Intelligible TinyOS Sensor Systems: Explanations for Embedded Software

Doina Bucur

Innovation Centre for Advanced Sensors and Sensor Systems (INCAS^3),
The Netherlands
doinabucur@incas3.eu

Abstract. As embedded sensing systems are central to developing pervasive, context-aware services, the applications running on these systems should be intelligible to system programmers and to users. Given that sensor systems are programmed in low-level languages, manually writing high-level explanations about their decision model requires knowledge about the system architecture, and is error-prone. We explore the possibility of extracting explanations which are small and expressive, but still preserve bit-level accuracy when needed. We contribute a tool which automatically and soundly generates compact, graphical explanations from sensor software implementation at compile-time. We base our algorithm on the techniques of (i) finite-state machine model extraction from software as used in model checking, and (ii) abstraction of program execution traces. We experiment with extracting explanations from heavyweight, low-level TinyOS applications for a mainstream sensor platform.

1 Introduction

Embedded sensor applications are inherently *context aware*; like other pervasive systems (surveyed in [1]), they most often implement a deterministic *rule-based decision logic* to trigger an output actuation out of a set of input events from the environment. For example, the classical TinyOS [2, 3] sensor application signals its state on LEDs, and outputs a network message every set number of samples read from an on-board sensor. The control flow of such an application consists of a main thread and a set of *event handlers* to which context may be switched to at any time (i.e., a timer has fired, a new sensor sample is available, a network message has finished sending); the main thread and each event handler are implemented essentially as a series of `if/else` statements.

On the other hand, the programming of these sensor applications is done close to the hardware; programmers of microcontroller compiler suites avoid higher-level languages, since C is as an inherently powerful and efficient system programming language. A natural consequence is that this efficiency in programming limits expressivity and consequently *intelligibility* of the resulting application. This last fact explains why embedded software engineering is still in the domain of professional programmers; intuitive programming abstractions

and middleware [4] for embedded systems are still in their infancy, and research on *end-user software engineering* [5] for embedded systems is yet to develop.

To partly reconcile the need for high-level intelligibility with the low-level programming of embedded context-aware applications, one needs tools for software analysis which would automatically (i) document the program for users, and (ii) provide feedback to programmers, either during or post-development. 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 (as motivated by [6]). In particular, these analysis tools should automatically extract *explanations* about the context-awareness decision logic, with explanation types as categorized by e.g. [7, 1]. We focus on this latter task.

We make the technical distinction between extracting explanations about the program context-aware behaviour either statically (i.e., offline at compile-time, from program source), or at runtime (i.e., during the sensor application's execution, in the field). Given their definition, some of the explanation types from [7, 1] suit mostly a static approach, e.g.: 'What are the environmental *Inputs* to this system?', 'What are the *Outputs*?, '*How to* drive this system into reaching this Output?', and '*What* would the Output be *if* this were the Input?'. On the other hand, other explanation types suit a runtime approach (e.g., '*What* is the current state of the system?'), and the rest a combination. In this, we look at the static approach, and at extracting the four corresponding explanation types.

Our contribution. We show that program analysis algorithms and tools from the field of *software model checking*[1] are a solution to the problem of automatically and soundly generating explanations of context-aware behaviour from TinyOS application source. Our tool chain, *TOSExp*, takes as input a TinyOS application and information about the sensor platform, and:

- Generates *Input* and *Output* explanations [7], i.e., (i) that set of external events the application takes as inputs, and (ii) which system interfaces are used to output signals, respectively.
- Generates compact *What if* and *How to* explanations [7], i.e., state machines showing an abstracted model of program logic: the model shows (i) *what* happens in the logic *if* certain *Input* values are imposed, and (ii) *how to* reach certain *Output* values without dictating the *Inputs*.

The former is done through an inspection of the program for (a) functions which are registered as event handlers[2]—these point directly to the input events, and (b) those microcontroller peripherals[3] which are written by the application—

---

[1] Software model checking is also known as *formal software verification.*

[2] In operating systems, event or interrupt handlers are functions whose execution is triggered by an hardware interrupt. At the occurrence of an *interrupt request* (IRQ), the microprocessor switches context to the interrupt handler.

