.se/contiki/>; accessed 02-2011.