[3] Embedded systems are built around a microcontroller: a chip integrating a microprocessor, RAM memory and registers, and input/output ports. The latter are used to read from or command system components (on-board sensors or actuators) external to the microcontroller.

these point to the program's outputs. For the latter, we make use of a tool suite composed of: (i) a language translator from embedded C to ANSI-C from the previous TOS2CProver (TinyOS to CProver) tool [8], together with (ii) CBMC (C Bounded Model Checker) [9], a state-of-the-art verification tool for ANSI-C programs, from the CProver suite [10]. Embedded TinyOS applications are translated to ANSI-C by TOS2CProver, and *Input* values to the application may be specified. A *predicate* over the state of a program *Output* serves as a specification, and both program and specification are inputted to CBMC. The counterexample offered by CBMC is abstracted into a compact model of how input states lead to output states. We experiment with generating these types of explanations over standard TinyOS applications programmed in embedded C, as built from the nesC [11] source code by the TinyOS compiler; we include applications built for the mainstream TelosB [12] sensor platform.

In summary. In what follows, Section 2 gives background information on sensor platforms, embedded software, and on formal analysis techniques for C programs. Section 3 presents our algorithms, with a focus on generating *What if* and *How to* explanations, and exemplifies experimental results. Sections 4 and 5 overview related work and conclude, respectively.

2   Background: Sensor platforms, embedded software, and predicate abstraction for C

2.1   Sensor platforms and embedded software

A variety of platforms exist for wireless sensor nodes; TelosB [12] includes a MSP430 microcontroller [13], whose *input/output pins* connect to the on-board three LEDs, five sensors (some indirectly, through an analog-to-digital converter), and the radio and wired serial communication interfaces. These pins are mapped to registers, which form *peripheral ports*—e.g., microcontroller pins 48 to 50 connect to the three on-board LEDs, and are mapped to three bits in a register of peripheral port 5, P5OUT.

The software can access these peripherals by direct memory access; all on-board memory, including these ports, are mapped into a unique address space, with peripherals at low addresses, `0x10` to `0xFF`. Thus, for the case of software written in embedded C for the MSP compiler, `msp430-gcc` [14], the assignment `0x31 ^= 0x01 << 6;` toggles the yellow LED; this is because P5OUT resides at address `0x31` on this example microcontroller.

Finally, a C function declared with `__attribute((interrupt(..)))` is an interrupt handler. E.g., function `sig_ADC_VECTOR` is executed when a new sensor sample from the analog-to-digital converter is available.

2.2   TinyOS

For a complex embedded system such as TelosB, a programmer will program the microcontroller not in embedded C, but in a higher-level C dialect for a sensor

operating system. For the mainstream TinyOS [2, 3], the application is written in component-based *network embedded systems C*, nesC [15], as is the operating system itself. NesC is a more structured, I/O-centric, dialect of C, without memory allocation. Many system calls in TinyOS (e.g., sampling a sensor) are non-blocking; their completion is signalled to the application by an interrupt, whose handler may further post *tasks* to a *task queue* for further execution. Event handlers run with highest priority and may preempt tasks, which execute most of the program logic.

For an application, nesC components are wired together by the nesC compiler to form an OS-wide program; nesC is in fact designed under the expectation that a final, inlined embedded C program will be generated from all the necessary OS components. It is this inlined, OS-wide program for the TelosB platform that we analyze in this contribution.

2.3   Model extraction from software

Program analysis tools (such as model checkers) start by automatically extracting a mathematical *model* from the program given in input; this is essentially a nondeterministic finite-state machine consisting of a set of *states* and a set of *transitions* defined by a transition relation[4].
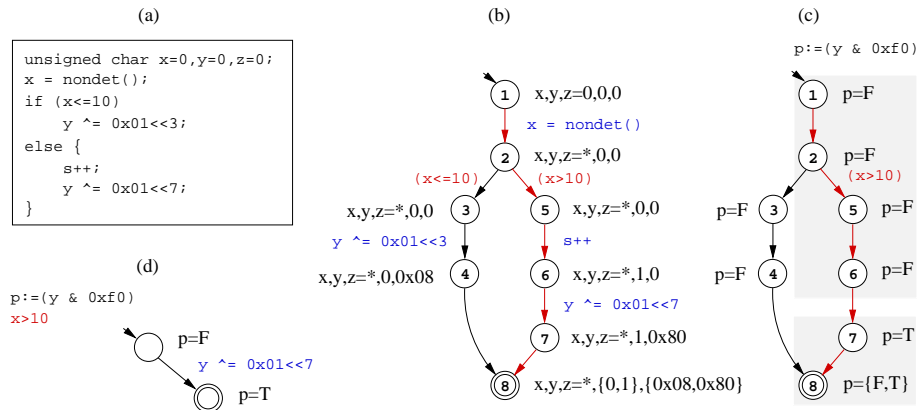

Fig. 1. Program fragment (a) with a bit-accurate model in (b). States are labelled with a valuation of the program's variables: in (b) state 7, `x` has a random value, `s` is 1, and `y` is 0x80; in (c) state 7, predicate `p` is true. Edges are labelled with corresponding C statements and conditions. (c) shows the truth values of the predicate `p` over the model's states, and (d) abstracts only that trace in whose final state `p` is true.

---

[4] Allowing for nondeterminism in models of software is particularly relevant for the case of embedded software, in which inputs are assigned unknown values from the environment.

In the particular case of software model checkers, the state of the program is defined as the valuation of the program's variables, while a transition corresponds to one basic program assignment. An example is given in Fig. 1(b); it models the C fragment from Fig. 1(a). We use this state-machine syntax as the syntax for our explanations in what follows.

2.4 Model checking and abstraction

Model checking is a technique for exploring the states in a mathematical model for conformance to a given specification. Take again the example in Fig. 1(b) and the predicate `p := (y & 0xf0)` as a specification; a model checker (such as [9]) would return, as proof that the program can reach a final state in which `p` is true, the program trace including the `x>10` branch and excluding the `x<=10` branch, as in Fig. 1(c).

It is such program traces that allow us to extract *What if* and *How to* program explanations in what follows. A final step consists in *abstracting* the program trace for readability: by only tracking the change in value of relevant program variables, and removing other variables altogether from the model (as in Fig. 1(d)), a compact program model is possible.

3 TOSExp: TinyOS explained

Fig. 2 overviews our program-analysis technique. TOSExp is based on our previous program translation and analysis tool, TOS2CProver [8], together with a C software model checker, CBMC [9].
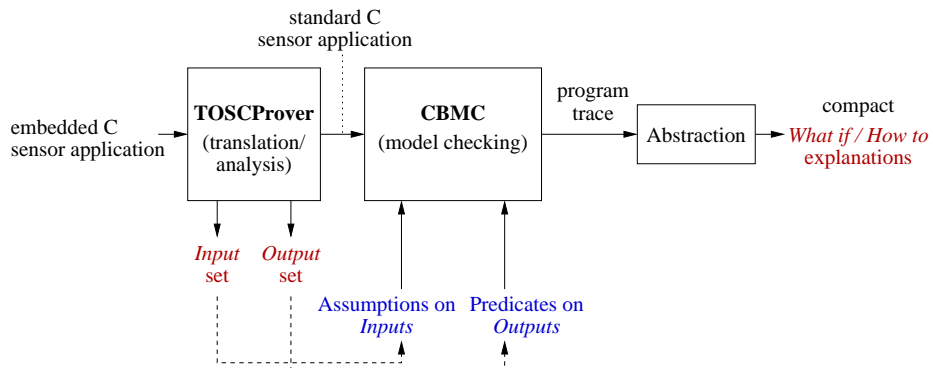

Fig. 2. The TOSExp tool chain. TOSCProver [8] analyses the sensor application for *Inputs* and *Outputs*, and further translates the program for use with CBMC. In order to obtain a *How to* explanation, we leave the *Inputs* nondeterministic; for *What if* explanations, the *Inputs* are given explicit values. Then, for both explanation types, we write a predicate over the values of *Outputs*, and run CBMC. The result is a program trace of input-output execution, which is finally abstracted into a compact form.

An inlined, TinyOS-wide embedded C sensor application (as overviewed in the background Subsections 2.1 and 2.2) serves as input to the tool chain. In the following, we experiment on demonstrative applications provided with TinyOS; *Blink* is a simple system-testing application which repeatedly toggles the state of the three on-board LEDs, with each toggle triggered by a timer alarm. *Sense* samples an on-board sensor, and displays part of the reading on the LEDs.

3.1 *Input* and *Output* explanations

In order to obtain the sets of *Inputs*, the sensor application is searched by TOS2CProver for the set of functions registered as interrupt handlers; it is this which provides the *Input* set. For *Blink*, the only functional interrupt handler is `sig_TIMERB0_VECTOR`; this gives that the *Input* set is simply {time}. For *Sense* and *Oscilloscope*, the *Input* sets are {sensors, time} and {sensors, time, network}, respectively. A dedicated hardware interrupt also exists to announce the software about a completed network send operation; thus, it is determined that the network is also an *Output* for *Oscilloscope*.

An application's other *Outputs* are also discerned through a TOS2CProver analysis step: out of all output peripheral registers characteristic to the TelosB platform, only some will be *assigned* values by the application, in the course of its execution after its booting and initialization phase. The three on-board LEDs are *Outputs* for *Sense* and *Blink*, as the three LED-controlling bits (bits 4, 5 and 6) [12, 13] in the 8-bit peripheral register P5OUT are modified by the program following an input timer event or an event signalling the sampling of a sensor. The same reasoning would hold if the sensor node had an on-board actuator, which would be managed by the software through another peripheral port. Thus, for *Blink, Sense* and *Oscilloscope*, the *Output* sets are {LEDs}, {LEDs} and {LEDs, network}, respectively.

This analysis step is sound: whenever an application does have an certain *Input* or *Output*, it will be included in the relevant set. On the other hand, whether or not the application does implement logic which makes effective use of a particular *Input* can only be determined by a *How to* explanation.

3.2 *What if* and *How to* explanations: Assigning the *Input*

For a *How to* explanation (which essentially asks how the application's environmental inputs could hypothetically be given values in order for the application to then reach a particular output), the *Inputs* are not assigned—it is the *Outputs* which are dictated values. Input values are left nondeterministic, and the model checker will explore traces leading from every possible input value to the particular, relevant output state. If such a program trace exists, it will return this trace as proof of *How to* application behaviour.

For a {time} *Input*, nondeterminism means that the main program will be interrupted by timer events, with each timer event arriving at an unpredictable point in time (i.e., for any local time on the system itself). This is again a sound approach: as TinyOS is not a real-time operating system, scheduling the timer to

fire an interrupt every half second will have a result which only approximates the desired behaviour. Technically, local system time is implemented by a clock-tick count stored in the 16-bit timer register; time is thus made nondeterministic by assigning an 'any value' to the timer register, and allowing the program to be explored as it unfolds from any initial value.

For a *What if* explanation, on the other hand, which asks what would the application do given particular environmental inputs, it is the *Inputs* which are assigned explicit values, and subsequently all reachable *Output* values (and traces to these) should be identified by our analysis—the solution to the latter is discussed in the next subsection. In what regards this assigning of explicit values to inputs, CBMC allows adding *assumption* statements in the C program; for a {sensor} *Input*, assigning a value to a sensor reading incoming from the analog-to-digital converter is done with a statement such as:

```
assume(ADC12MEM == {0x0001, [..]})
```

3.3   *What if* and *How to* explanations: Predicates on the *Output*

In order to extract both types of explanations using a model checker, we write boolean predicates over output values. For example, to find '*How to* turn LED 1 on' or '*What if* an amount of time passes' application behaviour, predicate `p`:

```
(P5OUT & 0x10) == 0      /* i.e., LED1 is on */
```

checks the value of bit 4 in P5OUT: a value of 0 in this peripheral bit turns this LED on. Predicates may be composed of any number of such basic predicates as the one above, composed with logical operators such as *and* or *or*.

This solution is generally correct also for both explanation types in that, for any given output's (range of) values, a predicate can be written—this is because all outputs are program variables, and a variable in software for the MSP430 microcontroller we use is (i) finite, and more, (ii) up to 16 bits long. The solution is especially fit for 'short' outputs, such as this 1-bit LED control.

This predicate is added to the program source in the form of an assertion, `assert(p)`; this then serves as a program condition against which CBMC is able to explore the program for behaviour leading to states in which the predicate holds[5]. If the predicate, on the other hand, can never hold, CBMC returns with this result and without a program trace; this signals that an explanation for this input-output situation does not exist, i.e. the situation is not possible in this application.

---

[5] On a technical note, the predicates we describe are then *negated* for use with CBMC, as the model checking algorithm will search for program traces leading to a *violation* of any such predicate. In the remaining of this section, for readability, we present our predicates before negation.

3.4   Abstraction and results: *What if / How to* **explanations of application logic**

We give two *How to* explanations in Fig. 3; these pertain to the *Blink* application, for which *Input*={time} and *Output*={LEDs}.
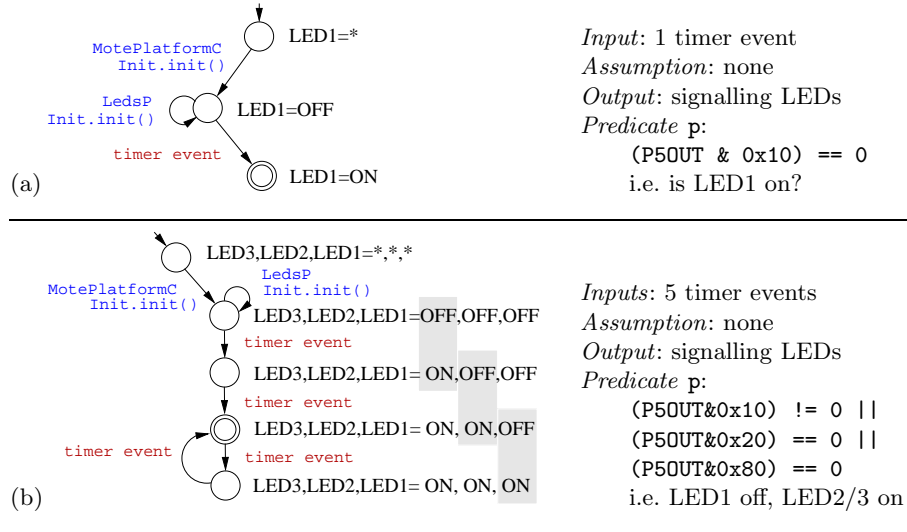

Fig. 3. *How to* explanations for *Blink*. In (a), the question posed is '*How to* have LED1 turned on?'. In (b), it is '*How to* have an off-on-on pattern on the three LEDs?'. For each example, we overview (on the right) the predicates used to generate the program trace, and we label the abstract states with valuations of the predicate (in (a)), and with valuations of each of the three subpredicates (in (b)). Edge labels are either input events (in red) or major internal program calls (in blue).

In Fig. 3(a), we ask: '*How to* have LED1 turned on?' i.e., including the boot process, system initialization, and main decision logic, what is the sequence of inputs leading to this LED state? For this, CBMC receives the program and the boolean predicate p asking whether LED1 is on, and returns one of the possible linear program execution traces leading to the desired state; initially, p has an unknown value, denoted by *. Similarly to the previous Fig. 1(b-d), we abstract the trace by organizing original states into fewer abstract states, one for each distinct value of a predicate, and present the result in the figure. We label the states with the corresponding predicate values, writing the predicate in full, for readability. We label the edges either simply with input events (highlighted in red), or also with adding checkpoints internal to the TinyOS implementation, which are relevant for programmers; for example, labelling an edge with the Init.init() interface command in the LedsP nesC component (in blue in the figure) signals that the LEDs are initialized at this point in the program.

To give an impression of the degree of compression in presenting this simple explanation, the *Blink* embedded source code is over 3000 lines of code; the program trace in this example contains 1192 states, which are abstracted into 3; the second abstract state subsumes almost 900 original program states.

Similarly, asking a slightly more complex question over how to obtain a particular lighting pattern on the three LEDs gives the answer from Fig. 3(b); for this, we label the states with individual truth values for each of the three sub-predicates in p. On a related note, an explanation to such *How to* questions may also *not exist*: asking TOSExp for the answer to the same question while allowing too few input events will return without a possible program execution trace.
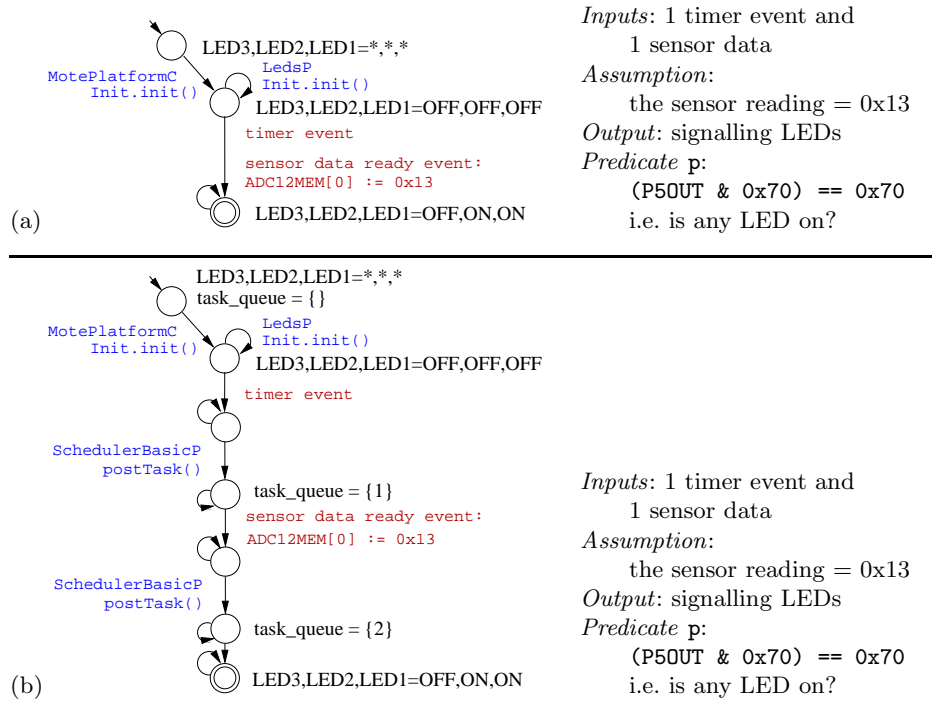

Fig. 4. *What if* explanations for *Sense*. In both (a) and (b), the question posed over *Sense* is '*What if* the voltage sensor reading would equal 0x13?'. An assumption over the ADC memory ADC12MEM implements this input value. The final state is determined by TOSExp to be an off-on-on LED pattern. In the explanation in (b), we allow more detail about the system's internal state, and thus include some of the values of the TinyOS scheduler's task queue (as in Subsection 2.2).

Fig. 4 gives *What if* explanations for *Sense*; *Sense* has twice the number of *Blink* states. An assumption over the value of an incoming sensor reading is

made, and the final output state depends on this: e.g., if we were assumed the data to be equal to 0x08 instead, the final state would be the existing state in which all LEDs are turned off.

3.5   Discussion: efficiency and limitations

Our technique does not suffer significantly from the expected efficiency issues related to the state-space explosion problem expected from model checking: all our CBMC runs terminate in under one minute. This efficiency is due to the fact that TOS2CProver already implements a sound state-space reduction technique, which we inherit in TOSExp.

However, a number of *limitations* remain for this approach to generating explanations. Specifically, a fair amount of knowledge on the internals of TinyOS is necessary for a programmer to experiment with generating new explanations; this, because both any assumptions on inputs and any predicate on outputs need to be written over low-level registers and peripherals, as seen in the examples above. This problem may be solved by also providing higher-level, pre-written templates for writing these assumptions and predicates.

Also, using bounded model checking as a program analysis method introduces one limitation in particular: with the method as is, it is difficult to recreate repetitive behaviour in the program—a program feature which may be of interest to programmers and users. This, because the method unwinds program loops of any size, effectively linearizing the explanation it extracts.

Furthermore, because our explanations are so precise to be bit-accurate and interrupt-accurate (as expected in the case of embedded systems, where precision of logic is key), when analysing large applications, a degree of user-friendliness in these models lacks: experimenting with input events which consist of a large number of repetitive interrupts (such as network events) crowds the model, and may require another degree of abstraction.

4   Related work: explaining context-aware embedded systems

Static extraction of explanations for embedded sensor systems

The standard technique to explain a TinyOS application from source is with `nesdoc`, the documentation tool included in TinyOS. This documentation is targeted to the specialist programmer, as it only preserves the program structure (i.e. nesC components, their interfaces and wiring, as overviewed in Subsection 2.2). This cannot be a solution to the problem of extracting explanations of context-aware behaviour, as it does not document the program logic.

Similarly to TOSExp, *FSMGen* [16] extracts from the source code a visual, user-readable, high-level finite-state machine modelling part of the `nesC` program's logic. This is done with the aim of aiding the programmer's understanding of the program logic across the same OS-wide program source, and—while is

uses symbolic execution as an analysis technique, it requires similar amount of time per analysis as TOSExp. The FSMGen models have a wider scope than our explanations, in that they represent all possible sequences of input transitions at once. However, unlike TOSExp, none of the inputs or outputs are quantified, a fact which precludes the extraction of the explanation types in [7]; the FSMGen models come closest to *How to* explanations.

Other program-analysis tools [17–19] for embedded sensor languages come from the validation and verification domain: they are designed to help the programmer detect if erroneous behaviour (e.g., standard programming errors such as memory violations) can arise in the application at compile-time, and provide a program trace (i.e., a *How to* explanation) of this behaviour. *T-Check* [17] and *KleeNet* [18] are based on model checking, testing, and symbolic execution; while, technically, the powerful analysis algorithms are eminently fitted to generate explanations out of program source (including at the level of a network of sensor nodes), they do not explicitly experiment with in this direction, or structure their program traces based on explanation types targeted at system users.

Explanations at runtime

Runtime monitoring techniques for sensor systems, such as *Safe TinyOS* [20] and *NodeMD* [21] attempt to offer *What* and *Why* explanations to a user or programmer located remotely from the sensor testbed. SafeTinyOS is an extension of the `nesC` language and compiler, which enables the sensor node running the program to have its status monitored remotely in the event of an operational exception, i.e., to (wirelessly) send to a preprogrammed destination a compressed-status message explaining the state of the program wherever it is determined that the application has reached an 'unsafe' state. For remote testbeds, it is the monitoring safety more than of status which takes priority, thus these explanations do not cover the system's output state, but the system's internal state instead.

As argued in [22], it is important that the system explains itself also when nothing goes wrong; therein is where the tools above still lack.

Explanations for other context-aware systems

Tools such as the Intelligibility Toolkit [1], Enactor [23] or Panoramic [24] form the state of the art in explanation generators for rule-based, context-aware systems which are heavier-weight than our embedded sensor systems. Moreover, these tools are targeted at explaining the *inference of complex context* (such as the type of activity currently taking place involving devices, locations and people), instead of sensing-to-actuation program logic. The Intelligibility Toolkit thus covers a more extensive set of decision logics for context inferences and both static and runtime explanation types, yet has the single disadvantage that the application must be programmed using this toolkit in the first place—for embedded sensor systems, a domain in which high-level programming and macroprogramming are not yet mainstream and embedded C programming is the norm, the

TOSExp approach in which the application itself needs no manual modification is more feasible.

5   Conclusions

In this contribution, we take the step of interfacing between (i) human-computer-interaction studies of *explaining* context-aware systems such as [7], and (ii) the area of embedded systems software analysis.

We experiment with automatically generating compact and readable input-to-output *Input,Output,What if* and *How to* explanations of rule-based sensor system logic, in a few degrees of detail, for sample TinyOS applications; these applications can thus be explained to users and programmers also in situations when the logic does not go wrong. We base our algorithm in known formal techniques for the extraction of mathematical models out of software, and model checking. TOSExp may make it easier for TinyOS programmers and the system's users to understand which behaviour the application can exhibit.

Besides addressing the current limitations of our algorithm (as overviewed in Section 3.5), future work includes user surveys to quantify the intelligibility of this style of explanations, and extending TOSExp to generate the remaining relevant types of explanations, e.g. *What, Why* and *Why not*; as these types require runtime generation, the difficulty resides less in the heavy-weight analysis, and more in efficiently instrumenting the application with status-logging mechanisms.

References

1. Lim, B.Y., Dey, A.K.: Toolkit to support intelligibility in context-aware applications. In: Proceedings of the 12th ACM international conference on Ubiquitous computing. Ubicomp '10, New York, NY, USA, ACM (2010) 13–22
2. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., Pister, K.: System architecture directions for networked sensors. SIGPLAN Not. **35**(11) (2000) 93–104
3. Levis, P., Gay, D., Handziski, V., Hauer, J.H., Greenstein, B., Turon, M., Hui, J., Klues, K., Sharp, C., Szewczyk, R., Polastre, J., Buonadonna, P., Nachman, L., Tolle, G., Culler, D., Wolisz, A.: T2: A second generation OS for embedded sensor networks. Technical Report TKN-05-007, Technische Universität Berlin (2005)
4. Rubio, B., Diaz, M., Troya, J.M.: Programming approaches and challenges for wireless sensor networks. In: Proceedings of the Second International Conference on Systems and Networks Communications. ICSNC '07, Washington, DC, USA, IEEE Computer Society (2007) 36–43
5. Ko, A.J., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M., Erwig, M., Scaffidi, C., Lawrance, J., Lieberman, H., Myers, B., Rosson, M.B., Rothermel, G., Shaw, M., Wiedenbeck, S.: The state of the art in end-user software engineering (2011) Accepted for publication in ACM Computing Surveys.
6. Ko, A.J., Myers, B.A., Coblenz, M.J., Aung, H.H.: An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. IEEE Trans. Softw. Eng. **32** (December 2006) 971–987

7. Lim, B.Y., Dey, A.K.: Assessing demand for intelligibility in context-aware applications. In: Proceedings of the 11th international conference on Ubiquitous computing. Ubicomp '09, New York, NY, USA, ACM (2009) 195–204
8. Bucur, D.: On Software Verification for Sensor Nodes. The Journal of Systems Software (2011) To appear. DOI 10.1016/j.jss.2011.04.054.
9. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS. Volume 2988 of LNCS., Springer (2004) 168–176
10. Kroening, D.: Formal verification. `http://www.cprover.org/` (2010)
11. Gay, D., Levis, P., von Behren, R.: The nesC language: A holistic approach to networked embedded systems. In: ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI), ACM (2003) 1–11
12. Moteiv Corporation: Telos. Ultra low power IEEE 802.15.4 compliant wireless sensor module. Revision B : Humidity, Light, and Temperature sensors with USB. `http://www.moteiv.com` (2004)
13. Texas Instruments: MSP430x1xx family — user's guide (Rev. F). `www.ti.com, slau049f.pdf` (2006)
14. Underwood, S.: Mspgcc—A port of the GNU tools to the Texas Instruments MSP430 microcontrollers. `http://mspgcc.sourceforge.net` (2003)
15. Gay, D., Levis, P., Culler, D.: Software design patterns for TinyOS. In: Proceedings of the ACM SIGPLAN/SIGBED Conference on languages, compilers, and tools for embedded systems (LCTES), ACM (2005) 40–49
16. Kothari, N., Millstein, T., Govindan, R.: Deriving state machines from TinyOS programs using symbolic execution. In: Proceedings of the international conference on Information Processing in Sensor Networks (IPSN), IEEE (2008) 271–282
17. Li, P., Regehr, J.: T-Check: bug finding for sensor networks. In: Proceedings of the 9th International Conference on Information Processing in Sensor Networks (IPSN), ACM (2010) 174–185
18. Sasnauskas, R., Landsiedel, O., Alizai, M.H., Weise, C., Kowalewski, S., Wehrle, K.: 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). (2010) 186–196
19. Mottola, L., Voigt, T., Österlind, F., Eriksson, J., Baresi, L., Ghezzi, C.: Anquiro: Enabling efficient static verification of sensor network software. In: Proceedings of Workshop on Software Engineering for Sensor Network Applications (SESENA) ICSE(2). (2010)
20. Cooprider, N., Archer, W., Eide, E., Gay, D., Regehr, J.: Efficient memory safety for TinyOS. In: Proceedings of the conference on Embedded Networked Sensor Systems (SenSys), ACM (2007) 205–218
21. Krunic, V., Trumpler, E., Han, R.: NodeMD: Diagnosing node-level faults in remote wireless sensor systems. In: Proceedings of the international conference on Mobile Systems, Applications and Services (MobiSys), ACM (2007) 43–56
22. Kofod-Petersen, A., Cassens, J.: Explanations and context in ambient intelligent systems. In: Proceedings of the 6th international and interdisciplinary conference on Modeling and using context. CONTEXT'07, Springer-Verlag (2007) 303–316
23. Dey, A.K., Newberger, A.: Support for context-aware intelligibility and control. In: Proceedings of the 27th international conference on Human factors in computing systems. CHI '09, New York, NY, USA, ACM (2009) 859–868
24. Welbourne, E., Balazinska, M., Borriello, G., Fogarty, J.: Specification and verification of complex location events with panoramic. In: Pervasive. (2010) 57–75